



Optimisation du gaz en Solidity

Mettre les variables de stockage en cache

- La lecture à partir d'une variable de stockage coûte au moins 100 gas
- Les écritures sont beaucoup plus cher
- Mettre en cache les variables de stockage pour effectuer une seule lecture et écriture

```
contract Caching {
         uint256 public number;
         function noCache(uint numberOfLoops) public view returns(uint result) {
             for(uint i = 0; i < numberOfLoops1; ++i) {</pre>
 8
                 result += number;
10
11
12
         function cache(uint numberOfLoops1) public view returns(uint result) {
13
14
             uint cachedVar = number;
             for(uint i = 0; i < numberOfLoops1; ++i) {</pre>
15
16
                 result += cachedVar;
18
19
```

Regrouper les structures

- Regrouper les éléments d'une structure pour réduire les coûts de gas
- Permet a minimiser les opérations coûteuses liées au stockage
- Les éléments de la premier structure sont stockés dans trois emplacements séparés
- ➤ Les éléments de la deuxième structure sont stockés dans seulement deux emplacements séparés => lecture et l'écriture moins cher

```
contract Packed_Struct {
         struct unpackedStruct {
             uint64 time;
             uint256 money;
 4
             address person;
 5
 6
 8
         struct packedStruct {
             uint64 time;
 9
             address person;
10
             uint256 money;
13
14
```

Utiliser des variables immuables et constantes

- Les variables qui ne sont jamais mises à jour devraient être immuables ou constants
- Les constantes et les valeurs immuables sont intégrées directement dans le bytecode du contrat et n'utilisent pas de stockage

```
4 contract ConstantAndImmutable {
5     uint256 constant public CONSTANT_VALUE = 123;
6     uint256 immutable public IMMUTABLE_VALUE;
7     constructor(uint256 _initialValue1) {
9         IMMUTABLE_VALUE = _initialValue1;
10     }
11     // rest of the contract code...
13 }
14
```



Timestamp & numéros de blocs n'ont pas besoin d'être uint256

- Un horodatage de taille uint48 fonctionnera pour des millions d'années à venir
- Un numéro de bloc s'incrémente toutes les 12 secondes => uint32 sera suffisant

```
contract TimestampAndBlockNumber {
         uint48 public timestamp;
         uint32 public blockNumber;
         constructor() {
             timestamp = uint48(block.timestamp);
             blockNumber = uint32(block.number);
10
11
12
         function updateData() public {
13
             timestamp = uint48(block.timestamp);
14
             blockNumber = uint32(block.number);
15
16
17
18
```

(1)

Calldata est moins cher que memory

- ➤ Uniquement pour les arguments de type référence des fonctions externes
- > L'accès aux données depuis calldata nécessite moins d'opérations
- ➤ Utiliser la mémoire uniquement lorsque les données doivent être modifiées

```
contract CalldataContract {
    function getDataFromCalldata(bytes calldata data) public pure returns (bytes memory) {
    return data;
}

contract MemoryContract {
    function getDataFromMemory(bytes memory data) public pure returns (bytes memory) {
    return data;
}

return data;
}
```



Utiliser ++i au lieu de i++ pour incrémenter

- i++ renvoie sa valeur précédente (i) avant d'incrémenter => deux valeurs sont stockées dans le stack
- > ++i incrémente i puis renvoie i => un seul valeur est stocké dans le stack

```
contract IncrementExample {
         uint256 public counter;
 6
         function incrementWithPrefix() public {
             counter = 0;
 9
             for (uint256 i; i < 10;) {
10
                  counter += 1;
11
                  unchecked {
12
                      ++i;
13
14
15
16
17
```

(

Les boucles do-while sont moins cher que les boucles for

```
12 v contract LoopDoWhile {
         function loop(uint256 times) public pure {
13 V
             if (times == 0) {
14 V
15
                 return;
16
17
18
             uint256 i;
19
20 V
             do {
21
                 // execute desired code ...
22 V
                 unchecked {
23
                     ++i;
24
             } while (i < times );
25
26
27
```

(1)

N'utiliser pas public pour les variables d'état si ce n'est pas nécessaire

- Une fonction publique (getter) est crée pour les variable de stockage publique
- Augmente la taille de la "jump table"
- > Augmente la taille du bytecode
- ➤ Le contrat devient plus volumineux

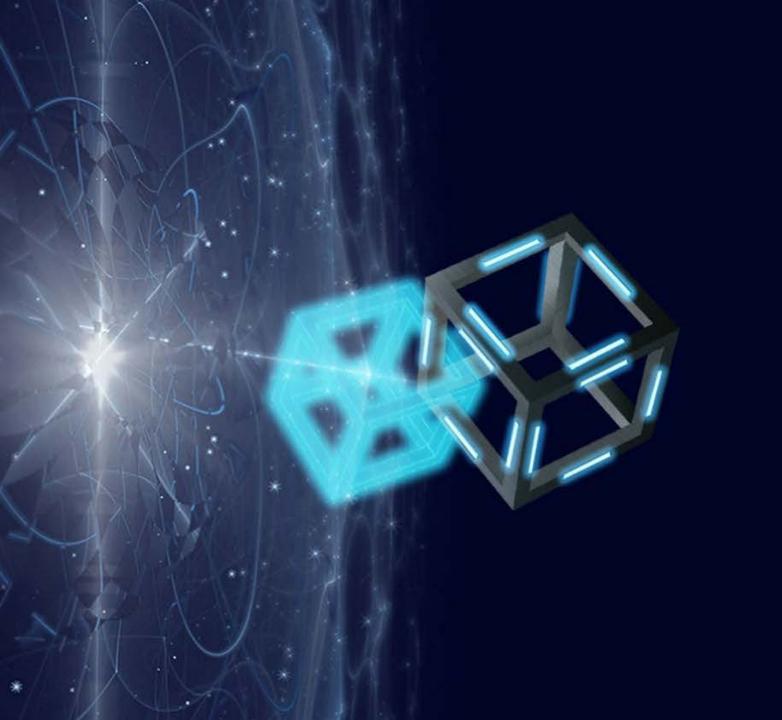
```
contract StateVariables {
 3
 4
         uint256 private privateVar = 100;
 6
         uint256 internal internalVar = 200;
 8
         uint256 public publicVar = 300;
 9
10
11
         // rest of the contract code...
12
13
```

Quelques ressources supplémentaires

- The RareSkills Book of Solidity Gas Optimization :
 https://www.rareskills.io/post/gas-optimization
- How to Optimize Smart Contracts in Solidity :
 https://medium.com/@0xkaden/how-to-write-smart-contracts-that-optimize-gas-spent-on-ethereum-30b5e9c5db85
- Solidity Gas Optimizations Cheat Sheet :
 https://0xmacro.com/blog/solidity-gas-optimizations-cheat-sheet



© R. Spadinger



Vulnérabilités des Contrats Intelligents

Vulnérabilités des Contrats Intelligents

Les vulnérabilités de sécurité des contrats intelligents posent des risques significatifs => pertes financières, rendre un protocole inutilisable...

Un audit et des tests approfondis sont essentiels avant le déploiement des contrats intelligents

Contrôle d'accès manquant

- Placer des restrictions sur qui peut appeler des fonction sensible, comme le retrait d'Ether, le changement du propriétaire du contrat...
- Même si un modificateur est en place, il y a eu des cas où le modificateur n'a pas été utilisé

```
1 ∨ contract MissingAccesControl {
         address public owner;
         modifier onlyOwner {
             owner == msg.sender;
 6
9
         function changeOwner(address newOwner) public {
10
11
             owner = newOwner1;
12
13
14 \
         function changeOwnerWithModifier(address newOwner1) public onlyOwner {
15
             owner = newOwner1;
16
17
```

(

Validation d'entrée incorrecte ou manquant

```
contract Auction {
         address public highestBidder;
         uint public highestBid;
 6
         function placeBid() public payable {
             require(msg.value < highestBid);</pre>
8
             highestBidder = msg.sender;
10
             highestBid = msg.sender;
11
12
13
```

Gas Griefing & déni de service (DoS)

Un contrat peut consommer malicieusement tout le gaz en entrant dans une boucle infinie.

```
contract DistributeETH {
       address[] users;
       function distribute(uint256 total1) public {
           for (uint i; i < users.length; ++i) {</pre>
              9
    contract Attacker {
       fallback() external payable {
12
          // infinite loop uses up all the gas
13
14
           while (true) {
15
16
17
                                            111/1/1/1/19
```

Insecure Randomness

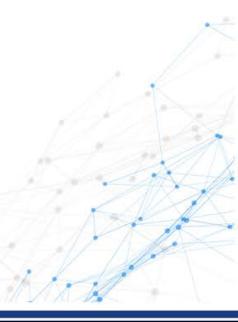
- Il n'est actuellement pas possible de générer de l'aléatoire de manière sécurisée sur la blockchain
- Les blockchains doivent être entièrement déterministes, sinon les nœuds ne pourraient pas parvenir à un consensus sur l'état
- Peu importe comment vous générez l'aléatoire, un attaquant peut le reproduire exactement

```
contract UnsafeDice {
         function randomNumber() internal view returns (uint256) {
             return uint256(keccak256(abi.encode(msg.sender, tx.origin, block.timestamp,
                 tx.gasprice, blockhash(block.number - 1))));
10
11
12
         function rollDice() public payable {
13
             require(msg.value == 1 ether);
14
             if (((randomNumber() % 6) + 1) == 6) {
15
                 (bool success,) = msg.sender.call{value: 2 ether}("");
16
                 require(success, "Transacion failed");
17
18
19
20
21
```

(

Insecure Randomness

```
∨ interface IUnsafeDice {
         function rollDice() external payable;
24
25
26
     contract ExploitDice {
28
         IUnsafeDice unsafeDice;
29
30
         constructor(address _unsafeDice1) {
31
             unsafeDice = IUnsafeDice(_unsafeDice1);
32
33
34
35
         function randomNumber() internal view returns (uint256) {
             return uint256(keccak256(abi.encode(msg.sender, tx.origin, block.timestamp,
36
                 tx.gasprice, blockhash(block.number - 1))));
37
38
39
40
         function attack() public payable {
             if (((randomNumber() % 6) + 1) == 6) {
41
                  unsafeDice.rollDice{value: 1 ether}();
42
43
44
45
```



Variables privées

- Les variables privées sont toujours visibles sur la blockchain
- ➤ Il faut jamais stocker les informations sensibles
- > Pour lire une variable, il faut connaître son emplacement de stockage
- ➤ Dans l'exemple, l'emplacement de stockage de "secretNumber" est 2

```
contract PrivateExample {
    uint256 public someNumber;
    address internal someAddress;
    uint256 private secretNumber;

constructor(uint256 _initialValue)) {
    secretNumber = _initialValue);
}

//ethers.js : await provider.getStorageAt(contractAddress, slotNumber);

//ethers.js : await provider.getStorageAt(contractAddress, slotNumber);
```

Erreurs d'arrondi - multiplier avant de diviser

```
contract RoundingErrors {
        //factor = 1001
10
        function divideFirst(uint256 factor1) external pure returns (uint256) {
             return (1000 / factor) * 100;
11
12
13
        function test2(uint256 factor1) external pure returns (uint256) {
14
15
             return (1000 * 100) / factor1;
16
17
```

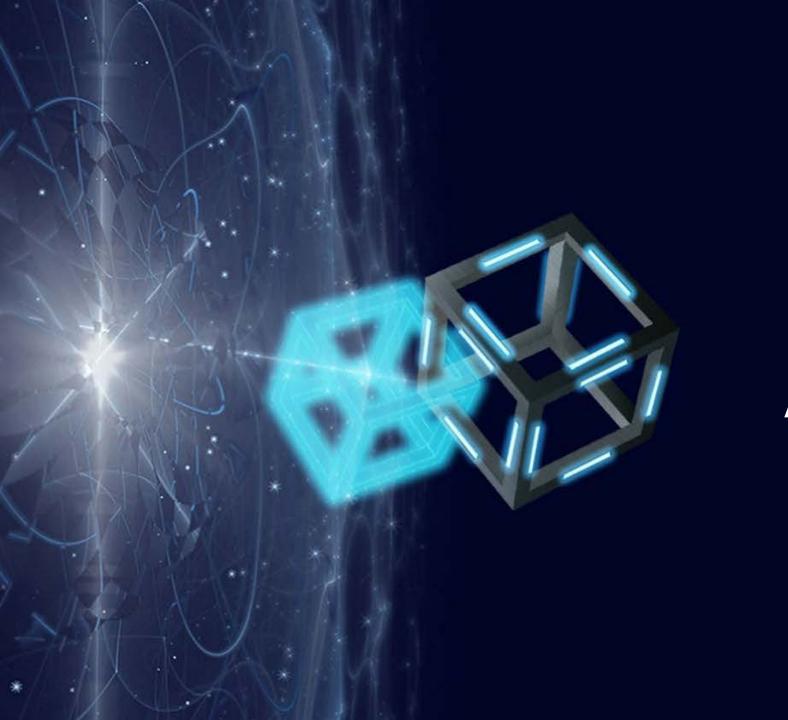
Solidity n'a pas de nombres à virgule flottante, donc les erreurs d'arrondi sont inévitables. La division doit toujours être effectuée en dernier.

Quelques ressources supplémentaires

- Solidity Smart Contract Attack Vectors :
 https://github.com/Quillhash/Solidity-Attack-Vectors
- Smart Contract Vulnerabilities :
 https://github.com/kadenzipfel/smart-contract-vulnerabilities
- Smart Contract Security :
 https://www.rareskills.io/post/smart-contract-security
- Solidity By Example Hacks : https://solidity-by-example.org/
- Security Vulnerability Aggregator : https://solodit.xyz/



© R. Spadinger



Slither Analyse Statique

Slither - Trail of Bits

- Outil d'analyse statique open-source
- Spécialisé dans la sécurité des contrats intelligents Ethereum
- > Recherche les vulnérabilités potentielles et les mauvaises pratiques de programmation dans le code
- > Fournit des recommandations pour améliorer la sécurité et la qualité du code
- Utilisé par de nombreux développeurs et auditeurs
- Page officielle Github : https://github.com/crytic/slither
- Prérequis :
 - Installer Python 3.8+
 - > Installer Solc-Select requis si Hardhat, Foundry... n'est pas utilisées

© R. Spadinger

Python 3.8+

- > Télécharger Python : https://www.python.org/downloads/
- Installer Python: https://www.datacamp.com/blog/how-to-install-python



24

Solc-Select

- > Outil pour passer rapidement entre les versions du compilateur Solidity
- > Page officielle Github : https://github.com/crytic/solc-select
- Installer Solc-Select : pip3 install solc-select
- Utilisation de Solc-Select :
 - Vérifier la version actuelle de solc : solc --version
 - Installer une version spécifique de sol : solc-select install 0.8.20
 - > Utiliser une version spécifique : solc-select use 0.8.20



Utilisation de Slither

- Installation de Slither : pip3 install slither-analyzer
- Ajouter slither.config.json au dossier racine du projet

> Exécution de Slither : slither . ou : slither ./contracts/myContract.sol

(1)

Slither Filters

- Les résultats peuvent être filtrés :
 - > Optimisation : --exclude-optimization
 - > Informations : --exclude-informational
 - Résultats faibles : --exclude-low

Utilisation du filtre : slither . --exclude-informational



Slither Printers

- > Par défaut, aucune imprimante n'est exécutée.
- Pour exécuter des imprimantes :

slither ./contracts/myContract.sol --print contract-summary,function-summary

- Autres imprimantes utiles :
 - > inheritance-graph
 - > call-graph

Liste de toutes les imprimantes : https://github.com/crytic/slither/wiki/Printer-documentation



Slither Detectors

- Slither exécute tous ses détecteurs par défaut.
- Pour exécuter uniquement des détecteurs sélectionnés : slither . --detect arbitrary-send,pragma
- Pour exclure des détecteurs : slither . --exclude naming-convention,unused-state
- Liste de toutes les détecteurs : https://github.com/crytic/slither/wiki/Detector-Documentation

