Para atender su solicitud, proporcionaré un desglose detallado de las tareas relacionadas con la identificación y documentación de módulos funcionales, la definición de interfaces y protocolos de comunicación, la asignación de responsabilidades de desarrollo, la redacción de un documento de arquitectura modular y la validación de la arquitectura para el **Centro centinela** proyecto. Dado que se trata de un proyecto estudiantil para un curso de ciberseguridad, adaptaré la respuesta para que sea práctica, clara y adecuada para un equipo de cuatro personas con distintos niveles de habilidad. También incorporaré mejoras del documento original y al mismo tiempo garantizaré que el alcance siga siendo manejable para un equipo de estudiantes.

**1. Identificar y documentar módulos funcionales**

**Enfoque**

El **Centro centinela** El proyecto está diseñado con una arquitectura modular, como se describe en el documento proporcionado. Los módulos funcionales se derivan de los objetivos del proyecto: centralizar el análisis de vulnerabilidad, implementar el concepto “Token de antesala” y brindar recomendaciones de mitigación. Cada módulo debe ser independiente, reutilizable y centrarse en una tarea específica para garantizar la escalabilidad y la mantenibilidad.

**Módulos funcionales**

Basado en la arquitectura del documento (Centinela/ Estructura del directorio) y requisitos, se identifican los siguientes módulos funcionales:

1. **Módulo central**
   * **Propósito**: Orquesta todo el sistema, coordinando interacciones entre módulos y gestionando configuraciones.
   * **Componentes**:
     + principal.py: Punto de entrada para la aplicación, inicializando y coordinando otros módulos.
     + despachador.py: Rutas de tareas entre módulos (por ejemplo, enviar archivos desde “Token de anteala” a Tokens especializados).
     + configurar/: Almacena archivos de configuración (por ejemplo, claves API, configuración de herramientas).
   * **Funcionalidad**:
     + Inicializa el sistema y carga configuraciones.
     + Gestiona el ciclo de vida del módulo (inicio, parada, manejo de errores).
     + Rutas de datos entre módulos según el tipo o la tarea.
2. **Módulo Token**
   * **Propósito**: Implementa la “Token de anteala” y Tokens especializados para el análisis de archivos de almacenamiento previo y el manejo de vulnerabilidades.
   * **Componentes**:
     + antesala.py: Intercepta y analiza archivos entrantes, determinando su tipo y redirigiendo a Tokens especializados.
     + static\_scan.py: Realiza análisis estáticos de archivos (por ejemplo, archivos de código para vulnerabilidades).
     + pdf\_sanitizer.py: Desinfecta los archivos PDF para detectar posibles malware o exploits.
   * **Funcionalidad**:
     + Identificación del tipo de archivo (usando mimetypes o libmagic).
     + Análisis de prealmacenamiento con ClamAV o herramientas similares.
     + Delegación a Tokens especializados según tipo de archivo o contexto de aplicación.
     + Colaboración entre Tokens para mejorar la detección de amenazas.
3. **Módulo de escáner web**
   * **Propósito**: Integra herramientas externas de escaneo de vulnerabilidades (por ejemplo, OWASP ZAP, Wapiti) para analizar aplicaciones web y API.
   * **Componentes**:
     + zap\_wrapper.py: Envuelve la API ZAP de OWASP para el escaneo de vulnerabilidades web.
     + resultados\_parser.py: Analiza los resultados del escaneo en un formato estandarizado para su almacenamiento e informes.
   * **Funcionalidad**:
     + Ejecuta escaneos web automatizados.
     + Procesa y normaliza las salidas de escaneo para su almacenamiento en la base de datos.
     + Se integra con escáneres basados en la nube si es posible (por ejemplo, VirusTotal API).
4. **Módulo de base de datos**
   * **Propósito**: Almacena resultados de escaneo, vulnerabilidades y recomendaciones de mitigación.
   * **Componentes**:
     + modelos.py: Define el esquema de la base de datos y las consultas para SQLite.
   * **Funcionalidad**:
     + Almacena los resultados del escaneo (por ejemplo, vulnerabilidades, marcas de tiempo, sistemas afectados).
     + Almacena sugerencias de mitigación basadas en OWASP/MITRE ATT&CK.
     + Proporciona interfaces de consulta para que otros módulos recuperen datos.
5. **Módulo GUI**
   * **Propósito**: Proporciona una interfaz de usuario (CLI o basada en web) para interactuar con el sistema.
   * **Componentes**:
     + simple\_web\_gui.py: Implementa una interfaz web básica usando Flask o FastAPI (CLI opcional para el MVP).
   * **Funcionalidad**:
     + Muestra los resultados del escaneo y las recomendaciones de mitigación.
     + Permite a los usuarios activar escaneos o configurar ajustes.
     + Implementa autenticación básica para proteger el acceso (por ejemplo, autenticación básica HTTP).
6. **Módulo de mitigación**
   * **Propósito**: Genera y aplica recomendaciones de mitigación para las vulnerabilidades detectadas.
   * **Componentes**:
     + Integrado en resultados\_parser.py y modelos.py para generar sugerencias de mitigación.
     + Scripts (futuros) para la mitigación automatizada de vulnerabilidades simples (por ejemplo, cerrar puertos abiertos con la aprobación del usuario).
   * **Funcionalidad**:
     + Genera informes HTML/PDF con pasos de mitigación.
     + Proporciona recomendaciones contextuales basadas en OWASP Top 10 o MITRE ATT&CK.
     + (Opcional) Ejecuta correcciones automatizadas con autorización del usuario.

**Documentación**

Cada módulo se documentará en un **Documento de especificación del módulo** con la siguiente estructura:

* **Nombre del módulo**: Por ejemplo, módulo central.
* **Propósito**: Breve descripción del rol del módulo.
* **Componentes**: Lista de archivos/scripts (por ejemplo, principal.py, despachador.py).
* **Entradas**: Datos o comandos recibidos (por ejemplo, flujos de archivos para antesala.py).
* **Salidas**: Datos o resultados producidos (por ejemplo, resultados de escaneo analizados).
* **Dependencias**: Bibliotecas o herramientas externas (por ejemplo libmagiaÉL, OWASP ZAP).
* **Consideraciones de Seguridad**: Autenticación, validación de entrada, etc.
* **Ejemplo de Flujo de trabajo**: Un caso de uso de muestra (por ejemplo, cómo un archivo es procesado por “Token de antesala”).

**Mejora**: Agregar a **README.md** por módulo en el repositorio del proyecto, resumiendo los detalles anteriores para una referencia rápida por parte del equipo y los evaluadores.

**2. Definir Interfaces y Protocolos de Comunicación**

**Enfoque**

Para garantizar la modularidad, cada módulo debe exponer interfaces claras (API o llamadas a funciones) y utilizar protocolos de comunicación estandarizados para las interacciones entre módulos. Dado el contexto del estudiante, las interfaces deben ser simples y utilizar las capacidades nativas de Python o marcos livianos.

**Interfaces**

1. **Interfaz del módulo principal**
   * **Métodos expuestos**:
     + inicializar\_sistema(config\_path): Carga la configuración e inicia los módulos.
     + dispatch\_task(tipo\_tarea, datos): Envía tareas a los módulos apropiados (por ejemplo, archivo al módulo Token, solicitud de escaneo al escáner web).
     + obtener\_estado(): Devuelve el estado del sistema (por ejemplo, módulos activos, tareas pendientes).
   * **Entrada/Salida**:
     + Entrada: archivos de configuración (JSON/YAML), solicitudes de tareas (por ejemplo, rutas de archivos, URL).
     + Salida: códigos de estado, identificadores de tareas o mensajes de error.
   * **Ejemplo**: dispatch\_task("scan\_web", "http://example.com") activa un escaneo web a través del Módulo de Escáner Web.
2. **Interfaz del Módulo Token**
   * **Métodos expuestos**:
     + anteala\_process(archivo\_path): Analiza un archivo y delega a Tokens especializados.
     + specialized\_scan(file\_type, file\_data): Realiza un análisis específico basado en el tipo de archivo.
     + sanitize\_file(file\_path): Desinfecta archivos (por ejemplo, archivos PDF) para un almacenamiento seguro.
   * **Entrada/Salida**:
     + Entrada: Rutas o flujos de archivos.
     + Salida: Resultados de análisis (seguro/inseguro, vulnerabilidades detectadas), archivos desinfectados.
   * **Ejemplo**: antesala\_process("documento.pdf") identifica el archivo como PDF y lo enruta a pdf\_sanitizer.py.
3. **Interfaz del módulo de escáner web**
   * **Métodos expuestos**:
     + start\_scan(url\_objetivo, herramienta="zap"): Inicia un escaneo usando OWASP ZAP o Wapiti.
     + analizar\_resultados(id\_escaneo): Convierte la salida de escaneo sin procesar a un formato estructurado.
   * **Entrada/Salida**:
     + Entrada: URL, selección de herramientas, parámetros de escaneo.
     + Salida: JSON con vulnerabilidades (p. ej., XSS, SQLi).
   * **Ejemplo**: start\_scan("http://example.com", herramienta="zap") ejecuta un escaneo OWASP ZAP.
4. **Interfaz del módulo de base de datos**
   * **Métodos expuestos**:
     + store\_result(scan\_id, vulnerabilidades): Guarda los resultados del escaneo.
     + store\_mitigation(id\_vulnerabilidad, texto\_mitigación): Guarda recomendaciones de mitigación.
     + resultados\_consulta(filtro): Recupera resultados según criterios (p. ej., fecha, gravedad).
   * **Entrada/Salida**:
     + Entrada: Resultados del escaneo (JSON), datos de mitigación.
     + Salida: Resultados de la consulta o estado de éxito/fracaso.
   * **Ejemplo**: store\_result("scan123", {"xss": "Alta gravedad"}) guarda la salida de un escaneo.
5. **Interfaz del módulo GUI**
   * **Métodos expuestos**:
     + mostrar\_resultados(id\_escaneo): Muestra los resultados del escaneo en la CLI o la interfaz web.
     + trigger\_scan(url\_objetivo): Permite a los usuarios iniciar un escaneo.
     + configure\_auth(nombre de usuario, contraseña): Configura la autenticación para la interfaz.
   * **Entrada/Salida**:
     + Entrada: Comandos de usuario, credenciales.
     + Salida: Resultados visualizados, informes HTML/PDF.
   * **Ejemplo**: mostrar\_resultados("scan123") representa una tabla de vulnerabilidades.
6. **Interfaz del módulo de mitigación**
   * **Métodos expuestos**:
     + generar\_mitigación(vulnerabilidad): Produce pasos de mitigación basados en el tipo de vulnerabilidad.
     + export\_report(scan\_id, formato="pdf"): Genera un informe en HTML o PDF.
     + (Futuro) apply\_fix(id\_vulnerabilidad, aprobación\_usuario): Ejecuta correcciones automatizadas.
   * **Entrada/Salida**:
     + Entrada: Datos de vulnerabilidad, autorización del usuario.
     + Salida: Texto de mitigación, archivos de informe.
   * **Ejemplo**: generar\_mitigación("xss") devuelve pasos basados en OWASP para mitigar XSS.

**Protocolos de comunicación**

* **Comunicación intramódulo**:
  + **Método**: Llamadas a funciones de Python o colas de mensajes.
  + **Protocolo**: Uso asyncio.Queue o multiprocesamiento.Queue para el paso asincrónico de tareas entre Tokens y otros módulos.
  + **Ejemplo**: El módulo principal envía un archivo a antesala.py a través de una cola, que luego delega a pdf\_sanitizer.py.
  + **Seguridad**: Validar entradas para evitar ataques de inyección (por ejemplo, desinfectar rutas de archivos con pathlib).
* **Comunicación entre módulos**:
  + **Método**: API REST (para módulos basados en web) o mensajes JSON (para comunicación interna).
  + **Protocolo**:
    - API REST (a través de Flask/FastAPI): para interacciones de GUI a núcleo o de escáner web a base de datos.
      * Ejemplo: POST/escaneo con carga útil JSON {"url": "http://example.com"}.
    - JSON sobre colas: para comunicación de Core a Token o de Token a Token.
      * Ejemplo: {"tarea": "escanear\_archivo", "ruta\_archivo": "/cargas/doc.pdf"}.
  + **Seguridad**:
    - Utilice HTTPS para las API REST.
    - Implemente la autenticación de clave API para las interacciones del módulo.
    - Registre todas las comunicaciones para depurarlas y auditarlas.
* **Integración de herramientas externas**:
  + **Método**: Llamadas API o ejecución de línea de comandos.
  + **Protocolo**:
    - OWASP ZAP/Wapiti: llamadas a la API HTTP (por ejemplo, POST/json/escanear).
    - ClamAV: Ejecución de línea de comandos o enlaces de Python (pyclamd).
  + **Seguridad**: Desinfecte las entradas a herramientas externas y valide las respuestas de la API.

**Mejora**:

* Uso **Redis** en lugar de colas de Python para una comunicación más sólida, ya que admite persistencia y escalabilidad.
* Implementar **OpenAPI/Swagger** para que la API REST documente los puntos finales y garantice la coherencia.
* Agregar **registro** (usando Python registro módulo) para rastrear interacciones y errores del módulo.

**3. Asignar responsabilidades de desarrollo**

**Composición del equipo**

* **Tamaño del equipo**: 4 estudiantes.
* **Supuestos**:
  + Todos los miembros tienen habilidades básicas de Python.
  + Al menos un miembro tiene experiencia en desarrollo web (Flask/FastAPI).
  + Al menos un miembro está familiarizado con las bases de datos (SQLite).
  + Todos están aprendiendo herramientas de ciberseguridad como parte del curso.

**Asignaciones de roles**

1. **Líder Técnico (Estudiante A)**
   * **Módulos**: Módulo principal, Módulo Token (parcial).
   * **Responsabilidades**:
     + Desarrollar principal.py y despachador.py.
     + Implementar antesala.py para el “Token de antesala.”
     + Definir protocolos de comunicación (colas, API REST).
     + Coordinar la integración y las pruebas del equipo.
   * **Habilidades requeridas**: Python, arquitectura del sistema, programación asíncrona.
2. **Desarrollador de Escáneres (Estudiante B)**
   * **Módulos**: Módulo de escáner web, módulo Token (parcial).
   * **Responsabilidades**:
     + Desarrollar zap\_wrapper.py y resultados\_parser.py.
     + Implementar static\_scan.py para análisis de archivos estáticos.
     + Integre las API de OWASP ZAP y Wapiti.
     + Pruebe la precisión de las salidas del escáner.
   * **Habilidades requeridas**: Python, integración de API, herramientas de ciberseguridad.
3. **Desarrollador de Interfaz (Estudiante C)**
   * **Módulo**: Módulo GUI.
   * **Responsabilidades**:
     + Desarrollar simple\_web\_gui.py (Flask o FastAPI).
     + Implemente CLI como alternativa.
     + Configure la autenticación básica para la interfaz.
     + Garantizar una visualización fácil de usar de los resultados.
   * **Habilidades requeridas**: Python, desarrollo web (Flask/FastAPI), HTML/CSS básico.
4. **Especialista en Base de Datos y Mitigación (Estudiante D)**
   * **Módulos**: Módulo de Base de Datos, Módulo de Mitigación.
   * **Responsabilidades**:
     + Desarrollar modelos.py para esquemas y consultas SQLite.
     + Generar recomendaciones de mitigación (integrando OWASP/MITRE ATT&CK).
     + Crear scripts de generación de informes (HTML/PDF).
     + Pruebe el rendimiento de la base de datos y la precisión de las consultas.
   * **Habilidades requeridas**: Python, SQLite, generación de informes (por ejemplo, reportlab para archivos PDF).

**Mejora**:

* **Entrenamiento cruzado**: Asigne tiempo en el primer sprint para que todos los miembros aprendan sobre cada módulo, garantizando flexibilidad si alguien no está disponible.
* **Programación en pares**: Para tareas complejas (por ejemplo, “Token de antesala”), empareje el Líder Técnico con el Desarrollador de Escáneres para compartir conocimientos.
* **Roles de respaldo**: Asignar responsabilidades secundarias (por ejemplo, el estudiante C ayuda con las pruebas, el estudiante D ayuda con la documentación).

**4. Redactar un documento de arquitectura modular**

**Documento de arquitectura modular para SentinelCore**

**1. Introducción**

**Centro centinela** es una herramienta de ciberseguridad centralizada para el análisis y mitigación de vulnerabilidades en entornos web y de nube. Su arquitectura modular garantiza escalabilidad, mantenibilidad y adaptabilidad, con un sistema único “Token de antesala” para el análisis de archivos de prealmacenamiento.

**2. Descripción general del sistema**

* **Propósito**: Centralizar el escaneo de vulnerabilidades, brindar recomendaciones de mitigación y prevenir amenazas a través de una arquitectura modular “Token”.
* **Características clave**:
  + Panel de vulnerabilidad centralizado.
  + Análisis de archivos de prealmacenamiento con “Token de antesala.”
  + Integración con herramientas de código abierto (OWASP ZAP, Wapiti, ClamAV).
  + Sugerencias e informes de mitigación automatizados.
* **Usuarios objetivo**: Pequeñas y medianas empresas, equipos DevSecOps y entornos educativos.

**3. Componentes de arquitectura**

El sistema está dividido en seis módulos funcionales, cada uno con responsabilidades e interfaces definidas.

**3.1 Módulo central**

* **Propósito**: Orquesta las interacciones del módulo y la configuración del sistema.
* **Componentes**:
  + principal.py: Punto de entrada al sistema.
  + despachador.py: Enrutador de tareas.
  + config/: Archivos de configuración (JSON/YAML).
* **Interfaces**:
  + inicializar\_sistema(config\_path): Inicia el sistema.
  + dispatch\_task(tipo\_tarea, datos): Enruta tareas a módulos.
  + get\_status(): Devuelve el estado del sistema.
* **Dependencias**: Python 3.8+, asincio.

**3.2 Módulo de Token**

* **Propósito**: Analiza archivos antes del almacenamiento y delega a Tokens especializados.
* **Componentes**:
  + antesala.p: Intercepción y enrutamiento de archivos.
  + estático\_scan.py: Análisis de archivos estáticos.
  + pdf\_sanitizer.p: Desinfección de PDF.
* **Interfaces**:
  + anteala\_process(archivo\_path): Analiza y enruta archivos.
  + specialized\_scan(file\_type, file\_data): Maneja tipos de archivos específicos.
  + sanitize\_file(file\_path): Desinfecta archivos.
* **Dependencias**: libmagia, ClamAV, asyncio.Queue.

**3.3 Módulo de escáner web**

* **Propósito**: Integra herramientas de escaneo externas para vulnerabilidades web.
* **Componentes**:
  + zap\_wrapper.py: Integración OWASP ZAP.
  + resultados\_parser.py: Normalización de resultados.
* **Interfaces**:
  + start\_scan(url\_objetivo, herramienta): Inicia escaneos.
  + analizar\_resultados(id\_escaneo): Procesa las salidas de escaneo.
* **Dependencias**: API ZAP de OWASP, Wapiti, solicitudes.

**3.4 Módulo de base de datos**

* **Propósito**: Almacena resultados de escaneo y datos de mitigación.
* **Componentes**:
  + modelos.py: Esquema y consultas de SQLite.
* **Interfaces**:
  + store\_result(scan\_id, vulnerabilidades): Guarda datos de escaneo.
  + store\_mitigation(id\_vulnerabilidad, texto\_mitigación): Guarda recomendaciones.
  + resultados\_consulta(filtro): Recupera datos.
* **Dependencias**: sqlite3.

**Módulo GUI 3.5**

* **Propósito**: Proporciona interacción del usuario a través de CLI o interfaz web.
* **Componentes**:
  + simple\_web\_gui.py: Interfaz basada en Flask/FastAPI.
* **Interfaces**:
  + mostrar\_resultados(id\_escaneo): Muestra resultados.
  + trigger\_scan(url\_objetivo): Inicia escaneos.
  + configure\_auth(nombre de usuario, contraseña): Configura la autenticación.
* **Dependencias**: Flask/FastAPI, passlib para autenticación.

**3.6 Módulo de mitigación**

* **Propósito**: Genera y aplica recomendaciones de mitigación.
* **Componentes**: Integrado en resultados\_parser.py y modelos.py.
* **Interfaces**:
  + generar\_mitigación(vulnerabilidad): Produce medidas de mitigación.
  + export\_report(id\_escaneo, formato): Genera informes.
* **Dependencias**: reportlab (PDF), datos de OWASP/MITRE ATT&CK.

**4. Protocolos de comunicación**

* **Intramódulo**: Llamadas a funciones de Python o asyncio.Queue para pasar tareas.
* **Intermódulo**: API REST (Flask/FastAPI) con cargas útiles JSON o Redis para comunicación basada en colas.
* **Herramientas externas**: API HTTP (OWASP ZAP, Wapiti) o ejecución de línea de comandos (ClamAV).
* **Seguridad**:
  + Validación de entrada para todas las interfaces.
  + HTTPS para API REST.
  + Registro de todas las interacciones.

**5. Flujo del sistema**

sirena

graph TD

A[User] -->|Triggers Scan| B[GUI Module]

B -->|REST API| C[Core Module]

C -->|Dispatch Task| D[Token Module]

C -->|Dispatch Task| E[Web Scanner Module]

D -->|File Analysis| F[Antesala]

F -->|Delegate| G[Specialized Tokens]

E -->|Scan Results| H[Database Module]

H -->|Mitigation Data| I[Mitigation Module]

I -->|Reports| B

**6. Consideraciones de seguridad**

* **Autenticación**: Autenticación básica HTTP para puntos finales de GUI y API.
* **Validación de entrada**: Desinfecte las rutas de archivos, las URL y las entradas de API.
* **Registro**: Uso registro Módulo para rastrear errores e interacciones.
* **Aislamiento**: Ejecute Tokens en contenedores Docker para evitar interferencias.

**7. Escalabilidad y extensibilidad**

* **Diseño modular**: Se pueden agregar nuevos Tokens o escáneres sin modificar el núcleo.
* **Contenerización**: Docker para una fácil implementación y escalamiento.
* **Sistema de colas**: Redis para una comunicación sólida en iteraciones futuras.

**8. Dependencias**

* **Bibliotecas de Python**: asyncio, solicitudes, sqlite3, libmagic, matraz/fastapi, reportlab, pyclamd.
* **Herramientas externas**: OWASP ZAP, Wapiti, ClamAV.
* **Infraestructura**: Máquinas locales o nube (AWS Free Tier, Heroku).

**9. Mejoras futuras**

* Soporte multilingüe para documentación e informes.
* Módulo de gamificación para uso educativo.
* Integración con escáneres nativos de la nube (por ejemplo, AWS Security Hub).

**Mejora**:

* Incluir a **Composición de Docker** archivo en la arquitectura para definir dependencias del módulo y redes.
* Añadir un **sección de pruebas** con pautas para el uso de pruebas unitarias y de integración pytest.

**5. Validar la Arquitectura con el Equipo**

**Enfoque**

La validación garantiza que la arquitectura sea factible, se alinee con los objetivos del proyecto y sea comprensible para todos los miembros del equipo. Esto implica revisiones y pruebas iterativas.

**Pasos de validación**

1. **Reunión de revisión de arquitectura**
   * **Objetivo**: Presentar el documento de arquitectura al equipo.
   * **Actividades**:
     + Recorra cada módulo, sus interfaces y protocolos de comunicación.
     + Discuta la viabilidad dadas las habilidades del equipo y el cronograma del proyecto.
     + Identificar posibles cuellos de botella (por ejemplo, complejidad de la integración de API).
   * **Herramienta**: Utilice un documento compartido (Google Docs, Notion) para recibir comentarios en tiempo real.
   * **Salida**: Lista de aclaraciones o modificaciones (por ejemplo, simplificar la API REST para el MVP).
2. **Prueba de prototipos**
   * **Objetivo**: Construya un prototipo mínimo para probar las interacciones del módulo.
   * **Actividades**:
     + Implementar un básico “Token de antesala” que procese un archivo de muestra y delegue a un Token especializado simulado.
     + Pruebe la integración de OWASP ZAP con una URL de muestra.
     + Almacene los resultados en SQLite y muéstrelos a través de CLI.
   * **Herramienta**: Docker para pruebas aisladas, pytest para pruebas unitarias.
   * **Salida**: Prueba de concepto que valida la comunicación de núcleo a extintor y de núcleo a escáner.
3. **Iteración de retroalimentación**
   * **Objetivo**: Refinar la arquitectura en función de los resultados del prototipo y los comentarios del equipo.
   * **Actividades**:
     + Ajuste las interfaces si falla la comunicación (por ejemplo, cambie de asyncio.Queue a Redis si es necesario).
     + Simplifique los módulos si el desarrollo es demasiado complejo (por ejemplo, concéntrese en la CLI en lugar de la GUI web).
   * **Herramienta**: Trello para realizar un seguimiento de los comentarios y los elementos de acción.
   * **Salida**: Documento de arquitectura actualizado.
4. **Validación final**
   * **Objetivo**: Confirme que la arquitectura esté lista para su desarrollo completo.
   * **Actividades**:
     + Presentar la arquitectura revisada al equipo y al instructor del curso (si corresponde).
     + Simule un flujo de trabajo completo (análisis de archivos, escaneo web, informe de mitigación).
     + Asegúrese de que todos los miembros del equipo comprendan sus roles y dependencias del módulo.
   * **Herramienta**: Zoom/Discord para discusión, GitHub para control de versiones.
   * **Salida**: Documento de arquitectura aprobado y plan de desarrollo.

**Mejora**:

* Uso **Miro** o **Gráfico lúcido** crear un diagrama de arquitectura interactivo para la reunión de revisión.
* Grabe la sesión de prueba del prototipo para documentar los problemas y compartirla con los miembros ausentes del equipo.
* Invite a un revisor externo (por ejemplo, un profesor o un estudiante de último año) para que proporcione comentarios objetivos.

**Conclusión**

Los módulos identificados (Core, Token, Web Scanner, Database, GUI, Mitigation) proporcionan una estructura clara para **SentinelCore**, con interfaces bien definidas y protocolos de comunicación (REST API, colas). Las asignaciones de roles se alinean con las habilidades asumidas por el equipo, y el documento de arquitectura formaliza el diseño para mayor claridad y escalabilidad. La validación a través de revisiones y prototipos garantiza la viabilidad de un equipo de estudiantes. Mejoras como Docker, Redis y gamification mejoran la robustez y el valor educativo de los proyectos.