Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA DE COMPUTADORES

Trabajo Fin de Grado

Estudio e implementación de mecanismos de autenticación de usuarios en aplicaciones usando FIDO2 y criptografía ligera

Autor: Germán Esteban López

Tutor/es: Bernardo Alarcos Alcázar y Javier Junquera Sánchez

2022

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Estudio e implementación de mecanismos de autenticación de usuarios en aplicaciones usando FIDO2 y criptografía ligera.

Autor: Germán Esteban López.

Tutor: Bernardo Alarcos Alcázar.

Co-Tutor: Javier Junquera Sánchez.

TRIBUNAL:

Presidente: Antonio García Herraiz.

Vocal 1º: Juan Antonio Rodrigo Yanes.

Vocal 2º: Bernardo Alarcos Alcázar.

Suplente: Iván Marsá Maestre.

Fecha: 08-09-2022

Agradecimientos

Quiero agradecer a Javier Junquera Sánchez y a Bernardo Alarcos Alcázar por la confianza y la ayuda recibida con este proyecto, las respuestas a las dudas que me han facilitado y los comentarios que han ido surgiendo mientras revisaban mi trabajo han sido una ayuda inmensa para poder dar lo mejor en este último paso en mi grado, gracias.

Por supuesto dar las gracias a mi familia, a mis padres por apoyarme y confiar desde el principio con la carrera que he escogido, a mi hermana por estar siempre ahí cuándo la he necesitado, a mis amigos por el apoyo constante y acompañarme en esta etapa, y quiero dar las gracias a mi pareja, Mauge, por ser mi principal apoyo estos últimos meses, que han sido especialmente duros.

Gracias por todo.

Índice

Índice de tablas

Índice de Figuras

Resumen

El mundo de la ciberseguridad pasa por un momento de evolución, la frecuencia en los ataques de ingeniería social exige cambios en la autenticación, fomentando iniciativas como el estandarizado FIDO2 que busca la desaparición de las contraseñas. Por otra parte, cada vez se utilizan más sistemas empotrados con poca potencia, por lo que se necesita criptografía ligera poco exigente, en este trabajo se busca unificar ambas tecnologías, comprobando si es una alternativa viable para sistemas empotrados cada vez más frecuentes que requieran de seguridad, en el que se ha comprobado que el uso de criptografía ligera supone una mejoría notable en la eficiencia por encima de la criptografía clásica.

Palabras clave: FIDO2, Criptografía Ligera, ASCON, autenticación, *passwordless*.

Abstract

The world of cybersecurity is going through a time of evolution, the frequency of social engineering attacks requires changes in authentication, promoting initiatives such as the standardized FIDO2 that seeks the disappearance of passwords. On the other hand, more and more embedded systems with little power are being used, so light cryptography is needed, this work seeks to unify both technologies, checking if it is a viable alternative for increasingly frequent embedded systems that require security, in which it has been found that the use of light cryptography is a significant improvement in efficiency over classic cryptography.

Keywords: FIDO2, Lightweight Cryptography, ASCON, authentication, passwordless.

1. Introducción

Nuestra industria está en constante cambio y evolución, actualmente nos encontramos en un punto de inflexión con la aparición o avances de nuevas tecnologías como la Computación Cuántica, Inteligencia Artificial, 5G, *Internet of Things*… son términos que cada vez están más asentados y que pueden cambiar por completo el paradigma y asentar un avance enorme en la tecnología actual, sin embargo, no todo son buenas noticias, en el terreno de la ciberseguridad, tanto el uso de estas tecnologías como el refinamiento en las técnicas de los ciberatacantes, también suponen nuevas vulnerabilidades y puntos de entrada para ataques en redes y sistemas de todo tipo, tanto a equipos personales como infraestructuras críticas.

En este último contexto, entra la arquitectura FIDO2, FIDO proviene de *Fast Identity Online*, que es una propuesta creada por la *FIDO Alliance*, en la que distintas organizaciones punteras en la industria de las tecnologías de la información como Apple, Google, Amazon, Microsoft, BSI, NIST… entre muchas otras. Estas buscan que el proceso de autenticación por contraseña desaparezca por sus evidentes carencias y vulnerabilidades en el factor humano, la autenticación se realizará entre un dispositivo o programa que tenga el usuario y el servidor que aloje el servicio, no hay ningún tipo de interacción humana intermedia que pueda comprometer la seguridad por ataques de ingeniería social, simplificándolo al máximo, FIDO2 busca que el usuario humano, que suele ser perezoso y poco prudente, no intervenga en el proceso de autenticación a sus cuentas/recursos.

Partiendo de FIDO2, podemos ya vislumbrar que con su uso generalizado podemos arreglar los problemas que supone el uso de credenciales con contraseña en texto plano, sin embargo, FIDO2 choca con una tendencia actual del sector, y es el uso de dispositivos con uso específico o sistemas empotrados/embebidos, este tipo de sistemas está pensado para optimizar la ejecución de una tarea específica tanto a nivel computacional como a nivel energético, disminuyendo la implementación de potencia desaprovechada y el gasto energético, por ende, el precio por unidad de los dispositivos, de manera que a diferencia de un ordenador personal que suele utilizarse para distintas tareas, varias de ellas muy exigentes a nivel computacional, estos dispositivos están pensados y desarrollados solo para dicha tarea específica. Esto supone un problema al implementar algoritmos de seguridad habituales como RSA o AES, que suelen ser costosos computacionalmente hablando y que, si se implementan en estos sistemas empotrados, podrían alterar su malfuncionamiento, pero por otra parte no podemos prescindir de ellos, dado que estos sistemas pueden estar intercambiando información sensible o necesaria para el correcto funcionamiento, por ejemplo, en *IoT* se habla mucho de el uso de sensores de proximidad para la conducción autónoma, que distintos vehículos intercambien datos de velocidad y dirección para prever un posible accidente, un atacante que pueda realizar un ataque Man in the Middle, obtendría los datos de ambos vehículos y podría reenviarlos alterados para provocar la colisión, por una parte buscamos que esto no suceda por obvias cuestiones de seguridad, pero por otra buscamos también que no sea necesario tener un dispositivo costoso que realice una tarea tan simple como el intercambio de directrices de velocidad/dirección a la vez que tenga que cifrar la conversación, la solución a este dilema pasa por la criptografía ligera.

La criptografía ligera (término proveniente de su traducción del inglés *lightweight cryptography*, con sus siglas *LwC*) es una rama emergente de la criptografía actual que busca aportar soluciones en dispositivos limitados en potencia computacional y energética que requieran de un algoritmo

criptográfico para la confidencialidad e integridad de la información que manejan, sin que la ejecución de dicho algoritmo afecte al funcionamiento habitual del dispositivo por falta de recursos.

Objetivos del proyecto

El objetivo principal del proyecto es unificar las tecnologías de FIDO2 y la criptografía ligera y comprobar si es una alternativa viable que suponga alguna ventaja por encima de una implementación de FIDO2 habitual con criptografía clásica, permitiendo creando una alternativa viable a la implementación de FIDO2 en dispositivos con pocas prestaciones tanto en computación como en energía, fomentando el uso de este estandarizado para sistemas empotrados y mejorando su seguridad ante ataques externos, siendo estos sistemas cada vez más habituales en la industria con la aparición del fenómeno tecnológico *Internet of Things*, que unifica cada vez más la conectividad e intercambio de datos entre dispositivos a través de Internet (potenciado por la tecnología 5G).

Como objetivos específicos, es necesario el desarrollo de un marco teórico extenso de las tecnologías de FIDO2 y LwC, en el primer caso, se busca un conocimiento detallado de las tecnologías FIDOU2F y FIDO2, junto a sus componentes CTAP2 y WebAuthn que faciliten su implementación práctica. Mientras que, por parte de la criptografía ligera, se busca crear un estado del arte de esta rama de la criptografía, centrándonos en aspectos como los procesos de estandarizado realizados por el NIST y el ISO/IEC, en las propiedades y requisitos del cifrado impuesto por el NIST, y haciendo un inciso en el algoritmo ASCON, que es el escogido para la implementación práctica. Es importante mencionar que otro objetivo menor del proyecto es la creación de documentación en español de las tecnologías, tanto de FIDO2 como de la criptografía ligera hay muy poca documentación en inglés al ser tecnologías emergentes y en español es casi inexistente.

Partiendo del marco teórico ya obtenido, podemos empezar a desarrollar otro objetivo específico como es la prueba de concepto (Proof of Concept - PoC) de FIDO2, siendo esta un simulador del funcionamiento de FIDO2 que incluye tanto una implementación con criptografía clásica como LwC, esto nos permite obtener los resultados deseados de eficiencia de ambas implementaciones mediante el tiempo de ejecución, obteniendo unos resultados con los que comprobar el objetivo principal de si es viable unificar FIDO2 y LwC y de si supone una ventaja de eficiencia el uso de la criptografía ligera respecto a la criptografía clásica. Este simulador supone un objetivo especialmente interesante para posibles futuros trabajos con tecnología FIDO2 o una nueva versión del proyecto con otros algoritmos criptográficos ligeros, por ejemplo, con un cifrado asimétrico ligero.

Estructura de la memoria

La memoria se divide en los siguientes capítulos:

- Estado del arte – Estudio de la autenticación.
- Metodología.
- FIDO.
- Criptografía ligera.
- Implementación práctica: Simulación FIDO2 + LwC.

2. Estado del arte - Estudio de la autenticación.

El concepto de la autenticación no es algo novedoso ni de nuestro siglo, desde tiempos inmemorables se ha requerido autenticar a las personas para poder acceder a recursos o a territorios concretos, por ejemplo, en la época del Imperio Romano para poder demostrar que eras un ciudadano romano debías decir ante el guardia "*Civis Romanus sum*", que traducido del latín quiere decir: "Soy ciudadano romano", a simple vista no parece una contraseña muy compleja ni muy segura, pero el guardia no solo se fijaba en la frase, también tenía en cuenta tu acento, la vestimenta que portabas o te podía hacer alguna pregunta más específica, de manera que un forastero ajeno a la cultura romana lo tenía extremadamente difícil para hacerse pasar por ciudadano romano.

Pero el ser humano no se ha quedado quieto desde entonces, cada vez se han requerido métodos de autenticación más y más sofisticados, especialmente desde la era digital, el uso de la informática tiene muchísimas virtudes y ventajas, pero tiene un componente especialmente difícil de solucionar cuándo se trata de ciberseguridad, y es el anonimato. Ya no hablamos de nombres y apellidos, hablamos de direcciones IP, dominios, credenciales de acceso… cualquier persona con los conocimientos necesarios, puede realizar una suplantación de identidad y tal como están planteados los sistemas, es imposible estar seguros del todo de que no tengamos un intruso en nuestra red.

La incorporación de nuevas tecnologías como el 5G, *Internet of Things* (IoT), Computación Cuántica, Inteligencia Artificial… no solo facilita el desarrollo de nuevas aplicaciones y mejoras en la sociedad, también genera nuevas vulnerabilidades y oportunidades para los atacantes de beneficiarse a nuestra costa, por ello es necesario actualizarnos, son necesarios nuevos métodos de autenticación que nos permitan cumplir la definición del NIST: *la autenticación consiste en la verificación de la identidad de un usuario, proceso o dispositivo, con frecuencia como un requisito previo para permitir acceso a unos determinados recursos en un sistema de información*.

Entrando en los distintos métodos de autenticación que existen, hay varios tipos de categorías y de divisiones entre los tipos de autenticación, pero en la industria se suelen dividir los métodos en tres categorías distintas, en base a qué aporta el usuario para identificarse al sistema:

1. Autenticación por conocimiento: el usuario conoce una información "secreta" que utiliza en el sistema para demostrar su identidad, dentro de esta categoría hay métodos como el uso de contraseñas, pines, preguntas personales…
2. Autenticación por posesión: el usuario posee un objeto que le sirve para identificarse, puede ser una tarjeta de acceso, un dispositivo USB con un token o un generador de claves aleatorias, un llavero NFC… algún utensilio que interactúe con el sistema y permita acceder a los recursos del usuario.
3. Autenticación por inherencia: se aprovechan las cualidades del usuario, incluyendo propiedades físicas únicas de cada persona, como el rostro facial, ojos, huella dactilar, voz, actividad cardiaca… pero también propiedades conductuales, como la manera de escribir o de caminar.

Hay métodos de autenticación que no encajan en ninguna de las categorías anteriormente mencionadas, como la autenticación basada en direcciones IP o geolocalización entre otras, pero también son destacables.

Claramente hay uno que destaca sobre los demás, y es el uso de contraseñas de texto plano, pero a pesar de que sea poco seguro, ya que estamos dando al usuario la opción de poner contraseñas débiles… ¿Por qué se utiliza? Principalmente por comodidad y gastos, es un método barato, rápido de utilizar y fácil de administrar, si por ejemplo quisiéramos introducir biometría ocular como método de autenticación, el usuario necesita un escáner que sea capaz de analizar su ojo, es decir, invertir en un periférico caro, no es viable económicamente para usuarios habituales, o si un atacante consigue entrar en la cuenta del usuario, tiene fácil arreglo: un administrador puede cambiar la contraseña y forzar un cierre de sesión, comunicándose con el usuario legitimo para que establezca una nueva contraseña. Es importante a la hora de pensar en métodos de autenticación, en que no solo importa la robustez y lo seguro que sea el propio método, también tiene que ser fácil de implementar y gestionar en un sistema, además de fácil de utilizar por un usuario con pocos o nulos conocimientos en informática.

A pesar de que sea una técnica muy cómoda de utilizar tanto para usuarios como para los administradores de sistema, la debilidad y los agujeros que tiene en su seguridad son evidentes, existen un gran número de ciberataques que han comenzado con una mala elección de contraseña, y algunos de ellos no precisamente pequeños ni alejados en el tiempo, por ejemplo, el 13 de diciembre de 2020 se hizo público el ataque a la compañía de software “SolarWinds”[4] que permitía a los atacantes incluir código maligno en el software “Orion” que utilizaban clientes como Microsoft, AT&T, Siemens o incluso Cisco, alcanzando la cifra de 425 de las 500 empresas de la lista Fortune 500, llegando también a organismos gubernamentales estadounidenses como el departamento del tesoro, la administración de seguridad nuclear nacional o el departamento de seguridad nacional, este ataque comenzó con, según el CEO de la compañía, la filtración de una contraseña de uno de sus servidores “solarwinds123” por un insider de la compañía, aunque teniendo en cuenta la complejidad de una contraseña como “solarwinds123”, cualquier ataque de diccionario mínimamente preparado puede sacar esa contraseña en menos de una hora sin necesidad de navegar por GitHub buscando la contraseña filtrada, el nombre de la compañía y una sucesión numérica como 123 es definible, como mínimo, imprudente.

Fuese por una filtración o por ser una contraseña muy débil, es evidente que el uso de unas credenciales tan pobres puede ser la vía de entrada a un ataque mucho más devastador, pero el ejemplo mostrado es un ataque fuertemente medido y realizado por un equipo con una buena base técnica, normalmente un ciber atacante busca algo más sencillo y rápido, a menor escala, pero con mayor cantidad de objetivos, a veces incluso no necesita tener conocimientos en informática, haciendo uso de técnicas como Phishing, Smishing y Vishing, estos tipos de ataques están al alza desde la pandemia provocada por la COVID-19, dado que medidas sanitarias como el confinamiento generalizado o reducir al máximo el contacto social forzaba el uso del teletrabajo y aún con la situación sanitaria se ha mantenido como una manera de trabajar a considerar por las empresas, creando situaciones y un grupo de posibles víctimas mucho mayor que facilita el trabajo a los ciberatacantes, en 2020, en medio del confinamiento, las estafas por Phishing aumentaron a nivel global un 667% en un solo mes[5].

Es evidente con lo explicado anteriormente que el uso de contraseñas de texto plano ha quedado obsoleto y que es necesario actualizarnos, o mejor dicho, cambiar de perspectiva, una contraseña puede ser un buen sistema si se utilizara bien: usuarios con la capacidad de recordar contraseñas largas y complejas, sin repetición de contraseñas en distintas credenciales… es mejor limitar al mínimo la relevancia del usuario en la seguridad de sus recursos, en vez de utilizar autenticación por conocimiento, explorar las posibilidades de la autenticación por

posesión o inherencia, y eso es lo que busca la FIDO Alliance, que es la organización que crea los estandarizados de FIDO y fomenta su uso a nivel global.

Actualmente FIDO2 está lejos de conseguir su objetivo, pero su accesibilidad y su uso está cada vez más globalizado, empresas miembros de la FIDO Alliance como Google o Microsoft tienen sus propias plataformas que permiten la utilización de FIDO2 como alternativa para la autenticación en sus servicios y aplicaciones, por ejemplo, Azure Active Directory de Microsoft permite la autenticación por passwordless FIDO2, o Google contribuye con distintas aportaciones al estándar, como la creación de una API de FIDO2 para Android con implementación de WebAuthn y el uso de BLE (Bluetooth Low Energy) o NFC, una línea de security keys compatible tanto en FIDO U2F como FIDO2 con distintas opciones para empresas y particulares llamada “Titan Security Key”[6]... también existen otras iniciativas con FIDO como puede ser Samsung Knox (Samsung también es miembro de la FIDO Alliance), centrada en la utilización de biometría como clave de autenticación en sus dispositivos móviles, haciendo uso de FIDO2. Como podemos ver, las empresas de la FIDO Alliance van incluyen

Por parte de la criptografía ligera, ya está reconocida como una rama emergente de la criptografía y se están empezando a desarrollar varios algoritmos que puedan funcionar como estándar, con el objetivo de crear algoritmos viables a nivel computacional y energético en dispositivos de pocos recursos, existen distintos proceso de estandarizado, destacando los realizados por las organizaciones NIST e ISO/IEC, el proceso de estandarizado de NIST está aún pendiente de finalizar, actualmente se encuentra en proceso de finalización con 10 finalistas entre los que se encuentran ASCON, Elephant, GIFT-COFB, Grain1238-AEAD, ISAP, Photon-Beetle, Romulus, Sparkle, TinyJambu y Xoodyak. Se estima que a finales de 2022 finalizará la elección de los cifrados como nuevo estándar en NIST. El proceso de estandarizado de ISO/IEC en cambio ya terminó en noviembre de 2019, y fueron seleccionados tres algoritmos, divididos en dos grupos por el tamaño de bloque:

· Tamaño de bloque de 64 bits: PRESENT.

· Tamaño de bloque de 128 bits: CLEFIA y LEA.

Actualmente no hay mucho uso de criptografía ligera, al no existir un estándar definido por el NIST y que aún no se ha generalizado el uso del 5G y el fenómeno IoT, no hay tanta demanda de este tipo de cifrados como se podría esperar dado que aún no se han comercializado tecnologías como redes de sensores o conducción autónoma, que sí requerirán este tipo de cifrados ligeros en sus comunicaciones. Esta situación se ha complicado por las repercusiones socioeconómicas de la pandemia de la COVID-19, siendo como principal obstáculo el retraso generalizado de la implementación del 5G en nuestra sociedad.

Hay que mencionar que el algoritmo elegido para este proyecto está incluido en la lista anteriormente mencionada de los finalistas del proceso de estandarización de NIST, siendo este ASCON, ASCON cuenta tanto con cifrado AEAD como función de hash, especialmente interesante en este proyecto dado que buscamos cambiar SHA-256 u otros.

3. Metodología.

La elaboración de este trabajo busca, como hemos incluido en el punto de la introducción, unificar las tecnologías del estandarizado FIDO2 y la criptografía ligera y comprobar si dicho experimento supone una alternativa viable con la que establecer un protocolo de autenticación robusto y aún más eficiente que con el uso de criptografía clásica.

En lo referente a la parte práctica de este proyecto, la idea es realizar una *Proof of Concept* (PoC) que consiste en una simulación de FIDO2 simplificada con dos versiones distintas, en una utilizamos criptografía clásica utilizada por el estandarizado de por sí, con AES-256 como algoritmo simétrico y SHA-256 como función hash, la otra versión utilizará criptografía ligera dos versiones del algoritmo ligero ASCON, DryGASCON y ASCON-Hash, en ambas se utilizará ECC256 como clave pública. La PoC se realizará con el lenguaje de programación de alto nivel Python, haciendo uso de librerías como pydantic para establecer los modelos de datos, y la comunicación entre los actores se realizará mediante Sockets. Una vez terminada la simulación, se compararán los tiempos de ejecución y se discutirá su viabilidad comparando el uso de un tipo de criptografía u otra.

En lo referente a la parte teórica del proyecto, la memoria se centrará en el estudio tanto de FIDO como de la criptografía ligera, por parte de FIDO, se hará hincapié en los sets de estandarizados más utilizados actualmente de los 3 que ha establecido la FIDO Alliance, estos son FIDO U2F y FIDO2, también se entrará en detalle con las tecnologías WebAuthn y CTAP, que diferencian a FIDO2 con el resto de los sets. Con la criptografía ligera, se revisará su estado actual dentro del sector, como expansión de la información proporcionada en el estado del arte y se explicará en detalle el algoritmo propuesto para este proyecto, siendo ASCON el algoritmo elegido, utilizando una versión modificada DryGASCON-256 como sustituto del algoritmo simétrico AES-256, no es posible usar ASCON-AEAD dado que el tamaño máximo de llave es 128 bits, mientras que utilizaremos ASCON-Hash como alternativa de SHA-256 como función Hash.

4. FIDO

Fast Identity Online Alliance o FIDO Alliance es una organización abierta compuesta por empresas líderes en el sector tecnológico, agencias gubernamentales, instituciones financieras… y otros tipos de entidades con un objetivo claro: reducir al mínimo la dependencia de la industria en el uso de contraseñas como método de autenticación.

La *FIDO Alliance* cuenta con más de 250 miembros, entre los que se encuentran Apple, Google, Amazon, American Express, Meta, Microsoft, Mastercard, Samsung, VMware, IBM Security… pero como se ha mencionado no se cuenta únicamente con entidades privadas, también podemos encontrarnos con agencias gubernamentales como la agencia de ciberseguridad alemana (BSI), el gabinete oficial del Primer Ministro británico, el instituto nacional de estándares y tecnología (NIST) y las agencias de tecnología japonesa, australiana, coreana y taiwanesa… entre muchas más empresas e instituciones.

Actualmente predomina el uso de contraseñas definidas bajo unas directrices exigidas por los administradores de sistemas, a veces acompañadas de una contraseña de un solo uso (One Time Password – OTP) enviada por SMS como segundo factor de autenticación doble, que incluso este segundo paso de autenticación es vulnerable a ciberataques centrados en factor humano como el Phishing o Vishing, o incluso a grandes filtraciones de credenciales por parte de cibercriminales que se lucran subiendo dicha información a la Dark Web, como por ejemplo la filtración de Yahoo de 2013 en la que se vio comprometida la seguridad de 3.000 millones de cuentas, incluido nombre real, número de teléfono, firmas digitales, conversaciones privadas... para combatir el uso de contraseñas y evitar este tipo de incidentes a gran escala *FIDO Alliance* actualmente ha publicado tres sets de especificaciones como alternativas para una autenticación más simple y robusta:

1. FIDO U2F (Universal Second Factor).
2. FIDO UAF (Universal Authentication Framework)
3. FIDO2 (Incluye W3C's Web Authentication y FIDO Client to Authentication Protocol – CTAP)

6.1. FIDO U2F

FIDO Universal Second Factor fue el primer paso de la FIDO Alliance en establecer un estandarizado que mejorar la autenticación, desarrollado por Yubico y Google, y probado con éxito con la plantilla de Google[7], FIDO U2F incluye un segundo factor de autenticación que busca complementar, no sustituir al uso de contraseñas como principal método de autenticación, para entender cómo funciona este proceso de autenticación es necesario primero explicar algunos conceptos, empezando por los actores principales que forman parte del protocolo:

· Autenticador: traducido literalmente del concepto creado por la FIDO Alliance "Authenticator" en inglés, un autenticador como su nombre indica es un dispositivo, normalmente un dispositivo USB o NFC, centrado en la autenticación de un usuario con dos tareas básicas, como son la generación de credenciales de usuario o registro y la autenticación en sí, en el caso del registro, un autenticador genera una clave pública y una clave privada, la clave pública es compartida con

el dispositivo del usuario, normalmente incluida en el navegador para ganar velocidad en el proceso de autenticación, mientras que la clave privada nunca debe salir del autenticador, ya que la seguridad de este factor de autenticación radica en parte en el aislamiento del autenticador de los dispositivos del usuario, aunque un autenticador no tiene por qué ser siempre una pieza de hardware externa, el autenticador también puede ser un programa que contenga la clave privada, aunque es menos recomendable al no poder aislarlo físicamente.

· Relying Party: una Relying Party o RP corresponde al servidor FIDO incluido en la arquitectura que procesa las peticiones de los clientes de acceder a determinados elementos que están guardados por el servicio online, como información personal de una cuenta o acceso a recursos.

· Cliente FIDO: un cliente FIDO es el intermediario entre el autenticador y la Relying Party, este suele ser un navegador web dado que la comunicación suele estar precedida por un servidor FIDO y un autenticador preparado con la clave privada.

· Key Handle: Un key handle en FIDO U2F es una clave creada del codificado del origen de la solicitud, que está pensada para servir de identificador junto con la clave pública, el autenticador genera y utiliza el key handle como identificador para encontrar la clave privada en futuras autenticaciones a la vez que se la manda a la RP en el proceso de registro, mientras que la RP la utiliza para encontrar la clave pública registrada en el servidor.

· Origen: en la documentación de FIDO aparece constantemente un elemento llamado “origin”, simplemente mencionar que es un hash que se obtiene del combinado del protocolo, hostname y puerto.

· Counter: Tanto el autenticador como la RP llevan un conteo del número de conexiones satisfactorias que se han realizado, es una manera de evitar ataques de clonación, una diferencia entre los dos contadores implica que existe otro dispositivo con el mismo par de claves privada/pública, aunque es una medida de seguridad que en determinadas circunstancias es inservible, por ejemplo, si el clon es único y el original nunca se ha utilizado.

El procedimiento de registro, como se muestra en la Figura 1, comienza con la identificación del servicio al que está accediendo el usuario, la RP envía tanto el identificador de la aplicación como un challenge para evitar un ataque de reproducción (replay attack), que será procesado por el autenticador y devuelto firmado con la clave privada. El identificador de la aplicación está revisado por el navegador o Cliente FIDO y se comunicará con el autenticador con la información necesaria para proceder a generar las claves pública y privada, además del handle “h”, que se genera en base a la información proporcionada por el Cliente FIDO (origen, identificador del canal utilizado…), una vez el autenticador tiene preparados los elementos mencionados, manda la clave pública, el handle, un certificado y los firma junto a los elementos que le fueron entregados por el navegador, al propio Cliente FIDO para poder reenviárselo a la Relying Party, que almacenará tanto el handle como la clave pública para futuras autenticaciones del usuario.

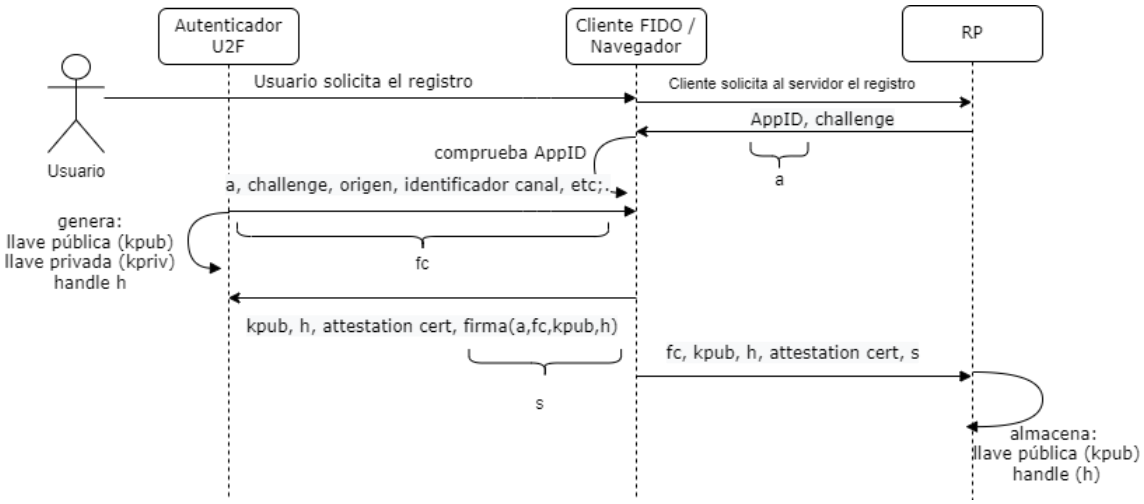


Figura 1. Procedimiento de Registro en FIDO U2F. [4]

En lo referente al procedimiento de autenticación, que corresponde a lo mostrado en la Figura 2, entramos en una situación distinta a la del procedimiento de registro en el que tanto el autenticador como la Relying Party disponen de una información previa con la que realizar el proceso, en este caso, el autenticador deberá tener guardada la clave privada, vinculada a un handle que comparte con la RP, esta RP lo vinculará a la clave pública. El procedimiento es simple y parecido al de registro, una vez el Cliente FIDO ha solicitado la autenticación, la RP se comunica con el Cliente FIDO mandando el handle, el identificador de aplicación y un challenge, el Cliente FIDO comprobará el identificador de aplicación para además definir un origin y un identificador de canal con el que establecer la comunicación (llamaremos a esta información "fc") una vez se haya finalizado el proceso de autenticación, esta información será enviada al autenticador y firmada con la clave privada, encontrada por el handle que en un principio estaba guardado por la RP. Además de la información firmada, el autenticador incluirá la actualización de un counter que va contando el número de veces que se ha realizado el proceso de autenticación, para comprobar posibles ataques de clonación teniendo un registro mutuo que debe coincidir para que no salte ninguna alarma. Tanto el valor actual del counter como la información firmada, será enviada del autenticador al Cliente FIDO, para después reenviarla a la RP, añadiendo la información *fc* con la que establecer la comunicación, una vez entregada, la RP revisará la información proporcionada cifrada por la clave privada, si todo ha funcionado correctamente, se actualizarán las cookies del navegador/Cliente FIDO.

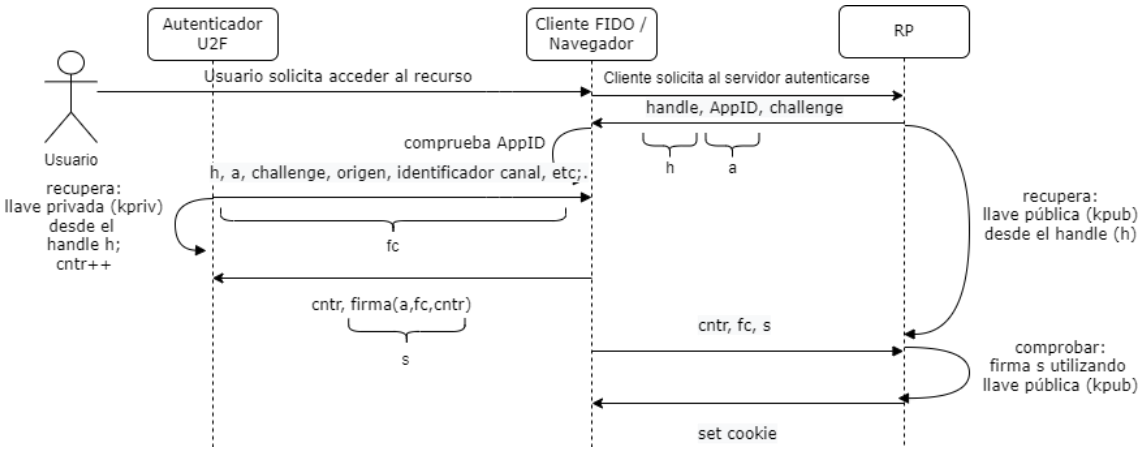


Figura 2. Procedimiento de Autenticación en FIDO U2F. [2]

Como podemos ver, el funcionamiento de FIDO U2F es bastante simple y por ello es sencillo de implementar como segundo factor de autenticación, siendo más seguro que por ejemplo autenticación *One Time Password* (OTP), este tipo de seguridad se ha visto comprometida varias veces de nuevo por el factor humano, mediante estafas y\o suplantaciones de identidad mediante Vishing, esta es otra gran ventaja de FIDO U2F y sus posteriores versiones, al aislar al propio usuario legítimo de la clave privada alojada en el autenticador, dificulta enormemente que el usuario pueda ser engañado por un ciberdelincuente.

6.2. FIDO2 – Visión General

FIDO2 se plantea como el principal set de estandarizado a nivel global al incluir las tecnologías de WebAuthn y CTAP2, está pensado principalmente para utilizar el navegador web o el sistema operativo como cliente FIDO, permitiendo una comunicación rápida y segura, además de contar con .

El protocolo cuenta con 3 componentes principales que son la RP, la plataforma cliente o Cliente FIDO y un autenticador, los pasos a seguir en líneas generales tanto para la operación de registro como de autenticación son los siguientes[8]:

1. En la RP, los casos de uso que involucran registro de credenciales o la autenticación del usuario, la aplicación de la RP llama a “navigator.credentials.create()” o “navigator.credentials.get()”, en el caso de que sea un sitio web, o se usan los métodos equivalentes de la API de la plataforma del cliente si se trata de una aplicación nativa. Otros casos de uso, como gestión de credenciales, establecimiento/mantenimiento de PIN o inscripción biométrica.

En “navigator.credentials”, los desarrolladores recuperan credenciales e interactúan con el almacén de credenciales (user agent’s credential store) a través de métodos expuestos en la interfaz de “CredentialsContainer”, la cual cuelga de un objeto tipo “Navigator” como “navigator.credentials”, como podemos ver en la imagen 1 son un total de 4 operaciones las que cuenta la interfaz “CredentialsContainer”:

1. Get().
2. Store().
3. Create().
4. preventSilentAccess().

```
[Exposed=Window, SecureContext]
interface CredentialsContainer {
  Promise<Credential?> get(optional CredentialRequestOptions options = {});
  Promise<Credential> store(Credential credential);
  Promise<Credential?> create(optional CredentialCreationOptions options = {});
  Promise<undefined> preventSilentAccess();
};

dictionary CredentialData {
  required USVString id;
};
```

Imagen 1. Interfaz de CredentialsContainer.[9]

2. La plataforma cliente establece la conexión con un autenticador disponible que haya utilizado un criterio obtenido por la aplicación de la RP y otra posible información que tenga la plataforma para seleccionar el autenticador.

3. La plataforma cliente obtiene la información sobre el autenticador utilizando el comando *authenticatorGetInfo*, esta información permite a la plataforma saber las capacidades del autenticador.

   authenticatorGetInfo, como indica su nombre, funciona como método que permite a las plataformas cliente recibir una lista con información relevante para la conexión entre la aplicación de la RP y el propio autenticador, *authenticatorGetInfo* devuelve un listado de información que incluye, siendo los siguientes los imprescindibles para el correcto funcionamiento de CTAP2:

   - *Versions:* lista de versiones compatibles con el dispositivo, sea FIDO_2_1 (CTAP2.1), FIDO_2_0 (CTAP2.0) o U2F_V2 (CTAP1 y U2F), entre otras versiones de FIDO.
   - *AAGUID*: Authenticator Attestation GUID es un identificador de 128 bits que indica el tipo del autenticador, de esta manera la plataforma cliente reconoce las versiones de firmware con las que es compatible la llave.

4. Dependiendo de la operación iniciada en el paso 1 de la aplicación de la RP, o de la propia plataforma cliente, las opciones proporcionadas en los métodos y las capacidades del autenticador, la plataforma cliente invocará uno o más comandos de la API del autenticador. Esta API cuenta con varias interfaces y diccionarios con las que se definen los métodos utilizados por los tres actores principales de FIDO2 (RP, Autenticador y Cliente FIDO) para el correcto funcionamiento del protocolo, destacando por ejemplo la *PublicKeyCredential* que cuenta con las extensiones *CredentialCreationOptions* y *CredentialRequestOptions*, con las que la Relying Party define la información inicial que enviar al Cliente FIDO al comenzar un proceso de registro o de autenticación respectivamente, o la interfaz *AuthenticatorResponse*, que cuenta con *clientDataJSON*, este atributo cuenta con el tipo de operación que se lleva a cabo, el challenge originado en la RP y otra información que requiere el cliente FIDO para enviarla a la RP.

Hablando de manera más específica, vamos a distinguir las operaciones entre registro y autenticación, los pasos a seguir son:

6.3. FIDO2 – Registro

En la operación de registro, se busca la creación y almacenamiento de las credenciales tanto en el autenticador como en la Relying Party, permitiendo futuras autenticaciones a los recursos o servicios protegidos por FIDO2.

Los pasos seguidos para el procedimiento de registro se pueden ver en la Figura 3, junto a una descripción detallada de las interfaces, diccionarios y objetos utilizados en el proceso para el correcto intercambio de información entre las entidades de FIDO2:

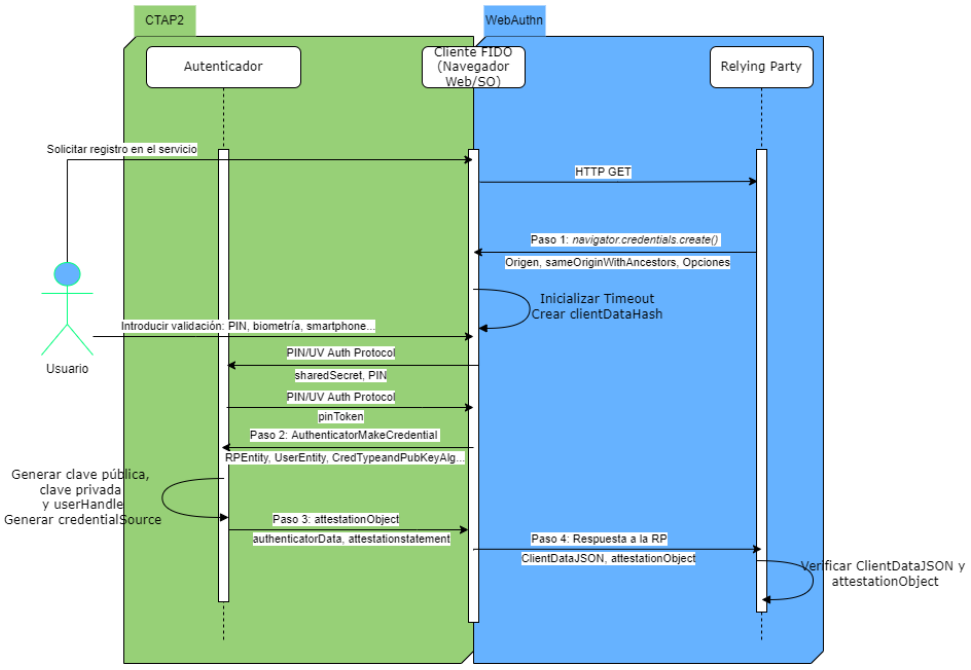


Figura 3. Procedimiento de registro FIDO2 [3]

0. El usuario mediante el Cliente FIDO/plataforma cliente, solicita a la RP el registro de una nuevo autenticador.

1. [WebAuthn] Respondiendo a la solicitud, la RP llama a la función "navigator.credentials.create()" para la creación de un nuevo registro de credenciales con clave pública, este envía al cliente FIDO la siguiente información:
    - Origen de la RP, que incluye información como dominio, IP y/o hostname.
    - "sameOriginWithAncestors" es un valor booleano que funciona como indicador para saber si el entorno de ejecución es el mismo que el del Cliente FIDO, en este caso el valor sería "true" y "false" si estamos ante un caso de *cross-origin*.
    - El atributo options cuenta con un listado de ítems necesarios para el correcto funcionamiento del protocolo, este listado es un diccionario llamado *PublicKeyCredentialCreationOptions*, del que obtenemos la siguiente información:
        - Rp (tipo: "PublicKeyCredentialRpEntity": contiene un identificador para la Relying Party responsable de la petición.
        - Usuario (tipo: "PublicKeyCredentialUserEntity"): *userEntity* contiene nombres de usuario y un identificador (user handle) para la cuenta usuario que ha realizado la petición de manera.
        - Challenge/desafío (tipo: "BufferSource"): corresponde al challenge que el autenticador firmará junto a otros datos, es importante que sean generados aleatoriamente por la RP dado su importancia a la hora de evitar ataques de reproducción (replay attacks), para asegurar su eficacia, W3C exige que contengan suficiente entropía para hacerlos imposibles de sacar con la tecnología actual, recomendable que tengan de tamaño mínimo 16 bytes (https://w3c.github.io/webauthn/#sctn-cryptographic-challenges).
        - PubKeyCredParams (Tipo: PublicKeyCredentialParameters): incluye un listado de tipos de cifrados y firmas que soporta la RP, ordenados de más a menos preferidos por la RP.
        - Timeout (Tipo: unsigned long): atributo opcional, especifica como su nombre indica, un tiempo límite, en milisegundos, para que la RP sea respondida antes de cesar en la petición.
        - ExcludeCredentials (Tipo: PublicKeyCredentialDescriptor): listado opcional de credenciales vinculadas al usuario solicitante, está pensado para evitar que se puedan crear nuevas credenciales repetidas en base a un mismo autenticador.
        - AuthenticatorSelection (Tipo: AuthenticatorSelectionCriteria): listado opcional con capacidades y configuraciones que

un autenticador debe o debería tener para poder participar en la operación *navigator.credentials.create*.

El user agent valida la información obtenida por la RP, y crea una instancia del ClientData que incluye la siguiente información:

- Type (tipo: DOMString): el tipo de operación, que puede variar entre "webauthn.create" y "webauthn.get", dependiendo de si se trata de registro o de autenticación.
- Challenge (tipo: DOMString): contiene el challenge incluido por la RP, en formato codificado como base64url.
- Origen (tipo: DOMString): contiene el origen de la petición, normalmente corresponde a un valor interno que no tiene estandarizado, o a una tupla que contiene un valor ASCII string como identificador, un hostname, un puerto y un dominio, pudiendo ser valor null los dos últimos.
- CrossOrigin (tipo: boolean): este elemento es el inverso al sameOriginWithAncestors.
- tokenBinding: como medida opcional, incluye el protocolo utilizado para la comunicación entre el user agent y la relying party, sin embargo, ninguno de los navegadores web más utilizados lo utiliza.

La instancia de *ClientData* se serializa en un formato JSON y se procede a pasarlo por SHA-256, creando *ClientDataHash*, seguidamente el user agent inicia el timer vinculado al timeout de la RP, localizando a su vez los autenticadores disponibles y comprobando el atributo *authenticatorSelection* para escoger el más compatible con el funcionamiento de la RP, seleccionando el primero que cumpla el criterio y que no esté incluido en la lista de *excludeCredential*.

2. [CTAP2] Una vez el Cliente FIDO cuenta con la información necesaria de la RP, solicita al autenticador mediante la función *authenticatorMakeCredential* a la vez que manda a este la información obtenida de la RP y el *ClientDataHash*, como *rpEntity* o *userEntity* exceptuando el *timeout* y el criterio de selección del autenticador, que se lo reserva el Cliente FIDO, la función también recibe varios parámetros que son: *requireUserPresence, requireUserVerification y credTypesAndPubKeyAlgs* (Incluye el tipo de credenciales de clave pública y los algoritmos pedidos por la Relying Party). El autenticador verifica la información recibida del cliente y consulta al usuario su consentimiento para empezar a crear las credenciales.

3. [CTAP2] Al comprobar el consentimiento del usuario, sea con alguna acción que pueda realizar el usuario con el autenticador y comprobar su presencia física (insertar su huella en un dispositivo con biometría, pulsar un botón en una security key por USB o NFC…), el autenticador genera el par de claves pública y privada junto al *user handle*, este elemento es un identificador

creado por la RP, vinculado de manera única a la cuenta de usuario que solicita la operación de registro, facilitando la gestión de cuentas de usuario. La información generada por el autenticador, junto a datos obtenidos desde el paso 2 por parte del Cliente FIDO, entre los que se encuentra el identificador de la RP (*rpId)*, el tipo de credenciales de clave pública utilizada (*type*) y un identificador propio (*id*), genera el credentialSource, que es el activo que utilizará el autenticador para vincular un futuro intento de autenticación con las credenciales necesarias.

El autenticador, además, activará un contador llamado "signatureCounter", este contador está pensado, al igual que en FIDO U2F, para llevar un seguimiento del número de autenticaciones exitosas realizadas por el autenticador (que se traduce en cada operación exitosa del método *authenticatorGetAssertion*), forzando así la detección de autenticadores clonados. El autenticador también responderá al Cliente FIDO enviando un objeto llamado *attestation object*, este objeto contiene dos elementos necesarios para la respuesta al cliente, estos son *authenticatorData* y *attestationStatement*, en el caso del primer elemento este contiene: RPid hash, *flags* que indican si el proceso de autenticación contiene la presencia/interacción del usuario, verificación y si tiene extensiones incluidas, el *signatureCounter* y un atributo nuevo llamado *attested credential*, que es, básicamente un byte array que contiene el AAGUID del autenticador, el tamaño del Credential ID, el propio Credential ID y *credentialPublicKey*, que es el parámetro *alg* sin ningún otro tipo de parámetro opcional, definido en CBOR. En lo referente al otro componente, siendo este *attestationStatement*, está pensado para incluir información de las credenciales de clave pública y del propio autenticador que las ha creado, este contiene una firma de atestación, creada mediante una llave de autoridad de atestación que contiene el autenticador, que funciona como un "testigo" de confianza para el Cliente FIDO y la Relying Party, sin embargo, no todos los autenticadores cuentan con una firma de atestación, de manera que hay casos en los que se firma simplemente con la clave privada generada.

4. [WebAuthn] Una vez recibida la contestación del autenticador, el Cliente FIDO vuelve a hacer de intermediario entre el autenticador y la RP, y con la información obtenida del autenticador, elabora el objeto *clientDataJSON* que cuenta con: el tipo de operación realizada (webauthn.create o webauthn.get), el challenge, el origen de la RP y el objeto *tokenBinding*. El Cliente FIDO una vez comprobada la conexión, enviará a la Relying Party los siguientes datos: el *attestation object*, el *ClientDataJSON*, el protocolo de transporte utilizado por el autenticador (NFC, BLE…) y las extensiones del cliente. La Relying Party, una vez ha recibido la información y , verifica la información del *clientData* y el hash de *clientDataJSON* junto con el *attestation Object* decodificándolo para almacenar la información necesaria, como el identificador del autenticador, el *userhandle*, las credenciales de clave pública y otros elementos necesarios para futuros intentos de autenticación.

6.4. FIDO2 - Autenticación:

En la operación de autenticación, con una etapa de registro ya concluida y con las credenciales de clave pública y la información necesaria intercambiada, se busca que el usuario legítimo pueda acceder a los recursos que tiene de una manera rápida y segura.

Los pasos seguidos para el procedimiento de autenticación se pueden ver en la Figura 4, junto a una descripción detallada de las interfaces, diccionarios y objetos utilizados en el proceso para el correcto intercambio de información entre las entidades de FIDO2:

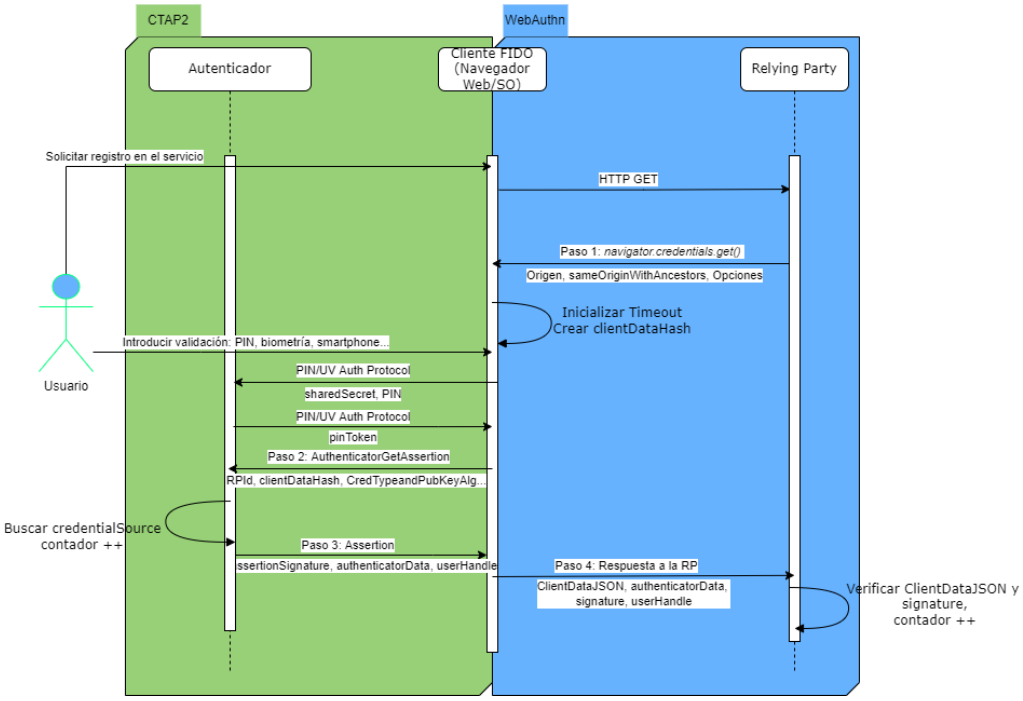


Figura 4. Procedimiento de autenticación FIDO2. [3]

0. El usuario solicita autenticarse mediante el Cliente FIDO a un servicio que cuenta con FIDO2, ya teniendo registro en la RP del servicio de un anterior proceso de registro.

7. [WebAuthn] A diferencia de con *navigator.credentials.create*, en el caso de *navigator.credentials.get*, contamos con dos funciones distintas para comenzar el envío de datos desde el RP hasta el Cliente FIDO: *CollectFromCredentialStore* y *DiscoverFromExternalStore*. La diferencia entre una función y otra es que en la primera se utiliza un autenticador en el que NO es necesaria la interacción del usuario y en la segunda SÍ. Con interacción del usuario nos referimos a tener que realizar un "gesto de autorización" con el que el usuario da su aprobación a una parte del procedimiento de registro o de autenticación, por ejemplo, pulsar un botón en una security key. En ambas funciones contamos con los mismos parámetros, que cuentan con elementos muy similares a los del procedimiento de registro: *origin* (por

ejemplo, un dominio o una dirección IP con puerto), *sameOriginWithAncestors* que nos indica el entorno de ejecución y el apartado *options*, este último apartado abarca un diccionario de datos de tipo *PublicKeyCredentialRequestOptions*.

Este diccionario es similar al del procedimiento de registro *PublicKeyCredentialCreationOptions*, pero con menos elementos como podemos ver en la imagen 2, como siendo el miembro *challenge* el único indispensable para proceder con la autenticación. En este diccionario encontramos dos nuevos elementos exclusivos de *PublicKeyCredentialRequestOptions*:

- *allowCredentials* (Tipo: PublicKeyCredentialDescriptor): utilizado por el cliente FIDO para encontrar autenticadores válidos con el proceso de autenticación listando las credenciales asociadas a la cuenta de usuario, funciona de dos maneras distintas, dependiendo de si el usuario se ha autenticado con su cuenta de usuario de antes (facilitando la identificación de las credenciales registradas) o si no lo ha hecho, en este caso la RP puede dejar este campo vacío, la identificación se realizaría con el *userhandle*.
- *userVerification* (Tipo: DOMString): la RP es posible que requiera verificación del usuario, de manera que se establece mediante una cadena de caracteres, tres posibles configuraciones de este miembro, “required” cuándo la RP establece como obligatoria la verificación del usuario, “preferred” cuándo la RP prefiere que se realice la verificación, pero si no es posible no fallará el procedimiento, siendo esta segunda configuración la predeterminada, y “discouraged” cuándo la RP rechaza que se utilice la verificación.

```
dictionary PublicKeyCredentialRequestOptions {
    required BufferSource                challenge;
    unsigned long                        timeout;
    USVString                            rpId;
    sequence<PublicKeyCredentialDescriptor> allowCredentials = [];
    DOMString                            userVerification = "preferred";
    AuthenticationExtensionsClientInputs extensions;
};
```

Imagen 2. Diccionario “PublicKeyCredentialRequestOptions”.[10]

Una vez incluidos y recibidos todos los valores necesarios, el Cliente FIDO comprobará la información y creará un *ClientDataJSON*, del que obtendremos también *clientDataHash*. Además, empezará un timer llamado *lifetimeTimer* que parte del parámetro timeout del RP (si no está este parámetro, lo establecerá el Cliente FIDO), elegirá un autenticador

entre los disponibles y comprobará si en la información distribuida por la RP se encuentran credenciales de clave pública vinculadas al usuario.

8. [CTAP2] Una vez inicializado el Timer y creado el *clientDataHash*, el Cliente FIDO envía el objeto equivalente a *authenticatorMakeCredential* para el proceso de autenticación: *authenticatorGetAssertion*. Este objeto cuenta con algunos atributos repetidos del otro objeto, como el *clientDataHash*, *requireUserPresence y requireUserVerification*. El autenticador pedirá que el usuario introduzca su credencial, como puede ser su huella dactilar, si el usuario ha conseguido identificarse correctamente con su dispositivo, se comprobarán los parámetros recibidos de *authenticatorGetAssertion*, si se les da el visto bueno y no salta ningún código de error, el autenticador dará por buena la autenticación y se incrementará el contador asignado a dichas credenciales, aunque hay autenticadores que tienen un contador global.

9. [CTAP2] El autenticador una vez ha dado su visto bueno, enviará la respuesta al cliente FIDO, en la que incluirá la siguiente información: *selectedCredential.id* (Credential ID, es un identificador único vinculado a una *publicKeyCredentialSource)*, *authenticatorData* idéntico al del proceso de registro, el *userhandle* de las credenciales seleccionadas y finalmente *assertionSignature*, que corresponde a la concatenación de *authenticatorData* y *clientDataHash*, firmadas con la clave privada seleccionada por el autenticador, esta firma funciona como confirmación para la Relying Party de que el usuario ha consentido la transacción concreta por la que se ha inicializado el proceso de autenticación, como puede ser un inicio de sesión o una transacción monetaria.

10. [WebAuthn] El cliente FIDO reenvía la información recibida por el autenticador a la Relying Party, añadiendo el *clientData* y las posibles extensiones que puedan estar habilitadas por el cliente. Una vez recibida la respuesta del cliente FIDO, la RP comprobará toda la información necesaria para confirmar su visto bueno (*clientData*, *clientDataHash*, *challenge*, origen, RPId)

6.5. FIDO2 – Protocolo PIN/UV

Hay un hueco en la seguridad de FIDO2 que no se ha detallado hasta ahora en el proyecto, y es que ocurre en ambos procesos, tanto en el de registro como en el de autenticación, el envío de información del Cliente FIDO al autenticador no tiene protección ante ataques de sniffing, un programa malicioso capaz de monitorizar la comunicación entre el navegador o el sistema operativo con la llave del autenticador, no tendría impedimentos para leer los paquetes de datos que llegan al autenticador. Como vemos en la imagen 3, es una vulnerabilidad que la FIDO Alliance ha tenido en cuenta y que cuenta con dos soluciones, pero siendo la segunda más interesante para el proyecto, y es el uso del “PIN/UV Auth Protocol”.

T-1.3.4	Sniffing Communication between ASM / FIDO Client and the Authenticator	Violates
AC5	In this threat, an attacker is able to read all data being sent to the authenticator (e.g. via CTAP). **Consequences:** When combined with T-1.3.3 it may undermine [SG-1] if an "external" user verification method (e.g. authenticatorClientPIN) is being used [FIDORegistry]. **Mitigations:** This threat is mitigated by [SM-1], more specifically by the ability for authenticators to either implement "internal" user verification methods [FIDORegistry] or to encrypt the communication to the authenticator using PIN/UV Auth Protocol One/Two [FIDOCTAP].	SG-1

Imagen 3. Amenaza/Threat número 1.3.4. en FIDO2. [11]

PIN/UV Auth Protocol es un protocolo de autenticación centrado en proteger la información compartida entre el autenticador y el Cliente FIDO, proporcionando un método de verificación de usuario mediante un PIN que proporciona el usuario para desbloquear el funcionamiento del usuario, este PIN consiste en un número de mínimo 4 números que el autenticador, el estandarizado de FIDO2 recomienda un máximo de 8 intentos introduciendo el PIN antes de bloquear la llave por posible vulneración de seguridad y un timeout definido en base al método de comunicación del autenticador (NFC, BLE, USB o en un autenticador interno), el procedimiento de cifrado de la comunicación es simple y hace uso de criptografía simétrica (normalmente AES-256) con un *sharedSecret* único establecido mediante el protocolo de curva elíptica Diffie-Hellman en cada iteración que se realice de *authenticationMakeCredential* o de *authenticationGetAssertion*, con el uso de la criptografía simétrica permitimos al autenticador y al FIDO Client establecer un *PINtoken* aleatorio que funcionará como Token en común para los dos a la hora de pasar información (no confundir el PINtoken, los pasos a seguir por el Cliente FIDO y el autenticador son:

1. El autenticador genera una clave pública por ECDH a la que llama “aG”, a la vez que genera un Token random al que llamaremos *pinToken*.

2. Después de comprobar que no se exceden el número de intentos impuesto en el valor *pinRetries*, el autenticador compartirá su clave pública “aG” al Cliente FIDO, el cliente FIDO generará su propia clave pública “bG” y generará el *sharedSecret* creando un hash mediante SHA-256 que concatena ambas claves públicas.

3. El cliente FIDO establecerá un PIN escogido por el usuario, y deberá mandárselo al autenticador junto a otros elementos, el Cliente FIDO mandará: su clave pública “bG” para que el autenticador pueda generar el *sharedSecret*, el nuevo PIN cifrado mediante AES-256 usando como clave el *sharedSecret* y el hash en HMAC-SHA-256 del nuevo PIN, también usando como clave el *sharedSecret*. Una vez entregados, el autenticador generará el *sharedSecret*, descifrará el ciphertext y hash entregado por el Cliente FIDO, los validará y almacenará el nuevo PIN como un SHA-256.

4. Aprovechando que el autenticador ya tiene registrado el PIN de ese cliente FIDO en particular, cada vez que el cliente quiera obtener el *pinToken*, solo tiene que compartir cifrado con AES el hash en SHA256 del PIN establecido y la clave pública “bG”, después de volver a generar el *sharedSecret*, descifra el ciphertext y obtiene el hash del PIN, lo compara con el almacenado en el paso anterior, si coincide y lo valida, envía el *pinToken* cifrado con el *sharedSecret* de vuelta al Cliente FIDO. Con el *pinToken*, el Cliente FIDO puede comenzar a llamar a las funciones *authenticationMakeCredential* y *authenticationGetAssertion*, únicamente tiene que crear un HMAC-SHA-256 con el pinToken y el *clientDataHash*, que será revisado por el autenticador para poder verificar si el origen de las peticiones es válido o no.

7. Criptografía Ligera

Lightweight Cryptography o LwC es el nombre que se le ha puesto internacionalmente a la que conocemos como criptografía ligera, esta es una rama del campo de la criptografía centrada en la creación de algoritmos AEAD y funciones hash con un objetivo claro, que es proponer alternativas a los cifrados convencionales de la industria como AES o SHA-256, estos cifrados son eficaces, pero poco eficientes a nivel computacional y energético en el caso de un dispositivo con pocas prestaciones, como sistemas empotrados específicos, etiquetas RFID o redes de sensores entre otros[12].Por ello tanto el NIST como ISO/IEC entre otras organizaciones, han realizado, algunas sin haber terminado aún, varios procesos de estandarizado o competiciones para escoger estándares y recolectar propuestas de algoritmos que cumplan unos determinados objetivos, siendo estos conseguir algoritmos robustos que cumplan las características de confidencialidad, integridad y disponibilidad con un coste a nivel energético y computacional mínimo para poder ser utilizado por cualquier tipo de dispositivo.

En el caso por ejemplo del proceso de estandarización del NIST, existe un documento, publicado en agosto de 2018, con los tipos de requisitos mínimos que deben cumplir todas las propuestas para este proceso si quieren aspirar a convertirse en estándar [13] en este documento se detallan varios aspectos en lo referente a la funcionalidad de los algoritmos, tamaño de claves mínimo para mantener la seguridad del cifrado… como se ha mencionado anteriormente, hay dos grupos de herramientas diferenciados a la hora de establecer los requisitos mínimos[13]:

1. Algoritmo AEAD: estos algoritmos deben tener operación de cifrado como de descifrado, teniendo como parámetros el ciphertext/plaintext, los datos asociados, *nonce* y por último la llave de cifrado. NIST exige también que se documente en detalle el proceso que siguen los algoritmos AEAD que utilizan un *nonce* repetido en varias iteraciones y que se compruebe que siga siendo viable la seguridad del algoritmo a pesar de esta medida. Los algoritmos AEAD se dice que tienen "familia" cuándo tienen varias versiones del algoritmo con distintos tamaños de llave o *nonce*, como de número de rondas que realiza el algoritmo, estas familias se crean para ofrecer una mayor variedad del algoritmo, y facilitar su implementación en distintos tipos de dispositivo, mencionar que el NIST exige que una propuesta de algoritmo no puede tener más de 10 miembros en su familia y que en dicha familia, una versión debe tener obligatoriamente al menos 128 bits, un *nonce* de mínimo 96 bits y los límites de entrada del texto plano y los otros parámetros, no deben bajar de 2^{50}-1 bytes.

2. Función Hash: las funciones hash coinciden en algunos requisitos con los algoritmos AEAD, por ejemplo, los ataques criptoanalíticos deben requerir mínimo 2^{112} computaciones por parte de un ordenador clásico con configuración "single-key setting". Las funciones hash también deben ser seguras ante un intento de ataques de extensión de longitud y ataques de colisión. Importante también que estas funciones no deben tener tamaños de salida menores a 256 bits y que han de aceptar cualquier tamaño de entrada menor al límite de tamaño establecido por la salida.

Sin embargo, además de requisitos mínimos, los procesos de estandarizado también han establecido distintos atributos que se utilizan como métricas para diferenciar los puntos fuertes

y débiles de cada cifrado LwC, estos atributos abarcan aspectos a nivel de hardware y software[14]:

- Rendimiento (Throughput-Software): rendimiento medido por texto plano cifrado por ciclos de reloj por bloque.

- Rendimiento (Throughput-Hardware): rendimiento medido por texto plano cifrado por unidad de tiempo, en bits por segundo.

- Latencia: tiempo que tarda el algoritmo en cifrar el texto plano, medido en número de ciclos de reloj por bloque.

- Consumo energético: la ecuación con la que obtenemos el consumo, en este caso la tenemos en la ecuación 1, se mide como la latencia en el contexto de implementación software, ciclos por bloque, por la potencia en microvatios, dividido por el tamaño de bloque en bits.

$$Energy[\mu J] = \frac{Latency[cycles/block] * Power[\mu W]}{blocksize[bits]}$$

Ecuación 1. Cálculo de la eficiencia software [14]

- Eficiencia(Software): la eficiencia en software se conoce como el rendimiento (Kbps) entre el tamaño del algoritmo en Kilobytes, como se muestra en la ecuación 2.

$$Software\ Efficiency = Troughput[Kbps]/CodeSize[KB]$$

Ecuación 2. Cálculo de la eficiencia software [14]

- Eficiencia(Hardware): en este caso, como vemos en la ecuación 3, la eficiencia es el rendimiento hardware (Kbps) dividido por la complejidad, siendo esta el espacio físico utilizado o gate área.

$$Hardware\ Efficiency = Troughput[Kbps]/Complexity[KGE]$$

Ecuación 3. Cálculo de la eficiencia hardware [14]

- Espacio físico: espacio o gate área utilizado para implementar el algoritmo en el circuito o placa, medido en µm².

- Requisitos de memoria: capacidad de la memoria RAM/ROM necesaria para ejecutar el cifrado.

Actualmente existen una cantidad considerable de distintos algoritmos, diferenciados tanto por funcionalidad como por tipo de cifrado utilizado, en la Tabla 1 se muestran los algoritmos que pasaron la segunda ronda del proceso de estandarizado del NIST, divididos en las dos subcategorías mencionadas:

Candidatos solo AEAD	
Permutación	Elephant, ISAP, Oribatida, SPIX, SpoC, Spook,WAGE
Cifrado por bloques	COMET, GIFT-COFB, HyENA, mixFeed, Pyjamask, SAEAES, SUN-DAE-GIFT, TinyJAMBU
Cifrado por bloques modificable	ESTATE, ForkAE, LOTUS-AEAD, LOCUS-AEAD, Romulus, Spook
Cifrado por flujo	Grain-128AEAD
Candidatos AEAD + función hash	
Permutación	ACE, ASCON, DryGASCON, Gimli, KNOT, ORANGE, PHOTON-Beetle, SPARKLE, Subterranean 2.0, Xoodyak
Cifrado por bloques	SATURNIN
Cifrado por bloques modificable	SKINNY-AEAD y SKINNY-HASH

Tabla 1 – Candidatos de la segunda ronda del proceso de estandarización del NIST.[1]

De todos estos cifrados, actualmente se encuentran en la fase final del proceso de estandarizado del NIST 10 cifrados distintos, de los cuales interesa ASCON para el proyecto de este trabajo, en concreto, la versión función hash del algoritmo y como algoritmo AEAD no se puede utilizar ASCON-AEAD ya que el tamaño máximo de clave es de 128 bits y el algoritmo simétrico a sustituir es AES-256, a cambio existe una alternativa que es una variante de ASCON que utiliza sus permutaciones y en otro cifrado llamado DrySponge: DryGASCON, este cifrado sí cuenta con una alternativa con clave de 256 bits.

7.1. ASCON

ASCON es un conjunto de cifrado con doble funcionalidad: algoritmo AEAD y función hash. Por la parte de AEAD, ASCON cuenta con 3 variantes distintas, en las que se incluyen ASCON-128, ASCON-128a y ASCON-80pq. Las variantes ASCON-128 y ASCON-128a, entran en la propuesta de cifrado para la *CAESAR competition*, en la que ASCON resultó vencedor en el apartado de "aplicaciones ligeras" junto al cifrado ACORN. En la parte de función hash, ASCON cuenta con 2 versiones: ASCON-HASH y ASCON-HASHA.

Según la documentación oficial del cifrado [15], el funcionamiento de ASCON es una permutación subyacente de 320 bits, definida únicamente con words de 64 bits y funciones booleanas bit a bit, entre las que encontramos las clásicas AND, NOT y XOR junto a rotaciones entre las words de 64 bits, esta combinación, junto a una S-box con un gasto reducido en memoria, permite tener una seguridad robusta a la vez que la implementación cuenta con un muy buen rendimiento tanto en software como en hardware, reduciendo los costes de computación y energéticos de manera notable. ASCON funciona de manera excepcional en escenarios en los que el propio dispositivo con prestaciones ligeras requiere hacer operaciones criptográficas y/o mantener comunicaciones con un servidor de altas prestaciones, en ambos casos encaja con el funcionamiento de FIDO2, siendo el dispositivo de prestaciones ligeras (en este caso, el Cliente FIDO), quién cifra la comunicación con el autenticador o establece la comunicación con un servidor FIDO/Relying Party. Hay que mencionar también que tanto por parte del seguimiento del NIST como en la ya terminada *CAESAR competition*, no se han encontrado vulnerabilidades dentro de ASCON, ni utilizando ataques de canal lateral.

ASCON cuenta con dos variantes, tanto para la función hash como para el cifrado AEAD, en la Tabla 2 podemos ver las dos variantes ASCON-128 y ASCON-128a que corresponden a las variantes del cifrado AEAD con los parámetros escogidos por los desarrolladores, mientras que en la Tabla 3 observamos las variantes ASCON-Hash y ASCON-Hasha que son las variantes de la función hash.

Nombre	Algoritmos	Tamaño en bits				Núm. Rondas	
		Llave	Nonce	Tag	Bloque	p^a	p^b
ASCON-128	$E,D_{128,64,12,6}$	128	128	128	64	12	6
ASCON-128a	$E,D_{128,64,12,8}$	128	128	128	128	12	8

Tabla 2 – Parámetros de los cifrados AEAD de la familia ASCON. [15]

Nombre	Algoritmo	Tamaño en bits		Núm. Rondas	
		Llave	Bloque	p^a	p^b
ASCON-HASH	$E,D_{128,64,12,6}$	256	64	12	6
ASCON-HASHa	$E,D_{128,64,12,8}$	256	128	12	8

Tabla 3 – Parámetros de las funciones hash de la familia ASCON. [15]

8. Implementación práctica: Simulación de FIDO2 + LwC

En esta implementación, se ha realizado una prueba de concepto de FIDO2 en Python, esta PoC no busca ser una recreación perfecta ni una aplicación real del protocolo FIDO2, sino comprobar los beneficios del uso de LwC en un entorno que utilice FIDO2 en comparación de un entorno con FIDO2 que utilice criptografía clásica.

Dado que esta prueba se trata de una comparativa, se han creado dos implementaciones distintas: "*SimulaciónFIDO2"* y "*SimulaciónFIDO2LwC*". En la primera implementación se utiliza como algoritmo AEAD AES-256 y como función hash SHA-256, en la segunda implementación el algoritmo AEAD es DryGASCON-256 y como función hash, ASCON-Hash; pero por el resto de la implementación, el código es idéntico línea por línea para realizar una comparación acertada, el cuál es accesible en el Anexo 1 de esta memoria.

8.1. Desarrollo del simulador.

Como se ha mencionado, el lenguaje de programación utilizado es Python, haciendo uso de sockets para realizar la comunicación entre los tres actores clave: Cliente FIDO, Autenticador y Relying Party. Estos actores intercambian información según la opción que escojamos en la interfaz de usuario del Cliente FIDO mostrada en la Imagen 4, que puede ser o bien el registro de credenciales o autenticarse en un inicio de sesión habiendo registrado previamente los datos.

```
PS C:\Users\Germán\Desktop\SimulacionFIDO2LwC - copia> python3 .\FIDOClient.py
////////// Simulador FIDO2 - Python. //////////
Introduzca el número de una opción para seleccionarla:
1. Registro de credenciales.
2. Inicio de sesión.
Cualquier otra opción cerrará el programa.


2
```

Imagen 4 – Interfaz de usuario Simulador FIDO2.

El desarrollo de las fases de registros y autenticación se ha basado en lo incluido en el punto de FIDO2 de este proyecto, simplificando las clases y los objetos para no complicar innecesariamente la comparativa entre un tipo de implementación y otra. Entrando en cada funcionalidad en específico, en el caso de registro de credenciales el procedimiento es simple, el usuario desde la consola del Cliente FIDO solo necesita introducir un nombre de usuario, este nombre se enviará a la Relying Party y junto a otra información incluida en la clase *createRegistration* como el identificador de la RP, challenge, PubKeyCredParams, etc; se enviará al Cliente FIDO para comenzar el manejo de información necesario entre actores, las clases y objetos se han creado mediante el uso de la librería pydantic [16], concentradas en un archivo llamado *classes.py* como se ve en el Código 1.

```
class makeCredential(BaseModel):
    clientDataHash: bytes
    rpid: str
    userEntity: usuario
    credTypesandPubKeyAlgs: pubKeyCredParams

class clientData(BaseModel):
    type: str
    challenge: str
    origin: origin
    user: usuario

class credentialSource(BaseModel):
    credentialID: str
    type = "public-key"
    clavePrivada: bytes
    rpid: str
    userHandle: usuario
    counter: int
```

Código 1 - Fragmento de código del archivo classes.py

Una vez el Cliente FIDO ha recibido los datos de la clase *createRegistration*, crea el clientDataHash y crea un objeto *makeCredential* para ser enviado al autenticador, donde esté en base a lo incluido en el atributo "*credTypesandPubKeyAlgs*", creará las claves pública y privada necesarias para firmar en ECDSA [17] y poder realizar futuras autenticaciones, además, para recrear la seguridad establecida entre el Cliente FIDO y el Autenticador, se ha utilizado ECDH Key Exchange (Elliptic Curve Diffie Hellman) como método de intercambio de claves entre los dos actores, utilizando un cifrado simétrico posterior para la comunicación entre ambos actores, esto es necesario dado que la comunicación entre el servidor FIDO de la Relying Party y el Cliente FIDO, que suele ser un navegador web, suele tener seguridad por defecto al tratarse de comunicación HTTPS, sin embargo, entre el cliente FIDO y el autenticador no hay nada que sirva de seguro como se detalla en la explicación del protocolo PIN/UV.

La implementación en el código del autenticador se puede ver en el Código 2, en lo referente del cifrado simétrico, varía según la versión del simulador en la que estemos, junto a cambios en las funciones hash, en el caso de la versión con criptografía ligera, se utiliza DryGASCON-256 como sustituto al AES-256 utilizado en la versión sin criptografía ligera, en lo referente a la función hash, en el caso de LwC se utiliza ASCON-Hash, mientras que sin LwC, SHA-256. En ambas simulaciones las implementaciones funcionan correctamente, haciendo uso de unas funciones desarrolladas en el archivo *functions.py* junto a otras funciones necesarias para el correcto funcionamiento de los simuladores, como funciones que establecen la comunicación de sockets, codifican o decodifican los objetos de las clases de *classes.py* para poder transmitirlas, buscan en los archivos externos los registros realizados en la fase de registro… entre otras funcionalidades.

```
 ### ECDH Key Exchange con el Cliente FIDO ###

        #1. Generamos las claves pública y privada.

        authPrivKey = f.generarClavePrivECC()
        authPubKey = f.generarClavePubECC(authPrivKey)

        authPubKeyob = c.publicKeyECDH(x = authPubKey.x,y = authPub-
Key.y, curve = 'brainpoolP256r1')

        #2. Intercambiamos las claves públicas con el Cliente FIDO

        FIDOClientPublicKeyob = f.recibirPaquete(HOST,8887,'fc')
        f.enviarPaquete(authPubKeyob,HOST,8888,'fc')

        FIDOClientPublicKey = Point(registry.get_curve(FIDOCli-
entPublicKeyob['curve']), FIDOClientPublicKeyob['x'], FIDOCli-
entPublicKeyob['y'])

        #3. Generamos la clave compartida.

        sharedKey = f.ecc_point_to_256_bit_key(FIDOClientPublicKey *
authPrivKey)
```

Código 2 - ECDH Key Exchange incluido en el archivo authenticator.py

Una vez establecida la seguridad y creadas las claves pública y privada, el autenticador almacena los datos correspondientes a *credentialSource* y los guarda como un json en la carpeta *credentialSources*, entre los que está la clave privada, el credentialID, el UserHandle generado… a su vez, el autenticador prepara y envía al cliente FIDO la respuesta al servidor FIDO de la RP, que una vez recibe y verifica los datos, almacena esos datos en un objeto *CredentialStoreData* como json en la carpeta *credentialStore*, siendo lo más importante la clave pública, tenemos como ejemplo de *CredentialStoreData* en la imagen 5.

La fase de autenticación/inicio de sesión no es muy diferente a la de registro, pero está centrada como es lógico en la verificación de la firma por ECDSA y en otros aspectos como pueda ser el counter que lleva el conteo del número de autenticaciones, que debe ser parejo tanto en el autenticador como en la RP, evitando ataques de clonación.

```
Se ha encontrado los datos del usuario  mauge.gross
Datos del usuario:
{'credentialID': 'c2857c7b-2e79-11ed-968b-a8934a007b61', 'rpid': 'uah.es', 'origen': {'host':
 '80.26.225.82', 'port': 7800, 'domain': 'www.uah.es', 'scheme': 'HTTPS'}, 'pubKey': '-----BE
GIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEd1gaA+KyDRz+dfX3n231RMFmSn3Y\nHny+KX
20exgVKuSUJTRP7X20OeyL+T8Fc4i20c9Vn8X7N2YTOcinfWNFvw==\n-----END PUBLIC KEY-----\n', 'user':
{'userID': 2761826, 'displayName': 'mauge.gross'}, 'counter': 0}
```

Imagen 5 – Ejemplo de CredentialStoreData.

Para poder visualizar la fase de autenticación contamos con un diagrama de secuencia localizado en la figura 5:

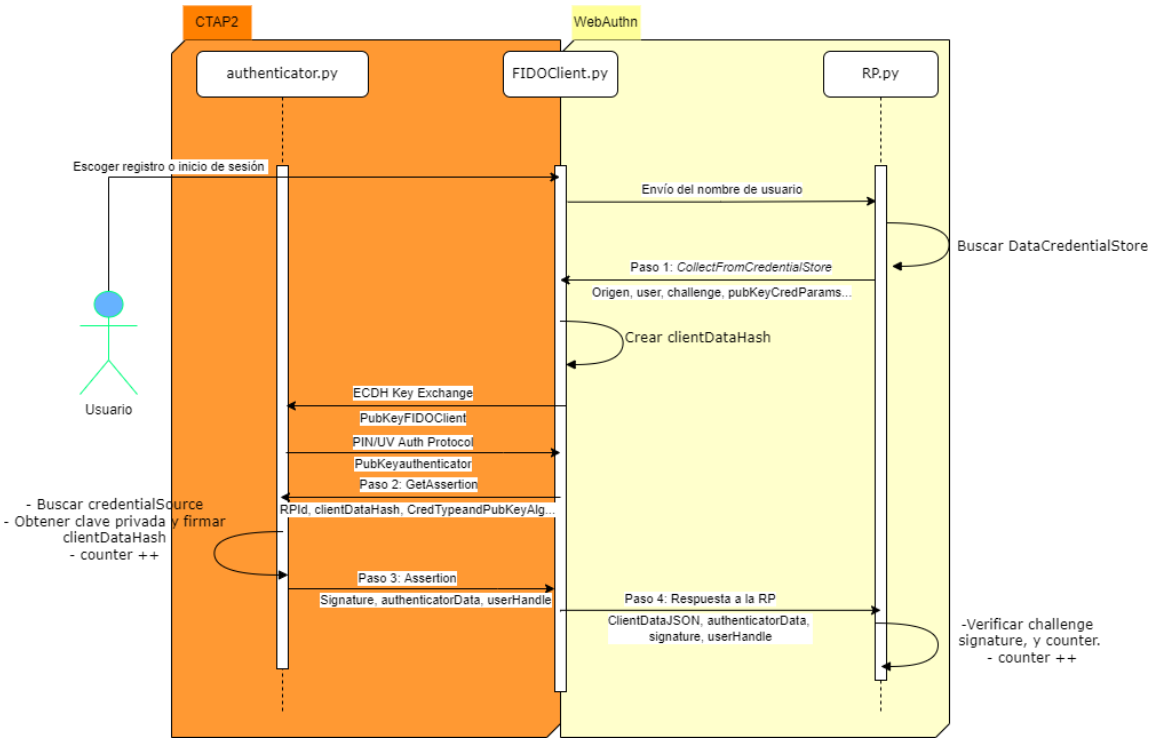


Figura 5 – Diagrama de secuencia de la fase de autenticación.

8.2. Obtención de resultados.

Una vez terminada la implementación de ambos simuladores, se procede a comprobar la eficiencia de cada una de las fases e implementaciones, distinguiendo entre programas sin criptografía ligera (SIN LwC) y programas con criptografía ligera (CON LwC), y entre fase registro y fase autenticación, contando con un total de 4 ejecuciones.

Para hacer la medición, se va a utilizar el tiempo de ejecución de las 4 variantes a comprobar, utilizando la clase *time* de Python para capturar el tiempo, junto a la librería matplotlib [18] para sacar las gráficas de las iteraciones, el cálculo de la media, la varianza y la desviación típica se ha hecho de manera manual al terminar las ejecuciones, se muestra en la Imagen 8 el código incluido al final del código de la Relying Party que se encarga de las mediciones, se ha escogido este programa para utilizar la medición, dado que es el programa encargado de acoger el paso 1 de FIDO2 y el encargado de realizar la última tarea de verificación, tanto en registro como en autenticación. Para conseguir una medición acertada, se han hecho un total de 100 ejecuciones seguidas en cada variante, cogiendo los datos en segundos.

```
    print("La ejecución ha tardado: ", tiempo_ejecucion)

    acumulacion_tiempo_ejecucion = acumulacion_tiempo_ejecucion +
tiempo_ejecucion

    count= count+1
    print("count:",count)
    lista_tiempos.append(tiempo_ejecucion)
    if count == 100:
        print("La media de la ejecución es: ")
        media = acumulacion_tiempo_ejecucion / 100
        print(media)

        for i in lista_tiempos:
            acumulacion_tiempo2 = acumulacion_tiempo2 + abs(i - media)
        desvtipica = acumulacion_tiempo2 / 100
        varianza = desvtipica**2

        print("Varianza: ",varianza)
        print("Desviación típica: ",desvtipica)
        plt.plot(lista_x,lista_tiempos,"-y", label="Rendimiento - Re-
gistro - LwC")
        plt.legend(loc="upper left")
        plt.show()

        break
```

Código 3 – Código de medición incluido en RP.py

8.3. Evaluación de la propuesta.

Al sacar las 100 iteraciones de cada variante, podemos obtener los datos incluidos más abajo en la Tabla 4 y las Gráficas del 1 al 4. A simple vista solo mirando la media, podemos ver una mejora notable en los tiempos de ejecución entre las ejecuciones con LwC y sin LwC, mostrando que efectivamente en base a estas mediciones en segundos, el uso de la criptografía ligera reduce los tiempos de ejecución en comparación de algoritmos de criptografía clásica, en el caso de la fase de registro, comparando la media “Con LwC” y “Sin LwC”, la versión con LwC es un 19,86% más rápida, mientras que la misma medición en la fase de autenticación es algo menor la diferencia, con un 14,06%.

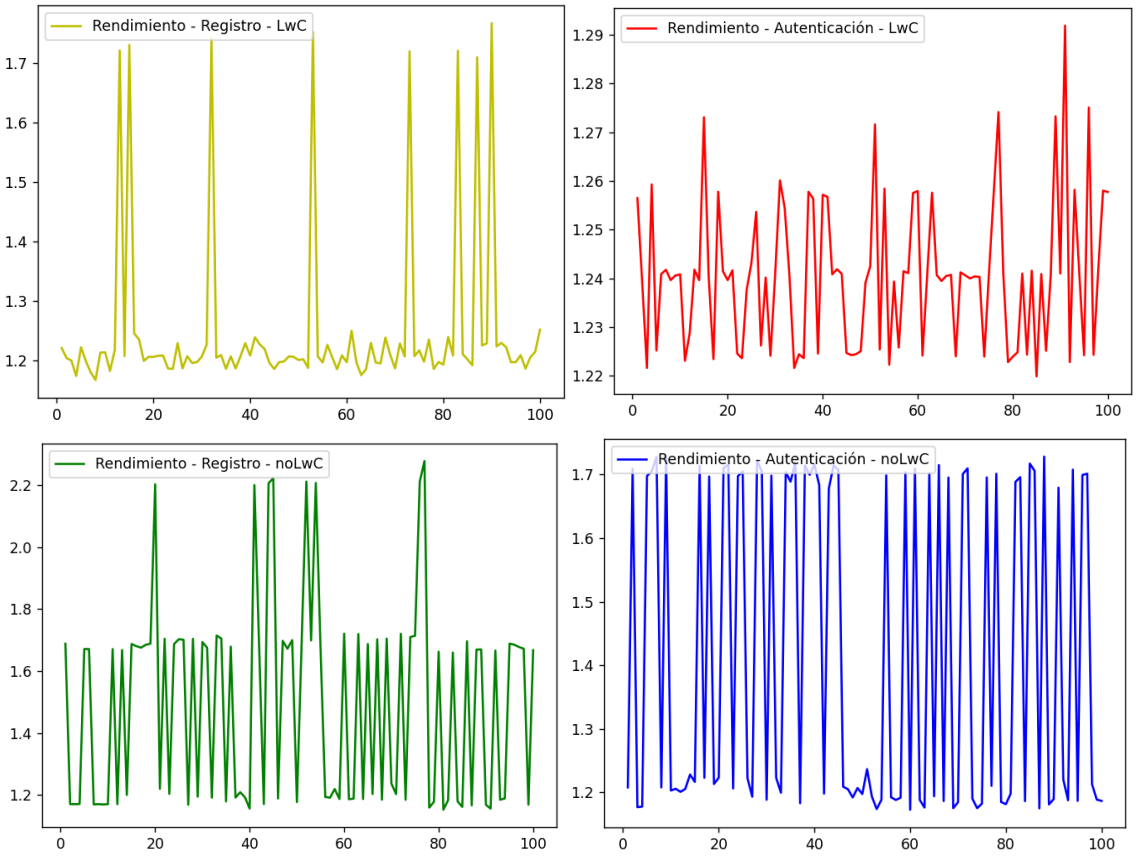
Sin embargo, no solo se trata de velocidad, también podemos observar en base a las gráficas y los valores de varianza y desviación típica, que en los casos “Sin Lwc” las medidas son notablemente más dispersas, mientras que por ejemplo los datos de autenticación con LwC abarcan de 1.22 a 1.29 segundos, los datos de autenticación sin LwC sí es cierto que tienen mínimos mayores, pero también picos mucho mayores, el rango está entre 1.1 (aprox) y 1.7 segundos. Estos picos se repiten también en la fase de registro, tanto con LwC como sin LwC,

aunque los picos en el primer caso alcanzan una máxima de casi 1.8 segundos y en el segundo caso de 2.2 segundos.

Es lógico que las ejecuciones de registro tarden ligeramente más que las de autenticación, dado que cuentan con más operaciones y manejo de datos que la autenticación, siendo la diferencia entre las dos versiones sin LwC de casi una décima de segundo, mientras que la diferencia con LwC es menor, de una centésima de segundo.

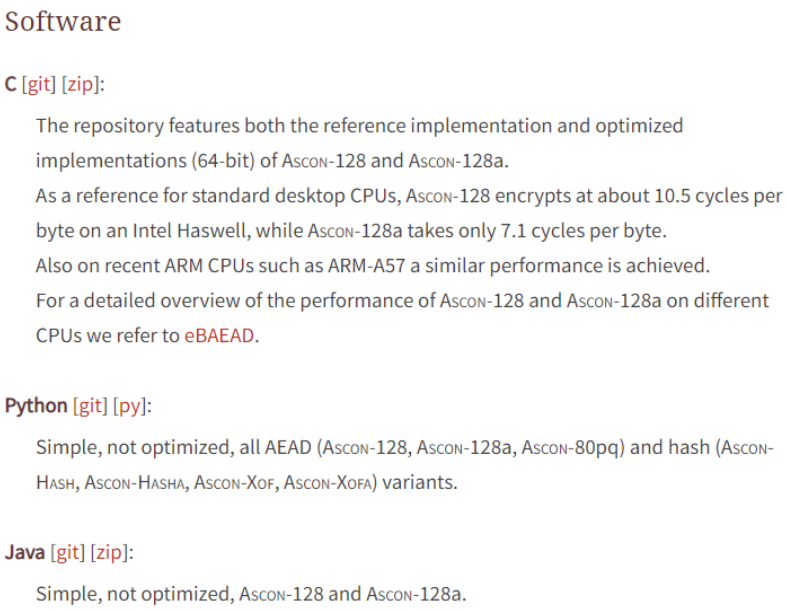
	Con LwC / Registro	Con LwC / Autenticación	Sin LwC / Registro	Sin LwC / Autenticación
Media	1.249553592	1.240542213	1.497778182	1.415081868
Varianza	0.005985754	0.000116304	0.084047056	0.062103047
Desv. Típica	0.077367651	0.010784444	0.289908704	0.249204830

Tabla 4 – Media, Varianza y DT de las ejecuciones.



Gráficas 1 al 4 – Gráficas de las ejecuciones, valores 1 al 100

Sin embargo, estos datos es importante tomarlos con una perspectiva adecuada, es importante mencionar que aun ejecutándose en el mismo entorno FIDO2 simulado, las condiciones de la implementación de los algoritmos son diferentes, en el caso de los algoritmos criptografía ligera, la función Hash ASCON-Hash es una implementación enteramente desarrollada en Python y sin optimizar, se puede observar en la Imagen 6 que viene en la descripción de la página oficial de ASCON. En el caso del algoritmo AEAD (DryGASCON-256), sí está optimizado y con parte del código escrito en C. Pero por parte de SHA-256 y AES-256, ambos se han obtenido de la librería pycryptodome [19] que cuenta con los algoritmos optimizados. De manera que la implementación con criptografía ligera cuenta con una ligera "desventaja" por la implementación de la función hash, por lo que, si la función hash estuviera optimizada, se podría esperar aún mejores resultados para la criptografía ligera.



Software

C [git] [zip]:

The repository features both the reference implementation and optimized implementations (64-bit) of Ascon-128 and Ascon-128a.
As a reference for standard desktop CPUs, Ascon-128 encrypts at about 10.5 cycles per byte on an Intel Haswell, while Ascon-128a takes only 7.1 cycles per byte.
Also on recent ARM CPUs such as ARM-A57 a similar performance is achieved.
For a detailed overview of the performance of Ascon-128 and Ascon-128a on different CPUs we refer to eBAEAD.

Python [git] [py]:

Simple, not optimized, all AEAD (Ascon-128, Ascon-128a, Ascon-80pq) and hash (Ascon-Hash, Ascon-Hasha, Ascon-Xof, Ascon-Xofa) variants.

Java [git] [zip]:

Simple, not optimized, Ascon-128 and Ascon-128a.

Imagen 6 – Descripción de las implementaciones de ASCON [20]

9. Conclusión.

Actualmente la industria se encuentra en un punto de inflexión, hay distintas tecnologías emergentes como el 5G o la Computación Cuántica que pueden ser la base para nuevas aplicaciones y mejoras en el paradigma actual. En concreto con el 5G, se espera que suponga una mejora considerable en las telecomunicaciones, con la aparición del *Internet of Things* habrá todo tipo de dispositivos interconectados constantemente, este fenómeno fomentará una tendencia del sector que es el desarrollo de sistemas empotrados o embebidos, siendo estos dispositivos pensados para una o pocas tareas, cuentan con poca potencia computacional y energética, pero es más barato desarrollarlos a nivel industrial.

Este tipo de dispositivos al tener poca potencia es complicado introducirles algoritmos de criptografía clásicos como AES o RSA, dado que son computacionalmente costosos, por ello, organizaciones como el NIST o ISO/IEC han o están desarrollando procesos de estandarizado y competiciones con los que desarrollar nuevos algoritmos que utilicen poca potencia computacional y poca energía, a esta rama de la criptografía se le conoce como criptografía ligera.

Junto a la criptografía ligera, en la industria de la ciberseguridad hay otras cuestiones que resolver, ataques de ingeniería social como el phishing se encuentran en alza en parte gracias a la situación que ha desembocado la pandemia, fomentando el uso de las telecomunicaciones y el teletrabajo. Las empresas son conscientes y actualmente las empresas y algunas de las instituciones más importantes de la industria buscan crear procesos de autenticación sin contraseñas, que son un problema serio a nivel seguridad, estas empresas e instituciones han formado una organización llamada FIDO Alliance que dispone de varios procesos de autenticación, siendo el más destacable FIDO2, que junta tecnología CTAP2 y WebAuthn.

La propuesta de este proyecto es juntar ambas tecnologías, FIDO2 y criptografía ligera, para proporcionar una alternativa viable de autenticación segura sin contraseñas a dispositivos computacionalmente poco potentes. Después de un exhaustivo estudio de los procesos de registro y autenticación de FIDO2, junto a un estudio sobre requisitos y el estado del estandarizado del NIST sobre criptografía ligera, escogiendo el cifrado ligero ASCON como la mejor alternativa, se ha desarrollado una prueba de concepto, un simulador de FIDO2 con el que comparar la eficiencia del simulador con y sin criptografía ligera, una vez medidos los tiempos de ejecución, hemos podido observar que hay una mejora considerable al utilizar la criptografía ligera.

9.1. Trabajos futuros.

Partiendo de esta premisa, es interesante indagar más en la viabilidad de este tipo de implementación, en un futuro trabajo, en vez de centrarnos en una prueba de concepto, se podría probar ya en un entorno más real y desarrollado como en la idea original del proyecto: incluir criptografía ligera en una security key ya desarrollada que permita cambios en su firmware. También se podría mejorar la propia prueba de concepto, incluyendo una réplica exacta del protocolo PIN/UV y otras funcionalidades de FIDO2, y más importante, si se desarrolla un cifrado asimétrico ligero, un elemento que no he podido encontrar buscando información sobre la criptografía ligera, sería interesante cambiarlo por ECDSA y ver si es aún más eficiente, aunque con cifrado de curva elíptica ya se obtienen unos resultados más que satisfactorios.

10. Anexo – Código fuente de la implementación.

La implementación consta de dos versiones: *SimulacionFIDO2* y *SimulacionFIDO2LwC*. Para no excederme en la longitud del anexo, he incluido la versión LwC de los tres principales programas (*FIDOClient.py*, *authenticator.py* y *RP.py*), pero no la versión sin LwC, en cambio aparecen comentadas las líneas de código modificadas con la alternativa sin LwC, lo que sí aparecen son las dos versiones del archivo *functions.py*.

El código está también disponible en el siguiente repositorio público de GitHub: https://github.com/German7463/FIDO2LwC

10.1. Estructura de carpetas.

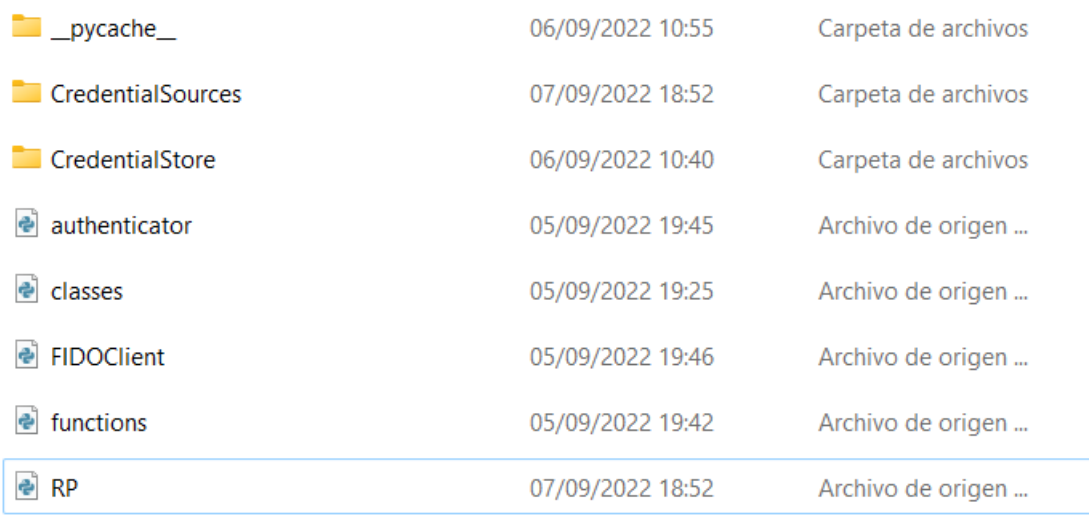


Imagen 7 – Estructura de las carpetas.

10.2. Código fuente.

10.2.1. Classes.py

```
from pydantic import *

class usuario(BaseModel):
    userID: int
    displayName: str

class origin(BaseModel):
    scheme = "HTTPS"
    host: str # dominio o dirección IP
    port: int # puerto
    domain: str # dominio

class pubKeyCredParams(BaseModel):
    type = "PUBLIC_KEY" #Establecido como el tipo de credencial
que se va a generar
    alg = "ECDSA" # Elliptic Curve Digital Signature Algorithm

class createRegistration(BaseModel):
    origin: origin
    rpid: str #RPid es el identificador de la Relying Party, suele
ser el dominio de la página web.
    user: usuario
    challenge: str
    timeout: float
    pubKeyCredParams: pubKeyCredParams

class makeCredential(BaseModel):
    clientDataHash: bytes
    rpid: str
    userEntity: usuario
    credTypesandPubKeyAlgs: pubKeyCredParams

class clientData(BaseModel):
    type: str
    challenge: str
    origin: origin
    user: usuario

class credentialSource(BaseModel):
    credentialID: str
    type = "public-key"
    clavePrivada: bytes
    rpid: str
    userHandle: usuario
    counter: int
```

```
class attestedCredentialData(BaseModel):
    AAGUID= "737935ab-318d-4350-8cf9-3f3110b3b804" ##Identificador
único para el autenticador.
    credentialID: str
    credentialPublicKey: bytes

class authenticatorData(BaseModel):
    rpidHash: str
    signCount: int
    attestedCredData: attestedCredentialData

class attestationObject(BaseModel):
    authenticatorData: authenticatorData
    attestationStatement: bytes #Este atributo en FIDO2 real es
una firma constrastada del autenticador, en mi caso utilizaré el
clientDataHash
                              # firmado con la clave privada como
seguro.

class publicKeyECDH(BaseModel):
    x: int
    y: int
    curve: str

class CredentialStoreData(BaseModel):
    credentialID: str
    rpid: str
    origen: origin
    pubKey: bytes
    user: usuario
    counter: int

### Authentication ###

class CollectFromCredentialStore(BaseModel):
    rpid: str
    origen: origin
    challenge: str
    timeout: float
    user: usuario

class authenticatorGetAssertion(BaseModel):
    rpid: str
    clientDataHash: bytes
    clientData: clientData

class attestedCredentialDataAuthentication(BaseModel):
    AAGUID= "737935ab-318d-4350-8cf9-3f3110b3b804" ##Identificador
único para el autenticador.
    credentialID: str
```

```
class authenticatorDataAuthentication(BaseModel):
    rpidHash: str
    signCount: int
    attestedCredData: attestedCredentialDataAuthentication

class assertionObject(BaseModel):
    authenticatorData: authenticatorDataAuthentication
    clientData: clientData
    signature: bytes
```

10.2.2. FIDOClient.py

```
from pydoc import cli
import classes as c
import functions as f
from tinyec import registry
from tinyec.ec import Point
from ecdsa import VerifyingKey, NIST256p
from pydantic import *
import ascon as asc
from Crypto.Hash import SHA256
HOST = "127.0.0.1"  # The server's hostname or IP address
PORTRP = 7800  # Puerto de comunicación con la RP
PORTAUTH = 8844  # Puerto de comunicación con el authenticator


print("////////// Simulador FIDO2 - Python. //////////")
print("Introduzca el número de una opción para seleccionarla: ")
print("1. Registro de credenciales.")
print("2. Inicio de sesión.")
print("Cualquier otra opción cerrará el programa.")
print("\n")

opcion = input()

f.enviar_dato(bytes(opcion,encoding="utf-8"), HOST, PORTRP-3)
f.enviar_dato(bytes(opcion,encoding="utf-8"), HOST, PORTAUTH-1)

print("\n")
print("************************************************")
print("***************    FIDOClient    ***************")
print("************************************************")
print("\n")

if opcion == '1':

    print("*************** Fase de Registro ***************")

    ############# Fase de Registro --- Parte 0: Solicitud al
```

```
servidor y establecimiento del nombre de usuario y PIN
#############

    print("Fase de Registro --- Parte 0: Solicitud al servidor y
establecimiento del nombre de usuario y PIN\n")

    print("Introduzca el nombre de usuario que desea utilizar: ")
    username = input()

    f.enviar_dato(username.encode("utf-8"),HOST,PORTRP-1)

    print("\n Se ha envíado el nombre de usuario")

    ############# Fase de Registro --- Parte 1: Creación del
ClientDataHash ###############

    print("Fase de Registro --- Parte 1: Creación del
ClientDataHash")

    paquetePaso1 = f.recibirPaquete(HOST, PORTRP, "rp")

    clientData1 = c.clientData(type="webauthn.create",
challenge=paquetePaso1['challenge'],
origin=paquetePaso1['origin'],
    user= paquetePaso1['user'])
    clientDataJSON1 = clientData1.json()
    clientDataHash1 =
asc.ascon_hash(clientDataJSON1.encode(encoding='utf-8')).hex()
    # Versión noLwC: clientDataHash1 =
SHA256.new(clientDataJSON1.encode(encoding='utf-8')).hexdigest()

    ############# Fase de Registro --- Paso 2: creación y envío
del authenticatorMakeCredential ###############

    print("Fase de Registro --- Paso 2: creación y envío del
authenticatorMakeCredential\n")

    ### ECDH Key Exchange con el Authenticator ###

    #1. Generamos las claves pública y privada.

    fcPrivKey = f.generarClavePrivECC()
    fcPubKey = f.generarClavePubECC(fcPrivKey)

    fcPubKeyob = c.publicKeyECDH(x = fcPubKey.x, y = fcPubKey.y,
curve = 'brainpoolP256r1')

    #2. Intercambiamos las claves pública y privada con el
Authenticator

    f.enviarPaquete(fcPubKeyob,HOST,8887,"authtor")
```

```
    authenticatorPublicKeyob =
f.recibirPaquete(HOST,8888,"authtor")

    authenticatorPublicKey =
Point(registry.get_curve(authenticatorPublicKeyob['curve']),
authenticatorPublicKeyob['x'], authenticatorPublicKeyob['y'])

    #3. Generamos la clave compartida.

    sharedKey = f.ecc_point_to_256_bit_key(authenticatorPublicKey
* fcPrivKey)

    ############################

    mkCredential1 =
c.makeCredential(clientDataHash=clientDataHash1,
                                    rpid=paquetePaso1['rpid'],
userEntity=paquetePaso1['user'],
credTypesandPubKeyAlgs=paquetePaso1['pubKeyCredParams'])

    f.enviar_datos_dryGascon(mkCredential1, sharedKey, HOST,
PORTAUTH+9)
    # Versión noLwC: f.enviar_datos_AES(mkCredential1, sharedKey,
HOST, PORTAUTH+9)

    ############ Fase de Registro --- Paso 3: recibir y comprobar
los datos del autenticador ############

    print("Fase de Registro --- Paso 3: recibir y comprobar los
datos del autenticador \n")
    paquetePaso3 = f.recibir_datos_dryGascon(HOST, PORTAUTH+1,
sharedKey)
    # Versión noLwC: paquetePaso3 = f.recibir_datos_AES(HOST,
PORTAUTH+1, sharedKey)

    Publickey =
paquetePaso3['authenticatorData']['attestedCredData']['credentialP
ublicKey']
    h = bytes(clientDataHash1, encoding="utf-8")

    firma = f.recibir_dato(HOST, PORTAUTH+2)

    attestationObject = c.attestationObject(authenticatorData=
paquetePaso3['authenticatorData'], attestationStatement=
paquetePaso3['attestationStatement'])

    if f.verificar_ECDSA(h, firma,
VerifyingKey.from_pem(Publickey)):
        print("Se verificado la firma obtenida del autenticador
satisfactoriamente")
        okFirma = True
```

```
    else:
        print("Ha fallado la verificación de la firma, se procede
a cerrar la sesión de registro.")
        okFirma = False

    ############ Fase de Registro --- Paso 4: envíar
ClientDataJSON y attestationObject a la Relying Party ############

    print("Fase de Registro --- Paso 4: envíar ClientDataJSON y
attestationObject a la Relying Party\n")

    if okFirma:
        print("\n Fase de Registro --- Paso 4: envíar
ClientDataJSON y attestationObject a la Relying Party \n")

        f.enviarPaquete(clientData1, HOST, PORTRP+1, "rp")
        f.enviarPaquete(attestationObject, HOST, PORTRP+2, "rp")

    else:
        print("Fin de programa.")

elif opcion == '2':
    print("**************** Fase de autenticación
****************")

    ############ Fase de autenticación --- Paso 0: Solicitud al
servidor, envío del nombre de usuario. ############

    print("Fase de autenticación --- Paso 0: Solicitud al
servidor, envío del nombre de usuario. \n")

    print("Introducza el nombre de usuario que desea utilizar: ")
    username = input()

    f.enviar_dato(username.encode("utf-8"),HOST,PORTRP-1)

    print("\n Se ha envíado el nombre de usuario")

    ############ Fase de autenticación --- Parte 1: Recibir
Objeto 'CollectFromCredentialStore', crear ClientDataHash y envíar
'GetAssertion'###############

    print("Fase de autenticación --- Parte 1: Recibir Objeto
'CollectFromCredentialStore', crear ClientDataHash y envíar
'GetAssertion'")

    dato = f.recibirPaquete(HOST,PORTRP,"rp")
    print(dato)

    clientData1 = c.clientData(type="webauthn.create",
challenge=dato['challenge'], origin=dato['origen'], user=
```

```
dato['user'])
    clientDataJSON1 = clientData1.json()
    clientDataHash1 =
asc.ascon_hash(clientDataJSON1.encode(encoding='utf-8')).hex()
    # Versión noLwC: clientDataHash1 =
SHA256.new(clientDataJSON1.encode(encoding='utf-8')).hexdigest()

    ############# Fase de autenticación --- Paso 2: Envíar
'GetAssertion'###############

    print("Fase de autenticación --- Paso 2: Envíar
'GetAssertion'")
    getAssertion = c.authenticatorGetAssertion(rpid=
dato['rpid'],clientDataHash=clientDataHash1, clientData=
clientData1)

    ### ECDH Key Exchange con el Authenticator ###

    #1. Generamos las claves pública y privada.

    fcPrivKey = f.generarClavePrivECC()
    fcPubKey = f.generarClavePubECC(fcPrivKey)

    fcPubKeyob = c.publicKeyECDH(x = fcPubKey.x, y = fcPubKey.y,
curve = 'brainpoolP256r1')

    #2. Intercambiamos las claves pública y privada con el
Authenticator

    f.enviarPaquete(fcPubKeyob,HOST,8887,"authtor")
    authenticatorPublicKeyob =
f.recibirPaquete(HOST,8888,"authtor")

    authenticatorPublicKey =
Point(registry.get_curve(authenticatorPublicKeyob['curve']),
authenticatorPublicKeyob['x'], authenticatorPublicKeyob['y'])

    #3. Generamos la clave compartida.

    sharedKey = f.ecc_point_to_256_bit_key(authenticatorPublicKey
* fcPrivKey)

    ############################

    f.enviar_datos_dryGascon(getAssertion,sharedKey,HOST,PORTAUTH)
    # Versión noLwC:
f.enviar_datos_AES(getAssertion,sharedKey,HOST,PORTAUTH)

    ############# Fase de autenticación --- Paso 3: Recibo de
información del autenticador y redirección de esta a la Relying
Party###############
```

```
    print("Fase de autenticación --- Paso 3: Recibo de información
del autenticador y redirección de esta a la Relying Party\n")

    assertionObject1 =
f.recibir_datos_dryGascon(HOST,PORTAUTH+1,sharedKey)
    # Versión noLwC:  assertionObject1 =
f.recibir_datos_AES(HOST,PORTAUTH+1,sharedKey)

    firma = f.recibir_dato(HOST,PORTAUTH+2)

    assertObj = c.assertionObject(authenticatorData=
assertionObject1['authenticatorData'], clientData=
assertionObject1['clientData'], signature=
assertionObject1['signature'])

    print(assertObj)

    f.enviarPaquete(assertObj,HOST,PORTRP+1,"rp")
    f.enviar_dato(firma,HOST,PORTRP+2)

    print("\nFin interacción con el Cliente FIDO, solo queda la
verificación de la Relying Party")


else:
    print("Fin de programa.")
```

10.2.3. RP.py

```
from random import randint
import classes as c
import functions as f
from pydantic import *
import json
from ecdsa import VerifyingKey, NIST256p
import ascon as asc
# from Crypto.Hash import SHA256

HOST = "127.0.0.1"  # The server's hostname or IP address
PORT = 7800   # Puerto de comunicación con el Cliente FIDO

print("\n")
print("*****************************************************")
print("***************    Relying Party    ***************")
print("*****************************************************")
print("\n")

opcion = f.recibir_dato(HOST,PORT-3).decode('ascii')
```

```
print("Introduzca el dominio del servidor: ")
domain = "uah.es"

if opcion == '1':

    ################### Fase de Registro ###################
    print("**************** Fase de Registro ****************")

    ############# Fase de Registro --- Parte 0: Solicitud de
registro, recibo del usuario. #############
    print("Fase de Registro --- Parte 0: Solicitud de registro,
recibo del usuario. \n")

    username = f.recibir_dato(HOST, PORT-1)

    origen1 = c.origin(host="80.26.225.82", domain=str("www." +
domain), port=7800)
    usuario1 = c.usuario(userID=randint(1000000,9999999),
displayName=username)
    pubkeycp1 = c.pubKeyCredParams()

    paquetePaso1 = c.createRegistration(origin=origen1,
rpid=domain, user=usuario1, challenge="abcdefg", timeout=6.5,
pubKeyCredParams=pubkeycp1)

    ############# Fase de Registro --- Paso 1: Envío de la
información de la RP ###############
    print("Fase de Registro --- Paso 1: Envío de la información de
la Relying Party.\n")

    f.enviarPaquete(paquetePaso1, HOST, PORT, "cf")
    print("Se ha envíado la información 'credential.create'.\n")

    ############# Fase de Registro --- Paso 4: Obtención del
attestationObject y del ClientDataJSON ###############

    print("Fase de Registro --- Paso 4: Obtención del
attestationObject y del ClientDataJSON\n")

    clientDataJSON = f.recibirPaquete(HOST, PORT+1, "cf")
    attestationObject = f.recibirPaquete(HOST, PORT+2, "cf")

    rpID = asc.ascon_hash(domain.encode(encoding='utf-8')).hex()
    # Versión noLwC: rpID =
SHA256.new(domain.encode(encoding='utf-8')).hexdigest()

    attestationObject1 =
c.attestationObject(authenticatorData=attestationObject['authentic
atorData'],
attestationStatement=attestationObject['attestationStatement'])
```

```
    collect = c.CredentialStoreData(credentialID=
attestationObject1.authenticatorData.attestedCredData.credentialID
,rpid=domain, origen=origen1, pubKey=
attestationObject1.authenticatorData.attestedCredData.credentialPu
blicKey, user= usuario1, counter=
attestationObject1.authenticatorData.signCount)

    cSDjson = collect.json()
    ruta = '.\CredentialStore\\' +
attestationObject1.authenticatorData.attestedCredData.credentialID
+ '.json'

    # Almacenamos credentialSource para futuras autenticaciones.
    with open(ruta, 'w') as json_file:
        json.dump(cSDjson, json_file)

    print("Se ha almacenado la siguiente información en el
servidor:\n ",cSDjson)

    print("\n Se ha finalizado el proceso de registro.")


elif opcion == '2':
    print("**************** Fase de autenticación
****************")

    ############# Fase de autenticación --- Paso 1: Comprobación
del nombre de usuario, envío de CollectFromCredentialStore
#############

    print("Fase de autenticación --- Paso 1: Comprobación del
nombre de usuario y recuperación de los datos de registro, envío
de CollectFromCredentialStore \n")

    username = f.recibir_dato(HOST, PORT-1)

    origen1 = c.origin(host="80.26.225.82", domain=str("www." +
domain), port=7800)
    usuario1 = c.usuario(userID=randint(1000000,9999999),
displayName=username)
    pubkeycp1 = c.pubKeyCredParams()

    challenge = "abcdefg"

    collectFromCredStoreP1 =
c.CollectFromCredentialStore(origen=origen1, rpid=domain,
user=usuario1, challenge=challenge, timeout=30.0)

    datos = f.buscar_Credenciales(username.decode("utf-8"))

    if datos == 0:
```

```
        print("No se ha encontrado el nombre de usuario nombrado,
Fin de programa.")
    else:
        print("Datos del usuario:")
        print(datos)
        print("\nSe envían los datos necesarios al Cliente FIDO.")

        f.enviarPaquete(collectFromCredStoreP1,HOST,PORT,"fc")


    ############# Fase de autenticación --- Paso 4: Comprobación
de los datos enviados por el autenticador #############

    print("Fase de autenticación --- Paso 4: Comprobación de los
datos enviados por el autenticador \n")

    assertObj = f.recibirPaquete(HOST,PORT+1,"fc")
    firma = f.recibir_dato(HOST,PORT+2)

    clientData1 = c.clientData(type=
assertObj['clientData']['type'], challenge=
assertObj['clientData']['challenge'], origin=
assertObj['clientData']['origin'], user=
assertObj['clientData']['user'])

    h =
bytes(asc.ascon_hash(clientData1.json().encode(encoding='utf-
8')).hex(), encoding="utf-8")

    try:

        if f.verificar_ECDSA(h, firma,
VerifyingKey.from_pem(datos['pubKey'])):
            print("Se ha verificado la firma obtenida del
autenticador satisfactoriamente.")

            if clientData1.challenge == challenge:
                print("Se ha verificado el challenge.")

                if assertObj['authenticatorData']['signCount'] ==
datos['counter']+1:
                    print("Verificación del counter
satisfactoria.")
                    print("Se da permiso al usuario a los recursos
del servidor y se actualiza el counter registrado en
CredentialStore.")

                    credStoreData =
c.CredentialStoreData(credentialID = datos['credentialID'], rpid=
datos['rpid'], origen= datos['origen'], pubKey= datos['pubKey'],
user= datos['user'], counter= datos['counter']+1)
```

```
                    ruta = '.\CredentialStore\\' +
datos['credentialID'] + '.json'
                    print("\n")

                    with open(ruta, 'w') as json_file: #
Actualizamos credentialSource
                        json.dump(credStoreData.json(), json_file)

                else:
                    print("Ha fallado la verificación del counter,
acceso denegado a los recursos del servidor.")
            else:
                print("Ha fallado la verificación del challenge,
acceso denegado a los recursos del servidor.")
        else:
            print("Ha fallado la verificación de la firma, acceso
denegado a los recursos del servidor.")
    except:
        print("Ha habido un error o ha fallado alguna verificación
del servidor.")

else:
    print("Fin de programa.")
```

10.2.4. authenticator.py

```
import uuid
import classes as c
import functions as f
from tinyec import registry
from tinyec.ec import Point
from ecdsa import SigningKey
import json
import ascon as asc
#from Crypto.Hash import SHA256

print("\n")
print("*****************************************************")
print("***************    Authenticator    ***************")
print("*****************************************************")
print("\n")

HOST = "127.0.0.1"
PORT = 8844
credentialID = uuid.uuid1()

opcion = f.recibir_dato(HOST,PORT-1).decode('ascii')

print(opcion)
```

```
if opcion == '1':

    print("**************** Fase de Registro ***************\n")

    ### ECDH Key Exchange con el Cliente FIDO ###

    #1. Generamos las claves pública y privada.

    authPrivKey = f.generarClavePrivECC()
    authPubKey = f.generarClavePubECC(authPrivKey)

    authPubKeyob = c.publicKeyECDH(x = authPubKey.x,y =
authPubKey.y, curve = 'brainpoolP256r1')

    #2. Intercambiamos las claves públicas con el Cliente FIDO

    FIDOClientPublicKeyob = f.recibirPaquete(HOST,8887,'fc')
    f.enviarPaquete(authPubKeyob,HOST,8888,'fc')

    FIDOClientPublicKey =
Point(registry.get_curve(FIDOClientPublicKeyob['curve']),
FIDOClientPublicKeyob['x'], FIDOClientPublicKeyob['y'])

    #3. Generamos la clave compartida.

    sharedKey = f.ecc_point_to_256_bit_key(FIDOClientPublicKey *
authPrivKey)

    ############# Fase de Registro ---Paso 2: Recibo de
información del FC, creación del par de claves pública y privada,
junto al UserHandle ###############

    print("Fase de Registro ---Paso 2: Recibo de información del
FC, creación del par de claves pública y privada, junto al
UserHandle\n")

    paquetePaso2 =
f.recibir_datos_dryGascon(HOST,PORT+9,sharedKey)
    # Versión noLwC: paquetePaso2 =
f.recibir_datos_AES(HOST,PORT+9,sharedKey)

    userHandle1 = paquetePaso2['userEntity']

    print("Generando clave pública y privada - Curva elíptica")
    clavePriv = f.generarClavePrivECDSA()
    clavePub = f.generarClavePubECDSA(clavePriv)

    credentialSource1 = c.credentialSource(credentialID =
str(credentialID), clavePrivada= clavePriv.to_pem(), rpid=
paquetePaso2['rpid'],
    userHandle= userHandle1, counter = 0)
```

```
    cSjson = credentialSource1.json()

    print("tipo del json: " , type(cSjson))
    print("\nSe ha creado el archivo: ")
    ruta = '.\CredentialSources\\' + str(credentialID) + '.json'
    print("\n")

    with open(ruta, 'w') as json_file: # Almacenamos
credentialSource para futuras autenticaciones.
        json.dump(cSjson, json_file)

    ############## Fase de Registro --- Paso 3: creación y envío
del clientDataHash y del attestationObject al Cliente FIDO
###############

    print("Fase de Registro --- Paso 3: creación y envío del
clientDataHash y del attestationObject al Cliente FIDO\n")

    rpIDHash1 =
asc.ascon_hash(paquetePaso2['rpid'].encode(encoding='utf-
8')).hex()
    # Versión noLwC: rpIDHash1 =
hashlib.sha256(paquetePaso2['rpid'].encode('utf-8')).hexdigest()

    attestedCredData1 = c.attestedCredentialData(credentialID =
str(credentialID), credentialPublicKey = clavePub.to_pem())
    authData = c.authenticatorData(rpidHash= rpIDHash1,
attestedCredData= attestedCredData1, signCount= 0)

    ### Firmar clientDataHash ###

    clientDataHashsign =
f.firmar_ECDSA(paquetePaso2['clientDataHash'].encode('utf-
8'),clavePriv)

    attestationObject1 = c.attestationObject(authenticatorData=
authData, attestationStatement= str(clientDataHashsign))


f.enviar_datos_dryGascon(attestationObject1,sharedKey,HOST,PORT+1)
    # Versión noLwC:
f.enviar_datos_AES(attestationObject1,sharedKey,HOST,PORT+1)

    f.enviar_dato(clientDataHashsign,HOST,PORT+2)

elif opcion == '2':
    print("*************** Fase de autenticación
***************")

    ### ECDH Key Exchange con el Cliente FIDO ###
```

```
    #1. Generamos las claves pública y privada.

    authPrivKey = f.generarClavePrivECC()
    authPubKey = f.generarClavePubECC(authPrivKey)

    authPubKeyob = c.publicKeyECDH(x = authPubKey.x,y =
authPubKey.y, curve = 'brainpoolP256r1')

    #2. Intercambiamos las claves públicas con el Cliente FIDO

    FIDOClientPublicKeyob = f.recibirPaquete(HOST,8887,'fc')
    f.enviarPaquete(authPubKeyob,HOST,8888,'fc')

    FIDOClientPublicKey =
Point(registry.get_curve(FIDOClientPublicKeyob['curve']),
        FIDOClientPublicKeyob['x'], FIDOClientPublicKeyob['y'])

    #3. Generamos la clave compartida.

    sharedKey = f.ecc_point_to_256_bit_key(FIDOClientPublicKey *
authPrivKey)

    ############# Fase de autenticación --- Paso 2: Recibo de
información del FC, obtención del CredentialSource, aumento del
counter y realizar la firma###############

    print("Fase de autenticación --- Paso 2: Recibo de información
del FC, obtención del CredentialSource, aumento del counter y
realizar la firma\n")

    getAssertionP2 =
f.recibir_datos_dryGascon(HOST,PORT,sharedKey)
    # Versión noLwC:
f.enviar_datos_AES(attestationObject1,sharedKey,HOST,PORT+1)

    credentialSource1 =
f.buscar_Credenciales_authenticator(getAssertionP2['clientData']['
user']['displayName'])

    print("Información encontrada del usuario: ",
credentialSource1)

    credSource = c.credentialSource(credentialID =
credentialSource1['credentialID'], clavePrivada=
credentialSource1['clavePrivada'],
    rpid= credentialSource1['rpid'], userHandle=
credentialSource1['userHandle'], counter=
credentialSource1['counter'] + 1)

    ruta = '.\CredentialSources\\' +
credentialSource1['credentialID'] + '.json'
```

```
    print("\n")

    with open(ruta, 'w') as json_file: # Actualizamos
credentialSource
        json.dump(credSource.json(), json_file)

    clientDataHashsign =
f.firmar_ECDSA(getAssertionP2['clientDataHash'].encode('utf-
8'),SigningKey.from_pem(credSource.clavePrivada))
    rpIDHash1 =
asc.ascon_hash(credSource.rpid.encode(encoding='utf-8')).hex()
    # Versión noLwC: rpIDHash1 =
SHA256.new(credSource.rpid.encode('utf-8')).hexdigest()

    ############# Fase de autenticación --- Paso 3: Envío de la
respuesta por parte del autenticador al Cliente
FIDO###############

    print("Fase de autenticación --- Paso 3: Envío de la respuesta
por parte del autenticador al Cliente FIDO\n")

    attestedCredData1 =
c.attestedCredentialDataAuthentication(credentialID =
credSource.credentialID)

    authData = c.authenticatorDataAuthentication(rpidHash=
rpIDHash1, signCount= credSource.counter, attestedCredData=
attestedCredData1, userHandle= credSource.userHandle)

    assertionObject1 = c.assertionObject(authenticatorData=
authData, clientData= getAssertionP2['clientData'], signature=
str(clientDataHashsign))


f.enviar_datos_dryGascon(assertionObject1,sharedKey,HOST,PORT+1)
    # Versión noLwC:
f.enviar_datos_AES(assertionObject1,sharedKey,HOST,PORT+1)

    f.enviar_dato(clientDataHashsign,HOST,PORT+2)

    print("\nDatos enviados, se ha terminado la interacción con el
autenticador.")

else:
    print("Fin de programa.")
```

10.2.5. Functions.py - noLwC

```
import json
import socket, hashlib, secrets
from ecdsa import SigningKey, NIST256p
from ecdsa.util import sigdecode_der, sigencode_der
import time
from tinyec import registry
import secrets
from drysponge.drygascon import DryGascon
import random
import os

### funciones útiles ###

def encodePaquete(paquete1):
    return bytes(paquete1.json(),encoding="utf-8")

def decodePaquete(paquete1):
    return json.loads(paquete1.decode('utf-8'))

def recibirPaquete(host,port,actororigen):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.bind((host,port))
        print("Creación del socket")
        sckt.listen()
        conn, addr = sckt.accept()
        if actororigen == "rp":
            print("Conexión establecida con la Relying Party.")
        elif actororigen == "cf":
            print("Conexión establecida con el Cliente FIDO.")
        elif actororigen == "authtor":
            print("Conexión establecida con el authenticator.")
        else:
            print("Conexión establecida con un actor
desconocido.")
        with conn:
            print(f"Conexión con: {addr}")
            while True:
                paquete = conn.recv(1024)
                print("Se ha recibido el paquete")
                break
            conn.close()
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.")
        print(e)
```

```
        sckt.close()

    return decodePaquete(paquete)

def enviarPaquete(paquete,host,port,actordestino):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.connect((host, port))
        if actordestino == "rp":
            print("Conexión establecida con la Relying Party.")
        elif actordestino == "cf":
            print("Conexión establecida con el Cliente FIDO.")
        elif actordestino == "authtor":
            print("Conexión establecida con el authenticator.")
        else:
            print("Conexión establecida con un actor
desconocido.")
        sckt.sendall(encodePaquete(paquete))
        print("Se ha enviado correctamente el paquete.")
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.")
        print(e)
        sckt.close()

    return 0

def enviar_dato(data,host,port):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.connect((host, port))
        print("Conexión establecida.")
        sckt.sendall(data)
        print("Se ha enviado correctamente el dato.")
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.")
        print(e)
        sckt.close()

    return 0

def recibir_dato(host,port):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.bind((host,port))
```

```
        print("Creación del socket")
        sckt.listen()
        conn, addr = sckt.accept()
        print("Conexión establecida.")
        with conn:
            print(f"Conexión con: {addr}")
            while True:
                data = conn.recv(1024)
                print("Se ha recibido el dato.")
                break
        sckt.close()

    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.\n")
        print(e)
        sckt.close()

    return data

def enviar_datos_dryGascon(plaintext,key,host,port):
    try:
        ciphertext, nonce  =
encrypt_DryGASCON(encodePaquete(plaintext),key)

        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.connect((host, port))
        print("Envío cifrado con dryGASCON - Conexión
establecida.\n")
        time.sleep(0.01)
        sckt.sendall(ciphertext)
        print("Se ha enviado el texto cifrado.")
        time.sleep(0.01)
        sckt.sendall(nonce)
        print("Se ha enviado el nonce.")
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.\n")
        print(e)
        sckt.close()

    return 0

def recibir_datos_dryGascon(host,port,key):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.bind((host,port))
        print("Creación del socket")
```

```
        sckt.listen()
        conn, addr = sckt.accept()
        print("Recibo de información cifrado con dryGASCON -
Conexión establecida.\n")
        with conn:
            print(f"Conexión con: {addr}")
            while True:
                ciphertext = conn.recv(2048)
                print("Se ha recibido el texto cifrado.")
                nonce = conn.recv(2048)
                print("Se ha recibido el nonce.")
                break
        sckt.close()

        plaintext = decrypt_DryGASCON(ciphertext, nonce, key)

    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.\n")
        print(e)
        sckt.close()

    return decodePaquete(plaintext)

def buscar_Credenciales(nombreusuario):
    credentialStoreList = os.listdir(".//CredentialStore")
    flag = False
    for i in credentialStoreList:
        ruta = '.\CredentialStore\\' + i
        with open(ruta) as f:
            data = json.loads(json.loads(f.read()))
            if data['user']['displayName'].strip() ==
nombreusuario.strip():
                print("Se ha encontrado los datos del usuario
",nombreusuario)
                flag = True
                break
    if flag:
        return data
    else:
        return 0

def buscar_Credenciales_authenticator(nombreusuario):
    credentialSourceList = os.listdir(".//CredentialSources")
    flag = False
    for i in credentialSourceList:
        ruta = '.\CredentialSources\\' + i
        with open(ruta) as f:
            data = json.loads(json.loads(f.read()))
            print(data['userHandle']['displayName'])
            print(type(data['userHandle']['displayName']))
```

```
            print(nombreusuario)
            if data['userHandle']['displayName'].strip() ==
nombreusuario.strip():
                print("Se ha encontrado los datos del usuario
",nombreusuario)
                flag = True
                break
    if flag:
        return data
    else:
        return 0


### funciones de cifrado/descifrado y generación de claves ###

def comprimir_clavePubECC(pubKey):
    return hex(pubKey.x) + hex(pubKey.y % 2)[2: ]

curve = registry.get_curve('brainpoolP256r1')

def generarClavePrivECC():
    return secrets.randbelow(curve.field.n)

def generarClavePubECC(clavePriv):
    return clavePriv * curve.g

def ecc_point_to_256_bit_key(point):
    sha = hashlib.sha256(int.to_bytes(point.x, 32, 'big'))
    sha.update(int.to_bytes(point.y, 32, 'big'))
    return sha.digest()

def generarClavePrivECDSA():
    return SigningKey.generate(curve=NIST256p)

def generarClavePubECDSA(clavePriv):
    return clavePriv.verifying_key

def firmar_ECDSA(plaintext, llavePriv):
    return llavePriv.sign(plaintext,sigencode= sigencode_der)

def verificar_ECDSA(plaintext, firma, llavePub):
    return llavePub.verify(firma,plaintext,sigdecode=
sigdecode_der)

#Python-oauth2/oauth2/__init__.py
https://github.com/joestump/python-
oauth2/blob/81326a07d1936838d844690b468660452aafdea9/oauth2/__init
__.py#L165
def generate_nonce(length):
    """Generate pseudorandom number."""
    return ''.join([str(random.randint(0, 9)) for i in
```

```
range(length)])

def encrypt_DryGASCON(msg, secretKey):
    nonce = bytes(generate_nonce(16), encoding='utf-8')
    dryGasconCipher = DryGascon.DryGascon256().instance()
    ciphertext = dryGasconCipher.encrypt(secretKey,nonce,msg)
    return (ciphertext,nonce)

def decrypt_DryGASCON(ciphertext, nonce, secretKey):
    dryGasconCipher = DryGascon.DryGascon256().instance()
    plaintext =
dryGasconCipher.decrypt(secretKey,nonce,ciphertext)
    return plaintext
```

10.2.6. Functions.py - LwC

```
import json
import socket, hashlib, secrets
from ecdsa import SigningKey, NIST256p
from ecdsa.util import sigdecode_der, sigencode_der
import time
from tinyec import registry
import secrets
from drysponge.drygascon import DryGascon
import random
import os

### funciones útiles ###

def encodePaquete(paquete1):
    return bytes(paquete1.json(),encoding="utf-8")

def decodePaquete(paquete1):
    return json.loads(paquete1.decode('utf-8'))

def recibirPaquete(host,port,actororigen):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.bind((host,port))
        print("Creación del socket")
        sckt.listen()
        conn, addr = sckt.accept()
        if actororigen == "rp":
            print("Conexión establecida con la Relying Party.")
        elif actororigen == "cf":
            print("Conexión establecida con el Cliente FIDO.")
        elif actororigen == "authtor":
```

```
            print("Conexión establecida con el authenticator.")
        else:
            print("Conexión establecida con un actor
desconocido.")
        with conn:
            print(f"Conexión con: {addr}")
            while True:
                paquete = conn.recv(1024)
                print("Se ha recibido el paquete")
                break
            conn.close()
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.")
        print(e)
        sckt.close()

    return decodePaquete(paquete)

def enviarPaquete(paquete,host,port,actordestino):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.connect((host, port))
        if actordestino == "rp":
            print("Conexión establecida con la Relying Party.")
        elif actordestino == "cf":
            print("Conexión establecida con el Cliente FIDO.")
        elif actordestino == "authtor":
            print("Conexión establecida con el authenticator.")
        else:
            print("Conexión establecida con un actor
desconocido.")
        sckt.sendall(encodePaquete(paquete))
        print("Se ha enviado correctamente el paquete.")
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.")
        print(e)
        sckt.close()

    return 0

def enviar_dato(data,host,port):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.connect((host, port))
        print("Conexión establecida.")
```

```
        sckt.sendall(data)
        print("Se ha enviado correctamente el dato.")
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.")
        print(e)
        sckt.close()

    return 0

def recibir_dato(host,port):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.bind((host,port))
        print("Creación del socket")
        sckt.listen()
        conn, addr = sckt.accept()
        print("Conexión establecida.")
        with conn:
            print(f"Conexión con: {addr}")
            while True:
                data = conn.recv(1024)
                print("Se ha recibido el dato.")
                break
        sckt.close()

    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.\n")
        print(e)
        sckt.close()

    return data

def enviar_datos_dryGascon(plaintext,key,host,port):
    try:
        ciphertext, nonce  =
encrypt_DryGASCON(encodePaquete(plaintext),key)

        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.connect((host, port))
        print("Envío cifrado con dryGASCON - Conexión
establecida.\n")
        time.sleep(0.01)
        sckt.sendall(ciphertext)
        print("Se ha enviado el texto cifrado.")
        time.sleep(0.01)
        sckt.sendall(nonce)
```

```
        print("Se ha enviado el nonce.")
        sckt.close()
    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.\n")
        print(e)
        sckt.close()

    return 0

def recibir_datos_dryGascon(host,port,key):
    try:
        sckt = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sckt.settimeout(30.0)
        sckt.bind((host,port))
        print("Creación del socket")
        sckt.listen()
        conn, addr = sckt.accept()
        print("Recibo de información cifrado con dryGASCON -
Conexión establecida.\n")
        with conn:
            print(f"Conexión con: {addr}")
            while True:
                ciphertext = conn.recv(2048)
                print("Se ha recibido el texto cifrado.")
                nonce = conn.recv(2048)
                print("Se ha recibido el nonce.")
                break
        sckt.close()

        plaintext = decrypt_DryGASCON(ciphertext, nonce, key)

    except Exception as e:
        print("Se ha producido un error en la obtención de los
datos, se procede a cerrar la conexión.\n")
        print(e)
        sckt.close()

    return decodePaquete(plaintext)

def buscar_Credenciales(nombreusuario):
    credentialStoreList = os.listdir(".//CredentialStore")
    flag = False
    for i in credentialStoreList:
        ruta = '.\CredentialStore\\' + i
        with open(ruta) as f:
            data = json.loads(json.loads(f.read()))
            if data['user']['displayName'].strip() ==
nombreusuario.strip():
                print("Se ha encontrado los datos del usuario
",nombreusuario)
```

```
                flag = True
                break
    if flag:
        return data
    else:
        return 0

def buscar_Credenciales_authenticator(nombreusuario):
    credentialSourceList = os.listdir(".//CredentialSources")
    flag = False
    for i in credentialSourceList:
        ruta = '.\CredentialSources\\' + i
        with open(ruta) as f:
            data = json.loads(json.loads(f.read()))
            print(data['userHandle']['displayName'])
            print(type(data['userHandle']['displayName']))
            print(nombreusuario)
            if data['userHandle']['displayName'].strip() ==
nombreusuario.strip():
                print("Se ha encontrado los datos del usuario
",nombreusuario)
                flag = True
                break
    if flag:
        return data
    else:
        return 0


### funciones de cifrado/descifrado y generación de claves ###

def comprimir_clavePubECC(pubKey):
    return hex(pubKey.x) + hex(pubKey.y % 2)[2: ]

curve = registry.get_curve('brainpoolP256r1')

def generarClavePrivECC():
    return secrets.randbelow(curve.field.n)

def generarClavePubECC(clavePriv):
    return clavePriv * curve.g

def ecc_point_to_256_bit_key(point):
    sha = hashlib.sha256(int.to_bytes(point.x, 32, 'big'))
    sha.update(int.to_bytes(point.y, 32, 'big'))
    return sha.digest()

def generarClavePrivECDSA():
    return SigningKey.generate(curve=NIST256p)

def generarClavePubECDSA(clavePriv):
```

```
    return clavePriv.verifying_key

def firmar_ECDSA(plaintext, llavePriv):
    return llavePriv.sign(plaintext,sigencode= sigencode_der)

def verificar_ECDSA(plaintext, firma, llavePub):
    return llavePub.verify(firma,plaintext,sigdecode=
sigdecode_der)

#Python-oauth2/oauth2/__init__.py
https://github.com/joestump/python-
oauth2/blob/81326a07d1936838d844690b468660452aafdea9/oauth2/__init
__.py#L165
def generate_nonce(length):
    """Generate pseudorandom number."""
    return ''.join([str(random.randint(0, 9)) for i in
range(length)])

def encrypt_DryGASCON(msg, secretKey):
    nonce = bytes(generate_nonce(16), encoding='utf-8')
    dryGasconCipher = DryGascon.DryGascon256().instance()
    ciphertext = dryGasconCipher.encrypt(secretKey,nonce,msg)
    return (ciphertext,nonce)

def decrypt_DryGASCON(ciphertext, nonce, secretKey):
    dryGasconCipher = DryGascon.DryGascon256().instance()
    plaintext =
dryGasconCipher.decrypt(secretKey,nonce,ciphertext)
    return plaintext
```

11. Bibliografía.

[1] M. Sönmez Turan *et al.*, «Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process», National Institute of Standards and Technology, NIST Internal or Interagency Report (NISTIR) 8369, jul. 2021. doi: 10.6028/NIST.IR.8369.

[2] A. Shikiar, «FIDO U2F & UAF Tutorial», *FIDO Alliance*, 24 de marzo de 2016. https://fidoalliance.org/fido-u2f-uaf-tutorial/?lang=zh-hans (accedido 8 de agosto de 2022).

[3] A. Angelogianni, I. Politis, y C. Xenakis, «How many FIDO protocols are needed? Surveying the design, security and market perspectives», *ACM Comput. Surv.*, p. 1122445.1122456, mar. 2022, doi: 10.1145/1122445.1122456.

[4] R. Contreras, «SolarWinds, el enemigo invisible».

[5] «ENISA Threat Landscape 2020 - Phishing», *ENISA*. https://www.enisa.europa.eu/publications/phishing (accedido 8 de agosto de 2022).

[6] «Titan Security Key - FIDO U2F USB-C NFC Bluetooth - Google Store». https://store.google.com/es/product/titan_security_key?hl=es (accedido 8 de agosto de 2022).

[7] «Google_Case_Study.pdf». Accedido: 8 de agosto de 2022. [En línea]. Disponible en: https://resources.yubico.com/53ZDUYE6/as/q3unyy-dmr8u0-fds0yi/Google_Case_Study.pdf

[8] «Client to Authenticator Protocol (CTAP)», p. 192.

[9] «Credential Management Level 1». https://w3c.github.io/webappsec-credential-management/#credentialscontainer (accedido 8 de agosto de 2022).

[10] «Web Authentication: An API for accessing Public Key Credentials - Level». https://w3c.github.io/webauthn/#dictdef-publickeycredentialrequestoptions (accedido 8 de agosto de 2022).

[11] «FIDO Security Reference», p. 24.

[12] G. Hatzivasilis, I. Papaefstathiou, C. Manifavas, y I. Askoxylakis, «Lightweight Password Hashing Scheme for Embedded Systems», en *Information Security Theory and Practice*, Cham, 2015, pp. 260-270. doi: 10.1007/978-3-319-24018-3_17.

[13] «Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process», p. 17.

[14] V. A. Thakor, M. A. Razzaque, y M. R. A. Khandaker, «Lightweight Cryptography Algorithms for Resource-Constrained IoT Devices: A Review, Comparison and Research Opportunities», *IEEE Access*, vol. 9, pp. 28177-28193, 2021, doi: 10.1109/ACCESS.2021.3052867.

[15] «Ascon – Specification». https://ascon.iaik.tugraz.at/specification.html (accedido 22 de agosto de 2022).

[16] «pydantic». https://pydantic-docs.helpmanual.io/ (accedido 7 de septiembre de 2022).

[17] B. Warner, «ecdsa: ECDSA cryptographic signature library (pure python)». Accedido: 7 de septiembre de 2022. [En línea]. Disponible en: http://github.com/tlsfuzzer/python-ecdsa

[18] «Matplotlib documentation — Matplotlib 3.5.3 documentation». https://matplotlib.org/stable/index.html (accedido 7 de septiembre de 2022).

[19] H. Eijs, «pycryptodome: Cryptographic library for Python». Accedido: 7 de septiembre de 2022. [MacOS :: MacOS X, Microsoft :: Windows, Unix]. Disponible en: https://www.pycryptodome.org

[20] «Ascon – Implementations». https://ascon.iaik.tugraz.at/implementations.html (accedido 7 de septiembre de 2022).

Universidad de Alcalá

Escuela Politécnica Superior