Ethereum: Seguridad y Buenas Prácticas (DApps)

-Sprint 3-

Informe de Auditoria

Jaime Contreras
Master en Blockchain, Metaverso y NFT's
Diciembre 2023

Tabla de Contenido

1. Enfoque y Metodología

La metodología de auditoría de contratos inteligentes es esencial para garantizar la seguridad y el funcionamiento correcto en el mundo de las Criptomonedas y Blockchain.
Por tal razón se ha definido un proceso que comienza revisando la documentación para comprender el contrato y sus funciones. Luego, analiza el diseño y la estructura, identificando posibles amenazas. Herramientas automáticas como Slither ayudan a encontrar vulnerabilidades. Las pruebas de cobertura son cruciales para asegurar una amplia cobertura de código. La revisión manual profundiza en la detección de vulnerabilidades críticas. La documentación de hallazgos y sugerencias de soluciones asegura que el contrato cumpla con estándares de seguridad. Esta metodología es fundamental para proteger activos y usuarios en este espacio emergente. A continuación las fases del proceso.

Fase	Categoría	Descripción
 **Entendimiento Inicial**	**Documentación y Especificaciones**	• Revisar whitepaper, wiki, docs técnicas. • Entender propósito, funciones, actores, assets, y casos de uso del contrato
 **Arquitectura y Código Alto Nivel**	**Diseño y Estructura del Código**	• Revisar estructura, librerías, herencia, modificadores. • Mapear relaciones entre funciones/contratos. • Identificar vectores de entrada/salida de datos y cripto-activos
 **Evaluación de Riesgos**	**Threat Model**	• Enumerar amenazas comunes (reentrancy, overflow, etc). • Identificar amenazas particulares del contrato. • Priorizar amenazas críticas
 **Análisis Estático**	**Herramientas Automatizadas**	• Ejecutar Slither. • Revisar vulnerabilidades reportadas. • Evaluar bugs, code smells, y exploits potenciales. • Calificar hallazgos por probabilidad e impacto
 **Análisis Dinámico**	**Coverage de Tests**	• Ejecutar con solidity-coverage. • Analizar líneas/ramas de código sin coverage. • Desarrollar pruebas adicionales para casos críticos
 **Lógica de Negocio**	**Análisis de Funcionalidad**	• Revisar llamadas externas y manejo de cripto-activos. • Validar transferencias de fondos. • Verificar entradas/salidas de Ether/tokens
 **Revisión de Código**	**Revisión Manual**	• Verificar inicialización y uso de variables. • Detectar condiciones de carrera y reentrancy. • Confirmar validaciones de entradas y salidas
 **Reporte y Cierre**	**Documentación de Hallazgos**	• Documentar vulnerabilidades y exploits. • Proveer escenarios de reproceso. • Sugerir mitigaciones y soluciones cuando sea posible

I. Lista de Contactos

Nombre	Organización	Rol	Email	Teléfono
Juan Pérez	CryptoSecure Inc.	Auditor Jefe	juan.perez@email.com	+34 555 1234
María Gómez	Blockchain Innov.	Desarrolladora	maria.gomez@email.com	+34 555 5678
Carlos López	Tech Solutions	Analista de Riesgo	carlos.lopez@email.com	+34 555 9012
Ana Pozo	EtherTrust	Consultora	ana.pozo@email.com	+34 555 3456
Santiago Ruiz	SmartAudit	Director Técnico	santiago.ruiz@email.com	+34 555 7890

II. Control de Versiones

Versión	Fecha	Descripción	Autor	Revisado Por

2. Scope

El presente documento de auditoría está circunscrito exclusivamente al análisis y evaluación de los siguiente Smart Contract ; **VulnerableBank.sol**, **VulnerableDAO.sol**, **VulnerableShop.sol** y **VulnerableVault.sol**.

El alcance de esta auditoría se limita a las versiones de los contratos alojadas en el repositorio indicado, accedibles a través del enlace **https://github.com/jcontrerasd/Vulnerabilidades-Smart-Contract**, y se restringe al **commit ID 3a46d1c387b2cffb1de5ebafa3fc12af37641b7d**.

Es importante enfatizar que no se incluirán en esta auditoría otros repositorios ni contratos adicionales que no se hayan mencionado en este documento. Cualquier extensión del alcance, que involucre la revisión de repositorios adicionales o contratos no enlistados previamente, requerirá una actualización formal del documento de auditoría, así como una aprobación explícita de todas las partes interesadas.

Los hashes proporcionados corresponden a los estados de los archivos en el momento exacto del commit especificado y sirven como referencia inmutable para verificar la integridad de los contratos en futuras consultas o auditorías.

Este enfoque garantiza la precisión, la relevancia y la integridad de la auditoría, proporcionando un marco de trabajo claro y bien definido que guía el proceso de revisión y garantiza que todos los involucrados tengan una comprensión común de los límites dentro de los cuales se realiza este ejercicio de seguridad y verificación.

Detalle

- Repositorio de código: **https://github.com/jcontrerasd/Vulnerabilidades-Smart-Contract**
- Lista de contratos a auditar:
    - **VulnerableBank.sol**
    - **VulnerableDAO.sol**
    - **VulnerableShop.sol**
    - **VulnerableVault.sol**
- Commit ID que ha sido auditado: **fb2934ed8be4dc3c0324670d3b89bf1fdcc2a244**
- Tiempo destinado a la auditoría: 2 semanas (160 Horas) por un equipo de 2 Consultores
- Hashes de los diferentes archivos en scope (***shasum -a 256 nombre_del_archivo.sol***):
    - **VulnerableBank.sol: 3bcaf99eb5c6f318a3ff700b643e385143274ce97cefb50cde6b02916ac92fa1**
    - **VulnerableDAO.sol: a13fc4dd46ddd658d03c4efeea4556bbafef0818826c0c498150b68a7cc683a6**
    - **VulnerableShop.sol: 791b4d18394d6e16aa98991a900d5948f7e468272b75185752d8a4b9594d7a20**
    - **VulnerableVault.sol: 0671816a0c944cea5b1b243f5e67e061e1aa8931e6d8a3043a321c8f15d33459**

3. Resumen Ejecutivo

En nuestra revisión de seguridad de los Smart Contract que sustentan sus operaciones, hemos identificado riesgos críticos que requieren atención inmediata. Estos riesgos, si no se abordan, podrían permitir a actores malintencionados tomar el control de los contratos, manipular fondos, y alterar el comportamiento esperado de nuestras operaciones automatizadas.

Dos riesgos destacan por su potencial impacto:

Autenticación Vulnerable:
Nuestros métodos de verificación de identidad dentro de los contratos están basados en un modelo obsoleto que podría ser explotado. Esto es similar a tener una cerradura anticuada en una puerta de alta seguridad; no importa qué tan fuerte sea la puerta si la cerradura es fácil de forzar. Si esta vulnerabilidad es explotada, podría permitir a un atacante autorizarse como administrador y realizar cambios no autorizados, poniendo en riesgo nuestros activos digitales.

Procedimientos de Retiro Inseguros:
Hemos encontrado que el proceso de retiro de fondos de sus contratos es susceptible a ataques de "reentrada", un método sofisticado que se asemeja a un ladrón que entra y sale repetidamente de una caja fuerte antes de que la alarma se active. Esto podría resultar en la extracción no autorizada de fondos, vaciando efectivamente las reservas financieras del contrato.

Para mitigar estos riesgos, es crucial actualizar los metodos de autenticación y procedimientos de transacción. Esto incluye la implementación de controles de acceso más robustos y patrones de programación seguros que son estándares en la industria. Estas mejoras son equivalentes a instalar cerraduras electrónicas avanzadas y sistemas de alarma actualizados en nuestra infraestructura digital.

Es importante notar que la seguridad no es un estado, sino un proceso continuo. Por lo tanto, recomendamos una revisión y actualización regulares de sus sistemas para protegernos contra amenazas emergentes. La inversión en la seguridad de sus contratos inteligentes es esencial para mantener la confianza de sus usuarios/clientes y la integridad de la plataforma.

La acción inmediata para abordar estos problemas asegurará que la infraestructura siga siendo sólida, segura y confiable.

4. Criterios de Clasificación

La siguiente tabla proporciona una estructura para evaluar y clasificar las vulnerabilidades encontradas durante auditorías de Smart Contract y las cuales hacen parte de nuestra metodología. Estos niveles de riesgo, desde crítico hasta bajo, ayudan a determinar la prioridad y la urgencia de las respuestas de mitigación necesarias. Las descripciones resumen el impacto potencial de cada tipo de vulnerabilidad, proporcionando un marco claro para la acción correctiva y preventiva en el desarrollo y mantenimiento de contratos inteligentes seguros.

Nivel de Riesgo	Descripción
Crítico	Vulnerabilidades que permiten atacantes drenar fondos o tomar control total del contrato.
Alto	Defectos que exponen a manipulaciones de transacciones o a pérdidas financieras moderadas.
Medio	Problemas que pueden conducir a comportamientos imprevistos sin comprometer directamente los fondos.
Bajo	Inconsistencias menores que tienen un impacto limitado y no afectan la seguridad financiera.

5. Disclaimers

El cliente reconoce que, a pesar de nuestros mejores esfuerzos y experiencia en auditoría de Smart Contract, el proceso de identificación de vulnerabilidades es inherentemente complejo y no puede garantizar la detección de todas las posibles amenazas. La consultoría de auditoría no asume responsabilidad por cualquier vulnerabilidad no detectada o cualquier perjuicio que pueda derivarse de ella. Nuestro objetivo es minimizar los riesgos al máximo, pero no podemos garantizar una cobertura absoluta. Se recomienda encarecidamente al cliente adoptar medidas adicionales de seguridad y diligencia debida para proteger sus activos digitales.

Por medio de esta auditoría de seguridad y análisis, hemos identificado diversas vulnerabilidades en el sistema evaluado. Es importante destacar que las modificaciones y ajustes sugeridos para resolver estas vulnerabilidades se presentan exclusivamente como recomendaciones para el cliente. La consultoría de seguridad y auditoría proporciona orientación y asesoramiento experto en la identificación de posibles riesgos y soluciones, pero la implementación de dichas recomendaciones y mejoras debe ser responsabilidad exclusiva del cliente así como su puesta en producción. Estas sugerencias tienen como objetivo mejorar la seguridad y la integridad del sistema, y su adopción es fundamental para mantener la robustez y la confiabilidad del mismo.

6. SC0001-VulnerableBank.sol

Descripción General del Contrato *(/contracts/VulnerableBank.sol)*

Es un contrato inteligente enfocado en la gestión de inversiones y la distribución de beneficios a beneficiarios específicos. Los usuarios que inician la distribución reciben una recompensa en forma de porcentaje del monto distribuido. El contrato establece un administrador con la capacidad de modificar configuraciones y fija un mínimo de inversión para prevenir problemas de redondeo. Aunque su propósito principal es administrar inversiones y distribuir ganancias, presenta vulnerabilidades críticas que pueden ser explotadas, incluyendo riesgos de reentrancy y el uso inseguro de tx.origin para autenticación.

I. Vulnerabilidades Identificadas

1. SC0001-001 USO INCORRECTO DE TX.ORIGIN

Descripción: El modifier "onlyOwner" incluye un control de acceso que en lugar de comprobar msg.sender comprueba tx.origin. De esta manera, en lugar de comprobar quién manda el mensaje estará comprobando quién ha iniciado la transacción.

Criticidad: Critico

Impacto: Pérdida de Control y Manipulación del Contrato. Permite a un atacante modificar la configuración del contrato, pudiendo resultar en malversación de fondos o alteraciones en la lógica del contrato.

Código Original: *(/contracts/VulnerableBank.sol :36-40)*

```
36      ///@notice  Checks that the caller is the admin
37      modifier onlyOwner() {
38          require(tx.origin == admin, "Unauthorized");
39          _;
40      }
```

Cómo Explotarlo : Se podría orquestar un ataque de phishing contra el administrador, de forma que consiguieras que hiciera uso de un contrato malicioso intermedio. Este contrato malicioso podría llamar a la función "updateConfig" desde su código, suplantando al administrador y consiguiendo elegir libremente el periodo de distribución.

Paso 1: Construcción de Contrato Proxy
El atacante desarrolla un contrato con una función (fakeUpdateConfig) que llama a updateConfig en VulnerableBank. Este contrato es presentado como beneficioso para atraer al administrador de VulnerableBank.

Paso 2: Ejecución de Ataque de Phishing
Mediante técnicas de phishing, el atacante persuade al administrador de VulnerableBank para que interactúe con el contrato proxy, bajo la falsa pretensión de obtener beneficios.

Paso 3: Activación de Cambios no Autorizados
La interacción del administrador con el contrato proxy desencadena la función fakeUpdateConfig, explotando la verificación de tx.origin en VulnerableBank para modificar ilegítimamente su configuración.

Explicación Mitigación: Cambiar tx.origin por msg.sender en el modificador onlyOwner para asegurar que solo el que invoca de manera directa (y no el originador de la transacción) pueda ejecutar acciones restringidas al administrador.

Código Corregido:

```
modifier onlyOwner() {
   require(msg.sender == admin, "Unauthorized");
   _;}
```

2. SC0001-002 REENTRACY

Descripción: La función "distributeBenefits" aparentemente sigue el patrón CEI, pero el modificador "returnRewards" hace una llamada externa. Esto hace que sea vulnerable a reentrada, ya que antes de que se ejecute la secuencia segura de "doInvest", "returnRewards" podrá ejecutar código externo sin que la variable de estado "total_vested" haya sido actualizada.

Criticidad: Alta

Impacto: El impacto de esta vulnerabilidad es significativo, permite a un atacante extraer fondos del contrato de manera repetida antes de que se actualice el estado del contrato, lo que podría resultar en la pérdida de todos los fondos.

Código Original: *(/contracts/VulnerableBank.sol :43-53 / 88-113)*

```
43    ///@notice Transfers a percentage of the vested tokens to the caller as reward
44    ///@param percentage The percentage of the vested tokens to transfer
45    modifier returnRewards(uint percentage) {
46        // A hundreth of the distributed amount will be rewarded to the distributor as incentive
47        uint reward = total_invested * percentage / 10_000;
48
49        (bool success, ) =  payable(msg.sender).call{value: reward}("");
50        require(success, "Reward payment failed");
51
52        _;
53    }
```

```
88    /**
89        @notice Distributes a percentage of the total vested to the beneficiaries. Before that, the caller will be
90            rewarded with a percentage of the distributed amount as detailed in the returnRewards modifier
91        @param percentage The percentage of the vested tokens to distribute
92     */
93    function distributeBenefits(uint percentage)
94        external
95        returnRewards(percentage)
96    {
97        //  Checks
98        require(total_invested >= MIN_INVESTED, "Not big enough to avoid rounding issues");
99        require(percentage < MAX_PERCENTAGE, "Not big enough to avoid rounding issues");
100       require(block.number - latest_distribution >= distribute_period, "Too soon");
101
102       // Effects
103       latest_distribution = block.number;
104       // Calculate the amount to distribute as a percentage of the total vested
105       uint amount = total_invested * percentage / PERCENT;
106       // Subsctract the distributed amount from the total vested
107       total_invested -= amount;
108
109       //Interactions
110       doDistribute(amount);
111
112       emit Benefits(amount);
113   }
```

Cómo Explotarlo : Un atacante podría desplegar un smart contract malicioso para, aprovechando el ataque anterior, llamar a "doInvest" haciendo que el modificador "returnRewards" ejecute la llamada. La función receive() a su vez volverá a llamar a "doInvest", y así sucesivamente. Con esto el atacante conseguirá vaciar el contrato extrayendo en forma de rewards todos los fondos.

Paso 1: Despliegue del Contrato Malicioso
El atacante despliega un contrato inteligente malicioso que está diseñado para llamar a la función distributeBenefits del contrato VulnerableBank.

Paso 2: Primera Llamada a distributeBenefits
El contrato malicioso llama a distributeBenefits, activando el modificador returnRewards. Esto lleva a una transferencia de recompensa hacia el contrato malicioso.

Paso 3: Reentracy mediante receive() o fallback()
El contrato malicioso utiliza su función receive() o fallback() para llamar nuevamente a distributeBenefits antes de que la primera llamada se complete, explotando la vulnerabilidad de reentrancy para extraer repetidamente fondos como recompensas.

Explicación Mitigación: Para mitigar la vulnerabilidad de reentrancy, se debe actualizar el estado del contrato antes de realizar cualquier transferencia de fondos. Se recomienda implementar el patrón de Check-Effects-Interactions, asegurando que todas las modificaciones de estado ocurran antes de las llamadas externas.

Código Corregido:

```
function distributeBenefits(uint percentage)
    external
{
    require(total_invested >= MIN_INVESTED, "Not big enough to avoid rounding issues");
    require(percentage < MAX_PERCENTAGE, "Not big enough to avoid rounding issues");

    // Check: Verificaciones realizadas antes de modificar el estado o interactuar
    require(block.number - latest_distribution >= distribute_period, "Too soon");

    // Effect: Actualizar el estado antes de la interacción
    latest_distribution = block.number;
    uint amount = total_invested * percentage / PERCENT;
    total_invested -= amount;

    // Interaction: Llamada externa después de la actualización del estado
    uint reward = amount * percentage / 10_000;
    (bool success, ) = payable(msg.sender).call{value: reward}("");
    require(success, "Reward payment failed");

    doDistribute(amount);

    emit Benefits(amount);
}
```

7. SC0002-VulnerableDAO.sol

Descripción General del Contrato *(/contracts/VulnerableDAO.sol)*

Es un contrato inteligente que gestiona la resolución de disputas mediante votaciones. Los usuarios pueden votar en disputas sobre transacciones, y el resultado de la votación determina si el vendedor debe reembolsar al comprador o si la venta se cierra. Además, el contrato ofrece una función de lotería para premiar a los votantes con NFTs. Las acciones clave requieren autenticación a través de una contraseña. A pesar de su propósito innovador, el contrato tiene debilidades significativas en seguridad y generación de números aleatorios, lo que pone en riesgo la integridad de sus operaciones.

I. Vulnerabilidades Detectadas

1. SC0002-001 WEAK PSEUDO-RANDOMNESS

Descripción: La función "lotteryNFT" intenta calcular un número aleatorio para entregar un NFT a los participantes. El algoritmo usa datos conocidos públicamente, con lo que se podrá predecir si un usuario conseguirá NFT premium en un bloque o no.

Criticidad: Alta

Impacto: El impacto principal es la previsibilidad de los resultados. Un atacante con conocimiento suficiente de los datos del blockchain podría anticipar o influir en los resultados de la lotería para obtener NFT premium de manera desproporcionada, comprometiendo la imparcialidad del juego y posiblemente llevando a pérdidas financieras para otros participantes.

Código Original: *(/contracts/VulnerableDAO.sol :137-159)*

```
137     /**
138         @notice Run a PRNG to award NFT to a user
139         @param user The address of the elegible user
140      */
141     function lotteryNFT(address user) internal {
142         uint randomNumber = uint8(
143             uint256(
144                 keccak256(
145                     abi.encodePacked(
146                         blockhash(block.number - 1),
147                         block.timestamp,
148                         user
149         ))));
150 
151         if (randomNumber < THRESHOLD    ) {
152             /*
153             * Award NFT logic goes here
154             */
155             emit AwardNFT(user);
156         }
157 
158 
159     }
```

Cómo Explotarlo : Un atacante podría crear un smart contract malicioso y esperar hasta que llegue un bloque en el que pueda recibir NFT premium para llamar a "checkLottery".

Paso 1: Monitoreo de Bloques
El atacante monitorea los bloques y calcula el resultado del número pseudoaleatorio basado en el blockhash del bloque anterior, el timestamp del bloque actual, y su propia dirección.

Paso 2: Despliega un contrato inteligente malicioso
Está programado para llamar a la función checkLottery en el momento preciso en que las condiciones del bloque coinciden con un resultado favorable.

Paso 3: Ejecución en el Momento Óptimo
El contrato malicioso ejecuta la llamada a checkLottery en el bloque específico donde la predicción del número pseudoaleatorio indica una alta probabilidad de ganar un NFT premium.

Explicación Mitigación: Para mitigar esta vulnerabilidad, se recomienda utilizar un oráculo para generar números aleatorios o implementar un mecanismo de compromiso-revelación. Evitar el uso de variables predecibles como block.timestamp y blockhash para la generación de números aleatorios, ya que estos pueden ser manipulados o anticipados por los mineros o los atacantes.

Código Corregido: (https://docs.chain.link/vrf/v1/best-practices)

```
import "@chainlink/contracts/src/v0.8/VRFConsumerBase.sol";

contract VulnerableDAO is VRFConsumerBase {

    bytes32 internal keyHash; // Identificador de la clave de Chainlink VRF
    uint256 internal fee; // Costo en LINK para solicitar un número aleatorio
    mapping(bytes32 => address) private requestToUser; // Mapeo de solicitud de número aleatorio a la dirección del usuario

    // Eventos para registrar solicitudes y entrega de NFT
    event RandomNumberRequested(bytes32 indexed requestId, address indexed user);
    event AwardNFT(address indexed user, uint256 randomNumber);

    // Constructor para inicializar el contrato con la configuración de Chainlink VRF
    constructor(address vrfCoordinator, address linkToken, bytes32 _keyHash, uint256 _fee)
    VRFConsumerBase(vrfCoordinator, linkToken) {
        keyHash = _keyHash;
        fee = _fee;
    }

    // Función para verificar la posibilidad de ganar un NFT en una lotería
    function checkLottery(uint disputeID) external {
        // Solicitar un número aleatorio a Chainlink VRF
        bytes32 requestId = requestRandomness(keyHash, fee);
        requestToUser[requestId] = msg.sender; // Asociar la solicitud con el usuario
        emit RandomNumberRequested(requestId, msg.sender); // Emitir evento de solicitud
    }

    // Sobrescritura de fulfillRandomness para manejar el número aleatorio recibido de Chainlink VRF
    function fulfillRandomness(bytes32 requestId, uint256 randomness) internal override {
        address user = requestToUser[requestId]; // Obtener el usuario asociado con la solicitud
        lotteryNFT(user, randomness); // Llamar a la función de lotería con el número aleatorio
        delete requestToUser[requestId]; // Limpiar el mapeo después de procesar la solicitud
    }
                    // Función para determinar si se otorga un NFT basado en el número aleatorio
    function lotteryNFT(address user, uint256 randomness) internal {
        if (randomness < THRESHOLD) {
            emit AwardNFT(user, randomness); // Emitir evento si se cumple la condición para entregar NFT
            // Implementar la lógica para asignar y enviar el NFT al usuario 'user'
        }
    }
}
```

2. SC0002-002 INFORMACIÓN SECRETA NO CIFRADA / CONTROLES DE ACCESO INSEGUROS

Descripción: El "owner" de este contrato ha incluido una contraseña durante el despliegue para usarla como control de acceso. Como esta información está almacenada en el storage, el cual es público, cualquier podrá acceder a las funciones privilegiadas.

Criticidad: Alta

Impacto: Cualquier persona con acceso al blockchain puede leer la contraseña almacenada en el storage del contrato y utilizarla para acceder a funciones restringidas. Esto podría resultar en la manipulación no autorizada de la lógica del contrato, incluyendo la creación, modificación o resolución de disputas, y la manipulación de la función de lotería.

Código Original: *(/contracts/VulnerableDAO.sol :34-51)*

```
34      string password;
35
36
37      /************************************ Events and modifiers ************************************************/
38
39      event AwardNFT(address user);
40
41
42      /**
43          @notice Check if the caller is authorized to access key features
44          @param magicWord The password to access key features
45       */
46      modifier isAuthorized(string calldata magicWord) {
47          require(
48              keccak256(abi.encodePacked(magicWord)) == keccak256(abi.encodePacked(password)),
49              "Unauthorized");
50          _;
51      }
```

Cómo Explotarlo : Un atacante podría leer el storage del contrato de diferentes maneras para obtener el string de la contraseña. También se podría revisar el mensaje de despliegue e intentar obtener el valor entregado, pero es más difícil.

Paso 1: Leer el Storage del Contrato
Utilizar herramientas de exploración de blockchain como Etherscan para inspeccionar el storage del contrato inteligente y encontrar la variable password.

Paso 2: Decodificar la Contraseña
Dado que la contraseña está almacenada como un string, no requiere decodificación adicional. El atacante simplemente copia la contraseña tal como aparece en el storage.

Paso 3: Acceder a Funciones Restringidas
Utilizar la contraseña obtenida para interactuar con las funciones protegidas del contrato, como updateConfig, newDispute y otras, que requieren autorización.

Explicación Mitigación: Es esencial no almacenar contraseñas o claves secretas en el storage de un contrato inteligente, ya que es accesible públicamente. Una mejor práctica es utilizar un patrón de control de acceso basado en roles, como el proporcionado por OpenZeppelin, que utiliza direcciones de Ethereum para gestionar permisos sin exponer secretos.

Código Corregido:

```
// Importamos Ownable de OpenZeppelin para una gestión segura y sencilla de la propiedad del contrato.
import "@openzeppelin/contracts/access/Ownable.sol";

contract VulnerableDAO is Ownable {
    // Constructor ahora vacío. Ownable automáticamente establece el desplegador como el propietario.
    constructor() Ownable() {
        // No se necesita configuración adicional aquí.    }
    // Esta función previamente requería una contraseña para acceso.
    // Ahora, usando onlyOwner, garantizamos que solo el propietario del contrato pueda llamar a esta función.
    function updateConfig(string calldata newConfig) external onlyOwner {
        // Aquí iría la lógica de actualización de la configuración.    }
    // Cualquier otra función que necesite protección de acceso también debe usar onlyOwner.
    // Por ejemplo, una función para resolver disputas:
    function resolveDispute(uint disputeId) external onlyOwner {
        // Lógica para resolver la disputa.    }
    // Nota: Todas las variables y funciones no relacionadas con la vulnerabilidad han sido omitidas para enfocarnos en
la corrección.
}
```

8. SC0003-VulnerableShop.sol

Descripción General del Contrato *(/contracts/VulnerableShop.sol)*

Es una plataforma para la compra-venta de bienes, donde los vendedores bloquean fondos como garantía contra comportamientos maliciosos. Los compradores insatisfechos pueden abrir reclamaciones, que son resueltas por una DAO. Este contrato administra transacciones entre compradores y vendedores, incluye un proceso de disputa y gestiona la retención de fondos. Sin embargo, el contrato presenta vulnerabilidades en su implementación que pueden afectar la seguridad de las transacciones y la integridad del proceso de disputa.

I. Vulnerabilidades Detectadas

1. SC0003-001 VISIBILIDAD PREDETERMINADA (DEFAULT VISIBILITY)

Descripción: La función “deleteSale” está pensada para ser “internal”, pero ha sido marcada como “public”. Esto hará que, en lugar de ser llamada solo de manera controlada desde “removeMaliciousSale”, también pueda ser usada por sí misma.

Criticidad: Media

Impacto: Esta vulnerabilidad podría permitir que cualquier usuario externo llame directamente a la función "deleteSale", lo que puede llevar a resultados no deseados o maliciosos, como la eliminación de ventas legítimas.

Código Original: *(/contracts/VulnerableShop.sol :172-185)*

```
172     /**
173         @notice Remove a sale from the list and do some slashing afterwards
174         @param itemId The ID of the item which sale is being removed
175         @param toBePaid If the seller should be paid or not
176      */
177     function deleteSale(uint itemId, bool toBePaid) public {
178
179         delete offered_items[itemId];
180
181         /*
182         * Slashing code goes here
183         */
184
185     }
```

Cómo Explotarlo : Como la función afectada recibe el argumento “itemID”, un atacante simplemente podría llamar a la función para borrar Sales de manera arbitraria. Esto causaría una denegación de servicio completa de la plataforma, ya que todas las ventas creadas podrían ser borradas.

Paso 1: Cuenta externa
Se utiliza una transacción para llamar a la función "deleteSale" con un valor arbitrario para el argumento "itemID". Esto eliminará una venta de manera no autorizada.

Paso 2: Repite el Paso 1
Para diferentes valores de "itemID", eliminando múltiples ventas y causando interrupciones en la plataforma.

Paso 3: Continúa ejecutando transacciones maliciosas
Para eliminar ventas hasta que la plataforma quede inutilizable.

Explicación Mitigación: Cambiar la visibilidad de la función "deleteSale" a "internal" para que solo pueda ser llamada desde otras funciones internas del contrato. Además, implementa un mecanismo de control de acceso para autorizar quién puede llamar a esta función, por ejemplo, solo el propietario del contrato.

Código Corregido:

```
function deleteSale(uint itemId, bool toBePaid) internal {
   delete offered_items[itemId];

   /*
   * Slashing code goes here
   */
}
```

2. SC0003-002 FALLOS DE LÓGICA

Descripción: La función “reimburse” hace una transferencia de la cantidad a devolver en tokens. En lugar de dirigir la transferencia al usuario beneficiario de la devolución, la dirige a “msg.sender”.

Criticidad: Alta

Impacto: Esta vulnerabilidad permite a un atacante redirigir fondos a su propia dirección, lo que puede resultar en pérdidas financieras significativas para los usuarios legítimos del contrato.

Código Original: *(/contracts/VulnerableShop.sol :128-141)*

```
128     // /**
129     //     @notice Reimburse a buyer.
130     //     @param itemId The ID of the item being reimbursed
131     //  */
132     function reimburse(address user) external onlyOwner {
133         uint amount = disputed_items[user].price;
134         /*
135         * Reimbursement logic goes here
136         */
137
138         // Send the tokens back to the user
139         token.safeTransfer(msg.sender, amount);
140         emit Reimburse(msg.sender);
141     }
```

Cómo Explotarlo : Cada vez que el owner realice una devolución, esa misma dirección recibirá el reintegro en lugar de su legítimo beneficiario.

Paso 1: El atacante crea una cuenta
Luego realiza una transacción para llamar a la función "reimburse" con la dirección del atacante como argumento.

Paso 2: El contrato erroneamente transfiere
Los fondos son desviados al atacante en lugar del beneficiario legítimo, lo que resulta en que el atacante reciba los fondos.

Paso 3: El atacante repite
Los pasos 1 y 2 cada vez que el owner realice una devolución, acumulando fondos indebidos.

Explicación Mitigación: Corregir la función "reimburse" para transferir los fondos al usuario beneficiario de la devolución en lugar de usar "msg.sender" para evitar que el atacante reciba fondos indebidos.

Código Corregido:

```
function reimburse(address user) external onlyOwner {
    uint amount = disputed_items[user].price;

    /*
    * Reimbursement logic goes here
    */

    // Corregir: Transferir fondos al usuario beneficiario en lugar de msg.sender
    // Cambio de: token.safeTransfer(msg.sender, amount);
    // A: token.safeTransfer(user, amount);
    token.safeTransfer(user, amount);
    emit Reimburse(user);
}
```

9. SC0004-VulnerableVault.sol

Descripción General del Contrato *(/contracts/VulnerableVault.sol)*

Es un contrato que permite a los usuarios apostar y retirar Ether. Cuando un vendedor publica un artículo en la tienda, los fondos se bloquean durante la venta. Si la DAO considera al usuario malicioso, los fondos son confiscados. Este contrato gestiona el bloqueo y desbloqueo de fondos, interactúa con un contrato NFT de "powerseller" y con un contrato de tienda. Aunque proporciona una funcionalidad esencial para asegurar transacciones comerciales, presenta vulnerabilidades críticas que pueden ser explotadas, afectando la seguridad de los fondos y la integridad del sistema.

I. Vulnerabilidades Detectadas

1. SC0004-001 OVER/UNDERFLOWS ARITMÉTICOS

Descripción: La función “unstake” comprueba correctamente que el usuario que llama tiene suficientes fondos no bloqueados para recuperar la cantidad deseada. Pero el modifier “enoughStaked” no comprueba si la resta que realiza puede dar un resultado por debajo de cero, como el contrato hace uso de solidity 0.7 hará underflow si la cantidad a reducir, efectivamente dando un número muy alto que será considerado válido. Después, la función “unstake” también resultará en un underflow al restar la cantidad a extraer, permitiendo a cualquier usuario malicioso vaciar el contrato.

Criticidad: Crítica

Impacto: Esta vulnerabilidad podría permitir que un usuario malicioso vacíe el contrato, causando pérdidas significativas de fondos para los usuarios legítimos.

Código Original: *(/contracts/VulnerableVault.sol :30-38 / 72-83)*

```
30    ///@notice Check if the user has enough unlocked funds staked
31    modifier enoughStaked(uint amount) {
32        require(
33            (balance[msg.sender] - amount) > 0,
34            "Amount cannot be unstaked"
35        );
36
37        _;
38    }
```

```
72    ///@notice Unstake unlocked funds from the vault, the user must do it on their own
73    ///@param amount The amount of funds to unstake
74    function doUnstake(uint amount) external enoughStaked(msg.sender, amount) {
75        require(amount > 0, "Amount cannot be zero");
76
77        balance[msg.sender] -= amount;
78
79        (bool success, ) = payable(msg.sender).call{value: amount}("");
80        require(success, "Unstake failed");
81
82        emit Unstake(msg.sender, amount);
83    }
```

Cómo Explotarlo : Sería posible llamar a unstake con balance cero y sacar cualquier cantidad de fondos.

Paso 1: Configurar un Balance Cero
El atacante asegura que el balance de su cuenta en el contrato es cero. Esto podría lograrse mediante una serie de operaciones legítimas o simplemente creando una nueva cuenta sin fondos.

Paso 2: Llamada a doUnstake con Monto Alto
El atacante llama a la función doUnstake con un monto alto, superior a su balance actual (que es cero). Debido a la vulnerabilidad del underflow, el contrato interpretará que el atacante tiene suficientes fondos para retirar.

Paso 3: Explotar el Underflow para Retirar Fondos
El contrato, debido al underflow, permitirá al atacante retirar una cantidad de fondos significativamente mayor que su balance actual, potencialmente vaciando el contrato.

Explicación Mitigación: Actualizar el contrato para usar Solidity 0.8.x al igual que los otros contratos, que incluye protecciones contra underflows/overflows por defecto. Si no es posible, asegúrese de que todas las operaciones aritméticas estén protegidas contra underflows y overflows, utilizando verificaciones explícitas o bibliotecas de matemáticas seguras como OpenZeppelin.

Código Corregido:

```
pragma solidity ^0.8.0; // Actualizado a Solidity 0.8.x que incluye protecciones contra underflows y overflows

contract VulnerableVault {
    // ... Resto del código del contrato ...

    // Modificador actualizado para Solidity 0.8.x
    modifier enoughStaked(uint amount) {
        // Solidity 0.8.x maneja automáticamente los underflows y overflows
        require(balance[msg.sender] >= amount, "Insufficient balance");
        _;
    }

    // Función doUnstake actualizada para Solidity 0.8.x
    function doUnstake(uint amount) external enoughStaked(amount) {
        require(amount > 0, "Amount cannot be zero");

        // No es necesario usar SafeMath en Solidity 0.8.x, las operaciones son seguras por defecto
        balance[msg.sender] -= amount; // La resta es segura por defecto en Solidity 0.8.x

        (bool success, ) = payable(msg.sender).call{value: amount}("");

        require(success, "Unstake failed");

        emit Unstake(msg.sender, amount);
    }
    }

    // ... Resto del código del contrato ...
}
```

2. SC0004-002 VERIFICACIÓN DE RETORNO DE LLAMADA NO CHEQUEADA (UNCHECKED CALL RETURN VALUE)

Descripción: La función "claimReward" comprueba que poseemos un NFT concreto haciendo una llamada al contrato externo "powerseller_nft" usando "call". El resultado de esta llamada no es comprobado, con lo cual no hace efectos de control de acceso real.

Criticidad: Alta

Impacto: El impacto es significativo ya que podría permitir que se reclamen recompensas de forma indebida, lo que no solo afectaría la economía del contrato, sino que también podría desincentivar a los poseedores legítimos de NFT de participar en el sistema.

Código Original: *(/contracts/VulnerableVault.sol :86-105)*

```
86      // /**
87      // @notice Claim rewards generated by slashing malicious users.
88      //     First checks if the user is elegible through the checkPrivilege function that will revert if not.
89      */
90      function claimRewards() external {
91          uint amount;
92
93          powerseller_nft.call(
94              abi.encodeWithSignature(
95                  "checkPrivilege(address)",
96                  msg.sender
97              )
98          );
99
100         // /*
101         // * Rewards distribution logic goes here
102         // */
103
104         emit Rewards(msg.sender, amount);
105     }
```

Cómo Explotarlo : Un atacante sin privilegios podría llamar libremente a "claimRewards", ya que nunca se sabría si es o no privilegiado.

Paso 1: El atacante identifica el contrato con la vulnerabilidad
Comprende cómo la función claimRewards omite la verificación del resultado de la llamada externa.

Paso 2: El atacante invoca directamente
La función claimRewards sin tener el NFT requerido, sabiendo que la función no valida realmente si el llamador tiene privilegios.

Paso 3: Repetición de Paso 2
Si la función claimRewards incluye lógica que distribuye fondos o beneficios, el atacante repite el paso 2 múltiples veces para maximizar la extracción de recursos del contrato.

Explicación Mitigación: Es crucial implementar controles de acceso robustos. Las llamadas a contratos externos deben ser siempre verificadas. Utiliza patrones como checks-effects-interactions y considera el uso de modificadores de funciones para manejar controles de acceso y reentrancy guards para evitar llamadas no deseadas a funciones críticas.

Código Corregido:

```
function claimRewards() external {
    // Checks: Verificar si el remitente tiene privilegios para reclamar recompensas.
    (bool success, bytes memory data) = powerseller_nft_address.call(
        abi.encodeWithSignature("checkPrivilege(address)", msg.sender)
    );
    require(success && abi.decode(data, (bool)), "Sender not privileged");

    // Effects: Establecer la cantidad de recompensas.
    uint amount = calculateRewards(msg.sender);

    // Interactions: Emitir el evento de recompensas.
    emit RewardsClaimed(msg.sender, amount);
}
```

10. Funciones Adicionales de Seguridad

Funciones	Descripción	Beneficios
Emergency Stop (Pausa de Emergencia)	Implementación de un mecanismo que puede detener ciertas funcionalidades del contrato en caso de descubrirse una vulnerabilidad.	Permite una respuesta rápida para prevenir daños mayores mientras se investiga y corrige el problema.
Escape Hatch (Salida de Emergencia)	Permite retirar fondos a una dirección segura en caso de que el contrato se vea comprometido.	Protege los fondos de los usuarios permitiendo que sean recuperados en situaciones críticas.
Multisig	Requiere que múltiples partes autoricen una transacción antes de que pueda ejecutarse, a menudo implementado a través de un contrato de cartera multisig.	Aumenta la seguridad al requerir consenso para acciones críticas, reduciendo el riesgo de mal uso o robo por parte de un solo actor.
Rate Limiting (Limitación de Frecuencia)	Restricciones en la frecuencia de transacciones para prevenir abusos y reducir posibles daños en caso de ataque.	Limita la cantidad de fondos que podrían ser afectados en un periodo de tiempo determinado.
Timelocks	Introduce demoras obligatorias en la ejecución de ciertas funciones críticas.	Da tiempo para que las acciones sospechosas sean notadas y potencialmente revertidas antes de que causen daño.
Upgradability (Actualizaciones)	Permite actualizar el contrato para introducir mejoras o corregir fallos sin perder el estado o los fondos.	Asegura la capacidad de mejorar la seguridad con el tiempo sin tener que migrar recursos a un nuevo contrato.

11. Anexo de Revisión con Slither

- slither contracts/VulnerableBank.sol --checklist --show-ignored-findings --markdown-root ../ > ./audit/VulnerableBank.sol.md

- slither contracts/VulnerableDAO.sol --checklist --show-ignored-findings --markdown-root ../ > ./audit/VulnerableDAO.sol.md

- slither contracts/VulnerableShop.sol --checklist --show-ignored-findings --markdown-root ../ > ./audit/VulnerableShop.sol.md (***No se crea el archivo producto que el Smart Contract se encuenta con errores lo que impidió su compilación***)

- slither contracts/VulnerableVault.sol --checklist --show-ignored-findings --markdown-root ../ > ./audit/VulnerableVault.sol.md.

Se creo un carpeta **audit** en **github** con las salidas de estos analisis, cada archivo esta en formato markdown que contienen links hacia las lineas de los archivos .sol con los smart contract.