스마트 컨트랙트 취약점 유형 #3

2021.11.4. 김경동

내용

- Smart Contract Vulnerability Category #3
  - Block Stamp Manipulation
  - Constructors with Care
  - Uninitialized Storage Pointers
  - Floating Pointers and Numerical Precision
  - tx.origin Authentication
- Case Study - Polygon’s double spending bug

Block Timestamp Manipulation

취약점 유형 소개

- Solidity에는 Random number generation이 없음
- Block의 Timestamp를 이용
  - Block을 만드는 Miner(채굴자)가 이를 약간 조절할 가능성 존재

Block Timestamp Manipulation

취약점 예제

```
1  contract Roulette {
2      uint public pastBlockTime; // Forces one bet per block
3
4      constructor() public payable {} // initially fund contract
5
6      // fallback function used to make a bet
7      function () public payable {
8          require(msg.value == 10 ether); // must send 10 ether to play
9          require(now != pastBlockTime); // only 1 transaction per block
10         pastBlockTime = now;
11         if(now % 15 == 0) { // winner
12             msg.sender.transfer(this.balance);
13         }
14     }
15 }
```

Note

In version 0.7.0, the alias `now` (for `block.timestamp`) was removed.

Block Timestamp Manipulation

취약점 방지

> **Note**
>
> Do not rely on `block.timestamp` or `blockhash` as a source of randomness, unless you know what you are doing.
>
> Both the timestamp and the block hash can be influenced by miners to some degree. Bad actors in the mining community can for example run a casino payout function on a chosen hash and just retry a different hash if they did not receive any money.
>
> The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

- Block Timestamp를 이용, Random number 생성이나 활용으로 사용 금지
  - 특히, 게임 승부 관련이나 중요 상태 변화
- Block Number 사용이 낫다

Constructors with Care

취약점 유형 소개

- Solidity 0.4.22 이전에서는 Contract와 **같은 이름의 함수**가 Constructor 였음
- 개발 중 Contract 이름이 바뀌는데 Constructor 함수는 그대로 있는 경우, 호출 가능한 함수가 되어 버림
- 그 이후 버전에서는 해당 syntax가 deprecated 됨
  - constructor 라는 키워드로 정의

> **Warning**
>
> Prior to version 0.4.22, constructors were defined as functions with the same name as the contract. This syntax was deprecated and is not allowed anymore in version 0.5.0.

Uninitialized Storage Pointers

취약점 유형 소개

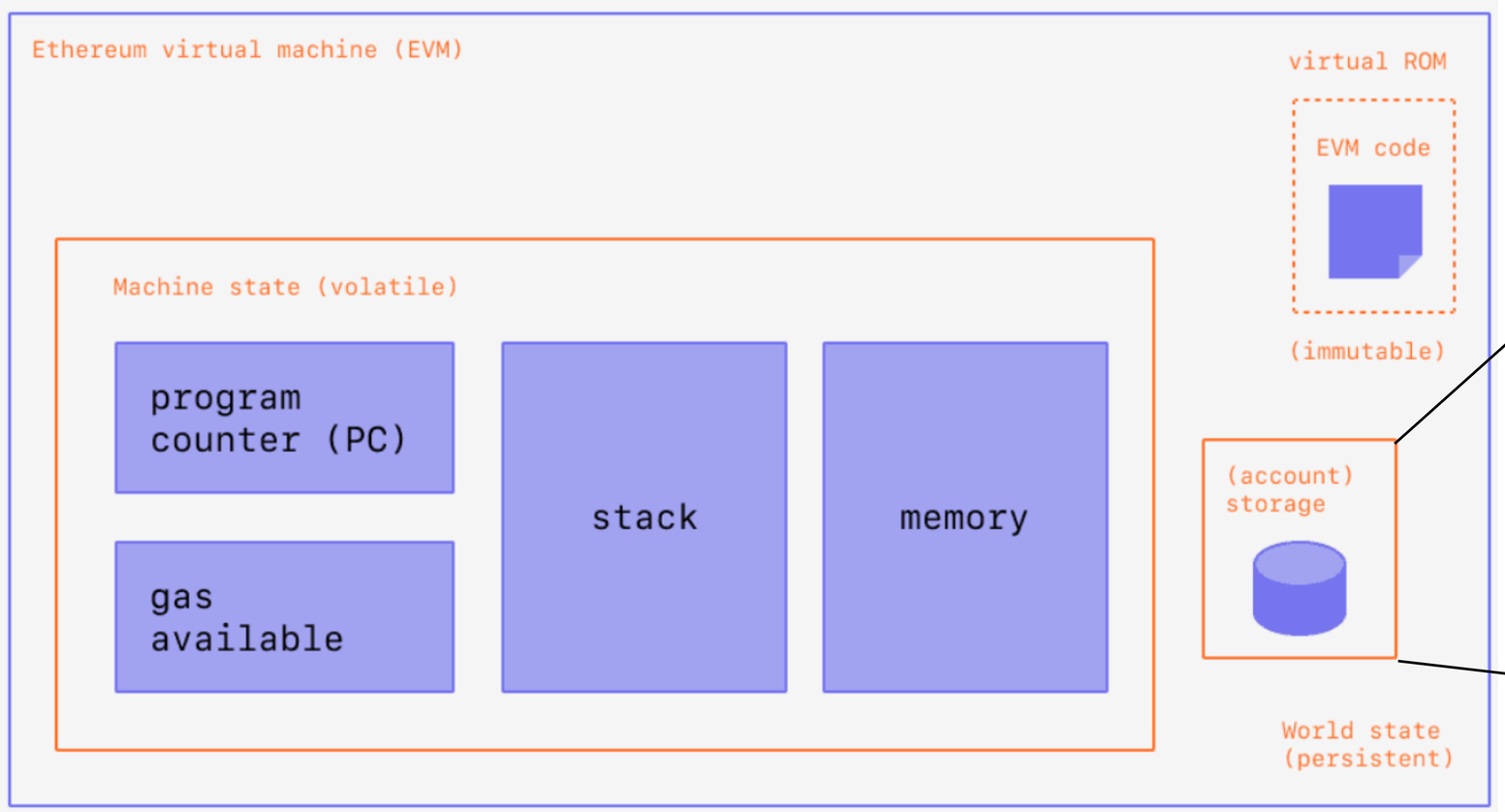
- 초기화되지 않은 변수로 인한 취약점
  - 기존 SW 취약점에서도 자주 발생되는 유형
  - Memory Leak, Authentication Bypass, …

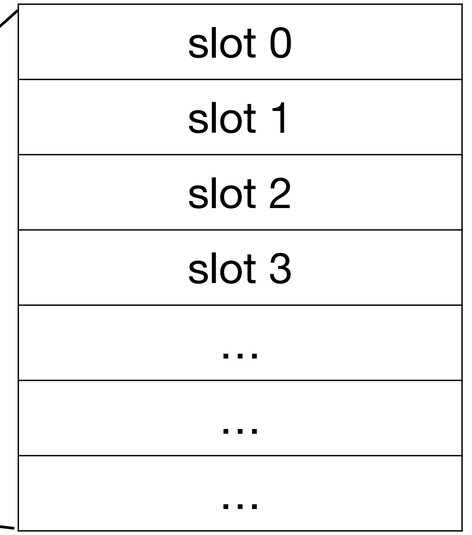
Example Language: **C**

```
char *test_string;
if (i != err_val)
{
  test_string = "Hello World!";
}
printf("%s", test_string);
```

Uninitialized Storage Pointers

EVM 메모리 구조





상태 변수, 매핑, 구조체 타입 등 저장

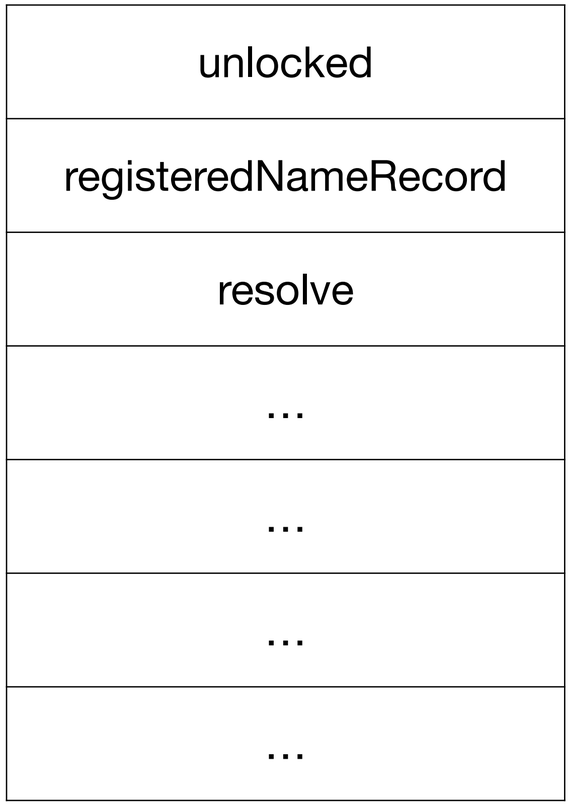
Uninitialized Storage Pointers

취약점 예제

```
// A Locked Name Registrar
contract NameRegistrar {

    bool public unlocked = false;  // registrar locked, no name updates

    struct NameRecord { // map hashes to addresses
        bytes32 name;
        address mappedAddress;
    }

    mapping(address => NameRecord) public registeredNameRecord; // records who
    mapping(bytes32 => address) public resolve; // resolves hashes to addresse

    function register(bytes32 _name, address _mappedAddress) public {
        // set up the new NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;

        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;

        require(unlocked); // only allow registrations if contract is unlocked
    }
}
```

Uninitialized Storage Pointers

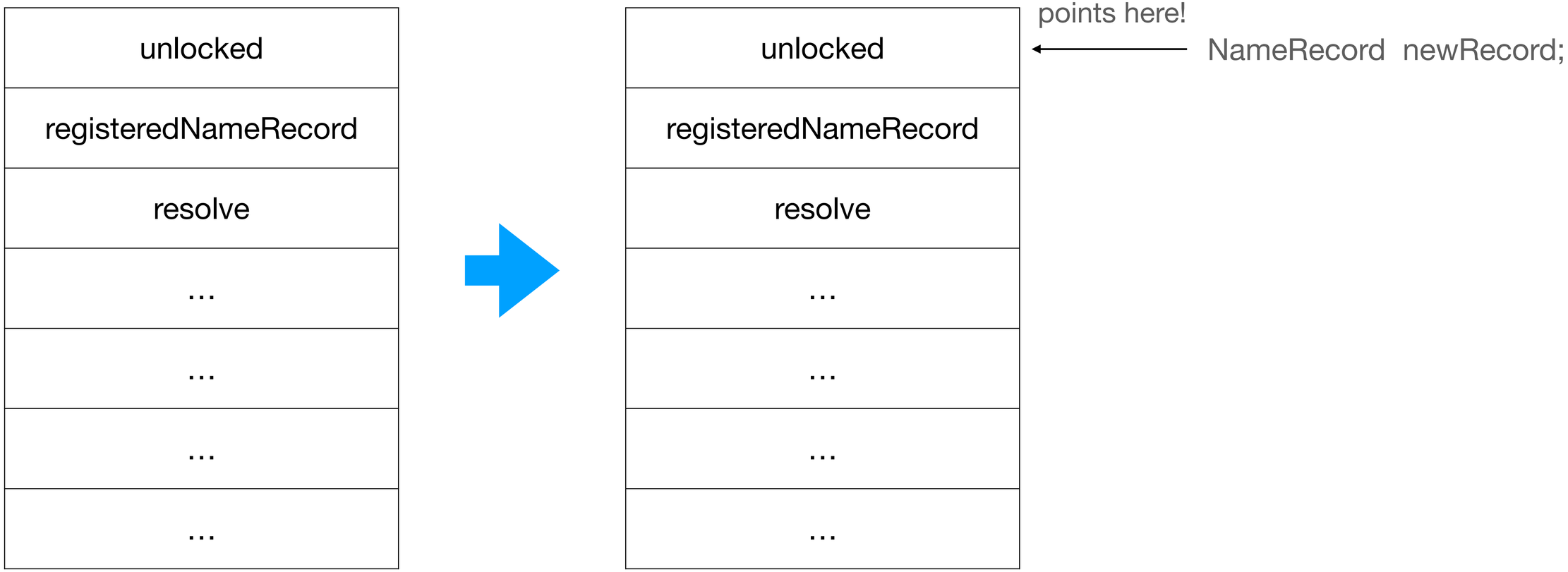
예제 Storage 메모리 상태



State Variables in contract Sequentially Stored
(Don’t consider the optimization in this example)

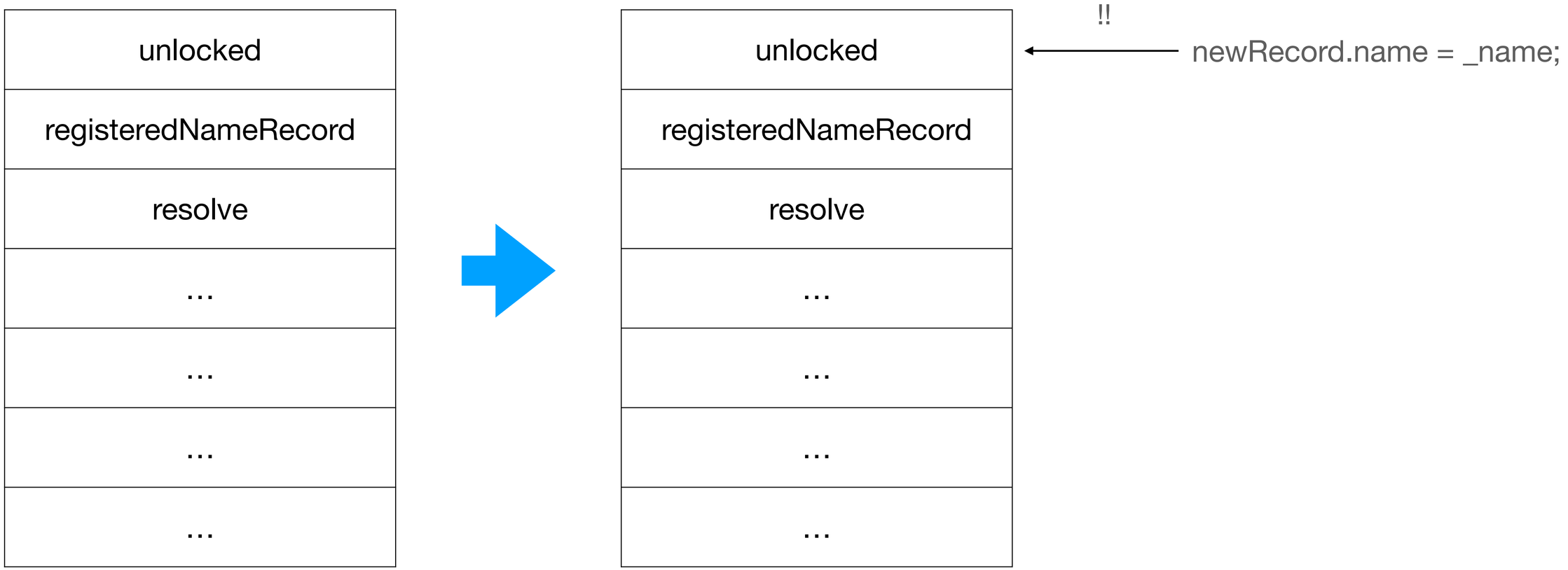
Uninitialized Storage Pointers

예제 Storage 메모리 상태



Uninitialized Storage Pointers

예제 Storage 메모리 상태



Uninitialized Storage Pointers
취약점 방지

- Solidity 컴파일러가 Uninitialized Storage Variables 존재시 Warning 줌
  - 컴파일이 안되도록 하기도 함

```
Compiling 1 file with 0.7.3
contracts/Token.sol:75:9: TypeError: Data location must be "storage", "memory" or "calldata" for variab
le, but none was given.
        NameRecord newRecord;
        ^------------------^

Error HH600: Compilation failed
```

  - 명시적으로 memory, storage 키워드 사용
    - Uninitialized Storage Variable 접근 코드에서 컴파일 에러

Floating Points and Precision

취약점 유형 소개

- Solidity에서는 Floating Point 타입에 대해 제대로 지원이 되지 않음(not fully supported)
  - Solidity 0.8.9 기준
- 따라서 개발자가 Integer 타입으로 Floating Point를 구현
  - 취약점 발생 가능

Floating Points and Precision

취약점 예제

```
contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping(address => uint) public balances;

    function buyTokens() public payable {
        uint tokens = msg.value/weiPerEth*tokensPerEth; // convert wei to eth
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        uint eth = tokens/tokensPerEth;
        balances[msg.sender] -= tokens;
        msg.sender.transfer(eth*weiPerEth); //
    }
}
```

Floating Points and Precision

취약점 방지

- 보다 큰 수로 나누어지게
  - weiPerEth(1e18) -> weiPerTokens(1e19)
- 곱하기/나누기 계산 순서 생각
- Solidity 0.8.9 기준 fixed/ufixed 타입 지원/사용
- 라이브러리 사용?

Floating Point Arithmetic

- `Multiprecision.sol` 89, by alianse777 2.
- `ABDKMathQuad.sol` 62, by ABDK 9. Smart contract library of mathematical functions operating with IEEE 754 quadruple-precision binary floating-point numbers (quadruple precision numbers). ⚠️ Weird license.

Fixed Point Arithmetic

- `FixidityLib.sol` 77, by CementDAO. This library provides fixed point arithmetic with protection against overflow.
  Users:
  - DeltaCamp's fork 13 used by PoolTogether 4.
  - Fork 13 used by Celo 7.
- `SafeDecimalMath.sol` 86, by Synthetix 1. Safely manipulate unsigned fixed-point decimals at a given precision level.
- `Exponential.sol` 71, by Compound 5. Exponential module for storing fixed-precision decimals.
- `ABDKMath64x64` 33, by ABDK 9. Smart contract library of mathematical functions operating with signed 64.64-bit fixed point numbers. ⚠️ Weird license.
- `FixedPoint.sol` 84 by UMA 5. Library for fixed point arithmetic on uints with 18 digit precision.
- `RealMath.sol` 53 by Macroverse 3.
- `BNum.sol` 53 by Balancer 2. Library for fixed point arithmetic with 18 digit precision.
- `PRBMathSD59x18.sol` 2 and `PRBMathUD60x18.sol` 2 by `@PaulRBerg`. Smart contract library for advanced fixed-point math operating with signed 59.18-decimal and 60.18-decimal fixed-point numbers.

Tx.Origin Authentication
취약점 유형 소개

- tx.origin
  - Global Variable
  - Sender of the Transaction(full call chain)
- 해당 값을 인증으로 사용시 피싱같은 공격에 취약

Tx.Origin Authentication

취약점 예제

```
contract Phishable {
    address public owner;

    constructor (address _owner) {
        owner = _owner;
    }

    function () public payable {} // collect ether

    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
```

Tx.Origin Authentication

취약점 예제

```
contract Phishable {
    address public owner;

    constructor (address _o
        owner = _owner;
    }

    function () public payal

    function withdrawAll(ad
        require(tx.origin ==
        _recipient.transfer
    }
}
```

```
import "Phishable.sol";

contract AttackContract {

    Phishable phishableContract;
    address attacker; // The attackers address to receive funds.

    constructor (Phishable _phishableContract, address _attackerAddress) {
        phishableContract = _phishableContract;
        attacker = _attackerAddress;
    }

    function () payable {
        phishableContract.withdrawAll(attacker);
    }
}
```

Tx.Origin Authentication

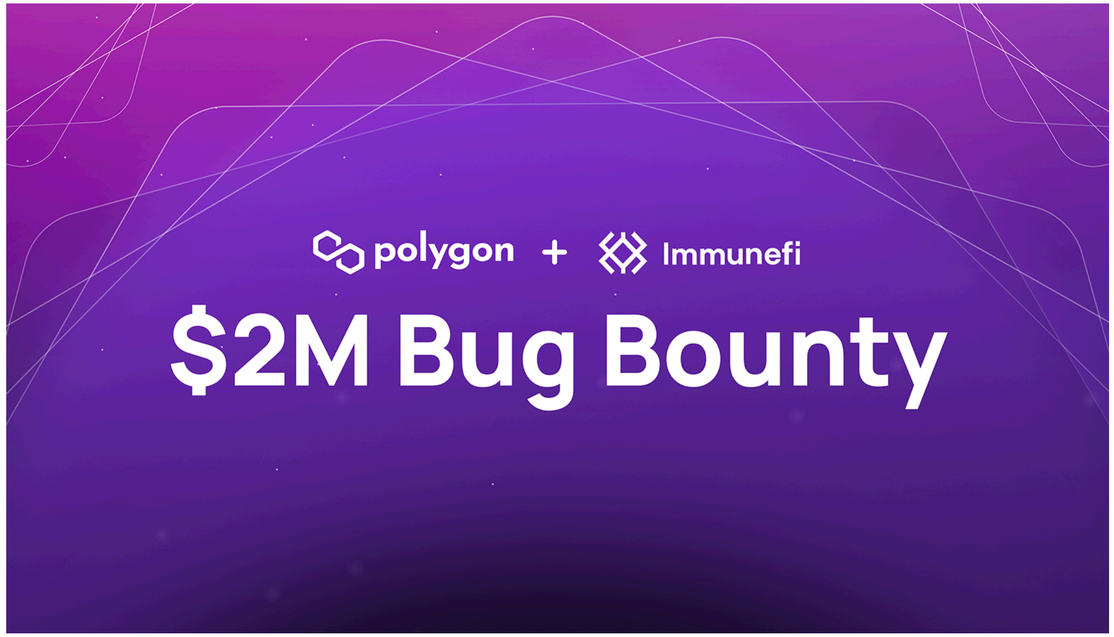
취약점 예제 - 공격 시나리오

- 공격자가 AttackContract 를 배포
- Phishable Contract의 Owner가 Ether를 전송하도록 낚음
- fallback function -> Phishable.withdrawAll(attacker_address)

Tx.Origin Authentication

취약점 방지

- tx.origin은 인증용으로 사용 금지



Case Study

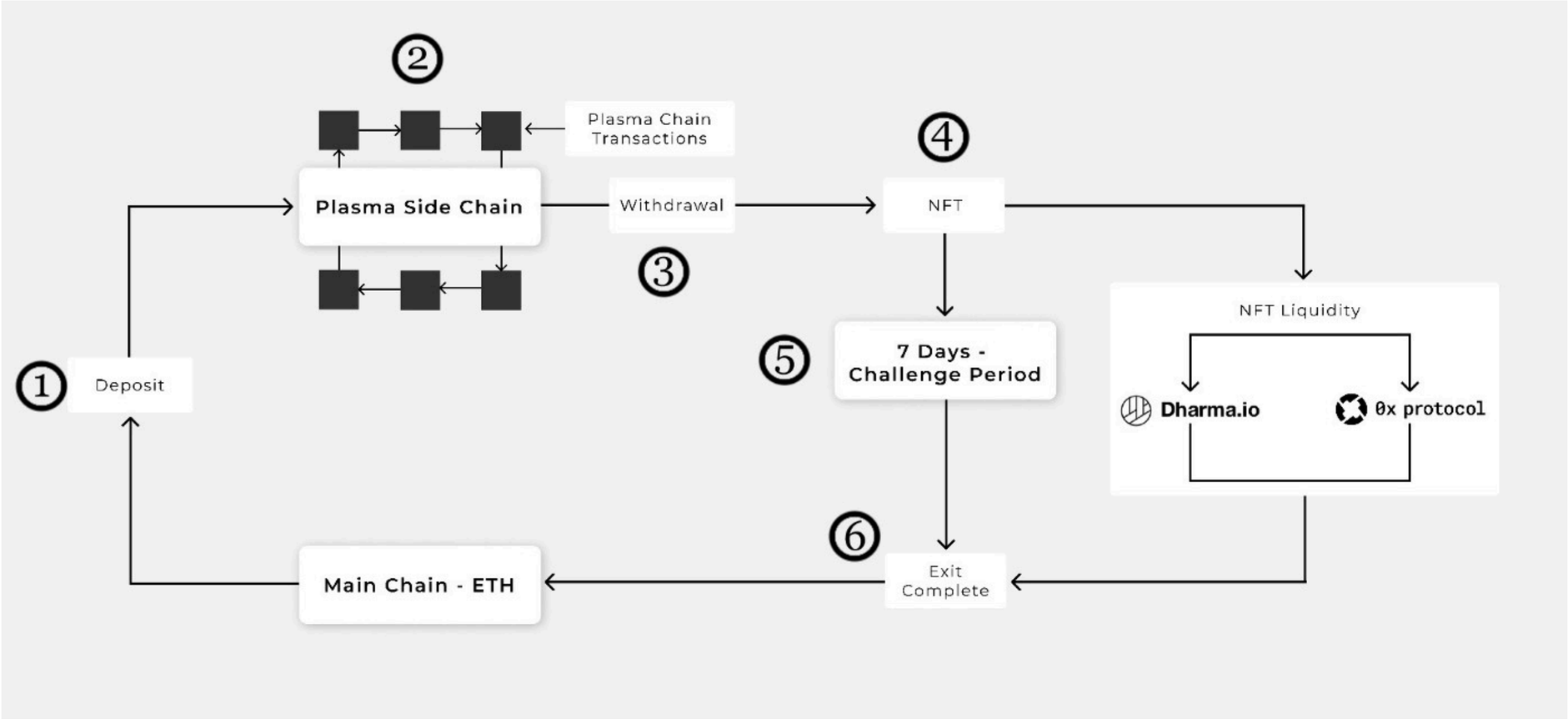
- Double spending bug in Polygon’s Plasma bridge

Polygon’s Plasma bridge

- Polygon은 Ethereum과 Polygon 체인 간 이동을 위한 Bridge를 제공
  - Plasma / PoS

Plasma bridge

High Level Flow of Assets



Plasma bridge

High Level Flow of Assets

1. 사용자가 Ethereum Chain 내 Polygon Contract에 자금을 전송
2. Polygon 네트워크상의 token이 발급
3. 사용자가 Polygon 네트워크에서 자금을 인출하려고 하면
4. 인출하고자 하는 금액의 EXIT NFT Token이 생성
5. 7일간 대기
6. 대기 시간이 끝나면 Ethereum 계좌로 자금을 돌려받음

Plasma bridge

취약점 설명

- 인출 과정이 시작되면 Polygon Token을 태움(Burn)
- Token을 태우는 Transaction이 성공적임이 제대로 확인(Confirm)이 되면 자금 EXIT 과정이 진행
  - Polygon의 WithdrawManager가 해당 기능을 담당

Plasma bridge
취약점 설명

- branchMask
  - Must be unique
  - Used to generate Exit ID
  - Hex Prefix Encoded

```
96      function verifyInclusion(
97          bytes calldata data,
98          uint8 offset,
99          bool verifyTxInclusion
100     )
101         external
102         view
103         returns (
104             uint256 /* ageOfInput */
105         )
106     {
107         ExitPayloadReader.ExitPayload memory payload = data.toExitPayload();
108         VerifyInclusionVars memory vars;
109
110         vars.headerNumber = payload.getHeaderNumber();
111         vars.branchMaskBytes = payload.getBranchMaskAsBytes();
112         vars.txRoot = payload.getTxRoot();
113         vars.receiptRoot = payload.getReceiptRoot();
114         require(
115             MerklePatriciaProof.verify(
116                 payload.getReceipt().toBytes(),
117                 vars.branchMaskBytes,
118                 payload.getReceiptProof(),
119                 vars.receiptRoot
120             ),
121             "INVALID_RECEIPT_MERKLE_PROOF"
122         );
123
124         if (verifyTxInclusion) {
125             require(
```

Plasma bridge

취약점 설명

- MerklePatriciaProof.verify 에서의 Decoding 방식 != verifyInclusion 에서의 Decoding 방식

```
124         if (verifyTxInclusion) {
125             require(
126                 MerklePatriciaProof.verify(
127                     payload.getTx(),
128                     vars.branchMaskBytes,
129                     payload.getTxProof(),
130                     vars.txRoot
131                 ),
132                 "INVALID_TX_MERKLE_PROOF"
133             );
134         }
135 
136         vars.blockNumber = payload.getBlockNumber();
137         vars.createdAt = checkBlockMembershipInCheckpoint(
138             vars.blockNumber,
139             payload.getBlockTime(),
140             vars.txRoot,
141             vars.receiptRoot,
142             vars.headerNumber,
143             payload.getBlockProof()
144         );
145 
146         vars.branchMask = payload.getBranchMaskAsUint();
147         require(
148             vars.branchMask & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00000000 == 0,
149             "Branch mask should be 32 bits"
150         );
151         // ageOfInput is denoted as
152         // 1 reserve bit (see last 2 lines in comment)
153         // 128 bits for exitableAt timestamp
154         // 95 bits for child block number
155         // 32 bits for receiptPos + logIndex * MAX_LOGS + oIndex
156         // In predicates, the exitId will be evaluated by shifting the ageOfInput left by 1 bit
157         // (Only in erc20Predicate) Last bit is to differentiate whether the sender or receiver of the in-fli
158         return (getExitableAt(vars.createdAt) << 127) | (vars.blockNumber << 32) | vars.branchMask;
159     }
```

Plasma bridge

취약점 설명

- MerklePatriciaProof.verify 에서의 Decoding 방식
  - 첫 값이 1이나 3이 아니면 첫 바이트 무시
  - 서로 다른 224개(14x16)의 Encoding이 존재 -> 서로 다른 224개의 Exit ID 생성 가능

```
137             uint8 hpNibble = uint8(_getNthNibbleOfBytes(0, b));
138             if (hpNibble == 1 || hpNibble == 3) {
139                 nibbles = new bytes(b.length * 2 - 1);
140                 bytes1 oddNibble = _getNthNibbleOfBytes(1, b);
141                 nibbles[0] = oddNibble;
142                 offset = 1;
143             } else {
144                 nibbles = new bytes(b.length * 2 - 2);
145                 offset = 0;
146             }
147
148             for (uint256 i = offset; i < nibbles.length; i++) {
149                 nibbles[i] = _getNthNibbleOfBytes(i - offset + 2, b);
150             }
```

```
Encoded  -> Decoded
0x00819c -> 0x0801090c
0x01819c -> 0x0801090c
0x02819c -> 0x0801090c
...
0x20819c -> 0x0801090c
0x21819c -> 0x0801090c
0x22819c -> 0x0801090c
...
0x40819c -> 0x0801090c
0x41819c -> 0x0801090c
0x42819c -> 0x0801090c
..
0xff819c -> 0x0801090c
```

Plasma bridge
취약점 패치

- branchMask의 인코딩 첫 값이 무조건 0이어야 함

1 contracts/root/withdrawManager/WithdrawManager.sol

```
@@ -109,6 +109,7 @@ contract WithdrawManager is WithdrawManagerStorage, IWithdrawManager {
109  109
110  110          vars.headerNumber = payload.getHeaderNumber();
111  111          vars.branchMaskBytes = payload.getBranchMaskAsBytes();
     112  +       require(vars.branchMaskBytes[0] == 0, "incorrect mask");
112  113          vars.txRoot = payload.getTxRoot();
113  114          vars.receiptRoot = payload.getReceiptRoot();
114  115          require(
```

Case Study
기억해볼만한 점

- 다른 사람의 코드(여기서는 Merkle Patricia Proof)를 가져다쓰면서 100% 이해를 못한 것에서 취약점이 기인
  - 다른 Github 프로젝트에서도 사용한 곳을 찾을 수 있음
- 누가 작성했든 나중에는 당신의 코드다! - from Whitehat Hacker

- 기존 SW 개발에서도 많은 오픈소스 프로젝트들을 가져다 사용(-> 공급망 공격)
  - 취약점 탐지의 첫번째 단계가 취약 오픈소스를 사용하는지 확인하는 것!
- 가져다쓰는 코드의 이해 뿐아니라 지속적인 업데이트(특히 보안)에 대한 관심 증가 추세

Reference

- https://blog.sigmaprime.io/solidity-security.html
  - Smart Contract Vulnerability Category
- https://docs.soliditylang.org/en/v0.8.9/
- https://ethereum.org/en/developers/docs/evm/
- https://takenobu-hs.github.io/downloads/ethereum_evm_illustrated.pdf
- https://medium.com/immunefi/polygon-double-spend-bug-fix-postmortem-2m-bounty-5a1db09db7f1
- https://gerhard-wagner.medium.com/double-spending-bug-in-polygons-plasma-bridge-2e0954ccadf1