View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Repositorio Universidad de Zaragoza

Trabajo Fin de Grado en Ingeniería en Tecnologías y Servicios de Telecomunicación

Sistema de fuzzing para identificación de tarjetas NFC

Pablo Del Val Egido

Director: Ricardo J. Rodríguez Fernández

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre de 2020
Curso 2019/2020

Resumen

Near Field Communication (NFC) es una tecnología de identificación por radiofrecuencia que permite la comunicación inalámbrica entre dos dispositivos. En los últimos años esta tecnología se ha ido imponiendo en numerosos elementos, entre ellos el DNI electrónico, las tarjetas del transporte público o las tarjetas de crédito y débito. Estas últimas son utilizadas para realizar pagos en múltiples comercios con tan solo acercar la tarjeta a un terminal punto de venta y verificar la identidad del usuario mediante la introducción de un código PIN. Por la importancia que tienen como medio de pago, estas tarjetas deben cumplir unas especificaciones concretas (EMV) que ayudaron a definir las tres grandes compañías que operaban con tarjetas de crédito: Europay, Mastercard y Visa.

En este trabajo de fin de grado se pretende desarrollar un sistema para verificar, mediante técnicas automáticas, el normal funcionamiento y adecuación al estándar EMV de las tarjetas NFC. En concreto, el sistema hará uso de las técnicas de fuzzing, consistentes en proporcionar datos inválidos, inesperados o aleatorios a la entrada de un sistema. Con el uso de estas técnicas se pretende encontrar vulnerabilidades o problemas de seguridad en el objetivo. En este trabajo se utilizan estas técnicas para alterar los comandos que se establecen en una transacción EMV. Tras ello se analizan las respuestas obtenidas y se trata de encontrar particularidades entre las tarjetas estudiadas. El objetivo principal de la aplicación de estas técnicas consiste en detectar elementos que permitan identificar de manera inequívoca una tarjeta NFC.

Índice general

1. Introducción **5**
1.1. Motivación 5
1.2. Objetivos 5
1.3. Organización 6

2. Conocimientos preliminares **7**
2.1. Tecnología NFC y normas ISO/IEC 14443 y 7816 7
2.1.1. Norma ISO/IEC 14443 7
2.1.2. Norma ISO/IEC 7816 8
2.2. Tarjetas EMV 9
2.2.1. EMV Book 3 9

3. Herramientas de fuzzing **11**
3.1. Historia del fuzzing 11
3.2. Técnicas de fuzzing 11
3.3. Comparativa de Fuzzers 12
3.3.1. SPIKE 12
3.3.2. Peach 13
3.3.3. Sulley 13
3.3.4. Boofuzz 14
3.4. Comparativa de fuzzers estudiados 14

4. Sistema desarrollado **15**
4.1. Integración de las herramientas 15
4.2. Problemas encontrados 16

5. Experimentos y resultados **19**
5.1. Pruebas realizadas 19
5.2. Resultados obtenidos 21

6. Conclusiones y trabajo futuro **29**

A. Extensión temporal del trabajo **33**

Capítulo 1

Introducción

1.1. Motivación

En la actualidad, los pagos mediante *Near Field Communication* (NFC, del inglés comunicación en el campo cercano) ya sean con tarjetas de crédito o con dispositivos móviles están muy presentes en el día a día en numerosos comercios [8]. Una de sus principales ventajas es la rapidez con la que se realizan los pagos. Sin duda, tras la pandemia producida por el coronavirus, los pagos con tarjetas de crédito han aumentado, llegando incluso en algunos comercios a rechazar los pagos con dinero en efectivo (como por ejemplo, en el pago del transporte público en Zaragoza).

Estas tarjetas de crédito deben cumplir ciertas especificaciones para realizar las transacciones de forma segura ya que contienen datos personales y confidenciales. Por tanto, es importante comprobar la seguridad ofrecida por estas tarjetas y el correcto funcionamiento de las aplicaciones integradas en las mismas.

Por ello mediante el uso de un sistema de fuzzing, se va a tratar de encontrar diferencias entre las implementaciones de los protocolos necesarios en diferentes tipos de tarjetas. Observando las diferentes respuestas de cada tarjeta, se espera poder clasificarlas en base a las funciones que implementan. En cuanto a líneas de trabajo relacionadas, se ha encontrado un trabajo similar donde se realizaba fuzzing sobre tarjetas EMV [15]. En dicho trabajo se encuentran fallos de seguridad en la implementación de algunas de las tarjetas utilizadas.

1.2. Objetivos

Los objetivos abordados en este TFG son:

a) Conocer cómo funciona el protocolo NFC y las herramientas y hardware necesarios para comunicarse con tarjetas NFC.

b) Realizar un análisis de la evolución en las técnicas de fuzzing y el software disponible para ejecutar dichas técnicas.

c) Utilizando estas técnicas de fuzzing, detectar características únicas de tarjetas NFC para identificarlas.

1.3. Organización

El trabajo se divide en seis capítulos. En el capítulo 1 se especifican tanto la motivación para realizar este trabajo como los objetivos abordados. En el capítulo 2 se realiza una introducción a la tecnología NFC y las normas que cumplen las tarjetas EMV. En el capítulo 3 se habla sobre el fuzzing y se realiza una comparativa sobre la evolución de los sistemas para la creación de fuzzers. En el capítulo 4 se detalla la integración de las distintas herramientas en el sistema desarrollado y los principales problemas encontrados. En el capítulo 5 se explican las pruebas que se han realizado y los resultados obtenidos. Finalmente, en el capítulo 6 se realiza una conclusión del trabajo realizado.

Capítulo 2

Conocimientos preliminares

En este capítulo se introducen los conceptos, normativas y protocolos necesarios para la comprensión de las comunicaciones establecidas en este trabajo. En primer lugar, se define la tecnología NFC junto a los estándares que utiliza y a continuación se definen las comunicaciones en las tarjetas EMV.

2.1. Tecnología NFC y normas ISO/IEC 14443 y 7816

La tecnología *Near Field Communication* (NFC) engloba a la familia de estándares para establecer una comunicación inalámbrica entre dos dispositivos en proximidad. Estos estándares NFC están basados en otros estándares de identificación por radio frecuencia (*Radio Frequency Identification,* RFID) como ISO/IEC 14433 [10–13].

2.1.1. Norma ISO/IEC 14443

Dentro de ISO 14443 se pueden distinguir dos tipos de tecnologías: 14443-A y 14444-B, que aunque trabajan en la misma frecuencia, se diferencian en la modulación utilizada, la codificación y los protocolos de inicialización. El estándar consta de 4 partes:

1. ISO/IEC 14443-1 [11]: Describe las características físicas.
2. ISO/IEC 14443-2 [13]: Describe la potencia y señal de la radiofrecuencia.
3. ISO/IEC 14443-3 [10]: Detalla los algoritmos de inicialización y anti-colisión.

   La comunicación está formada por un elemento activo, *Proximity coupling device* (PCD) y un elemento pasivo, *Proximity inductive coupling card* (PICC). El PCD hace un sondeo del medio mediante órdenes *Request* REQA, hasta que un PICC entra en el campo electromágnetico y es activado, momento en el que responde con un *Answer to request* (ATQA) a la espera de ser seleccionado.

   Si el PCD recibe más de un ATQA, ejecuta un algoritmo de anti-colisión. Este algoritmo consiste en enviar un SELECT con una máscara de bits a los PICCs. Estos solo responderán si la máscara coincide con su identificador. En cada iteración, esta máscara se restringe más hasta que tan solo contesta un PICC con un mensaje *Select Acknowledge* (SAK), donde determina el tipo de tarjeta que es y los protocolos que soporta.

4. ISO/IEC 14443-4 [12]: Protocolo de transmisión.

   En esta parte de la norma se define el protocolo de transmisión, en el que mediante unas órdenes RATS y PPS, se definen los parámetros necesarios para el canal lógico de comunicación.

   En función del SAK recibido, el PCD envía una orden *Request ATS* (RATS), que el PICC debe responder con la respuesta *Answert To Select* (ATS). Esta respuesta contiene parámetros como los tamaños de ventana, los tiempos de respuesta máximos para la configuración del canal, bytes históricos... Una vez establecido este canal lógico, será sobre el que se envíen los mensajes APDU.

2.1.2. Norma ISO/IEC 7816

El estándar ISO/IEC 7816 se reparte en un total de 15 documentos donde se recogen las especificaciones de las tarjetas inteligentes. En la parte 4 [14] se define la organización y seguridad de las órdenes para intercambiar información.

Los mensajes *Application Protocol Data Unit* (APDU), incluidos en la parte 4 de esta norma ISO 7816, son los que utiliza NFC para enviarlos a través del canal lógico establecido en ISO 14443-4.

Hay dos tipos de APDUs, los comandos y las respuestas. Los comandos son enviados desde el lector (PCD) a la tarjeta (PICC) y las respuestas en sentido contrario.

Como se muestra en la figura 2.1, los comandos están formados por una cabecera obligatoria de 4 bytes y opcionalmente hasta un máximo de 255 bytes de datos.

El primer byte es el byte de clase. A continuación se envía un byte que contiene la instrucción a realizar. Tras este byte se envían dos bytes que indican los parámetros de la instrucción. Por último, opcionalmente se envía un byte con el tamaño del bloque de datos enviado, seguido por el propio bloque de datos. Tras este bloque de datos se envía un byte con el tamaño de la respuesta esperada por parte de la tarjeta.

Como se muestra en la figura 2.2 las respuestas contienen hasta 256 bytes de datos con la respuesta en caso de que exista y siempre terminan con dos bytes donde se indica el estado del procesamiento del comando (SW1 y SW2 en la figura).

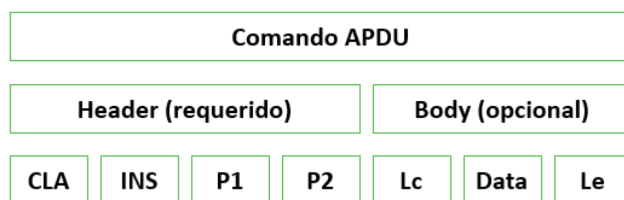

Figura 2.1: Comando APDU

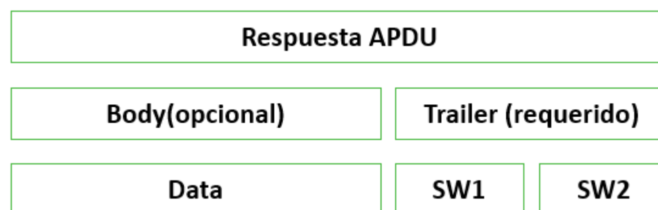

Figura 2.2: Respuesta APDU

2.2. Tarjetas EMV

EMV es un estándar de comunicación entre tarjetas con circuito integrado y TPV (terminal punto de venta). El nombre EMV es un acrónimo de las tres compañías que colaboraron en el desarrollo del estándar "Europay, Mastercard y VISA".

EMV presenta un marco general para la definición de pagos seguros. Estas especificaciones se distribuyen en cuatro documentos:

1. Libro 1: Interfaz entre la tarjeta y el terminal, independiente de la aplicación.
2. Libro 2: Seguridad y gestión de claves.
3. Libro 3: Especificación de la aplicación.
4. Libro 4: Interfaz de titular de tarjeta, comerciante y comprador.

En este trabajo resulta de interés las especificaciones de la aplicación definidas en el libro 3 [9], el cual se explica con un poco más de detalle a continuación.

2.2.1. EMV Book 3

EMV constituye tanto un conjunto de protocolos como un marco a partir del cual se pueden desarrollar protocolos propios. Las reglas establecidas en estas especificaciones imponen funcionalidades mínimas a ser implementadas.

Una transacción EMV se compone de 8 etapas como se muestra en la figura 2.3, adaptado de [15]. En primer lugar se selecciona una aplicación mediante el comando `SELECT`, que permite al terminal seleccionar una aplicación presente en la tarjeta por su identificador único, el AID. A continuación, se inicializa la transacción mediante el comando `GET PROCESSING OPTIONS`, que permite informar a la tarjeta del inicio de la transacción e intercambiar la información necesaria para la misma. Una vez inicializada la transacción, se leen los datos de la tarjeta mediante los comandos `READ RECORDS` y `GET DATA`. Tras leer los datos, se pasa a realizar la autenticación de la tarjeta, que se puede realizar de manera estática o dinámica.

En el caso de la autenticación estática, el terminal realiza una firma criptográfica de clave pública de los datos estáticos leídos de la tarjeta para validar su autenticidad. En cambio, la autenticación dinámica se realiza mediante el comando `INTERNAL AUTHENTICATE`, que permite al terminal autenticar la tarjeta. En este caso, se debe recalcular una nueva firma para cada transacción. Para autenticar al titular de la tarjeta, se utiliza el código PIN (número de identificación personal), que el usuario envía directamente al terminal mediante el comando `VERIFY`. En este punto el terminal decide si continúa con la transacción o no para evitar una transacción cuyo resultado sería el rechazo en cualquier caso. Una vez comprobado, el terminal envía

a la tarjeta un primer comando `GENERATE APPLICATION CRYPTOGRAM` que representa el núcleo de la transacción y refleja la decisión tomada por el terminal. Este comando indica la moneda utilizada, la cantidad y las capacidades del terminal. En este punto el terminal indica si acepta la transacción o no y para evitar posibles fraudes, se valida la identidad del emisor enviando un segundo comando `GENERATE APPLICATION CRYPTOGRAM`, procesado mediante el comando `EXTERNAL AUTHENTICATE` [15].

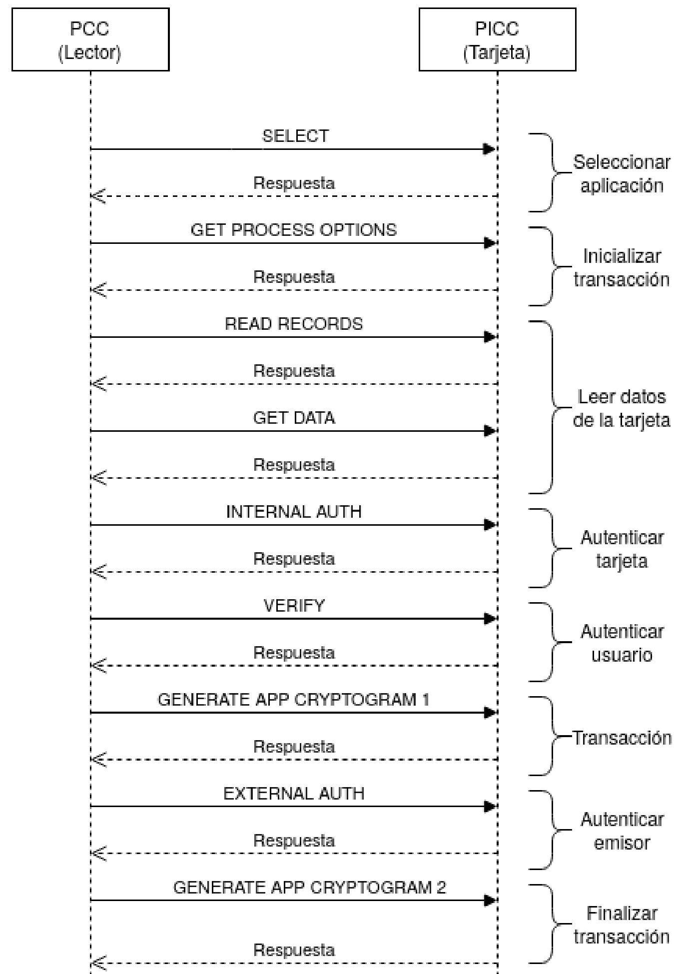

Figura 2.3: Transacción EMV

Capítulo 3

Herramientas de fuzzing

En este capítulo se describe brevemente la historia del fuzzing y se realiza una clasificación de las técnicas utilizadas para llevarlo a cabo. Por último, se realiza una comparativa entre algunas herramientas que se pueden utilizar para este propósito.

3.1. Historia del fuzzing

El término fuzzing fue utilizado por primera vez en 1990 por el profesor Barton Miller en la Universidad de Wisconsin. Mientras estaba realizando una conexión serie con un terminal, se dio cuenta que debido al ruido en la línea producido por una tormenta, se estaban enviando carácteres aleatorios junto a los comandos que enviaba. Además de los problemas de usabilidad que esto causaba, se observó que algunas de las entradas corruptas causaban fallos en el sistema UNIX que recibía estos mensajes.

Esto llevó a Miller, junto a otros investigadores de la universidad, a crear un programa que generara datos de entrada aleatorios (*fuzz*) y otro para testear utilidades del sistema de manera interactiva (*ptyjig*) [16].

Desde entonces, las técnicas de fuzzing se han hecho más populares y se han ido mejorando para detectar fallos de seguridad y vulnerabilidades en software mediante el envío de datos inválidos inesperados o aleatorios a las entradas de un programa de ordenador.

3.2. Técnicas de fuzzing

Las técnicas de fuzzing se pueden clasificar de distintas formas. Si se piensa en el objetivo, se pueden dividir las técnicas en función del conocimiento que se tenga sobre el sistema en tres categorías: cajas negras, cajas blancas y cajas grises. En la técnica de cajas negras, no se considera la lógica interna del programa y simplemente se observa la salida generada para distintos datos de entrada. En el lado opuesto, se encuentran las cajas blancas, donde se obtiene información detallada sobre el objetivo y acceso a todos los recursos necesarios como documentación, especificaciones de diseño o código fuente. Por último, en un punto intermedio se sitúan las cajas grises, donde se obtiene algo de información acerca del objetivo pero no es necesario estudiarlo en detalle.

Se podría pensar que las técnicas de caja blanca son más efectivas que las de caja negra, ya que se tiene acceso al código fuente completo. Sin embargo, esto no es necesariamente cierto ya que el proceso de compilación del software puede provocar cambios al compilar el código fuente en

código ensamblador, por lo que puede haber variaciones entre el código revisado y el ejecutado. Por otro lado, las técnicas de caja blanca son la manera más eficiente de realizar pruebas, ya que usando cajas blancas los métodos de cajas negras están igualmente disponibles pero no al revés. Una buena práctica es combinar ambas técnicas, ya que aunque las cajas blancas ofrezcan todos los elementos a probar, puede quedar alguna parte del software fuera de la documentación. Además, el acceso a todos los componentes puede ser una distracción.

Otra forma de clasificarlas es en función de la manera en la que se generan los datos. Se pueden tener sistemas basados en mutaciones de datos o en generación de datos. En los primeros los datos se crean de manera aleatoria dada una semilla de entrada. Esto puede provocar que el objetivo rechace los datos debido a que no cuadran con el formato esperado. En el caso de la generación de datos, se estudia el objetivo y se acomodan los datos al protocolo que utilice el mismo. Esto resulta más costoso en tiempo pero se obtienen resultados más precisos.

También se pueden clasificar estos sistemas dependiendo de si existe realimentación o no. En el caso de tener realimentación, se analizan las respuestas a determinadas entradas con el objetivo de mejorar la precisión en la siguiente ejecución. Estos análisis se pueden realizar de manera dinámica o estática. Si se realiza de manera dinámica, la información se procesa en tiempo de ejecución y las entradas van variando en función de las salidas. En cambio, si se realiza un análisis estático, las salidas se analizan una vez termina la ejecución.

3.3. Comparativa de Fuzzers

En esta sección se van a analizar diferentes herramientas de fuzzing disponibles en el mercado. Existe una gran cantidad de herramientas para este propósito pero este análisis se centra en las específicas para fuzzing NFC o fuzzing de protocolos, ya que es el que se va a emplear en este trabajo.

3.3.1. SPIKE

Un gran paso en el desarrollo de herramientas fuzzing se dio en 2001, cuando Dave Aitel publicó SPIKE y posteriormente lo presentó en la conferencia Black Hat USA en 2002 [7].

SPIKE está escrito en C y presenta una API para el desarrollo rápido y eficiente de fuzzers de protocolo de red. SPIKE se presenta como una herramienta de código abierto con licencia *GNU General Public License.*

SPIKE define una serie de primitivas que permiten a los programadores construir mensajes que se envían a través de la red. Este framework no está muy bien documentado pero viendo los scripts escritos en la distribución se puede sacar una lista de algunas primitivas que permite usar.

Entre las funciones más útiles de SPIKE, destacan las siguientes:

a) Tiene una gran cantidad de cadenas integradas para fuzzing que producen una amplia variedad de errores en los programas. Determina qué valores se deben enviar a la aplicación objetivo para causar fallos que resulten de interés.

b) El concepto de bloques consiste en representar la capa de datos de un protocolo mediante bloques de datos anidados. Estos bloques anidados pueden definirse con diversos formatos. Se puede usar este concepto para calcular los tamaños de secciones específicas que se pueden insertar dentro de los propios bloques. Esto es de gran utilidad cuando se utilizan protocolos en los que un campo requiere especificar el tamaño.

c) SPIKE admite varios tipos de datos diferentes que se usan en protocolos de red y se pueden insertar en diferentes formatos, lo que permite cortar y pegar fácilmente desde distintos programas.

3.3.2. Peach

Peach se creó originalmente como un framework de código abierto para realizar fuzzing. Sin embargo, en 2014, la herramienta se actualizó a una versión cerrada 2.0 y ha pasado a ser una herramienta enfocada al uso profesional. Recientemente esta herramienta ha sido adquirida por GitLab [18].

En comparación con otras herramientas de fuzzing, la arquitectura de Peach es la más flexible y permite una mayor reutilización de código. Este framework ofrece una serie de componentes para la construcción de fuzzers entre los que se incluyen: generadores, transformadores, protocolos, editores y grupos.

Los generadores se encargan de generar los datos ya sean cadenas simples o binarios. La abstracción de la generación de datos permite la reutilización de fragmentos de código de otros fuzzers ya implementados. Los transformadores pueden ser un codificador que se encarga de transformar datos entre diferentes codificaciones. De nuevo estos se pueden abstraer y permiten reutilizar código ya implementado. Los editores se encargan de transportar los datos generados a través de un protocolo de manera transparente. Los grupos están formados por uno o más generadores y se utilizan para variar los datos que el fuzzer produce.

La arquitectura de Peach permite al programador centrarse en los componentes individuales de un protocolo para luego juntarlos todos en un fuzzer. Esta arquitectura es más lenta de crear que la basada en bloques pero con el uso y la reutilización de código se va facilitando la creación de nuevos fuzzers.

Aunque Peach está implementada con Python, la configuración y protocolos se describen mediante lenguaje XML, lo que hace que sea más complicado su uso. Se puede decir que Peach es una herramienta enfocada a un uso profesional y que aparte de las opciones de fuzzing permite ejecutar otro tipo de pruebas de seguridad para probar software y protocolos.

3.3.3. Sulley

Sulley es un framework de fuzzing escrito en Python formado por varios componentes extensibles. El objetivo de Sulley es simplificar tanto la respresentación de los datos como la transmisión de los mismos y el monitoreo del objetivo [2].

La mayoría de fuzzers actuales se centran únicamente en la generación de datos. Sulley no solo tiene una generación de datos interesante, sino que incluye muchos otros aspectos importantes que debe incluir un fuzzer. Sulley observa la red y mantiene registros metódicamente, instrumenta y monitorea el objetivo y es capaz de recuperarlo de un estado de error. Sulley detecta, rastrea y categoriza los fallos detectados, puede hacer fuzz en pararelo y determinar qué secuencia de casos de prueba desencadena fallos.

El uso de Sulley se puede dividir en tres partes:

1. **Representación de datos**: Este es el primer paso para usar cualquier fuzzer. Sulley, basado en SPIKE, utiliza una representación de datos basada en bloques. Cada bloque está formado por primitivas y éstas se pueden alterar de manera individual.

2. **Sesión**: Esta es una de las principales ventajas de Sulley frente a otros fuzzers. En una sesión, se vinculan los distintos bloques desarrollados, junto a los agentes de supervisión de Sulley (socket, depurador, etc.) y se inicializa el fuzzing.

3. **Post mortem**: Por último, una vez que la sesión ha finalizado se revisan los datos generados y los resultados monitoreados. A partir de aquí se pueden reproducir casos de prueba individuales.

Una de las principales desventajas de Sulley es que salvo algún manual de uso, posee poca documentación. Además, con la aparición de Boofuzz (descrito a continuación) como su sucesor se quedó sin soporte.

3.3.4. Boofuzz

Boofuzz es un framework escrito en Python que tiene a Sulley como su predecesor pero que trae muchas mejoras [5]. Al igual que Sulley, Boofuzz incorpora una manera fácil y rápida de generar los datos, detección de fallos, recuperación del objetivo tras fallos y el monitoreo de los datos testeados. Sin embargo Boofuzz también trae algunas mejoras como la inclusión de una documentación online, una instalación más sencilla, soporte para extender la herramienta y la corrección de algunos fallos en el software. También viene con soporte incorporado para fuzzing en serie sobre capas IP, Ethernet y transmisiones UDP y la posibilidad de exportar los resultados en formato CSV.

Boofuzz permite desarrollar una herramienta de fuzzing especificando un protocolo y su formato y con eso él se encarga de generar las mutaciones específicas.

En Boofuzz el objeto más importante es la sesión. Cuando se crea una sesión es necesario pasarle un objetivo con una conexión donde se encuentran las posibilidades de incluir a sockets TCP, UDP o SSL entre otros. Una vez la sesión está preparada, hay que definir la estructura de mensajes del protocolo elegido. Para ello, igual que en Sulley, se utiliza una estructura de bloques.

Una vez definidos todos los mensajes, solo queda conectarlos utilizando la sesión que se ha creado anteriormente y ejecutar el fuzzing.

3.4. Comparativa de fuzzers estudiados

En la tabla 3.1 se muestra una comparativa de las características que se han tenido en cuenta para la selección del fuzzer. Se ha elegido Boofuzz porque es la única herramienta que siendo software libre tiene documentación y soporte. Además, la creación de nuevos bloques de datos resulta más sencillo.

	Documentación	Soporte	Licencia	Lenguaje	Generación de datos	Monitoreo	Usabilidad
Spike	No	No	Sofware libre	C	Basada en bloques	No	Base para crear otros fuzzers
Sulley	No	No	Software libre	Python	Basada en bloques	Sí	Facilidad generar bloques Python
Boofuzz	Online	Sí	Software libre	Python	Basada en bloques	Sí	Facilidad generar bloques Python
Peach	No	Sí	Comercial	XML	Reutilización de código	Sí	Más líneas de código con XML

Tabla 3.1: Comparativa de fuzzers

Capítulo 4

Sistema desarrollado

En este capítulo se explican las distintas partes que componen el sistema desarrollado y la integración de las mismas. También se muestran algunos de los problemas encontrados durante el desarrollo del sistema, así como la resolución de estos problemas.

4.1. Integración de las herramientas

El sistema desarrollado se compone de cuatro elementos: el generador de datos, el programa para comunicarse con el lector NFC, el propio lector NFC para la comunicación con las tarjetas y el programa para analizar los resultados. En la figura 4.1 se muestra un diagrama de alto nivel donde se pueden ver todas las partes que forman el sistema.

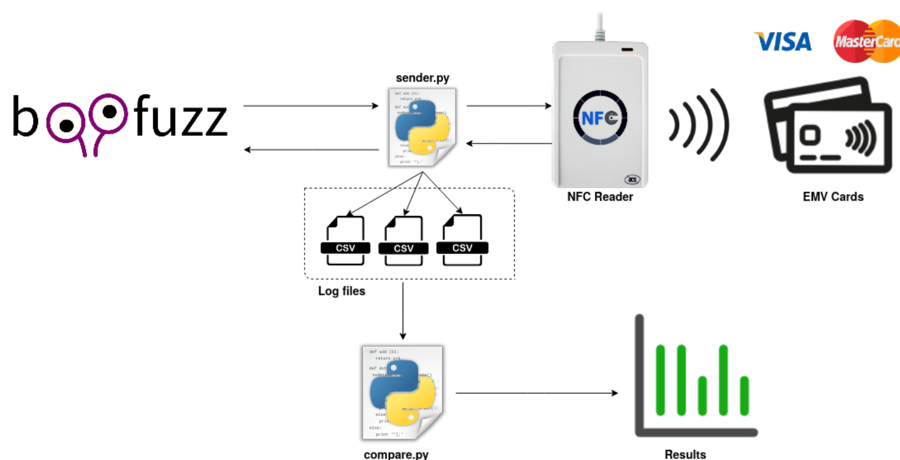

Figura 4.1: Diagrama alto nivel

Como se ha explicado anteriormente, se ha elegido Boofuzz como generador de datos porque actualmente es el que más documentación tiene para desarrollar nuevos bloques de datos. Para el uso de esta herramienta es necesario crear un script de Python en el que se definen los bloques de datos, la sesión y el objetivo. En el capítulo 5 se explican en detalle los bloques de datos creados. Para crear la sesión, únicamente se ha definido el tiempo entre cada mensaje enviado y

la opción para que tras cada mensaje se quede esperando a recibir una respuesta antes de enviar el siguiente. Tras esto se conectan los distintos bloques a la sesión para que siempre que se envíe un comando APDU, sea con la aplicación de la tarjeta seleccionada tal y como se muestra en la figura 4.2.

Figura 4.2: Envío de APDUs

A continuación, estos datos son recibidos por el programa `sender.py`, que es el encargado de la comunicación con el lector NFC. Boofuzz y el programa `sender.py` se comunican mediante el uso de dos socket UDP (uno en cada sentido de la comunicación). Para la comunicación con el lector NFC, se utiliza la librería `nfcpy` [4]. Esta librería incluye la clase `nfc.clf.ContactlessFrontend`, que permite abrir una conexión entre el dispositivo conectado por USB y la tarjeta NFC que se le aproxime. De este modo, se establece el canal lógico para el envío de datos y mediante el objeto `tag` se envían y reciben bytes de datos entre la tarjeta y el lector. Los bytes de datos recibidos con las respuestas se pasan a formato hexadecimal y se almacenan en ficheros con formato `Comma Separated Value` (CSV) para su posterior análisis.

El lector NFC escogido ha sido el `ACR122U-A9` por ser un lector asequible y que funciona bien con este tipo de tarjetas. Este lector es el encargado de realizar la comunicación con la tarjeta. En la siguiente sección se detallan algunos problemas y su resolución para su correcta instalación.

Por último, para realizar el análisis de los datos se ha creado el programa `compare.py`. Este programa es el encargado de abrir los distintos ficheros de log generados. Utilizando la librería `pandas` [6] se pueden almacenar todos los datos para poder procesarlos conjuntamente. Los datos recibidos se separan en función de a qué tarjeta pertenecen y a qué tipo de APDU están respondiendo. De esta manera se obtiene el número de mensajes recibidos de cada tipo para cada comando enviado y a partir de estos datos se construyen las gráficas para una mejor visualización de los resultados.

4.2. Problemas encontrados

En esta sección se describen los problemas encontrados durante el desarrollo del sistema y la integración de las distintas partes que lo componen. Entre ellos están la instalación del hardware y software necesarios, la generación de los datos y entrega de los mismos a la tarjeta, el formato de almacenamiento de las respuestas y el análisis de los resultados obtenidos.

El primer problema es encontrar la manera de establecer comunicaciones con las tarjetas EMV. Para ello es necesario la instalación de un lector NFC. En este caso se ha optado por el lector `ACR122U-A9`, como se ha indicado anteriormente. Para la instalación de este lector se ha utilizado como referencia la guía de [17] en la que se instalan los paquetes necesarios, se descargan los drivers y adicionalmente se modifica el archivo `/etc/modprobe.d/blacklist.conf` para evitar conflictos entre el proceso pcscd y el driver pn533 durante la inicialización. Tras esta instalación, siguen apareciendo problemas de inicialización debido a que esta versión del lector tiene fallos cuando están activados el LED y la vibración. Para desactivar la vibración, es necesario enviar un comando en modo directo mediante una herramienta proporcionada por el propio lector. Siguiendo el manual, se envía el comando `FF 00 52 00 00` para apagar la vibración, descrito en la sección 6.7 del manual [1]. Por último y tras tener la vibración desactivada, el lector

sigue comportándose de manera extraña, dando fallos de inicialización en algunas ocasiones y funcionando correctamente en otras. Para solucionar este problema, es necesario inicializar y parar el servicio `etc/init.d/pcscd` haciendo uso de los comandos `start` y `stop` cada vez que se conecta el lector.

Una vez resuelta la inicialización del lector NFC, mediante un script de Python y haciendo uso de la librería `nfcpy`, se establece una comunicación con la tarjeta NFC para ver que responde correctamente. El problema encontrado aquí es que para conectar el lector NFC es necesario pasarle al programa la ruta donde está conectado el lector USB y esta ruta cambia cada vez que se conecta. Para solucionar este problema, desde el programa `sender.py` se lanza el comando `lsusb` y se busca el lector ACR122 para obtener su ruta y abrir la interfaz correctamente en cada ejecución de manera autónoma.

El siguiente paso es realizar la generación de datos con la herramienta Boofuzz. Para ello se utiliza un script en Python. Con la ayuda de la documentación se pueden definir los bloques con facilidad. El problema surge cuando se quieren entregar estos bloques a la tarjeta. Boofuzz permite la comunicación mediante Sockets, pero no a través del puerto USB. Por ello es necesario enviar los datos a través de un socket UDP al programa `sender.py`, que se encarga de recibir estos paquetes y reenviarlos al lector NFC mediante la conexión establecida sobre NFC.

A continuación, es necesario almacenar las respuestas que devuelve la tarjeta para poder analizarlas posteriormente. Por un lado, hay que guardar la respuesta recibida por la tarjeta y por otro los datos que se han enviado para obtener esa respuesta. A estos datos hay que añadirle un campo que represente para qué comando APDU se ha realizado la ejecución y una etiqueta que represente la tarjeta.

Capítulo 5

Experimentos y resultados

En este capítulo se definen los diferentes tests realizados sobre las diferentes tarjetas y un análisis de los resultados obtenidos.

5.1. Pruebas realizadas

Una vez está establecida la conexión entre el programa generador de datos y las tarjetas, hay que definir los mensajes con los que se quiere realizar el fuzzing. En este caso se han definido mediante el uso de bloques de datos que proporciona Boofuzz los mensajes necesarios para realizar una transacción EMV.

Los archivos de una tarjeta inteligente están organizados en una estructura de árbol. El archivo superior es el archivo maestro `Master File` (MF). El MF tiene uno o más archivos de definición de aplicaciones, llamados `Application Definition Files` (ADF). Dentro de un ADF hay archivos de aplicación elementarios, denominados `Application Elementary Files` (AEF), que contienen datos. Se puede seleccionar un ADF con el identificador de aplicación `Application Identifier` (AID). Dentro de un ADF se pueden seleccionar los AEF con el identificador corto de archivos `Short File Identifier` (SFI).

Si se conocen los AIDs de las aplicaciones que contiene la tarjeta, se pueden seleccionar directamente. En cambio, si no se conocen, primero se tiene que enviar un comando `SELECT` al directorio `Directory Definition File` (DDF) llamado `1PAY.SYS.DDF01`. Este DDF es lo que corresponde al archivo maestro que contiene entradas para una o más aplicaciones.

Este bloque de datos es el primero que se define en Boofuzz. A continuación se define otro `SELECT` para seleccionar la aplicación. En una primera ejecución no se conocen las aplicaciones de la tarjeta, pero tras observar la respuesta del primer `SELECT` se obtienen las dos aplicaciones que hay dependiendo del tipo de tarjeta. En la figura 5.1 se ve la respuesta al primer `SELECT` de una tarjeta Mastercard. Para mostar el contenido del mensaje se ha utilizado la herramienta `emv-bertlv` [3].

Figura 5.1: Respuesta Mastercard `SELECT 1PAY.SYS.DDF01`

En la imagen se ve que esta tarjeta contiene la aplicación `A0000000041010`. En el siguiente `SELECT` que se envía, se selecciona esta aplicación. En el caso de una tarjeta Visa, se selecciona la aplicación `A0000000032010`. Si no se seleccionara la aplicación antes de enviar un comando APDU, la respuesta obtenida por parte de la tarjeta será siempre un código de error.

A continuación de estos dos mensajes iniciales para seleccionar la aplicación, en cada ejecución del programa se selecciona qué tipo de comando se va a enviar. En la tabla 5.1 se muestran los mensajes que se han seleccionado para realizar el fuzzing sobre las tarjetas EMV. La tabla muestra los valores necesarios para cado campo del mensaje, y con XX los campos que son variables.

APDU	CLA	INS	P1	P2	Lc	DATA	Le
SELECT	00	A4	04	00	XX	XX	00
GET ATTRIBUTE	00	34	00	00	00	-	-
GENERAL AUTHORIZATION	00	86	00	00	XX	XX XX	00
READ RECORDS	00	B2	XX	XX	00	-	-
GET PROCESS OPTIONS	80	A8	00	00	XX	XX XX	00
GET DATA	80	CA	9F	00	00	-	-
INTERNAL AUTHENTICATE	80	88	00	00	XX	XX XX	00

Tabla 5.1: Comandos APDU generados

Para definir estos mensajes, se utilizan bloques de datos en los que se define cada parte del comando APDU. En el código 5.1 se muestra la definición del comando `Internal Authenticate`.

```
1  def internal_auth():
2
3      request = "Internal Auth"
4      s_initialize(request)
5      s_byte(0x80, name="cla", full_range=True, fuzzable=
          False)  # CLA
6      s_byte(0x88, name="ins", full_range=True, fuzzable=
          False)  # INS
7      s_byte(0x00, name="p1", full_range=True, fuzzable=
          False)  # P1
8      s_byte(0x00, name="p2", full_range=True, fuzzable=True
          )  # P2
9      s_byte(0x02, name="Lc", full_range=True, fuzzable=
          False)  # Lc
10     s_static("\x83\x00", name="data")  # Data
11     s_byte(0x00, name="Le", full_range=True, fuzzable=
          False)  # Le
12     return request
```

Código 5.1: Definición del comando `Internal Authenticate`

Estos bloques de datos comienzan siempre con `s_initialize` para darle nombre al bloque y poder conectarlo a la sesión posteriormente. A continuación se definen las primitivas para construir el mensaje. Mediante el uso de `s_byte` se puede definir un byte con un valor por defecto y especificar si será susceptible de fuzzing o no con la opción `fuzzable=False/True`. Para definir más de un byte se ha utilizado la primitiva `s_bit_field` si se quieren alterar los datos o `s_static` en el caso de que se envíen los datos fijos. En este caso solo se ha puesto a

`True` el byte P2, por lo que se enviará el mensaje con todos las combinaciones de bits posibles en esta posición. En este caso los bytes a variar se han definido como una secuencia de 16 bits en la que se alteran todos ellos.

Resulta de interés el comando `Read Records`, donde variando los bytes P1 y P2 se recorren todos los sectores de la tarjeta. En el siguiente código 5.2 se muestra el bloque definido para este comando.

```
1  def read_records():
2      request = "Read records"
3      s_initialize(request)
4      s_byte(0x00, name="cla", full_range=True, fuzzable=
          False)  # CLA
5      s_byte(0xB2, name="ins", full_range=True, fuzzable=
          False)  # INS
6      s_bit_field(0, 16, name="p1:p2", full_range=True,
          fuzzable=True)  # P1;P2
7      s_byte(0x00, name="Lc", full_range=True, fuzzable=
          False)  # Lc
8      return request
```

Código 5.2: Definición del comando `Read Records`

5.2. Resultados obtenidos

En esta sección se van a exponer los resultados obtenidos y un análisis de las distintas respuestas recibidas.

Para las pruebas se han utilizado cuatro tarjetas, que han sido clasificadas otorgando un identificador a cada una de ellas como se muestra en la tabla 5.2. Las tarjetas elegidas han sido una Mastercard, una Visa y dos Visa Electron.

Tarjeta	Etiqueta
Mastercard	1
Visa Electron	2
Visa	3
Visa Electron	4

Tabla 5.2: Tarjetas utilizadas

Para cada una de estas tarjetas se han lanzado individualmente las pruebas. Para cada tipo de APDU que se ha enviado a las tarjetas, se han analizado las respuestas recibidas, siendo de especial interés las respuestas erróneas.

Como se ha descrito en la Sección 2.1.2, estos mensajes de error se codifican en los dos últimos bytes de la respuesta enviada por la tarjeta (SW1 y SW2). En la figura 5.2 obtenida del libro EMVBook3 [9] se muestra el estado de las respuestas en función de estos dos bytes.

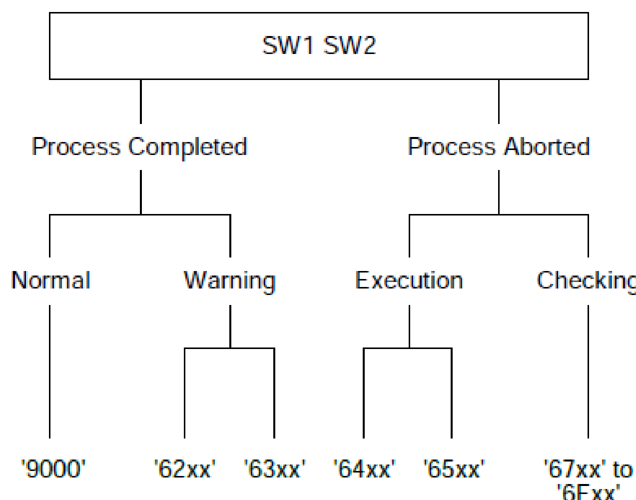

Figura 5.2: Clasificación bytes de estado (extraída de [9])

En la tabla 5.3 se muestran los 10 mensajes de error diferentes recibidos en los experimentos realizados.

ERROR	SW1	SW2
Wrong Length	67	00
Logical Channel not supported	68	81
Conditions of use not satisfied	69	85
Function not suppported	6A	81
File not found	6A	82
Record not found	6A	83
Incorrect P1 or P2 parameter	6A	86
Referenced data not found	6A	88
Instruction code not supported	6D	00
Class not supported	6E	00

Tabla 5.3: Errores respuestas APDU

En primer lugar, el error 6700 corresponde a cuando se envía una orden APDU donde el byte Lc y el tamaño de los datos no coincide. El error 6881 se ha obtenido al variar el primer byte (CLA) del paquete. El error 6985 se obtiene cuando no se satisfacen las condiciones de uso por lo que la transacción no se puede completar con la aplicación seleccionada. Más adelante se verá en detalle qué tarjetas han respondido con este error. Todos los errores donde el primer byte corresponde con 6A se refieren a errores por los parámetros P1 y P2. Estos son los principales parámetros que se han variado en la generación de datos, por lo que se han obtenido un mayor número de errores de este tipo. Por último, el error 6D00 corresponde al segundo byte de la orden APDU (INS) y el error 6E00 a un formato incorrecto en la orden.

Para analizar los resultados, se han almacenado en un mismo directorio los ficheros correspondientes a cada tarjeta, teniendo un directorio para cada test realizado y cuatro ficheros en cada uno de ellos. Mediante el programa `compare.py`, se selecciona el directorio que se quiere analizar y se accede a los ficheros contenidos en él.

Una vez concatenados los cuatro ficheros, se filtran los datos eliminando las respuestas correctas (que son las que terminan en 9000). De esta manera se van a analizar únicamente las respuestas erróneas. Dentro de estas respuestas erróneas, se busca cuántos tipos de respuestas diferentes hay y se agrupan los datos por número de veces que se ha recibido ese error para cada tarjeta. Por último, se utiliza un gráfico de araña para mostrar el número de veces que se repite cada error para cada una de las tarjetas. Como este dato en algunos casos es muy elevado y lo que resulta de interés es ver las diferencias entre las diferentes tarjetas, se ha acotado el límite de las gráficas a 20. De esta manera, el análisis se centra en ver las tarjetas que han dado mensajes de error diferentes y las que no han respondido ninguna vez con algún tipo de error.

A continuación se muestran los gráficos con los datos obtenidos para los 6 tipos de comandos que se han ejecutado finalmente y se realiza la explicación de cada uno de ellos.

Comando `Get Attribute`

En la figura 5.3 se muestran los mensajes de error recibidos por las tarjetas para las variaciones del comando `Get Attribute`. En el envío de este comando únicamente se ha modificado el byte P2. Se puede ver que se obtienen ceros en 3 tipos de errores (6881, 6a86 y 6e00). Los errores 6881 y 6e00 únicamente se obtienen de la tarjeta 3. El resto de tarjetas no devuelven este error. Por otro lado, se observa que la respuesta 6a86 únicamente no se recibe de la tarjeta 1.

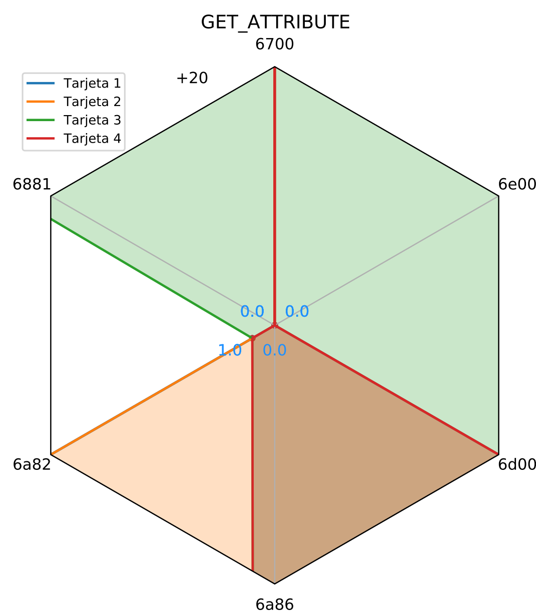

Figura 5.3: Resultados de experimentos del comando `Get Attribute`

Comando `General Authorization`

En la figura 5.4 se muestran los mensajes de error recibidos para el comando `General Authorization`. En este caso se ha variado el byte INS y el byte P2. Aquí se puede ver que se obtienen ceros de alguna tarjeta para 4 tipos de errores (6881, 6985, 6a81 y 6e00). Se puede observar una diferencia clara entre la tarjeta Mastercard (tarjeta 1) y las tarjetas Visa (tarjetas 2, 3 y 4). La única que responde el error 6985 es la tarjeta 1 y a su vez esta es la única que no devuelve el error 6a81. También resulta de interés que la única que responde a los errores 6881 y 6e00 es la tarjeta 3.

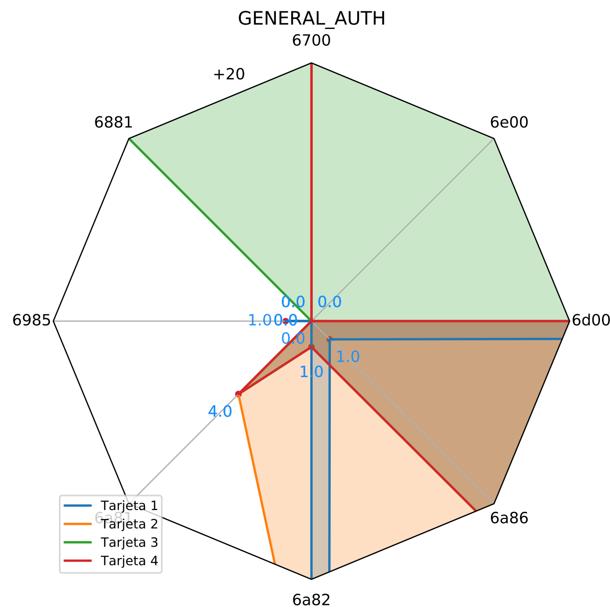

Figura 5.4: Resultados de experimentos del comando `General Authorization`

Comando `Read Records`

En la figura 5.5 se muestran los mensajes de error recibidos para el comando `Read Records`. En este comando se han variado los bytes P1 y P2, pasando por todas las posibles combinaciones. Con este comando se obtienen hasta 9 códigos de error distintos. Resulta de interés que la única tarjeta que devuelve el error 6985 es la tarjeta 1. Esta misma tarjeta no devuelve ninguna vez los errores 6881, 6a81, 6d00 ni 6e00, siendo la única que no responde el código de error 6e00 ni el 6a81.

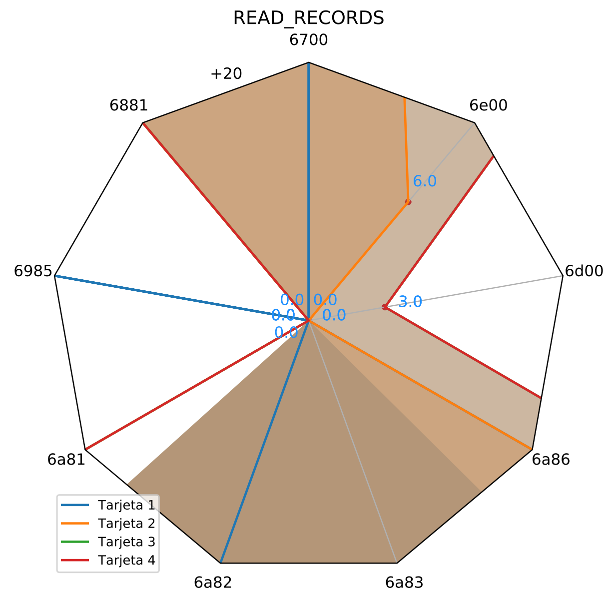

Figura 5.5: Resultados de experimentos del comando `Read Records`

Comando `Get Process Options`

En la figura 5.6 se muestran los mensajes de error recibidos para el comando `Get Process Options`. En este comando se ha variado el byte Lc. En este caso, la tarjeta 1 únicamente responde con tres tipos de errores el 6700, 6985 y 6a82, siendo nuevamente la única que no responde el error 6a86. También cabe destacar que para este comando se observan diferencias entre las demás tarjetas.

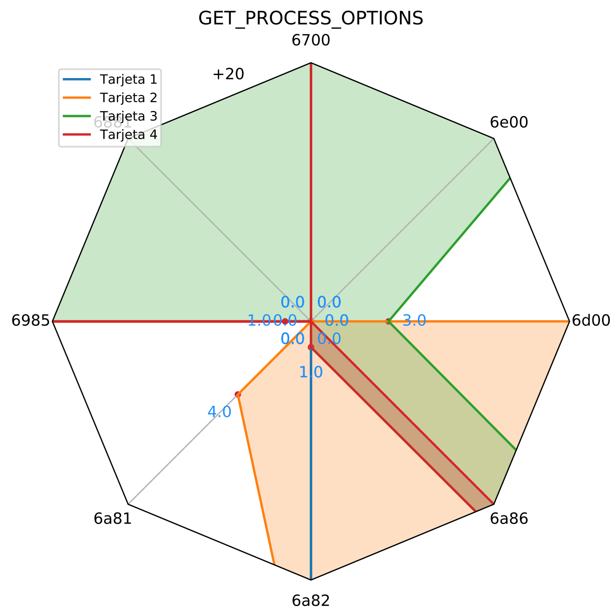

Figura 5.6: Resultados de experimentos del comando `Get Process Options`

Comando `Get Data`

En la figura 5.7 se muestran los mensajes de error recibidos para el comando `Get Data`. En este comando se ha variado el byte P2. En este caso la tarjeta Mastercard (tarjeta 1) responde con cuatro tipos de errores (6700, 6985, 6a82 y 6a88). Llama la atención que la única tarjeta que responde con los errores 6881, 6d00 y 6e00 es la tarjeta 3.

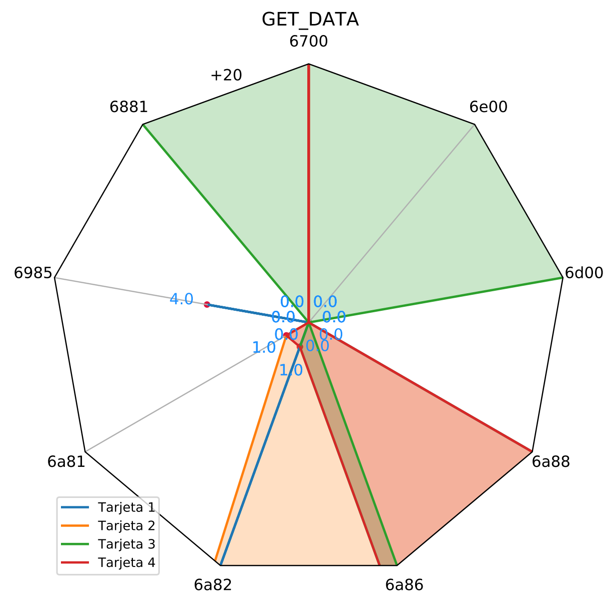

Figura 5.7: Resultados de experimentos del comando `Get data`

Comando `Internal Authenticate`

En la figura 5.8 se muestran los mensajes de error recibidos para el comando `Internal Authenticate`. En este comando también se ha variado el byte P2. La única tarjeta que responde con todos los errores es la tarjeta 3. Resulta de interés que la única tarjeta que no responde con el error 6a86 es la tarjeta 1.

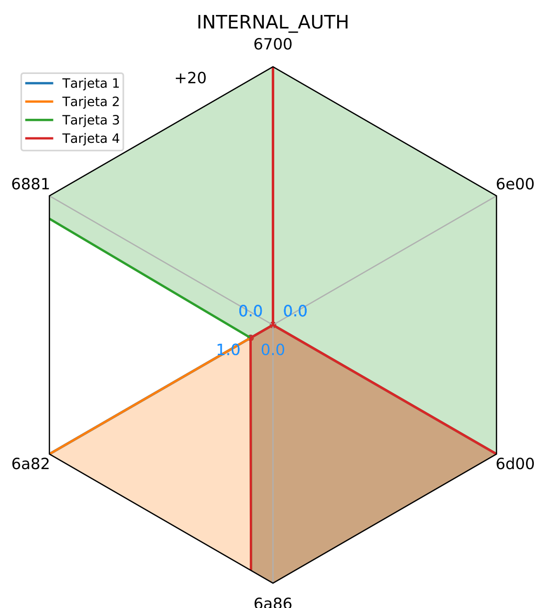

Figura 5.8: Resultados de experimentos del comando `Internal Authenticate`

Discusión de los resultados

A modo de resumen, se puede observar claramente que hay diferencias en las respuestas obtenidas por las diferentes tarjetas. Lo que más llama la atención es que la tarjeta 1 (Mastercard) es la única tarjeta que en ningún caso responde con el código de error 6a86. Este error se da cuando los parámetros P1 o P2 son incorrectos y precisamente estos son los parámetros que más se han variado. Por lo general, las tarjetas Visa escogidas para la experimentación responden con un número de errores más variado, siendo la tarjeta Mastercard la que responde con mayor frecuencia el código de error 6985.

Es evidente que la implementación de las especificaciones entre las tarjetas analizadas es diferente, pero sería necesaria una mayor muestra de tarjetas para comprobar que todas las tarjetas de Mastercard se comportan de la misma manera.

A modo de resumen, se puede observar claramente que hay diferencias en las respuestas obtenidas por las diferentes tarjetas. Estas diferencias resultan útiles para distinguir con qué tipo de tarjeta se está realizando la comunicación NFC.

Capítulo 6

Conclusiones y trabajo futuro

En este capítulo se exponen las conclusiones sobre el trabajo realizado y el posible trabajo futuro.

En este trabajo se ha realizado el desarrollo de un sistema capaz de comunicar la herramienta Boofuzz con varias tarjetas NFC. Se ha estudiado el protocolo necesario para establecer comunicaciones con tarjetas que cumplen las especificaciones EMV y se han implementado algunos de estos mensajes como bloques de datos en la herramienta Boofuzz. Tras realizar una serie de tests se han analizado los resultados obtenidos.

A la vista de los resultados, se puede decir que no todas las tarjetas implementan las especificaciones EMV de la misma manera. Se han encontrado diferentes respuestas entre las tarjetas Mastercard y varias versiones de tarjeta Visa. Estas diferencias permitirían a un adversario reconocer con qué tipo de tarjeta se está estableciendo una comunicación NFC, pudiendo determinar la forma de ataque.

Debido a que el número de tarjetas disponibles para experimentación era limitado no se pueden sacar conclusiones generalizadas para estos tipos de tarjetas, más alla de que a pesar de ser unas especificaciones comunes, cada fabricante gestiona las excepciones de manera diferente.

Como posible continuación de este trabajo, se pueden ampliar el número de comandos sobre los que se realiza fuzzing hasta llegar a definir todos los comandos APDU existentes para tarjetas inteligentes. Otra posible extensión sería realizar otros tipos de análisis sobre las respuestas obtenidas, por ejemplo, centrándose en qué variaciones de bits provocan determinados códigos de error. También sería interesante realizar este tipo de análisis sobre tarjetas NFC de cualquier tipo viendo las especificaciones que tiene que cumplir cada una de estas tarjetas.

Debido a que NFC es una tecnología en constante crecimiento y que cada día está presente en un mayor número de aplicaciones, es necesario seguir realizando análisis de fuzzing para encontrar posibles vulnerabilidades e incumplimientos de las especificaciones que puedan resultar en un riesgo de seguridad para los usuarios.

Bibliografía

[1] Manual API ACR122U. [Online; http://www.acs.com.hk/download-manual/419/API-ACR122U-2.03.pdf].

[2] Herramienta fuzzing Sulley v1.0. [Online; https://github.com/OpenRCE/sulley], June 2014.

[3] Herramienta parser emv-bertlv v-0.1.8. [Online; https://github.com/binaryfoo/emv-bertlv], 2019.

[4] Librería nfcpy v1.0.3. [Online; https://nfcpy.readthedocs.io/en/latest/], June 2019.

[5] Herramienta fuzzing Boofuzz. [Online; https://boofuzz.readthedocs.io/en/stable/], April 2020.

[6] Librería pandas v1.1.2. [Online; https://pandas.pydata.org/docs/], September 2020.

[7] Dave Aitel. An Introduction to SPIKE, the Fuzzer Creation Kit. [Online; http://www.immunitysec.com/downloads/usingspike3.ppt], 2002.

[8] Consejo Económico y Social de España. Nuevos hábitos de consumo, cambios sociales y tecnológicos. techreport 4/2016, Gobierno de España, February 2017.

[9] EMV. Integrated Circuit Card Specifications for Payment Systems - Book 3 Application Specification v4.3. [Available; https://www.emvco.com/wp-content/uploads/2017/05/EMV_v4.3_Book_3_Application_Specification_20120607062110791.pdf], November 2011.

[10] ISO/IEC. 14443-3 Identification cards - Contactless integrated circuit cards - Proximity cards - Part 3: Initialization and anticollision. [Available; https://www.iso.org/standard/70171.html], August 2016.

[11] ISO/IEC. 14443-1 Cards and security devices for personal identification - Contactless proximity objects — Part 1: Physical characteristics. [Available; https://www.iso.org/standard/73596.html], 2018.

[12] ISO/IEC. 14443-4 Cards and security devices for personal identification - Contactless proximity objects - Part 4: Transmission protocol. [Availbale;https://www.iso.org/standard/73599.html], June 2018.

[13] ISO/IEC. 14443-2 Cards and security devices for personal identification - Contactless proximity objects - Part 2: Radio frequency power and signal interface. [Available; https://www.iso.org/standard/73597.html], July 2020.

[14] ISO/IEC. 7816-4 `Identification cards - Integrated circuit cards - Part 4: Organization, security and commands for interchange`. [Available; https://www.iso.org/standard/77180.html], May 2020.

[15] Julien Lancia. Un framework de fuzzing pour carte a puce : application aux protocoles EMV. *SERMA Technologies*, June 2011.

[16] So B. Miller BP, Fredriksen L. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, (33(12)):32–44., 1990.

[17] oneguyoneblog. Guía instalación `ACR122U NFC USB READER`. [Online; https://oneguyoneblog.com/2016/11/02/acr122u-nfc-usb-reader-linux-mint/], November 2016.

[18] Christina Weaver. GitLab Acquires Peach Tech and Fuzzit to Expand its DevSecOps Offering. *GitLab*, June 2020.

Apéndice A

Extensión temporal del trabajo

En la figura A.1 se muestra un diagrama de Gannt con la extensión temporal del proyecto. La realización de este trabajo ha suspuesto un coste de 380 horas aproximadamente. Estas horas se han realizado a lo largo de 7 meses y se han separado en distintas fases, siendo la documentación la parte más extensa en el tiempo y el desarrollo del sistema la que más carga de trabajo ha supuesto.

La primera fase es la documentación sobre la tecnología NFC y las especificaciones que utiliza y un estudio sobre qué es el fuzzing, las técnicas utilizadas para llevarlo a cabo y los principales softwares que hay disponibles.

A continuación se llevaron a cabo una serie de pruebas con el hardware y diferentes librerías para establecer la comunicación con las tarjetas NFC. Al final se optó por utilizar la librería `nfcpy` por estar escrita en Python y existir un mayor número de herramientas relacionadas con esta librería.

Una vez decidida la librería se pasó a la preparación del entorno y se estableció una primera comunicación entre las tarjetas y el lector. Se comprobó que se podían enviar y recibir mensajes en ambas direcciones sin problemas.

Tras esto se pasó al desarrollo del sistema, con la integración de las diferentes herramientas: Boofuzz, el lector NFC y el programa para comunicar ambas partes. Además se desarrolló un programa para analizar los resultados obtenidos y generar gráficas con ellos.

Por último, una vez que estaba desarrollada la herramienta, se ejecutaron todos los tests con las tarjetas que había disponibles y se comenzó con la redacción de la memoria.

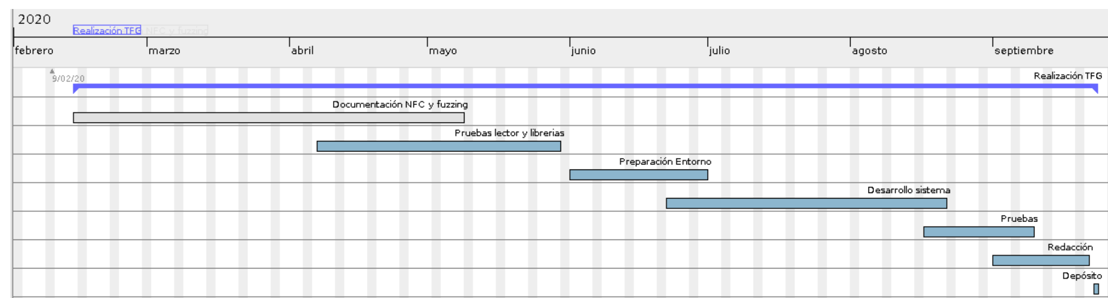

Figura A.1: Diagrama Gannt con la extensión temporal del trabajo