



Security issues in smart contracts

Lecture 7



- 1) Main vulnerabilities in smart contracts
- 2) Vulnerabilities in Algorand smart contracts
- 3) Practical recommendations for ensuring the security of smart contracts
- 4) Optimising the performance of smart contracts on Algorand



Security is the most important aspect of smart contract development. If a smart contract is hacked, the costs can be extremely high.

Therefore, when building smart contracts, it is necessary to ensure that the following **criteria are met**:

- Minimalism
- Code reuse
- Code quality
- Auditability
- Test coverage

Overcomplication is not good for security. Errors are less likely to occur if the code **is simple, short, and performs only one function**. Always try to keep **the code as simple as possible** to avoid unwanted effects that could lead to security loopholes.

Simpler means safer.

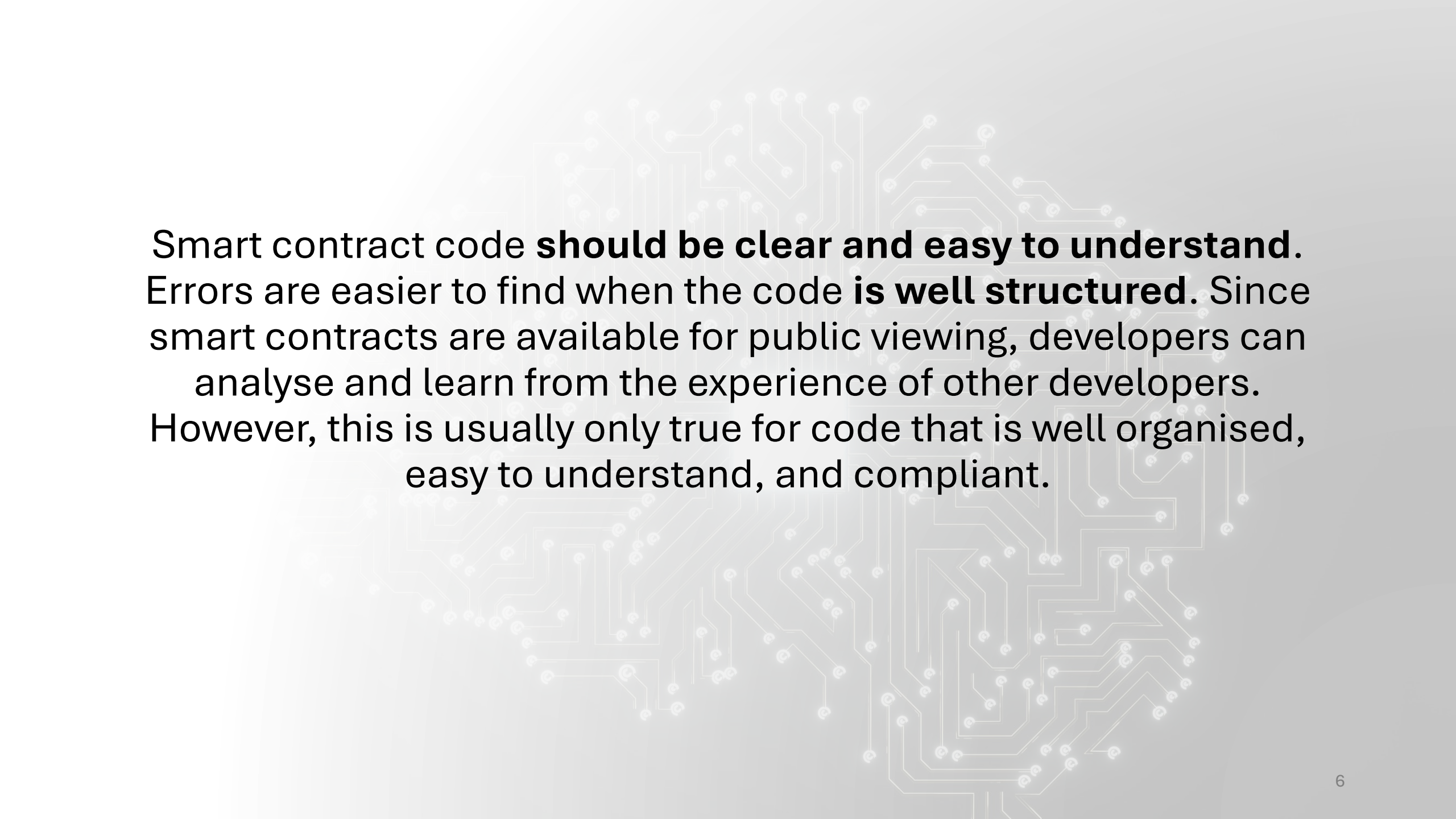
+

•

○

Developers are encouraged to use existing libraries or contracts whenever possible. This helps reduce the risk of security vulnerabilities in new contracts. Smart contracts that are **already** in use and **tested** by others are much **more secure** than new ones.

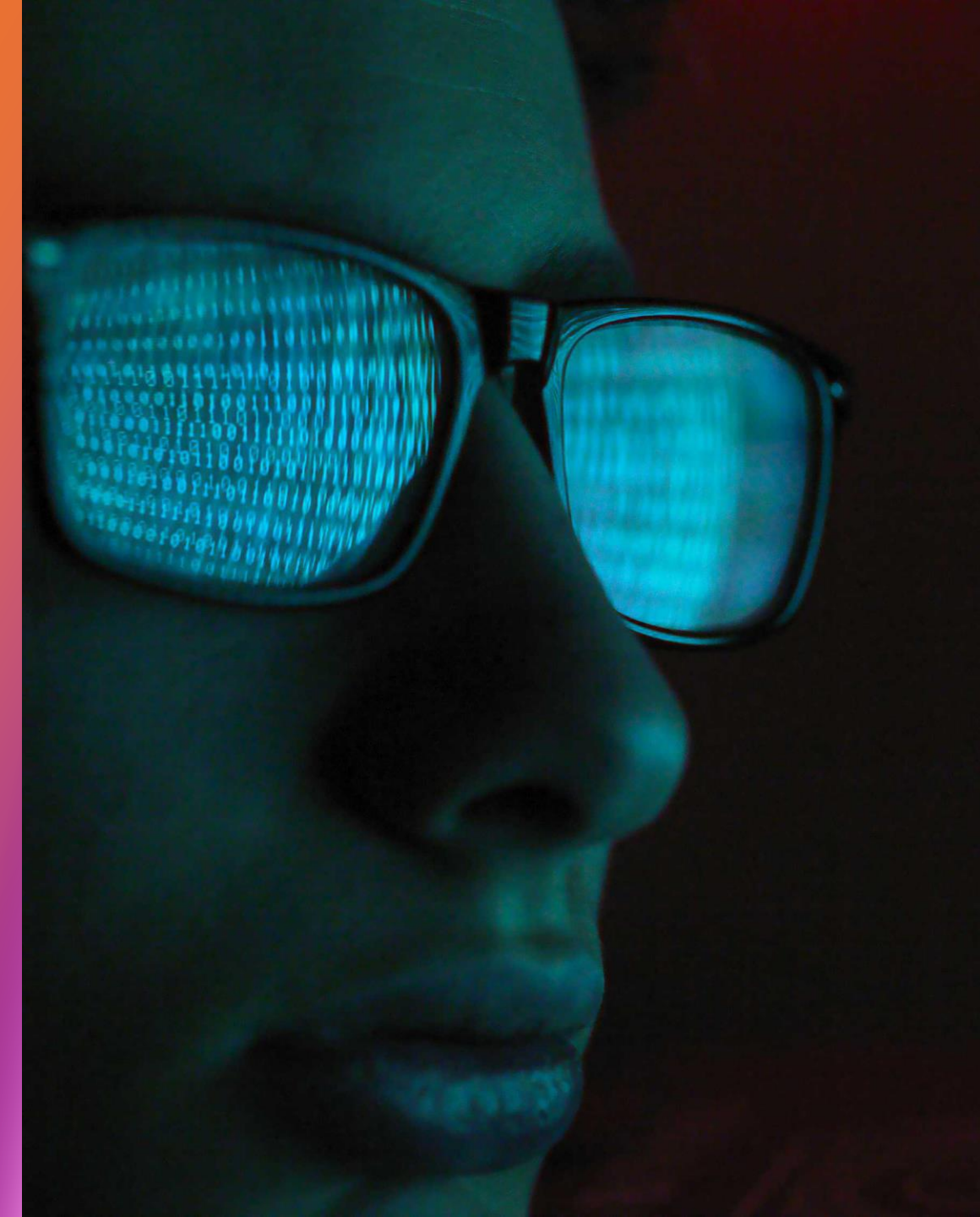
Therefore, reuse the code to reduce the risk.



Smart contract code **should be clear and easy to understand**. Errors are easier to find when the code **is well structured**. Since smart contracts are available for public viewing, developers can analyse and learn from the experience of other developers. However, this is usually only true for code that is well organised, easy to understand, and compliant.

The main vulnerabilities in smart contracts

Vulnerabilities in smart contracts are weaknesses or errors in the code of smart contracts that can be exploited by attackers to carry out attacks or misconduct. Vulnerabilities can lead to undesirable consequences, such as loss of funds, distortion of business logic, unauthorised execution of transactions, or even complete deletion of the contract.



Reentrancy Attacks

A re-entry attack exploits a vulnerability in smart contracts where a function makes an external call to another contract before updating its own state. This allows the external contract, possibly a malicious one, to re-enter the original function and repeat certain actions, such as withdrawals, using the same state. With such attacks, an attacker can extract all funds from the contract.

Example: The most famous example of this was the DAO hack, which resulted in the withdrawal of \$70 million worth of ether.

+

•

○

Reentrancy in Algorand smart contracts

In Algorand, AVM has built-in restrictions that **prohibit the reentry** of smart contract functions. This means that if the smart contract code tries to re-call the same smart contract during another operation, the system will automatically block this action.

Denial of service (DoS) attacks

A Denial of Service (DoS) attack is a type of attack aimed at making a particular service, system, or smart contract unavailable to users. In the context of blockchains and smart contracts, a DoS attack usually consists of depleting the contract's resources, which leads to the inability to perform its functions.

Smart contracts have limited resources, such as the number of computational operations (gas), memory, or balance, that can be used to execute transactions. If an attacker finds a way to use up these resources before they are completely exhausted, the smart contract will no longer be able to process new transactions.

DoS attack. Features of Algorand

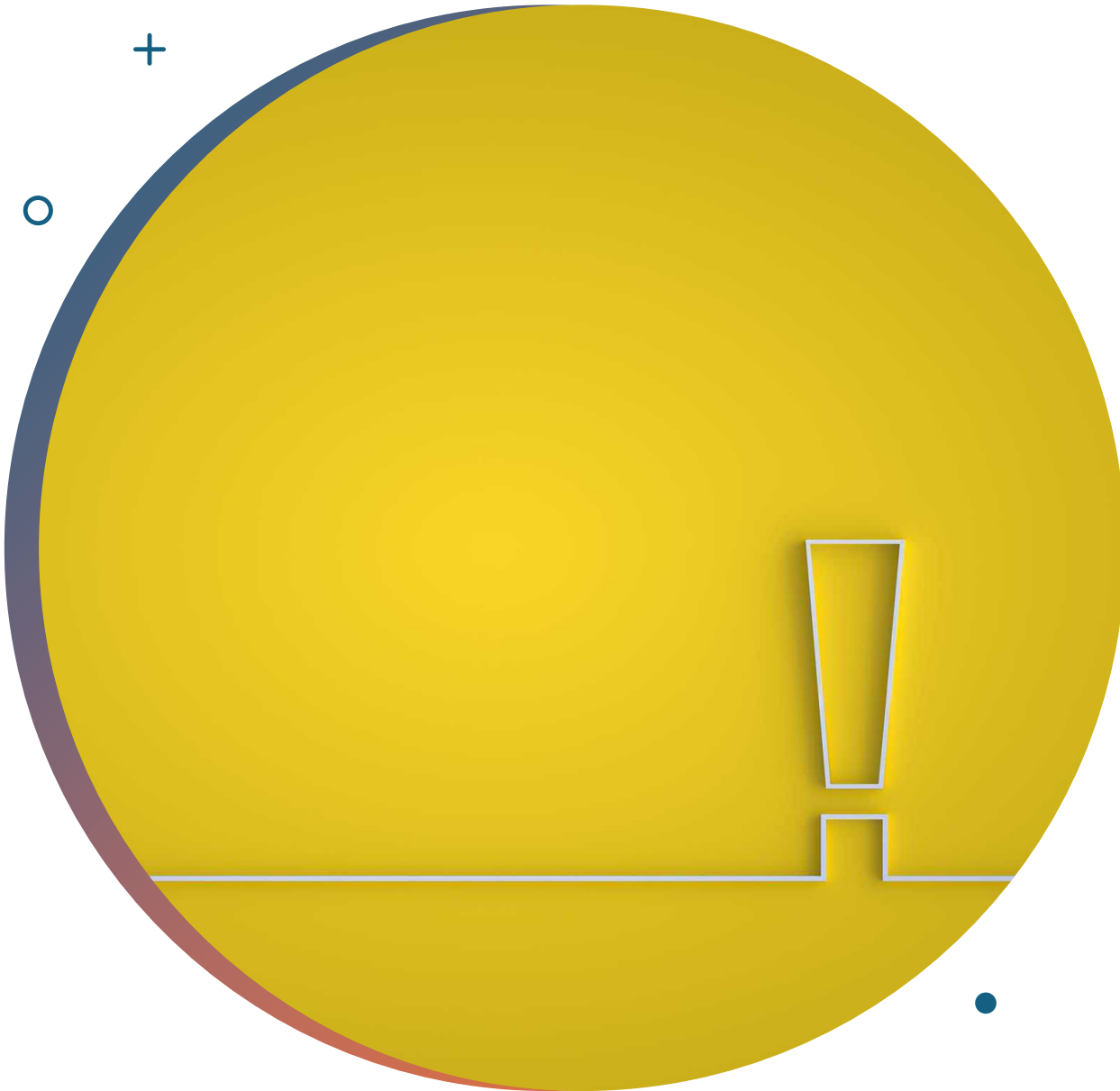
Gas limit: Algorand does not use the concept of "gas" like Ethereum, but has a limit on the number of instructions (opcode) for each smart contract. This limit can help avoid some types of DoS attacks, as an attacker cannot execute an infinite number of instructions.

Atomic Transaction Groups: Algorand allows you to combine multiple transactions into groups that are executed atomically (all transactions are executed or none). This can be used for secure transaction validation, but if the group size or transaction types are not checked, it can be a vulnerability to a DoS attack.

Payment mechanism: in Algorand, all transactions must be paid for, which reduces the likelihood of attacks as the attacker will have to spend real money. However, an attacker can still try to drain the smart contract's resources by sending a large number of transactions if there are weaknesses in the contract code.

Possible scenarios of a DoS attack on Algorand

- **Group Transaction Attack:** If the contract does not check the size of the transaction group or the types of transactions, an attacker can add additional transactions to the group, which can lead to unintended consequences such as resource consumption or blocking access to the contract.
- **Depleting the contract balance:** If the contract performs internal transactions, an attacker may attempt to deplete the contract balance through unlimited transactions or through poor management of transaction details.



Logical errors

Logical errors, also known as business logic vulnerabilities, are hidden flaws in smart contracts. They occur when the contract code **does not match the expected behaviour or business logic** that was intended when it was designed. These errors are insidious because they can go unnoticed for a long time until someone discovers and exploits them.

Logical errors

Impact: Logical errors can cause a smart contract to behave in unexpected ways or even render it completely unusable. Such errors can result in the loss of funds, misallocation of tokens, or other negative consequences that can lead to serious financial and operational losses for users and stakeholders.

Correction:

- Always test your code by creating comprehensive test cases that cover all possible variants of business logic.
- Conduct thorough code reviews and audits to identify and fix potential logic errors.
- Document the expected behaviour of each feature and module, and then compare it to the actual implementation to ensure they are consistent.



Lack of or improper validation of input data

If a smart contract does not properly validate data **entered by users or received from external sources**, it may allow attackers to manipulate the contract logic. This could include passing incorrect parameters or values that could lead to undesirable behaviour, such as loss of funds or compromising the security of the contract.

Vulnerabilities in Algorand smart contracts

Missing transaction group size check

If the smart contract does not check the size of the transaction group, attackers can add their transactions to the group. This can lead to loss of funds if these additional transactions include asset transfers.

Prevention: Always check the size of the transaction group and define clear terms and conditions for each transaction in the group.

```
def check_customer_payment():
    return Seq(
        Assert(Global.group_size() == Int(2)), # Check if the group size is 2
        Assert(Gtxn[0].asset_amount() == Int(50)), # Check if the asset amount is 50
        Assert(Gtxn[0].xfer_asset() == Int(1063265)), # Check if the asset ID is 1063265
        Assert(Gtxn[0].asset_receiver() == Addr("2ERPX0LR7IESFS6FWX45CDA4Z3447SPNMC6YQHF2UXOXRRNLX5KZAMT4DI")),
        Assert(Gtxn[0].rekey_to() == Global.zero_address()), # Check if RekeyTo is the zero address
        Approve()
    )
```

+

•

○

Missing access control

If the smart contract code does not contain checks for application calls such as `UpdateApplication` and `DeleteApplication`, an attacker can modify the application code or delete it completely.

Prevention: always add smart contract-level checks for calls that can modify or delete an app. For example, that only the `creator_address` can delete or modify the app.

```
def application_deletion(self):  
    return Return(Txn.sender() == Global.creator_address())
```

Asset ID verification is missing

If a smart contract does not verify the ID of the asset it interacts with, an attacker can manipulate the contract logic by substituting a fake or other (less valuable) asset for the correct one.

Precaution: always check the asset identifiers of the assets with which the smart contract works.

```
def check_customer_payment():
    return Seq(
        Assert(Global.group_size() == Int(2)), # Check if the group size is 2
        Assert(Gtxn[0].asset_amount() == Int(50)), # Check if the asset amount is 50
        Assert(Gtxn[0].xfer_asset() == Int(1063265)), # Check if the asset ID is 1063265
        Assert(Gtxn[0].asset_receiver() == Addr("2ERPXOLR7IESFS6FWX45CDA4Z3447SPNMC6YQHF2UXOXRRNLX5KZAMT4DI")),
        Assert(Gtxn[0].rekey_to() == Global.zero_address()), # Check if RekeyTo is the zero address
        Approve()
    )
```

+

•

○

No verification of internal transaction fees

If a smart contract does not explicitly set the amount of the fee for an internal transaction, an attacker can create operations that perform internal transactions and burn through the app's balance in the form of fees.

Prevention: always check and explicitly set the fees for internal transactions. The simplest solution for the internal transaction fee is to set it to 0 and rely on fee pooling.

RekeyTo parameter is not checked

If the smart contract does not have a RekeyTo check, an attacker can set this parameter to his or her address, which will allow him or her to directly control the contract assets or take control of the signature account.

Prevention: always check the RekeyTo parameter in transactions.

```
def check_customer_payment():
    return Seq(
        Assert(Global.group_size() == Int(2)), # Check if the group size is 2
        Assert(Gtxn[0].asset_amount() == Int(50)), # Check if the asset amount is 50
        Assert(Gtxn[0].xfer_asset() == Int(1063265)), # Check if the asset ID is 1063265
        Assert(Gtxn[0].asset_receiver() == Addr("2ERPXOLR7IESFS6FWX45CDA4Z3447SPNMC6YQHF2UXOXRRNLX5KZAMT4DI")),
        Assert(Gtxn[0].rekey_to() == Global.zero_address()), # Check if RekeyTo is the zero address
        Approve()
    )
```

No verification of the transaction receiver

In Algorand smart contracts, an atomic transaction group can be used to validate payment transactions, which links payment transactions to application calls. If a smart contract does not verify the receiver of a payment or asset transfer transaction, an attacker can specify a different recipient address.

Prevention: always verify the receiver's address in transactions.

```
def check_customer_payment():
    return Seq(
        Assert(Global.group_size() == Int(2)), # Check if the group size is 2
        Assert(Gtxn[0].asset_amount() == Int(50)), # Check if the asset amount is 50
        Assert(Gtxn[0].xfer_asset() == Int(1063265)), # Check if the asset ID is 1063265
        Assert(Gtxn[0].asset_receiver() == Addr("2ERPXOLR7IESFS6FWX45CDA4Z3447SPNMC6YQHF2UXOXRRNLX5KZAMT4DI")),
        Assert(Gtxn[0].rekey_to() == Global.zero_address()), # Check if RekeyTo is the zero address
        Approve()
    )
```



+

•

○

Overflow and underflow (Overflow/Underflow)

In the Algorand Virtual Machine (AVM), by default, the system stops the execution of a contract in case of overflow, underflow, or division by zero, which results in a transaction error. To prevent this, you can add restrictions on the values of the variables involved in transactions.

Warning: Always add checks on variable values that could lead to overflow or underflow.

+ Practical recommendations for • ensuring the security of Algorand ◦ smart contracts

Correct OnComplete handling

It is important not to allow the method you created to be called if the value of "OnComplete" is not "NoOp".

Recommendation: At the beginning of each entry into a smart contract, you should branch the code based on the OnComplete value, and only if it is **NoOp**, proceed to the execution of a specific method.

```
program = Cond(  
    [Txn.application_id() == Int(0), handle_creation],  
    [Txn.on_completion() == OnComplete.OptIn, handle_optin],  
    [Txn.on_completion() == OnComplete.CloseOut, handle_closeout],  
    [Txn.on_completion() == OnComplete.UpdateApplication, handle_update],  
    [Txn.on_completion() == OnComplete.DeleteApplication, handle_delete],  
    [Txn.on_completion() == OnComplete.NoOp, handle_noop],  
)
```

And already in `handle_noop`, you can call a specific smart contract method.

Minimising the number of entry points

The entry points (or methods) in a smart contract are the places where the contract can be invoked to perform a specific action. The more such entry points there are, the more difficult it is to test and maintain the contract, and the more opportunities for error or abuse.

Recommendation:

Reducing the number of entry points to the most essential ones simplifies testing and increases contract security. This allows you to more clearly define what actions are possible at any given time and more easily verify that they are performed correctly.



Optimising the performance of smart contracts on Algorand

Limitations on contract size and performance costs

Algorand has a [budget for transactions](#), which limits the number of transactions that can be performed in one contract.

Recommendation: Pay attention to the cost of opcodes and avoid using expensive operations, such as byte operations, if possible.

+

•

○

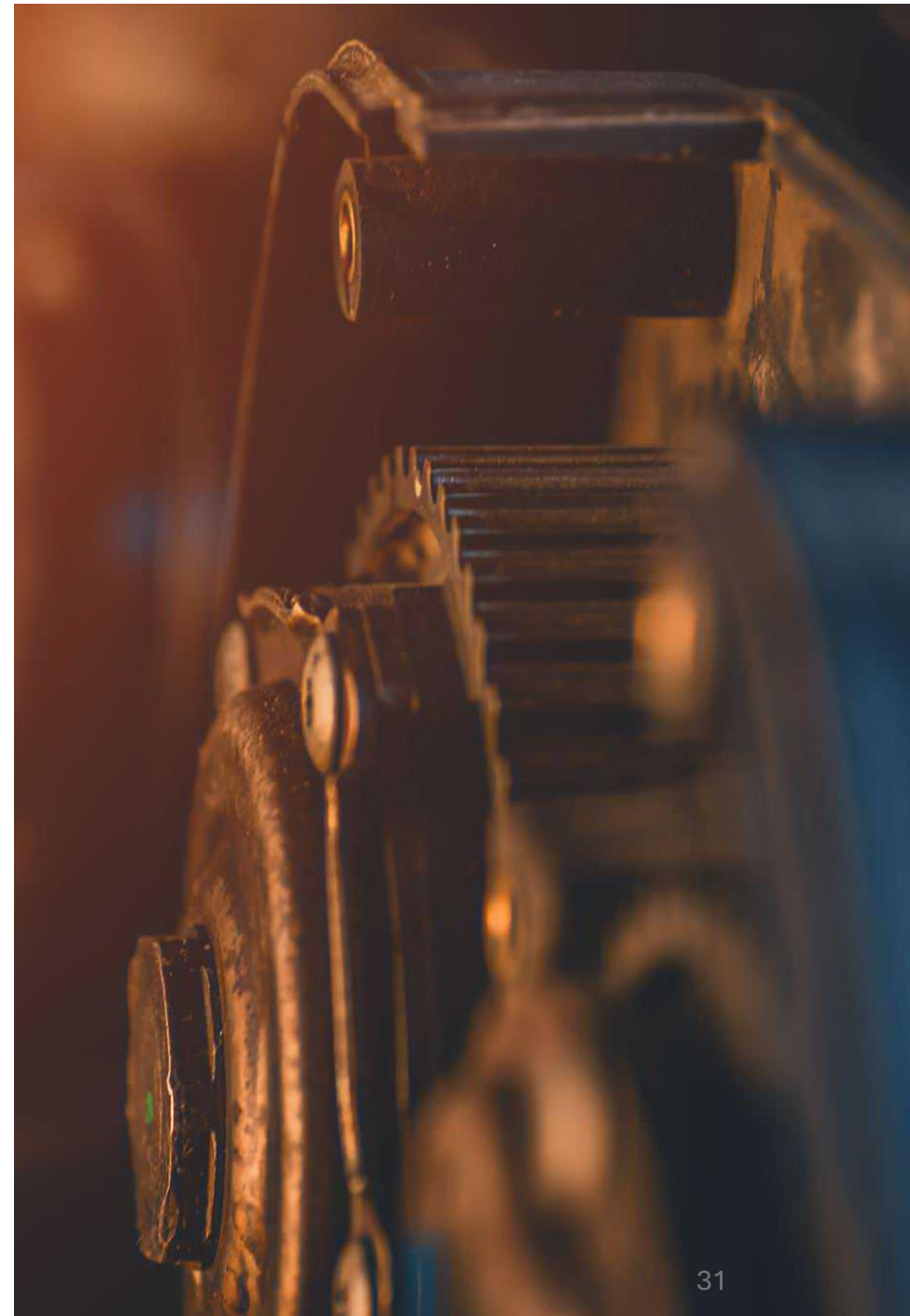
Example

A simple operation like adding two numbers will have a cost of 1. Of course, loading those numbers will have its own cost, but the simple addition costs 1 unit. When you add two bytes, the cost is 10. But when you want to calculate a hash of something, the cost skyrockets into the hundreds.

Moving complex computing off-chain

The limit on the number of operations (700 operations per transaction) in Algorand dictates the need to perform complex calculations outside the smart contract (off-chain). This reduces the cost of transactions and increases the efficiency of the contract.

Recommendation: move complex computations off-chain if possible to optimise the use of smart contract resources.



+

•

○

Example

Imagine that you need to find the shortest path in a graph. This task may be too complex to be performed within a smart contract due to the limitations on the number of operations. Instead, the path can be found on the backend, and the result can be passed to the contract for further verification and execution.

Avoid using byte maths

Byte operations in Algorand can be convenient in certain cases. For example, they allow you to work with larger numbers and wider integers than the standard `uint64`. However, these operations are **much more expensive** in terms of computational cost in smart contracts, which can greatly affect the efficiency of contract execution.

Recommendation: avoid using byte operations when the task can be solved using **standard `uint64`**. This will reduce the cost of contract execution and make it more efficient.



Minimise the size of the contract

Avoid storing large amounts of data in a smart contract and use the most compact code possible to reduce the size of the contract and reduce the cost of its execution.



Standard safety rules

NEVER store a key or mnemonic in the source code. It is almost impossible to safely remove a key or mnemonic from a git repository.

Avoid using unknown or untested libraries as they may contain vulnerabilities.

Use Algorand's test network to test your smart contract before deploying it to your production network to identify potential vulnerabilities.

Conduct regular audits.

Monitor changes in the Algorand ecosystem and update your smart contract to meet new security standards.

Avoid storing confidential data in the contract

Smart contracts are transparent and available for viewing by all network participants. Storing confidential data in a contract can lead to information leakage.

Rule: do not store private keys, passwords, or other sensitive data directly in the smart contract. Use encryption if necessary.

+

•

○

Document and test the code

Without proper documentation and testing, bugs and vulnerabilities can arise that are difficult to detect in the early stages.

Rule: always document the functions and modules of your code. Write tests that cover all possible scenarios for using the contract.

Additional materials

- [Modern guidelines for smart contracts and smart signatures on Algorand](#): official Algorand documentation
- [\(Not So\) Smart Contracts](#): This repository contains examples of common vulnerabilities in Algorand smart contracts, including code from real smart contracts.

