Contents

Introduction

Blockchain technology has revolutionized the field of finance, security and decentralization, enabling transactions that are transparent and immutable without the need for trusted intermediaries.Algorand is one of the next-generation blockchains that aims to solve the blockchain trilemma (scalability, security and decentralization) thanks to its protocol based on **PureProof of Stake (PPoS)** . This approach ensures confirmation of transactions quickly and securely, eliminating fork risk and increasing network resilience .

Algorand is a layer 1 blockchain that uses a mechanism of consensus based on PPoS. Unlike other blockchains that require high computational requirements (such as Bitcoin' s Proof of Work),Algorand randomly selects network validators based on their amount of ALGOs possessed, making the system more efficient and sustainable.

Main features of Algorand:

- **High scalability**: confirmation of blocks in seconds.
- **Advanced cryptographic security**: based on VRF (Verifiable Random Function) signatures .
- **Decentralization**: any ALGO holder can participate in the validation of transactions.
- **Support for smart contracts and tokenized assets**: Algorand provides advanced tools for the creation of decentralized applications (dApp) and digital assets .

However, like any technology, Algorand also has security vulnerabilities that must be carefully analyzed and mitigated. In this paper , some of the major vulnerabilities related to smart contracts and smart signatures by Algorand, highlighting how they can be exploited by attackers and what solutions can be implemented to mitigate them.

The information in this document is based on the vulnerability analysis reported at https://secure-contracts.com/not-so-smart-contracts/algorand/.

Smart Contract & Smart Signatures

Smart contracts in Algorand, known as **Algorand SmartContracts (ASC1)** , allow programmed logic to be executed on the blockchain in a secure and automated manner. They can be written using **PyTeal** , a Python-based language, or directly in **TEAL (Transaction Execution Approval Language)**.

Smart Signatures (or Stateless Smart Contracts) are TEAL scripts used to verify transaction conditions before approval, without maintaining state on the blockchain.

Example of a Smart Contract in TEAL

```
#pragma version 2

txna ApplicationArgs 0
byte "HelloWorld"
==
```

Questo semplice contratto verifica se l'argomento della transazione è `"HelloWorld"` e approva la transazione solo se la condizione è soddisfatta.

Example of Smart Signature in PyTeal

An example of **smart signature** in PyTeal that requires a signature with a specific public key:

```
from pyteal import *

def approval_program():
```

```
    return Txn.sender() == Addr("SPECIFIC_PUBLIC_KEY")

print(compileTeal(approval_program(), mode=Mode.Signature, version=2))
```

Differences between Smart Contracts and Smart Signatures

- **Smart Contracts (Stateful):** can store data on the blockchain and are used for more complex applications.
- **Smart Signatures (Stateless):** are used for verification of transactions without storing data on the blockchain.

Vulnerabilities

Rekeying

Description

Rekeying is a feature of Algorand that allows you to change the public key associated with an account without changing the balance or the permissions of the account itself. This mechanism is useful for security, but if not handled properly it can be exploited by attackers to take control of an account and divert funds.

Rekeying-related vulnerabilities can arise from:

- The absence of controls in the **smart contracts** and in the **smart signatures** to verify the legitimate owner of the account.
- The use of a malicious rekeying address that allows the attacker to sign transactions without the consent of the original owner.

Example

Imagine a PyTeal smart contract that **does not verify correctly the sender**, allowing an attacker to reassign the account signing key to another address.

Example of vulnerable code

```
from pyteal import *

def vulnerable_contract():
    return Txn.rekey_to() == Global.zero_address()

if __name__ == "__main__":
    print(compileTeal(vulnerable_contract(), mode=Mode.Signature, version=2))
```

In this case, the smart contract allows rekeying **without any control** , leaving room for possible exploits.

Exploitation

1. The attacker identifies an account using a smart contract that is vulnerable.
2. Performs a transaction that **resets the account key to an address controlled by the attacker**.
3. Once rekeying has been performed, the attacker can sign any transaction and steal the funds.

Solution

To prevent rekeying-based attacks , it is necessary to **block the ability to reassign the key** in smart contracts.

Example of safe code

```
from pyteal import *

def secure_contract():
    return Assert(Txn.rekey_to() == Global.zero_address())

if __name__ == "__main__":
    print(compileTeal(secure_contract(), mode=Mode.Signature, version=2))
```

This code prevents any attempt at rekeying,making the account **non-transferable to another private key**.

Conclusion

Rekeying can be a useful feature, but it must be managed carefully in smart contracts to avoid attacks. By implementing appropriate controls, it is possible to mitigate this vulnerability and ensure that only the legitimate account owner can sign transactions.

Unchecked Transaction Fees

Description

In smart contracts based on Algorand, the **smart signatures** can be used to approve transactions without the need for a traditional private key. However , if not implemented appropriate controls on transaction fees (`txn.fee`), an attacker can manipulate costs and overcharge the account owner.

Vulnerabilities arise from:

- **Absence of an upper limit** for transaction fees.
- Lack of verification to ensure that the user does not pay a excessive fees.
- Ability for an attacker to create transactions with **extremely high fees**, depleting the account balance.

Example

The following smart contract in PyTeal approves any transaction without controlling the fees:

```
from pyteal import *

def unchecked_fees():
    return Approve()

print(compileTeal(unchecked_fees(), mode=Mode.Signature, version=2))
```

What is the problem?

- The contract **Does not verify the value of the commission** (`txn.fee`), allowing an attacker to **set exorbitant rates**.
- If it is used in a smart signature for automatic transactions, it can deplete the account balance without control.

Exploitation

1. The attacker sends a transaction using the vulnerable smart signature.
2. Set a **high commission** (e.g., 1 ALGO per transaction).
3. After a few transactions, the account balance is **completely drained** by fees.

Solution

To avoid this vulnerability, it is necessary to **set an upperlimit on acceptable fees** .

Example of Secure Code

```
from pyteal import *

MAX_FEE = Int(1000)  # Set a cap on commissions (1000 microAlgos)

def safe_fees():
    return And(
        Txn.fee() <= MAX_FEE,  # Verify that the fee does not exceed the maximum
        ↪  allowed
        Approve()
    )

print(compileTeal(safe_fees(), mode=Mode.Signature, version=2))
```

Why is it safe?

- **Prevents account from being emptied** by high fees.
- **Limit the maximum amount** accepted for commissions.
- **Blocks any attempted abuse** by an attacker.

Conclusion

The absence of controls on transaction fees represents a significant risk. Developers must always implement limits on fees to avoid exploits and protect accounts that use smart signatures.

Closing Account

Description

Algorand allows users to **close an account** and transfer the entire balance to another address. If a smart signature does not properly check the field `txn.close_remainder_to`, an attacker can close the account and **steal all available funds**.

Vulnerabilities arise from:

- Lack of control that **prevents the closure of the account** .
- Ability for an attacker to **redirect all funds** to its own address.

Example

The following smart contract in PyTeal does not check the field `Txn.close_remainder_to()`, allowing anyone to close the account:

```
from pyteal import *

def unsafe_closing():
    return Approve()  # No control on close_remainder_to

print(compileTeal(unsafe_closing(), mode=Mode.Signature, version=2))
```

What is the problem?

- The contract approves any transaction , even if it includes a value for `Txn.close_remainder_to()`.
- An attacker can send a transaction with `Txn.close_remainder_to()` set to an **address under its control**, emptying the account.

Exploitation

1. The attacker identifies an account using the smart signature that is vulnerable.
2. Creates a transaction in which sets `Txn.close_remainder_to()` at its **own address**.
3. Since the contract does not block this operation, **all the funds are transferred to the attacker** .

Solution

To avoid this vulnerability, you must **block any transaction that attempts to close the account**.

Example of Secure Code

```
from pyteal import *

def safe_closing():
    return And(
        Txn.close_remainder_to() == Global.zero_address(),  # Impedisce la
        ↪  chiusura dell'account
        Approve()
    )

print(compileTeal(safe_closing(), mode=Mode.Signature, version=2))
```

Why is it safe?

- **Verify that `Txn.close_remainder_to()` is set to `Global.zero_address()`**, preventing the transfer of the balance remaining.
- **Protects account funds** from closure attacks fraudulent.

Conclusion

The ability to close an account without restrictions represents a serious risk to smart contracts based on smart signatures.Implementing a check on `Txn.close_remainder_to()` is essential to prevent the loss of funds.

Closing Asset

Description

In Algorand, an **asset** (ASA - Algorand Standard Asset) can be removed from an account if the entire balance of the assets to an other address. If a smart signature does not properly verify the field `Txn.asset_close_to()`, an attacker can **transfer the entire balance** of an asset to an address of his own, emptying the account.

Vulnerabilities arise from:

- Lack of control over `Txn.asset_close_to()`, allowing anyoneto close the asset and transfer all available tokens .
- Ability for an attacker to **hijack assets without the consent of the owner**.

Example

The following smart contract in PyTeal **does not check the field `Txn.asset_close_to()`**, allowing the closure of assets without restrictions:

```
from pyteal import *

def unsafe_closing_asset():
    return Approve()  # No control on asset_close_to

print(compileTeal(unsafe_closing_asset(), mode=Mode.Signature, version=2))
```

What is the problem?

- The contract **approves any transaction** , including one that closes an asset by transferring it to another address.
- An attacker can set `Txn.asset_close_to()` at an **own address** and empty the account of the assets.

Exploitation

1. The attacker identifies an account with a smart signature that is vulnerable.
2. Create an ASA transaction with `Txn.asset_close_to()` set to its **own address** .
3. Since the contract does not block this operation, **all the tokens of the asset are transferred to the attacker**.

Solution

To avoid this vulnerability, you must **block any transaction that attempts to close an asset**.

Example of Secure Code

```
from pyteal import *

def safe_closing_asset():
    return And(
        Txn.asset_close_to() == Global.zero_address(),  # Prevents the closure
        ↪ of the asset
        Approve()
    )

print(compileTeal(safe_closing_asset(), mode=Mode.Signature, version=2))
```

Why is it safe?

- **Verify that `Txn.asset_close_to()` is set to `Global.zero_address()`**, preventing the transfer of the entire balance of the asset.
- **Protects tokens from fraudulent transfers** without the consent of the owner.

Conclusion

The lack of control over `Txn.asset_close_to()` allows an attacker from **empty the assets of an account without authorization** . Implementing a control that prevents the closure of assets is essential to protect smart contracts based on smart signatures.

Group Size Check

Description

Algorand supports **group transactions** , in which multiple transactions are sent and processed together atomically. However, if a smart contract or a smart signature does not verify the size of the transaction group (`Txn.group_index()` and `Global.group_size()`), an attacker can:

- **Manipulating contract behavior** by sending isolated transactions instead of in an expected group.
- **Escape security restrictions** designed to work only in a specific group.

Vulnerabilities arise from:

- **Lack of control over `Global.group_size()`**, allowing the operation of the contract even when it is sent in isolated.
- **Lack of control of `Txn.group_index()`**, allowing the execution of the transaction with an unwanted order .

Example

The following smart contract in PyTeal does not verify the size of the group and allows a transaction to be executed even if it is not part of an intended group:

```
from pyteal import *

def unsafe_group_check():
    return Approve()  # No control on the size of the group

print(compileTeal(unsafe_group_check(), mode=Mode.Application, version=2))
```

What is the problem?

- The contract **approves any transaction** , even if it is **sent alone** instead of in a group.
- If the contract is designed to work only at within a group (e.g., conditional payment or atomic swap), an **attacker can circumvent restrictions** by sending transactions out of context.

Exploitation

1. The contract expects a **group transaction** for a conditional payment or a sequence of actions.
2. Instead, the attacker sends a **isolated transaction** , exploiting the lack of control over `Global.group_size()`.
3. The contract **still accepts** , allowing the attacker to circumvent the rules provided.

Solution

To avoid this vulnerability, it is necessary to **verify that the transaction is part of a group of the expected size and that the index in the group is correct**.

Example of Secure Code

```
from pyteal import *

EXPECTED_GROUP_SIZE = Int(2)  # For example, we always expect 2 transactions in
 ↪  the group
EXPECTED_INDEX = Int(0)  # The contract should always be the first transaction
 ↪  in the group

def safe_group_check():
    return And(
        Global.group_size() == EXPECTED_GROUP_SIZE,  # Check the size of the
         ↪  group
        Txn.group_index() == EXPECTED_INDEX,  # Controls the order of the
         ↪  transaction in the group
        Approve()
    )

print(compileTeal(safe_group_check(), mode=Mode.Application, version=2))
```

Why is it safe?

- **Prevents transaction execution if it is not part of a group of the correct size**.
- **Verifies that the transaction is in the expected position within the group**.
- **Protects smart contracts that depend on group transactions** from abuse and manipulation.

Conclusion

The absence of controls on group size and order of the transactions can lead to **unwanted executions and bypassing of security restrictions**. Implement audits on `Global.group_size()` and `Txn.group_index()` is essential to ensure that contracts function only in the intended context.

Time-based Replay Attack

Description

In Algorand , smart signatures can be used to authorize periodic transactions, such as recurring payments or pre-authorized. However, **if a contract does not use a lease value (`Txn.lease()`) to prevent the repetition of the same transaction**, an attacker can **reuse a signed transaction multiple times** to drain funds or perform unauthorized transactions.

Vulnerabilities arise from:

- **Failure to use the field `Txn.lease()`**, which prevents the repetition of the same transaction.
- **Possibility for an attacker to capture and repeat a legitimate transaction**, repeatedly obtaining a payment or action.

Example

The following smart contract in PyTeal **does not set a lease value** , allowing the transaction to be repeated:

```
from pyteal import *

def unsafe_replay():
    return Approve()  # No control over replay of transaction

print(compileTeal(unsafe_replay(), mode=Mode.Signature, version=2))
```

What is the problem?

- The contract **does not check whether the transaction has already been executed** , allowing indefinite replay.
- An attacker can **intercept a signed transaction** and reuse it several times, obtaining repeated payments.

Exploitation

1. A user authorizes a transaction for a recurring payment.
2. An attacker captures the transaction and resends it several times.
3. The contract **approves it each time** , allowing the attacker to receive more funds than he should.

Solution

To avoid this vulnerability, you must **use the field `Txn.lease()`**,which ensures that a transaction cannot be repeated.

Example of Secure Code

```
from pyteal import *

LEASE_ID = Bytes("unique-lease-id")  # A unique value that distinguishes the
 ↪  transaction

def safe_replay():
    return And(
        Txn.lease() == LEASE_ID,  # Impedisce la ripetizione della stessa
         ↪  transazione
        Approve()
    )

print(compileTeal(safe_replay(), mode=Mode.Signature, version=2))
```

Why is it safe?

- **It uses `Txn.lease()` to prevent the repetition of a transaction** already executed.
- **Each transaction will have a unique lease** , preventing replay by an attacker.
- **Protects periodic transactions** , such as recurring payments or one-time authorizations.

Conclusion

Failure to implement `Txn.lease()` exposes smart contracts to **time replay attacks**, allowing an attacker to repeat an already signed transaction. Using a unique lease value is an essential security measure to protect transactions that are sensitive.

Access Controls

Description

Smart contracts on Algorand allow developers to **create decentralized applications** (dApps) with functions to read and writing data . However, if the contract **does not implement appropriate access controls** , an attacker can:

- **Change critical data** of the application without permission.
- **Delete or update** the application arbitrarily.

Vulnerabilities arise from:

- **Failure to verify user identity (`Txn.sender()`)** when performing sensitive operations.
- **No control over who can update or delete the application** , allowing anyone to modify it.

Example

The following smart contract in PyTeal **does not verify the identity of the user** , allowing anyone to delete or update the application:

```
from pyteal import *

def unsafe_access_control():
    return Approve()  # No control over who can update or delete the app

print(compileTeal(unsafe_access_control(), mode=Mode.Application, version=2))
```

What is the problem?

- **Any user can update (`UpdateApplication`) or delete (`DeleteApplication`) the application**.
- **No control over who modifies critical data**, allowing unauthorized modifications.

Exploitation

1. An attacker examines the contract and finds that there is a missing check on `Txn.sender()`.
2. Executes a transaction of type `UpdateApplication` or `DeleteApplication`.
3. The contract **approves the request** , allowing the attacker to **modify or delete the app** .

Solution

To avoid this vulnerability, it is necessary to restrict the critical operations only to administrators, checking `Txn.sender()`.

Example of Secure Code

```python
from pyteal import *

ADMIN = Addr("ALGORAND_ADMIN_ADDRESS")  # Administrator's address

def safe_access_control():
    is_admin = Txn.sender() == ADMIN  # Checks whether the user is the
     ↪ administrator

    program = Cond(
        [Txn.on_completion() == OnComplete.UpdateApplication, is_admin],  # Only
         ↪ admin can update
        [Txn.on_completion() == OnComplete.DeleteApplication, is_admin],  # Only
         ↪ admin can delete
        [Txn.on_completion() == OnComplete.NoOp, Approve()],  # Other permitted
         ↪ operations
    )

    return program

print(compileTeal(safe_access_control(), mode=Mode.Application, version=2))
```

Why is it safe?

- **Verify that only the administrator (`ADMIN`) can update or delete the application.**
- **Blocks editing attempts by unauthorized users.**
- **Protects critical data and functionality from uncontrolled access attacks.**

Conclusion

The absence of access controls in an Algorand application allows **arbitrary changes and destructive attacks**. Check `Txn.sender()` and restrict sensitive operations to the administrators is critical to protecting the 'ntegrity of a dApp.

Asset Id Check

Description

Smart contracts on Algorand can manage **assets (ASA -Algorand Standard Assets)** to perform transfers, blocks and other operations. However, if the contract **does not verify the Asset IDassociated** with a transaction , an attacker can:

- **Manipulate the contract** to transfer unplanned assets .
- **Deceive the contract** by making it accept assets other than those desired.

Vulnerabilities arise from:

- **Lack of control on `Txn.assets[]`**, allowing the transfer of arbitrary assets .
- **Generic use of operations on assets**, without verifying that they belong to the application.

Example

The following smart contract in PyTeal **does not verify which asset is transferred** , allowing the transfer of any ASA:

```python
from pyteal import *

def unsafe_asset_transfer():
    return Seq([
        App.globalPut(Bytes("Receiver"), Txn.receiver()),  # Save the recipient
        Approve()  # No control over Asset ID
    ])

print(compileTeal(unsafe_asset_transfer(), mode=Mode.Application, version=2))
```

What is the problem?

- The contract **accepts any asset** without checking its ID.
- An attacker can **send other assets than those intended** ,altering the behavior of the contract.

Exploitation

1. A contract expects to receive a **specific ASA** , but does not verifies the Asset ID.
2. An attacker sends a **arbitrary asset** , which the contract mistakenly accepts.
3. This can cause **financial losses** or **unexpected behavior** in the contract.

Solution

To avoid this vulnerability, it is necessary to **verify that the asset used is the one intended**, checking `Txn.assets[]`.

Example of Secure Code

```python
from pyteal import *

EXPECTED_ASSET_ID = Int(123456)  # The Asset ID that the contract must accept

def safe_asset_transfer():
    return And(
        Txn.xfer_asset() == EXPECTED_ASSET_ID,  # Verify Asset ID
        Approve()
    )

print(compileTeal(safe_asset_transfer(), mode=Mode.Application, version=2))
```

Why is it safe?

- **Verifies that the transferred asset matches the expected ID (`EXPECTED_ASSET_ID`).**
- **Block attempts to transfer unauthorized assets.**
- **Avoid fraudulent manipulation of ASAs in the contract.**

Conclusion

The absence of control over Asset ID exposes a contract to **unwanted transfers and fraud** . Implement an explicit on `Txn.xfer_asset()` is essential to ensure that only authorized assets are processed by the contract.

Denial of Service (DoS)

Description

A **Denial of Service (DoS) attack** on Algorand can prevent the operation of a smart contract by blocking execution or exhausting available resources. One of the most common vulnerabilities occurs when an attacker can **opt-out from an asset associated to the contract , causing malfunctions**.

Vulnerabilities arise from:

- **Allowing anyone to opt-out of an ASA** ,causing errors in the contract.
- **Lack of verification of who can perform certain operations**, making the contract vulnerable to spam and DoS attacks.
- **Excessive use of contract resources**, blocking its legitimate execution.

Example

The following smart contract in PyTeal **does not control who can perform an opt-out from an ASA** , allowing anyone to do so and causing problems in asset management :

```
from pyteal import *

def unsafe_dos():
    return Approve()  # No control over asset opt-outs

print(compileTeal(unsafe_dos(), mode=Mode.Application, version=2))
```

What is the problem?

- **Any user can perform an opt-out from an ASA**, interrupting the workflow of the contract.
- **If the contract depends on a particular asset, an attacker can eliminate it**, blocking future operations .
- **It can be used for large-scale DoS attacks**, sending continuous requests and consuming resources.

Exploitation

1. A malicious user performs an opt-out transaction from an ASA used in the contract.
2. The contract **no longer finds the asset**, preventing further operations.
3. If the contract depends on a limited number of assets, the attacker can **perform the opt-out repeatedly**, blocking it completely.

Solution

To avoid this vulnerability, the contract must **verify that the opt-out is not arbitrarily executed** and block unauthorized actions authorized.

Example of Secure Code

```
from pyteal import *

ADMIN = Addr("ALGORAND_ADMIN_ADDRESS")  # Administrator's address

def safe_dos():
```

```
    is_admin = Txn.sender() == ADMIN  # Only the administrator can manage
     ↪ opt-outs

    program = And(
        Txn.on_completion() != OnComplete.OptIn,  # Prevents arbitrary opt-outs
        is_admin,  # Only the administrator can authorize certain actions
        Approve()
    )

    return program

print(compileTeal(safe_dos(), mode=Mode.Application, version=2))
```

Why is it safe?

- **Prevents anyone from performing an unauthorized opt-out**, preventing interruptions in the contract.
- **Restricts asset management to an administrator**, protecting the integrity of the system.
- **Blocks DoS attacks based on repeated opt-out**, ensuring business continuity of the contract.

Conclusion

A DoS attack based on asset opt-outs can **compromise the operation of a smart contract**. Implementing controls on authorized users and limiting critical operations is essential to prevent unwanted interruptions.

Inner Transaction Fee

Description

Smart contracts on Algorand can **perform internal transactions (Inner Transactions)** to send assets, make payments or interact with other applications. However, if the contract **does not explicitly sets the fee of inner transactions to zero** , it may incur unanticipated costs, quickly depleting the funds of the application.

Vulnerabilities arise from:

- **Failure to set the fee to zero (`fee=0`)**, charging the contract for inner transaction fees.
- **Ability for an attacker to force the contract to perform many inner transactions**, draining funds.
- **Excessive use of internal transactions without controlling costs**, causing financial sustainability problems.

Example

The following smart contract in PyTeal **does not set the fee to zero in the inner transactions**, causing ALGO consumption for each inner transaction executed:

```
from pyteal import *

def unsafe_inner_tx():
    payment_txn = InnerTxnBuilder.Begin()  # Start an internal transaction
    InnerTxnBuilder.SetFields({
```

```
        TxnField.type_enum: TxnType.Payment,
        TxnField.receiver: Txn.sender(),
        TxnField.amount: Int(100000),  # Send 0.1 ALGO
    })
    InnerTxnBuilder.Submit()  # No fee control
    return Approve()

print(compileTeal(unsafe_inner_tx(), mode=Mode.Application, version=6))
```

What is the problem?

- **Each inner transaction has a default fee** , which is subtracted from the balance of the application.
- **If the contract performs many inner transactions, it will quickly deplete the funds in ALGO**, stopping its operation.
- **An attacker can force the contract to generate many innertransactions**, causing a financial Denial of Service .

Exploitation

1. The contract executes an internal transaction **without setting the fee to zero**.
2. Each inner transaction consumes ALGO from the contract funds.
3. An attacker can exploit this vulnerability **generating many inner transactions**, emptying the application account.

Solution

To avoid this vulnerability, it is necessary to **explicitly set the fee of inner transactions to zero (`fee=0`)**, ensuring that the cost is covered by the initial group transaction.

Example of Secure Code

```
from pyteal import *

def safe_inner_tx():
    payment_txn = InnerTxnBuilder.Begin()  # Start an internal transaction
    InnerTxnBuilder.SetFields({
        TxnField.type_enum: TxnType.Payment,
        TxnField.receiver: Txn.sender(),
        TxnField.amount: Int(100000),  # Send 0.1 ALGO
        TxnField.fee: Int(0),  # Set the fee to zero
    })
    InnerTxnBuilder.Submit()
    return Approve()

print(compileTeal(safe_inner_tx(), mode=Mode.Application, version=6))
```

Why is it safe?

- **The fee of inner transactions is set to zero**, avoiding unwanted costs.
- **Prevents fund draining attacks**, protecting the balance of the application.
- **Ensures the sustainability of the contract**, avoiding problems of exhaustion of available ALGOs.

Conclusion

Not setting the inner transactions fee to zero can lead to **unforeseen costs and vulnerabilities of denial of service type financial**. Implement explicit control over `TxnField.fee = Int(0)` is essential to protect the integrity of the contract.

Clear State Transaction Check

Description

The **Clear State Transactions** in Algorand allow users to remove their state from a smart contract application.However, if the contract **does not properly check the type of the transaction (`OnComplete`)**, can expose critical vulnerabilities such as:

- **Loss of important data**, if users can perform a Clear State at unexpected times.
- **Ability to bypass access checks**, if an attacker is able to exploit a Clear State to change the state of the application.
- **Removal of assets associated with the contract**, compromising the financial logic of the application.

Example

The following smart contract in PyTeal **does not check the field `Txn.on_completion()`**, allowing any user to perform a Clear State Transaction without restrictions:

```
from pyteal import *

def unsafe_clear_state():
    return Approve()  # No restrictions on ClearState transactions

print(compileTeal(unsafe_clear_state(), mode=Mode.Application, version=6))
```

What is the problem?

- **Any user can remove their status from the contractat any time**, without any verification.
- **If the contract stores data critical to the user, this may be lost permanently**.
- **An attacker could use Clear State to remove assets or essential information**, causing problems in the management of the contract.

Exploitation

1. The contract does not verify the type of transaction (`OnComplete`).
2. A malicious user performs a Clear State transaction without restrictions.
3. The status associated with the user is deleted, causing potential loss of funds or application malfunctions.

Solution

To avoid this vulnerability, the contract must **block Clear State transactions explicitly**, or allow them only in controlled circumstances.

Example of Secure Code

```
from pyteal import *

def safe_clear_state():
    program = If(
        Txn.on_completion() == OnComplete.ClearState,
        Reject(),  # Reject all ClearState transactions
        Approve()
    )
    return program

print(compileTeal(safe_clear_state(), mode=Mode.Application, version=6))
```

Why is it safe?

- **Blocks all Clear State transactions**, preventing unauthorized removals.
- **Prevents users from deleting critical data from the contract**, protecting application logic.
- **It ensures that important assets and states remain intact**, preventing exploits related to state deletion

Conclusion

Failure to properly handle Clear State transactions can **compromise the security and integrity of the smart contract**. Blocking Clear State Transactions or implementing strict controls is essential to preventing data and asset loss.

Conclusion

The security of smart contracts on Algorand is critical to ensure the reliability and integrity of decentralized applications. We examined several critical vulnerabilities, including **Rekeying, Unchecked Transaction Fees, Account Closing, Asset Closing, Group Size Check, Time-based Replay Attack, Access Controls, Asset ID Check, Denial of Service, Inner Transaction Fee, and Clear State Transaction Check**.

These vulnerabilities can lead to **loss of funds, contract failures and DoS attacks , compromising the entire Algorand ecosystem**. However, by adopting development best practices in PyTeal , it is possible to mitigate them effectively:

- **Restrict access permissions**, to prevent unauthorized operations authorized.
- **Carefully verify transactions** , checking critical parameters such as `OnComplete`, `Txn.sender()` and `Txn.fee`.
- **Avoid unexpected costs**, setting `TxnField.fee = Int(0)` in inner transactions.
- **Protect assets associated with smart contracts**, preventing opt-outs and unsupervised closing transactions.
- **Use anti-replay attack measures**, such as leasing to avoid duplicate transactions.