

Sécurité des Smart Contracts

Vulnérabilités, Attaques et Outils d'Audit



Sécurité



Attaques



Audit

Novembre 2025

Agenda et Objectifs de la Formation

Objectifs

- ✓ Comprendre les fondamentaux de la sécurité des smart contracts
- ✓ Identifier les types d'attaques courants et leurs mécanismes
- ✓ Maîtriser les outils d'audit essentiels pour évaluer la sécurité
- ✓ Appliquer les bonnes pratiques de développement sécurisé

Agenda

- 1 Introduction aux smart contracts et sécurité
- 2 Vulnérabilités et types d'attaques
 - Attaques par réentrance
 - Débordements d'entiers
 - Vulnérabilités de contrôle d'accès
- 3 Processus et outils d'audit
 - Le processus d'audit de sécurité
 - Outils d'analyse statique et dynamique
 - Intégration CI/CD et automatisation
- 4 Bonnes pratiques et évolutions
 - Bonnes pratiques de développement
 - Cas d'études et leçons apprises
 - Évolution future de la sécurité

Introduction aux Smart Contracts

Qu'est-ce qu'un Smart Contract ?

 Un smart contract est un programme auto-exécutable stocké sur une blockchain. Il exécute automatiquement les termes d'un accord lorsque des conditions prédefinies sont remplies, sans nécessiter d'intermédiaire.

Immuabilité

 Une fois déployés, les smart contracts sont immuables et ne peuvent être modifiés, garantissant ainsi l'intégrité de leur logique.

Décentralisation

 Les smart contracts s'exécutent sur un réseau décentralisé de nœuds, éliminant le besoin d'un intermédiaire fiable.

Applications Décentralisées

 Les smart contracts jouent un rôle central dans les DApps (Applications Décentralisées), gérant des transactions, des actifs et des logiques complexes.

Automatisation

 Les conditions prédefinies déclenchent l'exécution automatique des termes du contrat, réduisant les risques d'erreur humaine.

Pourquoi la Sécurité est-elle Critique ?



Enjeux Financiers Colossaux

En 2024, les pertes dues aux incidents de sécurité liés aux smart contracts : **1,42 milliard de dollars** sur 149 incidents.



Impact sur la Réputation

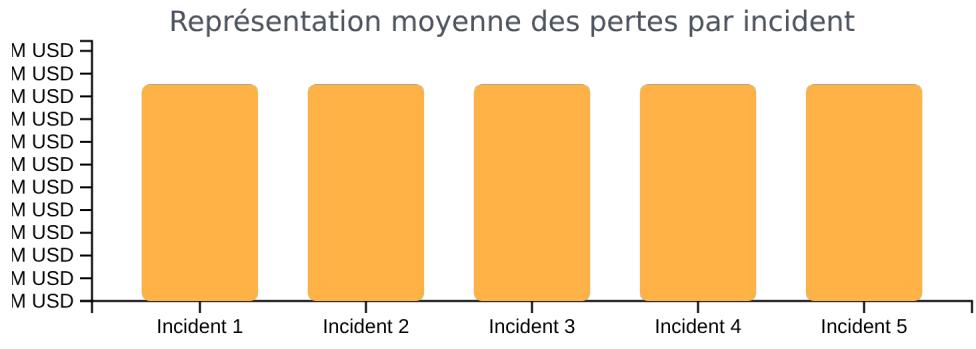
Les hacks entraînent une perte de confiance des utilisateurs. L'affaire du hack de The DAO en 2016 est un exemple marquant.



Irréversibilité des Transactions

Contraires aux systèmes traditionnels, les transactions blockchain sont irréversibles. Les fonds volés sont extrêmement difficiles à récupérer.

Incidents de Sécurité 2024



Moyenne par incident
1,42B 149 95,3M
Pertes totales (USD) Incidents documentés Moyenne par incident (USD)

Impact des Vulnérabilités sur l'Écosystème

Données sur les pertes financières

En 2024, les pertes financières dues aux incidents de sécurité liés aux smart contracts se sont élevées à :

1,42 milliard de dollars

sur 149 incidents documentés

(OWASP Smart Contract Top 10, 2025)



Enjeux Financiers

- Pertes financières directes lors d'attaques
- Invalidation économique des projets



Impact sur la Réputation

- Perte de confiance des utilisateurs et investisseurs
- Destruction d'un projet du jour au lendemain



Irréversibilité des Transactions

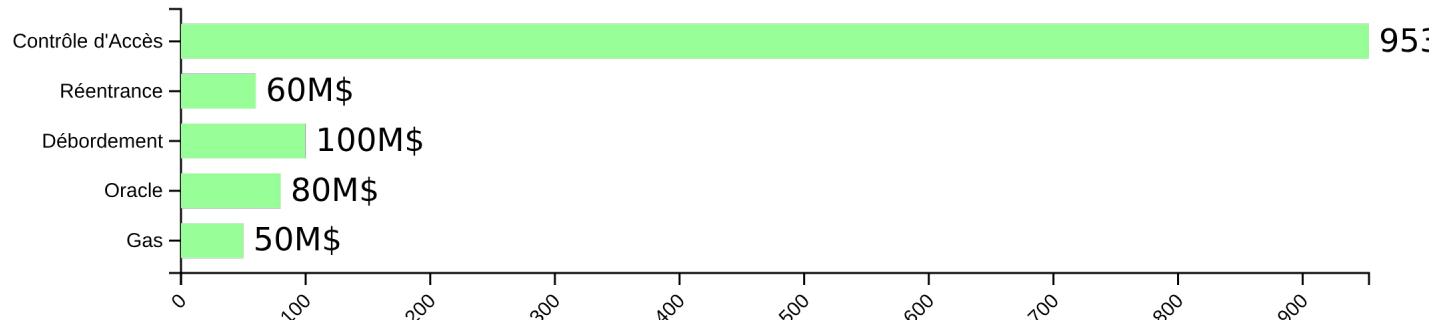
- Contrairement aux systèmes traditionnels
- Une fois les fonds volés, récupération extrêmement difficile

Classification OWASP Smart Contract Top 10



La liste des 10 vulnérabilités les plus critiques pour les smart contracts selon l'organisation OWASP.
Mise à jour annuelle pour aider les développeurs à sécuriser leurs contrats.

Pertes Financières par Vulnérabilité (2024)



1 **Contrôle d'Accès**

953,2M\$ pertes en 2024

2 **Réentrance**

60M\$ pertes (The DAO)

3 **Débordement**

SafeMath & Solidity 0.8+

4 **Oracle**

Manipulation de prix

5 **Gas**

Consommation optimale

Attaques par Réentrance (Re-entrancy)

💡 Qu'est-ce qu'une attaque par réentrance?

Une vulnérabilité où un contrat effectue un appel externe avant de mettre à jour son état, permettant à un contrat malveillant de "réentrer" la fonction et de répéter des actions.

1 Appel externe

Contrat vulnérable (A) appelle un contrat malveillant (B)



2 Préemption

Contrat B appelle la fonction de retrait avant mise à jour



3 Double dépôt

Contrat A envoie à nouveau des fonds avant mise à jour



⚠ Cas d'Étude : Le Hack de The DAO

- 2016 : Le DAO était un projet de fonds commun de capital-risque décentralisé
- Attaque par réentrance qui a permis de voler 3,6 millions d'Ether
- Équivalent à environ 60 millions de dollars à l'époque
- Résultat : Le projet a dû être sauvé par un hard fork



60 millions de dollars
perdus suite à l'attaque

Mitigation des Attaques par Réentrance

Trois stratégies essentielles pour prévenir les attaques par réentrance

Pattern

Checks-Effects-Interactions

Vérifications Effets Interactions

- ✓ Mettre à jour l'état avant d'interagir avec des contrats externes
- ✓ Effectuer les validations après avoir mis à jour l'état

```
function withdraw(uint256 amount) public {
    require(balances[msg.sender] >= amount);

    balances[msg.sender] -= amount; // Effets
    payable(msg.sender).transfer(amount); // Interactions
}
```



Gardes de Réentrance

Verrouillage

Empêche réentrance

Exécution

- ✓ Utiliser des modificateurs comme ReentrancyGuard
- ✓ Verrouiller la fonction pendant son exécution

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
contract MyContract is ReentrancyGuard {
    function withdraw(uint256 amount) public nonReentrant {
        payable(msg.sender).transfer(amount);
    }
}
```



Limitation des Appels Externes

Minimiser

"Pull" plutôt que "Push"

Sécurité

- ✓ Minimiser le nombre d'appels externes
- ✓ Préférer les mécanismes de "pull"

```
// Mauvaise pratique (Push)
function transfer(address to, uint256 amount) {
    to.call.value(amount)();
}

// Bonne pratique (Pull)
function requestWithdrawal(uint256 amount) {
    balances[msg.sender] += amount;
}

function withdraw() {
    payable(msg.sender).transfer(balances[msg.sender]);
}
```

Débordement d'Entiers (Integer Overflow/Underflow)

⚠️ Vulnérabilité

Les débordements d'entiers se produisent lorsque le résultat d'une opération dépasse la valeur maximale que le type peut représenter.

Exemple illustré

$$\begin{array}{r} \text{uint8} \\ 255 \end{array} + \begin{array}{r} 1 \end{array} = \begin{array}{r} 0 \end{array}$$
$$\begin{array}{r} \text{uint8} \\ 0 \end{array} - \begin{array}{r} 1 \end{array} = \begin{array}{r} 255 \end{array}$$

🔍 Cas d'Étude : Token BEC (2018)

Un attaquant a exploité une vulnérabilité de débordement d'entiers dans la fonction batchTransfer pour générer un montant de transfert nul.

Cette faille a permis de contourner la vérification de solde et de transférer illégalement des tokens.

🕒 Impact et Classement

Classement parmi les vulnérabilités les plus critiques par l'OWASP Smart Contract Top 10

3

Risque élevé

Protection contre les Débordements d'Entiers

Les débordements d'entiers (Integer Overflow/Underflow) sont une vulnérabilité courante dans les smart contracts Solidity.

🛡 SafeMath (Solidity < 0.8.0)

- ✓ Fonctions arithmétiques avec vérification
- ✓ Annule la transaction en cas de débordement

```
uint256 a = 255;  
uint8 b = 1;  
a = a.add(b); // Reverts en cas de débordement
```

</> Vérifications Intégrées (Solidity 0.8.0+)

- ✓ Vérifications automatiques des débordements
- ✓ Plus besoin de SafeMath pour les bases

```
uint256 a = 255;  
uint8 b = 1;  
a = a + b; // Reverts en cas de débordement
```

⚠ Exemple Réel : Attaque BEC (2018)

Attaquant a exploité une vulnérabilité dans la fonction **batchTransfer**.

```
function batchTransfer(address[] tos, uint256[] amounts) {  
    uint256 total = 0;  
    // Débordement non détecté  
    for(uint i = 0; i < tos.length; i++) {  
        total = total + amounts[i]; // Peut déborder  
    }  
    require(balanceOf[msg.sender] >= total);  
    // Transferts...  
}
```

Vulnérabilités de Contrôle d'Accès

#1

Rang dans OWASP Top 10 2025

\$

953,2 millions de dollars

pertes en 2024

Causes Courantes

 Visibilité `public` par défaut

Fonctions sans modificateur de visibilité sont publiques

 Utilisation incorrecte de `tx.origin`

Peut être exploité par des contrats malveillants

 Absence de modificateurs

Pas de restriction d'accès aux fonctions critiques



Exemple Marquant : Parity MultiSig

 Année : 2017

 Montant : 31 millions de dollars d'Ether

Faille de contrôle d'accès permettant à un attaquant de prendre le contrôle d'un contrat de bibliothèque, puis d'invoquer la fonction `selfdestruct`.

Conséquences

 Contrat rendu inaccessibles

 Fonds bloqués dans le contrat

 Perte de confiance dans la plateforme

Bonnes Pratiques de Contrôle d'Accès



Modificateurs de Fonction

Utilisez des modificateurs comme **onlyOwner** pour restreindre l'accès.

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    ...
}
```



Contrôle d'Accès RBAC

Implémentez des systèmes basés sur les rôles via OpenZeppelin AccessControl.

```
import "@openzeppelin/contracts/access/AccessControl.sol";

contract MyContract is AccessControl {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant USER_ROLE = keccak256("USER_ROLE");

    constructor() {
        _grantRole(ADMIN_ROLE, msg.sender);
    }
}
```



Utilisation de msg.sender

Toujours utiliser **msg.sender** pour les vérifications d'autorisation.

```
function transferOwnership(address newOwner) public {
    require(msg.sender == owner, "Not owner");
    owner = newOwner;
}
```



Déclaration Explicite de Visibilité

Toujours spécifier la visibilité des fonctions et variables.

```
public function publicFunction() {}
external function externalFunction() {}
internal function internalFunction() {}
private function privateFunction() {}
```

Manipulation d'Oracle de Prix et Flash Loans

Manipulation d'Oracle

Les oracles connectent les DApps aux données du monde réel, telles que les prix du marché.

Vulnérabilité majeure : Les oracles centralisés peuvent être exploités par les attaquants.

Mécanisme d'attaque

1. Attacker prend le contrôle d'un nœud oracle
2. Alimente le contrat avec des données falsifiées
3. Le contrat réagit aux prix faux, permettant des profits

Flash Loans

Les flash loans sont des prêts qui peuvent être pris et remboursés dans une seule transaction.

Avantage pour les attaquants : Manipuler les prix sans risque initial.

Comment les attaquants les utilisent

1. Emprunte une grande quantité de tokens
2. Manipule le prix du marché avec ces tokens
3. Rembourse le prêt + frais dans la même transaction

Autres Vulnérabilités Courantes



Génération de Nombres Aléatoires Faibles

- ⚠ **Définition :** Utilisation d'entropie prévisible ou insuffisante pour générer des nombres aléatoires
- ⚡ **Exploitation :** Les attaquants peuvent prédire les valeurs générées, affectant les jeux, les tirages au sort et la sécurisation
- 🛡 **Mitigation :** Utiliser des sources d'entropie sécurisées comme les variables de bloc (blockhash, timestamp) avec précaution



Déni de Service

- ⚠ **Définition :** Techniques qui empêchent l'exécution normale d'un smart contract
- ⚡ **Exploitation :** Boucles infinies, consommation excessive de gaz, ou conditions qui empêchent les transactions d'être traitées
- 🛡 **Mitigation :** Implémenter des limites de boucles, des vérifications de gaz, et des mécanismes de "pull" pour les transferts



Vulnérabilités de Logique Métier

- ⚠ **Définition :** Erreurs dans l'implémentation de la logique métier qui peuvent mener à des pertes financières
- ⚡ **Exploitation :** Incohérences dans les conditions, violations des invariants du contrat, ou comportements inattendus
- 🛡 **Mitigation :** Tests d'invariants, revue manuelle approfondie, et utilisation de bibliothèques éprouvées

Introduction au Processus d'Audit

Objectif de l'Audit

Identifier les failles de sécurité potentielles, les pratiques de codage inefficaces et les erreurs logiques dans les smart contracts.

Importance de l'Audit

Essentiel pour garantir la sécurité, la fiabilité et la performance des applications décentralisées (DApps) sur les plateformes Web3.



Audit de Smart Contract

Examen rigoureux et détaillé du code d'un protocole pour identifier les risques de sécurité.

Ce que les Auditeurs Recherchent

- Vulnérabilités connues (reentrancy, overflow)
- Erreurs logiques critiques
- Pratiques de codage inefficaces
- Problèmes de contrôle d'accès

Approche Globale

Les auditeurs combinent analyse statique, tests automatisés, revue manuelle du code et classifications des vulnérabilités par gravité.

Étapes Clés d'un Audit de Sécurité



1. Collecte & Gel du Code

- Documentation technique
- Code source
- Spécifications
- Livres blancs
- Diagrammes d'architecture



2. Tests Automatisés

- Vérification formelle
- Moteurs de test
- Tests de pénétration
- Évaluation des cyberattaques



3. Revue Manuelle

- Analyse ligne par ligne
- Expertise humaine
- Identification des anomalies
- Vérification des invariants



4. Classification des Vulnérabilités

- **Critique** : Impacte le fonctionnement
- **Majeure** : Logiques pouvant entraîner des pertes
- **Moyenne** : Affecte la performance
- **Mineure** : Inefficace, mais pas dangereux



5. Rapport & Corrections

- Rapport initial
- Solutions proposées
- Assistance aux corrections
- Vérification des fixes



6. Publication du Rapport

- Rapport complet
- Détail des découvertes
- Transparence aux parties prenantes
- Meilleures pratiques pour l'avenir

Classification des Vulnérabilités par Gravité



Critique

Impacte le fonctionnement sûr du protocole. Doit être corrigé immédiatement.



Majeure

Erreurs de centralisation ou logiques pouvant entraîner une perte de fonds ou de contrôle.



Moyenne

Affecte la performance ou la fiabilité de la plateforme. À corriger dans un bref délai.



Mineure

Code inefficace, mais pas immédiatement dangereux. Peut être corrigé à loisir.



Informationnelle

Liée au style ou aux bonnes pratiques de l'industrie. N'affecte pas la sécurité du protocole.

Analyse Statique vs Dynamique



Analyse Statique

Examine le code source ou le bytecode sans l'exécuter. Identifie les vulnérabilités et les erreurs par analyse de la structure et de la syntaxe.

Outils Courants



Slither



Mythril



Solhint

- + Couverture large et rapide des vulnérabilités connues
- + Détection efficace des problèmes comme les réentrances et les appels non vérifiés

! Limites

Peut générer des faux positifs et est limité aux smart contracts Solidity et Vyper.



Analyse Dynamique

Exécute le smart contract dans un environnement contrôlé avec des entrées variées pour découvrir des vulnérabilités non détectables par l'analyse statique.

Techniques Courantes



Fuzz Testing



Echidna



Tests d'Invariants

- + Recherche efficace de bugs logiques complexes
- + Découvre des cas limites inattendus qui ne seraient pas détectés statiquement

! Limites

Nécessite des entrées spécifiques au code et peut être gourmand en ressources pour les analyses complexes.

Slither - Outil d'Analyse Statique

Framework d'analyse statique open-source pour les smart contracts Solidity et Vyper, conçu pour identifier les vulnérabilités, les opportunités d'optimisation et les erreurs de codage sans exécuter le code.



Détecteurs Intégrés

Plus de 80 détecteurs pour une large gamme de vulnérabilités (reentrancy, pointeurs de stockage non initialisés, envois arbitraires, etc.)



Génération de Graphes

Capacité à générer des graphes d'héritage et des graphes d'appels de fonctions, facilitant la compréhension des architectures contractuelles complexes



Intégration CI/CD

S'intègre facilement dans les pipelines d'intégration et de déploiement continu (CI/CD) pour une analyse automatique à chaque modification



Rapports Détaillés

Fournit des rapports détaillés avec des niveaux de confiance pour chaque constatation, aidant les développeurs à prioriser les corrections



Optimisation du Gaz

Peut identifier des opportunités d'optimisation de la consommation de gaz, améliorant ainsi l'efficacité économique des déploiements



Limites

Peut générer un nombre significatif de faux positifs et est limité aux smart contracts Solidity et Vyper

Mythril - Exécution Symbolique



Qu'est-ce que Mythril?

Analyste de sécurité pour le bytecode EVM utilisant l'exécution symbolique et l'analyse de taint pour détecter les vulnérabilités.

Fonctionnalités et Avantages

Analyse Approfondie
Moteur d'exécution symbolique pour une analyse profonde

Compatibilité Multi-EVM
Prend en charge Ethereum, Hedera, Quorum et autres blockchains

Limites

- N'est pas personnalisable ; pas de création de détecteurs utilisateurs
- Peut être gourmand en ressources pour les analyses complexes

Approche Technique

Exécution Symbolique
Analyse le comportement sans exécuter avec des valeurs spécifiques

Calcul SMT
Résout des problèmes de logique pour détecter vulnérabilités

Echidna - Fuzzing Intelligent

Présentation

Echidna est un fuzzer pour smart contracts Ethereum utilisant le fuzzing basé sur les propriétés pour trouver des vulnérabilités.

Fonctionnalités

- ✓ **Tests basés sur les propriétés** : Écrivez des propriétés Solidity qui devraient toujours être vraies
- ✓ **Génération automatique** : Génère des entrées adaptées au code
- ✓ **Fuzzing guidé** : Utilise la couverture pour explorer l'espace d'état
- ✓ **Détection de bugs complexes** : Pour les transitions d'état inattendues

Limitations

- ✗ **Support Vyper** : Limité
- ✗ **Bibliothèques pour tests** : Limité

Fonctionnement



Comparaison des Outils d'Audit

Outil	Type d'Analyse	Avantages	Limitations
 Slither Analyse statique	 Statique	<ul style="list-style-type: none">+ Plus de 80 détecteurs+ Graphes d'héritage et d'appels+ Intégration CI/CD facile	<ul style="list-style-type: none">- Nombreux faux positifs- Limité aux smart contracts Solidity
 Mythril Exécution symbolique	 Symbolique	<ul style="list-style-type: none">+ Analyse approfondi du comportement+ Compatibilité multi-EVM+ Traces d'exécution détaillées	<ul style="list-style-type: none">- Non personnalisable- Ressources gourmandes
 Echidna Fuzzing	 Dynamique	<ul style="list-style-type: none">+ Tests basés sur les propriétés+ Génération automatique de tests+ Fuzzing guidé par la couverture	<ul style="list-style-type: none">- Support limité de Vyper- Limitations pour les bibliothèques

Autres Outils d'Analyse Importants

Manticore

Outil d'analyse symbolique pour la détection des vulnérabilités complexes.

- ✓ Analyse symbolique du bytecode Ethereum
- ✓ Génération automatique de cas de test
- ✓ Détection des invariants et des propriétés

Oyente

Analyseur statique pour la détection des vulnérabilités dans les smart contracts.

- ✓ Analyse statique du code source Solidity
- ✓ Détection des vulnérabilités par automates
- ✓ Modèle de comportement des contrats

Securify

Outil d'analyse sécurisée basé sur la vérification formelle.

- ✓ Vérification formelle des propriétés de sécurité
- ✓ Analyse des risques et des vulnérabilités
- ✓ Rapports de sécurité détaillés

Outils Complémentaires

Ensemble d'outils pour un audit complet et multi-facette.

- ✓ MythX: Analyseur basé sur Mythril
- ✓ Solgraph: Visualisation des graphes de contrôle
- ✓ ChainSecurity: Analyseur de sécurité Web3

Intégration CI/CD et Automatisation



Avantages de l'Intégration

- ✓ Détection précoce des vulnérabilités avant la mise en production
- ✓ Intégration continue de la sécurité dans le workflow de développement
- ✓ Réduction du coût et du temps de correction des failles de sécurité
- ✓ Assurance de la qualité du code et conformité aux standards de sécurité

Implémentation

- ❖ Outils d'Automatisation
 - Slither pour l'analyse statique
 - Mythril pour l'exécution symbolique
 - Echidna pour les tests de fuzzing
- ❖ Meilleures Pratiques
 - Configurer les outils pour s'intégrer aux tests unitaires
 - Utiliser des rapports standardisés pour une analyse facile
 - Mettre en place des webhooks pour les notifications

Bonnes Pratiques de Développement Sécurisé



Bibliothèques Éprouvées

- ✓ Intégrer des bibliothèques reconnues et régulièrement auditées
- ✓ Utiliser les versions stabilisées des contrats

```
import "@openzeppelin/contracts";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
```



Pattern C.E.I.

- ✓ Toujours effectuer les vérifications (**Checks**)
- ✓ Mettre à jour l'état du contrat (**Effects**)
- ✓ Interagir avec des contrats externes (**Interactions**)

```
function transfer(address to, uint256 value) public {
    require(balanceOf[msg.sender] >= value);
    balanceOf[msg.sender] -= value;
    balanceOf[to] += value;
    emit Transfer(msg.sender, to, value);
}
```



Gestion des Erreurs

- ✓ Utiliser **require()**, **assert()**, et **revert()** de manière appropriée
- ✓ Vérifications automatiques pour les débordements (Solidity $\geq 0.8.0$)
- ✓ Gérer les valeurs de retour des appels externes

```
function safeTransferFrom(address from, address to,
    uint256 value) public {
    require(balanceOf[from] >= value);
    require(to != address(0));
    balanceOf[from] -= value;
    balanceOf[to] += value;
    emit Transfer(from, to, value);
}
```

Tests et Validation Approfondis



Tests Unitaires Complètes

- ✓ Couverture exhaustive de tous les chemins d'exécution
- ✓ Vérification des cas limites et erreurs
- ✓ Tests des interactions entre contrats

"Les tests unitaires doivent couvrir 100% du code pour garantir la robustesse du contrat."



Fuzzing avec Foundry

- ✓ Génération automatique d'entrées aléatoires
- ✓ Recherche de bugs logiques complexes
- ✓ Tests d'invariants et propriétés

"Foundry permet de découvrir des vulnérabilités invisibles aux tests traditionnels."



Couverture des Chemins d'Exécution

- ✓ Analyse des chemins possibles dans le code
- ✓ Identification des branches non testées
- ✓ Génération de tests pour les cas rares



Gestion Sécurisée des Appels Externes



Vérification des Valeurs de Retour

Toujours vérifier les valeurs de retour des appels externes pour détecter les échecs d'exécution.

```
bool success = address(otherContract).call{value: 1 ether}("");  
if (!success) {  
    revert("Appel échoué");  
}
```

- Vérifiez les retours booléens des appels

- Gérez les exceptions avec des mécanismes appropriés



Pull vs Push

Préférez les mécanismes de "pull" plutôt que de "push" pour les transferts de fonds.

↓ Pull

L'utilisateur retire ses fonds

↑ Push

Le contrat envoie les fonds

- 🛡 Réduction des risques de réentrance
- 👤 Donne le contrôle à l'utilisateur



Validation Systématique

Validez toutes les entrées utilisateur pour prévenir les manipulations.

- ✓ Valeurs non nulles

- ✓ Plages valides

- ✓ Adresses correctes

- ✓ Types appropriés

</> Utilisez require() pour valider les entrées

⚠ Prévenez les comportements inattendus

Évolution Future de la Sécurité



Tendances Émergentes

- ↗ Évolution vers une approche plus proactive de la sécurité
- 🤖 Intégration de l'intelligence artificielle dans les audits
- 📋 Développement de standards plus robustes

● *Direction : Sécurité par conception*



Nouveaux Outils d'Audit

- 🔧 Outils hybrides analysant statique et dynamiquement
- 🧩 Testing basé sur les propriétés
- ✅ Outils d'intégration CI/CD pour la sécurité

● *Direction : Automatisation des audits*



Adaptation Écosystème

- ➡ Réaction aux nouvelles versions de Solidity
 - 🤝 Sécurité des contrats interopables
 - 👥 Collaboration entre développeurs et chercheurs
- *Direction : Écosystème sécurisé global*

Cas d'Études et Leçons Apprises

Cas d'Études

The DAO Hack (2016)

Attaque par réentrance qui a coûté 3,6 millions d'Ether (environ 60 millions)

Parity MultiSig Wallet (2017)

Vulnérabilité de contrôle d'accès causant une perte d'environ 31 millions de dollars

Leçons Apprises

 **Revue de code approfondie** : Mettre en place des processus de revue par les pairs

 **Tests complets** : Implémenter des tests unitaires et d'intégration couvrant tous les chemins

 **Bibliothèques éprouvées** : Utiliser des bibliothèques auditées comme celles d'OpenZeppelin

 **Audits professionnels** : Faire réaliser des audits de sécurité par des experts après chaque phase

Conclusion et Questions-Réponses

✓ Points Clés

- 🛡 La sécurité des smart contracts est critique en raison de leur nature immuable
- 👤 Les vulnérabilités courantes (réentrance, débordements) peuvent entraîner des pertes financières
- 🔍 Un audit rigoureux combiné à des outils d'analyse est essentiel pour identifier les vulnérabilités
- 💻 Les bonnes pratiques comme Checks-Effects-Interactions préviennent les attaques



Questions-Réponses

Comment appliquer ces principes de sécurité à mes projets?

Quels outils d'audit recommandez-vous pour un projet moyen?

Comment rester à jour avec les dernières vulnérabilités?



Pour approfondir vos connaissances:

- OWASP Smart Contract Top 10 (2025)
- OpenZeppelin Documentation