VANTAGEPOINT
security at the speed of development

Smart Contract Audit of Deflex Protocol

Final Report

Prepared for:
Deflex

21 Nov 2022

FOR GENERAL DISTRIBUTION

Table Of Content

1. Executive Summary

1.1. Overview

Vantage Point Security Pte Ltd was engaged by Deflex to conduct an Algorand smart contract audit of Deflex protocol. Deflex protocol is composed of a limit order application and an order router application. The limit order application allows users to place a limit order where anyone can fulfill the order within a single group transaction. The order router application allows various swaps by drawing on the liquidity available through Automated Market Makers such as Algofi, Humble and Pact. Deflex protocol supports both split swaps and multi-hop swaps to allow optimal routes in fulfilling users' limit orders placed. The Algorand smart contract audit was conducted based on the following materials provided.

Supporting Documents

- Limit-Order App
- Order-Router App
- Registry App
- Architecture Diagram of Deflex Protocol
- Audit Changes Summary

PyTeal Code Repo

- https://github.com/deflex-fi/deflex-contracts

Vantage Point performed this audit by first understanding the Deflex protocol's business logic based on the documents provided. We sought clarifications on potential issues, discrepancies, flaws and plans on how manual operations involved would be carried out through discussions with the Deflex team.

The smart contract audit was conducted on the provided PyTeal code to identify any weaknesses, vulnerabilities, and non-compliance to Algorand best practices.

Total of 1 medium-risk, 7 low-risk and 1 observational findings were identified from the smart contract audit.

Incorrect or Insufficient Transaction Fee Validation

- In logicsig.py, the logic signature rejects all transactions with transaction fee higher than 1000 microAlgo. In times of congestion in Algorand blockchain network, there may be instances where the transaction fee must be higher than 1000 microAlgo to ensure a successful transaction. In such cases, the logic that rejects any transaction that has transaction fee higher than 1000 microAlgo would eventually deny all legitimate transactions.

Unregistered Algo or ASA Transfers Interfere with Smart Contract Logic

- Transfer of ASA or Algo without appropriate application calls such as User_create_order operation can be done to limit order application. If any amount of ASA is transferred to the limit order application without appropriate application calls, the creator of the limit order would not be able to unlock the 100000 microAlgo which has been deposited to meet the minimum balance requirements. For Algo payment transactions without appropriate application calls, as the delete_application() call does not transfer or close the remaining Algo balance, any Algos transferred cannot be recovered too.

Lack of Validation for Compound Types

- It is strongly recommended to validate inputs for compound types such as abi.Address as the Router class does not validate inputs. Especially, abi.Address is not guaranteed to have exactly 32 bytes and therefore, it is recommended to manual verification on the its length.

Improvements in Code Visibility and Maintenance

- For testing purposes, mock swap functions and delete_application logic were added to smart contracts. However, for the smart contract code to be deployed, such test functions and scripts should be removed to minimize the risk of wrong parameters being supplied during deployment.

Insufficient Validation of closeRemainderTo and assetCloseTo Field

- Close_remainder_to and asset_close_to are fields that can be set to close an account or an Algorand Standard Asset (ASA). As a general best practice, it is recommended to validate the value against Global.zero_address() unless the transaction is expected to close an account or an ASA.

ARC-04 Smart Contracts with Router Class without Pragma or pragma

- PyTeal introduced Router class which adheres to ARC-04 ABI conventions and handles routing of base app calls and methods. However, as the Router class is still expecting backward-incompatible changes, it is recommended to use Pragma/pragma expression to pin the version of the PyTeal compiler in the source code. In limit_order_app.py, order_router_app.py and registry_app.py, no PyTeal compiler version pinning through the use of Pragma or pragma was observed.

Incorrect Use or Calculation of Minimum Balance

- All Algorand accounts require a minimum balance of 0.1 Algo. However, the minimum balance requirement for an account increase with opt-ins to Algorand Standard Assets (ASA) and Algorand Smart Contracts (ASC). The limit order application's subroutine get_algo_balance() calculates the minimum balance wrongly as the logic uses Global.min_balance() instead of min_balance(account: Expr).

Insufficient Validation for RekeyTo Field

- Rekeying is a powerful feature in Algorand that can be used to change the auth-addr of an account or a logic signature and is useful for specific scenarios where another smart contract or an account has to take over the right to authorize transactions. As a general best practice, it is recommended to validate the value against Global.zero_address() unless the transaction is expected to rekey an account to a specific address.

Inaccurate Comments

- Based on limit_order_app.py, the smart contract's code comments contained an incorrect description of expected transactions. Having incorrect comments may create unnecessary confusion in providing context to future development efforts in making changes to or integrating with the reviewed smart contract.

The outcome of this Algorand smart contract audit is provided as a detailed technical report that provides the project owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendation that will resolve the identified technical issue.

2. Project Details

2.1. Scope

App Name	Smart Contract Audit of Deflex Protocol
Testing Window	12 Sep 2022 to 14 Nov 2022
Target URL	https://github.com/deflex-fi/deflex-contracts
Svn / Git Revision Number	b2ddd7fa54f8657ce3dbfdcedc493e095072ba02
Project Type	Smart Contract Audit
App Type	Algorand Smart Contract
Items Completed	https://github.com/deflex-fi/deflex-contracts/blob/main/src/alop/contracts/limit_order_app.py https://github.com/deflex-fi/deflex-contracts/blob/main/src/alop/contracts/order_router_app.py https://github.com/deflex-fi/deflex-contracts/blob/main/src/alop/contracts/registry_app.py https://github.com/deflex-fi/deflex-contracts/blob/main/src/alop/contracts/commons.py
Issue Opening Date	12 Oct 2022 ● Incorrect/Insufficient/Lack of Transaction Fee Validation [Medium] ● ARC-04 Smart Contract with Router Class without Pragma or pragma [Low] 9 Nov 2022 ● Unregistered Algo or ASA Transfers Interfere with Smart Contract Logic [Low] 31 Oct 2022 ● Lack of Validation for Compound Types [Low] 18 Oct 2022 ● Improvements in Code Visibility and Maintenance [Low] 17 Oct 2022 ● Insufficient Validation of closeRemainderTo and assetCloseTo Field [Low] ● Insufficient Validation for RekeyTo Field [Low] 11 Oct 2022 ● Incorrect Use or Calculation of Minimum Balance [Low] ● Inaccurate Comments [Observational]

Issue Closing Date	14 Nov 2022 ● Incorrect/Insufficient/Lack of Transaction Fee Validation [Medium] ● Unregistered Algo or ASA Transfers Interfere with Smart Contract Logic [Low] ● Lack of Validation for Compound Types [Low] ● Improvements in Code Visibility and Maintenance [Low] ● Insufficient Validation of closeRemainderTo and assetCloseTo Field [Low] ● ARC-04 Smart Contract with Router Class without Pragma or pragma [Low] ● Incorrect Use or Calculation of Minimum Balance [Low] ● Insufficient Validation for RekeyTo Field [Low] ● Inaccurate Comments [Observational]

2.2. Status

Component	Review Type	Status
Security Audit	Smart Contract Audit	Completed

3. Risk Assessment

This chapter contains an overview of the vulnerabilities discovered during the project. The vulnerabilities are sorted based on the scoring categories CRITICAL, HIGH, MEDIUM and LOW. The category OBSERVATIONAL refers to vulnerabilities that have no risk score and therefore have no immediate impact on the system.

3.1. Summary of Vulnerabilities

Vulnerability Title	Risk Score	Closed
Incorrect/Insufficient/Lack of Transaction Fee Validation	Medium	☑
Unregistered Algo or ASA Transfers Interfere with Smart Contract Logic	Low	☑
Lack of Validation for Compound Types	Low	☑
Improvements in Code Visibility and Maintenance	Low	☑
Insufficient Validation of closeRemainderTo and assetCloseTo Field	Low	☑
ARC-04 Smart Contract with Router Class without Pragma or pragma	Low	☑
Incorrect Use or Calculation of Minimum Balance	Low	☑
Insufficient Validation for RekeyTo Field	Low	☑
Inaccurate Comments	Observational	☑

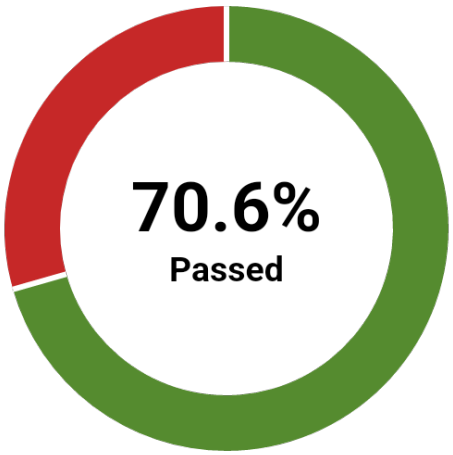
3.2. Vulnerabilities Statistics

Findings Overview

Total	Open	Closed
9	0	9

0	Critical	No findings
0	High	No findings
1	Medium	0 Open, 1 Closed.
7	Low	0 Open, 7 Closed.
1	Observational	0 Open, 1 Closed.

Test Case Status



Open	0.0%
Passed	70.6%
Failed	29.4%
Resolved	0.0%
N/A	0.0%

4. Detailed Description of Vulnerabilities

4.1. Incorrect/Insufficient/Lack of Transaction Fee Validation

Medium

Closed

BACKGROUND

Algorand transactions have a minimum transaction fee of 1000 microAlgo per each transaction and this amount may change if the network gets congested and variable transaction fees per byte are used instead. For both logic signatures and smart contracts, validating transaction fees are crucial in making sure the balances of logic signatures or smart contracts are not drained and transactions can be approved even in cases where higher transaction fees are required due to network congestion.

DESCRIPTION

Instance1
Affected Code/File:

- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/logicsig.py [limit_order_logicsig]

It was noted that the logic signature above had the following transaction fee validation.
Snippet - logicsig.py - Line 110-123

```
def limit_order_logicsig(params: LimitOrderParams):
    program = And(
        # the random nonce makes sure that two exact same limit orders (i.e.,
        # same user, assets, amounts, etc.) can be created simultaneously as
        # long as they have a different nonce
        Int(params.nonce) >= Int(0),
        # general safety conditions
        Txn.asset_close_to() == Global.zero_address(),
        Txn.fee() <= Int(1000),
        # we can either create a limit order or close the logicsig
        # create_order,
        Or(is_setup(params), is_teardown(params))
    )
    return compileTeal(program, Mode.Signature, version=6)
```

As noted above, the logic signature validates if the transaction fee is equal to or less than 1000 microAlgo.
Algorand network's transaction fee may increase from its minimum value of 1000 microAlgo if the network gets congested. To cater for such cases in the future where minimum transaction fee of 1000 microAlgo does not guarantee processing of the transaction due to congestion, the logic signature should either have the transaction fee paid through transaction fee pooling or have the validation that allows transaction fee to be of a certain reasonable value higher than 1000 microAlgo.

IMPACT

Incorrect or insufficient validation for transaction fees may cause rejection of legitimate transactions during network congestion or allow draining of balance.

RECOMMENDATIONS

It is recommended to perform validations for transaction fees in a way that fits into the context of the smart contract or logic signature.
1. Make senders or other participants of the transaction group pay for the transaction fee through transaction fee pooling, by settings the Txn.fee value to be 0.
2. In a scenario where only limited addresses can be used to have a transaction approved from the logic signature, if transaction fees are to be paid from the logic signature or smart contract, the limit can be set to an acceptable value with consideration for network congestion in Algorand blockchain.

COMMENT

Reviewed on 6 Nov 2022

Based on the updated repo https://github.com/deflex-fi/deflex-contracts/commits/main with commit ID b2ddd7fa54f8657ce3dbfdcedc493e095072ba02 from Deflex team, it was noted that the affected smart contract no longer uses logic signatures and instead, utilizes a normal Algorand account for maintaining necessary local states. In doing so, the logic signature logic which had validations for Txn.fee is no longer applicable and therefore, this issue is closed.

Reviewed on 18 Oct 2022

Based on the Deflex team, the changes are expected to be made in the future version of the smart contract when the logic signature is no longer used and replaced with a normal Algorand account. However, as such have not been audited in the current repo, this finding is kept open.

REFERENCES

Algorand Developer Portal - Fee
https://developer.algorand.org/docs/get-details/transactions/?from_query=Transaction%20Fee#fees

PyTeal Documentation - Transaction Fields
https://pyteal.readthedocs.io/en/stable/accessing_transaction_field.html?highlight=Transaction%20#id1

4.2. Unregistered Algo or ASA Transfers Interfere with Smart Contract Logic

Low

Closed

BACKGROUND

Unregistered Algo payment or asset transfer transactions sent without appropriate application calls may interfere with the smart contract's logic and cause an unexpected behavior. In scenarios where the assets or accounts has to be closed, having no path to transfer out the excess balance of Algorand Standard Assets (ASA) or Algo may prevent closure of the smart contract's app account and if such is part of group transactions, legitimate application calls to close the smart contract's application account may fail and lock user's funds indefinitely.

DESCRIPTION

Affected Code/File

- https://github.com/deflex-fi/deflex-contracts/blob/main/src/alop/contracts/limit_order_app.py [User_opt_out_assets]

It was noted that the User_opt_out_assets() operation of limit order application requires the Algorand Standard Asset (ASA) balance to be 0 before the asset can be closed. However, as long as the limit order application is opted in for an ASA, it is possible for anyone to do an asset transfer transaction to the limit order and increase the balance of ASA. This increased ASA balance will remain even after User_cancel_order() or Backend_fill_order_finalize() operations and prevent User_opt_out_assets() from succeeding as the following assert check at line 345 would always fail.

Code Snippet - limit_order_app.py [User_opt_out_assets] - Line 345

```
Assert(asset_holding.value() == Int(0)),
```

It may be possible for anyone to lock user's 100000 microAlgo within the his or her limit order application as there is no way to transfer the ASA balance which has been transferred without other application calls.
This is also the case for payment transactions as well since any Algo transfers not accompanied by appropriate operations such as User_create_order() will remain in the limit order application's account.

IMPACT

Smart contract logic which requires all Algo or asset transfer to be called together with appropriate application calls may fail if there were unregistered Algo or asset transfers to the app account. This may prevent legitimate operations of the application to always fail, due to an unexpected remaining asset balance which has not been accounted for in the application logic.

RECOMMENDATIONS

- Consider unexpected transfers of Algo or ASA (if any is opted in) and ensure all operations can be carried out.
- Create a privileged operation where any remaining balance of ASA can be transferred and opted-out
- During delete_application(), the application's Algo balance can be closed to the user

COMMENT

Reviewed on 11 Nov 2022

Based on commit ac32782b42352f5bcb4fd8b3774a1e7ad2219d7a at https://github.com/deflex-fi/deflex-contracts/tree/audit_fork, this issue is closed.
It was noted that the smart contract has a new method User_delete_app which allows the limit order app's creator to close the limit order application's account and transfer all remaining Algo balance to the creator.

Code Snippet - limit_order_app.py [User_delete_app]

```
    @ABIReturnSubroutine
    def User_delete_app() -> Expr:
        """Delete the app and close the app account to the user. This function
must
        be called with OnCompletion=DeleteApplication."""
        if is_test:
            # always allow delete for test apps - in some cases we want to
            # trigger an assertion failure which interrupts the global open
            # state from updating to 0
            return Return()
        else:
            return Seq(
                assert_safety_checks(),
                # only the creator can delete the app
                Assert(Txn.sender() == Global.creator_address()),
                # the app can only be deleted if there are no open orders
                Assert(App.globalGet(VAR_GLOBAL_NR_OPEN) == Int(0)),
                # the app can only be deleted if the user has opted out of all
assets
                Assert(App.globalGet(VAR_GLOBAL_NR_ASSET_OPT_INS) == Int(0)),
                # close the app account back to the creator to free up the
locked ALGO
                InnerTxnBuilder.Begin(),
                InnerTxnBuilder.SetFields({
                    TxnField.type_enum:          TxnType.Payment,
                    TxnField.sender:
Global.current_application_address(),
                    TxnField.receiver:           Global.creator_address(),
                    TxnField.close_remainder_to: Global.creator_address(),
                    TxnField.amount:
Balance(Global.current_application_address()),
                    TxnField.fee:                Int(0),
                }),
                InnerTxnBuilder.Submit(),
                Return(),
            )
```

In addition, the User_opt_out_assets method has been updated to allow any remaining balance of ASA to be transferred and closed to the limit order application's creator address.

Code Snippet - limit_order_app.py [User_opt_out_assets]

```
@ABIReturnSubroutine
    def User_opt_out_assets() -> Expr:
```

```
        """Opt the limit-order app out of all assets in the foreign assets
array."""
        self = Global.current_application_address()
        i = ScratchVar(TealType.uint64)
        nr_assets_opted_out = ScratchVar(TealType.uint64)
        return Seq(
            assert_safety_checks(),
            assert_is_initialized(),
            # only the creator can tell the app to opt out of an asset
            Assert(Txn.sender() == Global.creator_address()),
            # opting out of an asset is only possible if there aren't any open
orders
            Assert(App.globalGet(VAR_GLOBAL_NR_OPEN) == Int(0)),
            # iterate over assets and opt them out
            nr_assets_opted_out.store(Int(0)),
            For(i.store(Int(0)), i.load() < Txn.assets.length(),
i.store(i.load()+Int(1))).Do(Seq(
                # opting out of ALGO is not possible
                If(Txn.assets[i.load()] != ALGO_ID).Then(Seq(
                    asset_holding := AssetHolding.balance(self,
Txn.assets[i.load()]),
                    # we can only opt out of an asset that the user has opted
into in the first place
                    If(asset_holding.hasValue()).Then(Seq(
                        InnerTxnBuilder.Begin(),
                        InnerTxnBuilder.SetFields({
                            TxnField.type_enum:      TxnType.AssetTransfer,
                            TxnField.sender:
Global.current_application_address(),
                            TxnField.asset_receiver: Global.creator_address(),
                            TxnField.asset_close_to: Global.creator_address(),
                            TxnField.asset_amount:   asset_holding.value(),
                            TxnField.xfer_asset:     Txn.assets[i.load()],
                            TxnField.fee:            Int(0),
                        }),
                        InnerTxnBuilder.Submit(),
                        nr_assets_opted_out.store(nr_assets_opted_out.load() +
Int(1)),
                    )),
                ))
            )),
```

With the use of updated method User_opt_out_assets and new method User_delete_app, it is now possible for the limit order application's owner to claim any remaining ASA or Algo balance. As unregistered transfer of ASA or Algo can be properly handled, this issue is closed.

4.3. Lack of Validation for Compound Types

Low

Closed

BACKGROUND

PyTeal's Router class does not validate inputs for compound types such as abi.StaticArray, abi.Address, abi.DynamicArray, abi.String, or abi.Tuple). It is strongly recommended to have the methods immediately access and validate compound type parameters before committing them for later transactions.

DESCRIPTION

Affected File/Code

- https://github.com/deflex-fi/deflex-contracts/blob/main/src/alop/contracts/limit_order_app.py

It was noted that the affected smart contract did not have sufficient validation for a compound type parameter backend_address(abi.Address) within the User_create_order method as observed in the code snippet below.

Code Snippet - limit_order_app.py - User_create_order

```
    @ABIReturnSubroutine
    def User_create_order(
            escrow_optin_txn: abi.ApplicationCallTransaction,
            network_fee_txn: abi.PaymentTransaction,
            funding_txn: abi.Transaction,
            escrow: abi.Account,
            beneficiary: abi.Account,
            platform_treasury: abi.Account,
            asset_in: abi.Asset,
            amount_in: abi.Uint64,
            asset_out: abi.Asset,
            amount_out: abi.Uint64,
            expiration_date: abi.Uint64,
            fee_bps: abi.Uint64,
            registry_app: abi.Application,
            backend_address: abi.Address,
            note: abi.String) -> Expr:
        self = Global.current_application_address()
        order_state = App.localGetEx(escrow.address(),
Global.current_application_id(), VAR_LOCAL_STATE)
        app_asset_out_holding = AssetHolding.balance(self,
asset_out.asset_id())
        platform_treasury_holding =
AssetHolding.balance(platform_treasury.address(), asset_out.asset_id())
        beneficiary_holding_in = AssetHolding.balance(beneficiary.address(),
asset_in.asset_id())
        beneficiary_holding_out = AssetHolding.balance(beneficiary.address(),
asset_out.asset_id())
        return Seq(
            <REDACTED>
```

```
            App.localPut(escrow.address(),        VAR_LOCAL_BACKEND_ADDRESS,
backend_address.get()),
            App.localPut(escrow.address(), VAR_LOCAL_FEE_BPS,
fee_bps.get()),
            InnerTxnBuilder.Begin(),
            InnerTxnBuilder.MethodCall(
                app_id=registry_app.application_id(),
                method_signature=RegistryAppApi.get_signature(RegistryAppApi.E
S C R O W _ O P T _ I N ) ,
                args=[Global.current_application_id()],
                extra_fields={
                    TxnField.sender:         escrow.address(),
                    TxnField.on_completion:  OnComplete.OptIn,
                    TxnField.fee:            Int(0),
                },
            ),
            InnerTxnBuilder.Submit(),
            App.globalPut(VAR_GLOBAL_NR_OPEN, App.globalGet(VAR_GLOBAL_NR_OPEN)
+ Int(1)),
            Return(),
        )
```

As seen in the PyTeal documentation below, abi.Address may not necessarily be of a valid length for Algorand address.

PyTeal Documentation - Warning

```
The Router does not validate inputs for compound types (abi.StaticArray,
abi.Address, abi.DynamicArray, abi.String, or abi.Tuple). We strongly recommend
methods immediately access and validate compound type parameters before
persisting arguments for later transactions. For validation, it is sufficient
to attempt to extract each element your method will use. If there is an input
error for an element, indexing into that element will fail.
Notes:This recommendation applies to recursively contained compound types as
well. Successfully extracting an element which is a compound type does not
guarantee the extracted value is valid; you must also inspect its elements as
well.
Because of this, abi.Address is not guaranteed to have exactly 32 bytes.
To defend against unintended behavior, manually verify the length is 32 bytes,
i.e. Assert(Len(address.get()) == Int(32)).
```

As the smart contract commits the backend_address as a localstate of the escrow and use it to validate later on in Backend_fill_order_initialize method, at the minimum, it is recommended to validate the length of the backend_address prior to writing it as a localstate.

IMPACT

As the Router class does not validate inputs for compound types, if not validated, compound types with incorrect format may cause an unwanted behavior. For example, abi.Address is not guaranteed to be exactly 32 bytes and such should be manually verified.

RECOMMENDATIONS

Each compound type arguments, ensure there is sufficient validation for each methods prior to persisting them for future transactions.

For abi.Address, following assert check can be used to ensure the supplied input is of 32 bytes and can be considered a valid format for Algorand address.

Sample Validation for abi.Address Length

```
Assert(Len(address.get()) == Int(32))
```

COMMENT

Reviewed on 5 Nov 2022

This issue is closed.
Based on commit abe91a6542f58867549a35921db73ab7f6e03255, following validation on the length of compound type parameter backend_address was added before committing it to the escrow's local state. The commit has been made to audit_fork branch and Deflex team confirmed that audit_fork branch will be merged to the main repo prior to deployment.

Code Snippet - limit_order_app.py - Line 503-505

```
# an abi.Address is not guaranteed to be a proper address with 32 bytes,
            # make sure that it has indeed 32 bytes
            Assert(Len(backend_address.get()) == Int(32)),
```

REFERENCES

PyTeal Documentation - ABI Support - Warning
https://pyteal.readthedocs.io/en/stable/abi.html#registering-methods

4.4. Improvements in Code Visibility and Maintenance

Low

Closed

BACKGROUND

Code visibility and maintenance are important aspects in continuous development of smart contracts. For clarity, consistent naming conventions, explicit checks for specific transaction fields and removal of test code are some of the recommended practices to convey the purpose and meaning of the implement code for future developer team members.

DESCRIPTION

Affected Code/File

- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/registry_app.py [delete_application]
- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/limit_order_app.py [delete_application]
- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/order_router_app.py [delete_application, swap_test]

It was noted that the affected smart contracts included logic or methods meant for testing purposes.

Instance 1 - registry_app.py

Code Snippet - registry_app.py

```
def create_registry_app(limit_order_approval_program_bytes,
limit_order_clear_program_bytes, is_test):
    delete = Approve() if is_test else Reject()
    router = Router(
        name="Limit-Order App Registry",
        descr=("The registry app is used to index all limit orders for the
protocol, "
            "so that orders can be discovered for filling."),
        bare_calls=BareCallActions(
            no_op=OnCompleteAction(action=Approve(),
call_config=CallConfig.CREATE),
            # a bare opt-in is not supported
            opt_in=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            # closing out / clear state is irrelevant since there is no state
            close_out=OnCompleteAction(action=Approve(),
call_config=CallConfig.CALL),
            clear_state=OnCompleteAction(action=Approve(),
call_config=CallConfig.CALL),
            # updating & deleting is not allowed
            update_application=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            delete_application=OnCompleteAction(action=delete,
call_config=CallConfig.CALL),
```

```
        ),
    )
    <REDACTED>
    return router.compile_program(version=7)
```

Instance 2 - limit_order_app.py
Code Snippet - limit_order_app.py

```
def do_delete() -> Expr:
        if is_test:
            # always allow delete for test apps - in some cases we want to
trigger an assertion failure which interrupts
            # the global open state from updating to 0
            return Return(Int(1))
        return Seq(
            assert_safety_checks(),
            # only the creator can delete the app
            Assert(Txn.sender() == Global.creator_address()),
            # the app can only be deleted if there are no open orders
            Assert(App.globalGet(VAR_GLOBAL_NR_OPEN) == Int(0)),
            # the app can only be deleted if the user has opted out of all
assets
            Assert(App.globalGet(VAR_GLOBAL_NR_ASSET_OPT_INS) == Int(0)),
            Return(Int(1)),
        )
    router = Router(
        name="Limit-Order App",
        descr=("The limit order app implements the limit order protocol and
allows users to "
            "create, fill, or cancel limit orders."),
        bare_calls=BareCallActions(
            no_op=OnCompleteAction(action=do_create(),
call_config=CallConfig.CREATE),
            opt_in=OnCompleteAction(action=do_opt_in(),
call_config=CallConfig.CALL),
            close_out=OnCompleteAction(action=do_close_out(),
call_config=CallConfig.CALL),
            clear_state=OnCompleteAction(action=Approve(),
call_config=CallConfig.CALL),
            update_application=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            delete_application=OnCompleteAction(action=do_delete(),
call_config=CallConfig.CALL),
        ),
    )
    router.add_method_handler(User_initialize)
    router.add_method_handler(User_opt_into_assets)
    router.add_method_handler(User_opt_out_assets)
    router.add_method_handler(User_create_order)
    router.add_method_handler(User_cancel_order)
    router.add_method_handler(Backend_fill_order_initialize)
    router.add_method_handler(Backend_fill_order_finalize)
    pragma(compiler_version=constants.COMPILER_VERSION)
    return router.compile_program(version=constants.TEAL_VERSION)
```

Instance 3 - order_router_app.py

Code Snippet - order_router_app.py [delete_application]

```
 def do_delete() -> Expr:
        return Return(Int(1)) if is_test else Return(Int(0))
    router = Router(
        name="Order Router App",
        descr=("The order router app is used to atomically perform multiple
swaps through Algorand DEXes."
               "The swaps are performed using contract-to-contract calls.
Assets are sent "
               "to the app address before a group of swaps. After a swap, the
app can either"
               "close out all assets to a beneficiary, or return the assets
for future swaps in"
               "the transaction group (allowing users to perform multi-hop
swaps through different"
               "trading pairs)."),
        bare_calls=BareCallActions(
            no_op=OnCompleteAction(action=Approve(),
call_config=CallConfig.CREATE),
            opt_in=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            close_out=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            clear_state=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            update_application=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            delete_application=OnCompleteAction(action=do_delete(),
call_config=CallConfig.CALL),
        ),
    )
    router.add_method_handler(User_opt_into_assets)
    router.add_method_handler(User_swap)
    router.add_method_handler(User_swap_finalize)
    pragma(compiler_version=constants.COMPILER_VERSION)
    return router.compile_program(version=constants.TEAL_VERSION)
```

Code Snippet - order_router_app.py [swap_test]

```
# for testing purposes only
    @Subroutine(TealType.none)
    def swap_test(app_id, app_address, amount_in, asset_in_id, asset_out_id,
min_amount_to_receive) -> Expr:
        return Seq(
            InnerTxnBuilder.Begin(),
            If (asset_in_id == ALGO_ID).Then(
                InnerTxnBuilder.SetFields({
                    TxnField.type_enum: TxnType.Payment,
                    TxnField.receiver: app_address,
                    TxnField.amount: amount_in,
                    TxnField.fee: Int(0),})
            ).Else(
                InnerTxnBuilder.SetFields({
                    TxnField.type_enum: TxnType.AssetTransfer,
                    TxnField.xfer_asset: asset_in_id,
```

```
                TxnField.asset_receiver: app_address,
                TxnField.asset_amount: amount_in,
                TxnField.fee: Int(0),})
            ),
            InnerTxnBuilder.Next(),
            InnerTxnBuilder.SetFields({
                TxnField.type_enum: TxnType.ApplicationCall,
                TxnField.application_id: app_id,
                TxnField.application_args: [
                    Bytes('swap'),
                    Itob(WideRatio(
                        [Int(100000000), min_amount_to_receive],
                        [Int(99940009)])),],
                TxnField.assets: [asset_out_id],
                TxnField.fee: Int(0),}),
            InnerTxnBuilder.Submit(),)
```

IMPACT

Inconsistent naming conventions of methods and variables, implicit checks or validations and test scripts or methods may create confusion during continuous development, audit and deployment of smart contracts.

RECOMMENDATIONS

For better maintainability and clarity, it is recommended to have the following practices.

- If there are any test scripts or methods to be only used for internal testing, remove such code
- Name smart contract methods and variables with in consistent and meaningful ways
- Use explicit checks wherever possible, instead of implicit checks

COMMENT

Reviewed on 6 Nov 2022

Based on the updated repo at https://github.com/deflex-fi/deflex-contracts with commit ID b2ddd7fa54f8657ce3dbfdcedc493e095072ba02, following updates were noted.

Instance 1 & 2 - registry_app.py [delete_application] & limit_order_app.py [delete_application]

Instance 1 and 2 are closed. It was noted that the compile_teal_and_abi.py which is used to compile the registry_app.py and limit_order_app.py into approval.teal and clearstate.teal has IS_TEST parameter set to False by default and therefore, the application compiles to Teal approval program without any logic reserved for testing purposes. Such behavior has also been verified in the Teal code snippet below.

Code Snippet - registry_app.py compiled to approval.teal

```
txn OnCompletion
int DeleteApplication
==
bnz main_l16
err
main_l16:
txn ApplicationID
int 0
```

```
!=
assert
int 0
return
```

Code Snippet - limit_order_app.py compield to approval.teal

```
txn OnCompletion
int DeleteApplication
==
bnz main_l22
err
main_l22:
txn ApplicationID
int 0
!=
assert
callsub assertsafetychecks_0
txn Sender
global CreatorAddress
==
assert
byte "nr_open"
app_global_get
int 0
==<REDACTED>
```

Instance 3 - order_router_app.py [delete_application, swap_test]

Instance 3 of this issue is closed as it was confirmed that the compile_teal_and_abi.py which is used to compile order_router_app.py into approval.teal and clearstate.teal has IS_TEST parameter set to False by default. The application compiles to Teal approval program without the delete_application logic reserved for testing purposes. Such behavior has also been verified in the Teal code snippet below.

Code Snippet - order_router_app.py compiled to approval.teal [delete_application]

```
txn OnCompletion
int DeleteApplication
==
bnz main_l14
err
main_l14:
txn ApplicationID
int 0
!=
assert
int 0
return
```

Code Snippet - order_router_app.py compiled to approval.teal [swap_test]

```
load 35
int 0
==
bnz Userswap_5_l22
load 35
int 1
==
bnz Userswap_5_l16
```

```
load 35
int 2
==
bnz Userswap_5_l10
load 35
int 3
==
bnz Userswap_5_l9
load 35
int 0
!=
load 35
int 1
!=
&&
load 35
int 2
!=
&&
load 35
int 3
!=
&&
bnz Userswap_5_l8
err
Userswap_5_l8:
int 0
return
Userswap_5_l9:
int 0
return
```

REFERENCES

CWE-489: Active Debug Code
https://cwe.mitre.org/data/definitions/489.html

4.5. Insufficient Validation of closeRemainderTo and assetCloseTo Field

Low

Closed

BACKGROUND

Algorand transactions can have the closeRemainderTo and assetCloseTo field specified to close an account or an Algorand Standard Asset (ASA). For closeRemainderTo field, the remaining Algo balance after paying for Txn.fee and Txn.amount are sent to the specified address. If there are any asset holdings, one is required to close the assets prior to account closure through the use of CloseRemainderTo field.

DESCRIPTION

Affected Code/File

- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/logicsig.py [is_setup]
- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/order_router_app.py [User_opt_into_assets, User_swap, User_swap_finalize]
- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/registry_app.py [close_out, clear_state, Escrow_opt_in]

It was noted that the affected parts of the code did not have recommended validation for assetCloseTo or closeRemainderTo field.

Instance 1 - logicsig.py

Code Snippet - logicsig.py [is_setup]

```
def is_setup(params: LimitOrderParams) -> Expr:
    return And(
        Txn.type_enum() == TxnType.ApplicationCall,
        Txn.application_id() == Int(params.limit_order_app_id),
        Txn.on_completion() == OnComplete.OptIn,
        Txn.rekey_to() == Addr(util.app_address(params.limit_order_app_id)),
        is_valid_user_create_order_txn(Txn.group_index() + Int(3), params),
    )
def limit_order_logicsig(params: LimitOrderParams):
    program = And(
        Int(params.nonce) >= Int(0),        ..........
        Txn.asset_close_to() == Global.zero_address(),
        Txn.fee() <= Int(1000),
        Or(is_setup(params), is_teardown(params))
    )
    return compileTeal(program, Mode.Signature, version=6)
```

In the code snippet above, the affected logic signature has validations for Txn.asset_close_to() to a Global.zero_address(). However, validation for Txn.close_remainder_to() is missing.

Instance 2 - order_router_app.py

Code Snippet - order_router_app.py [User_opt_into_assets]

```
@ABIReturnSubroutine
def User_opt_into_assets() -> Expr:
```

```
    i = ScratchVar(TealType.uint64)
    self = Global.current_application_address()
    return Seq(
        Assert(is_txn_order_router_app_operation(Txn.group_index() + Int(2),
get_selector(OrderRouterAppApi.USER_SWAP))),
        For(i.store(Int(0)), i.load() < Txn.assets.length(), i.store(i.load()
+ Int(1))).Do(
            Seq(
                If(Txn.assets[i.load()] != ALGO_ID).Then(Seq(
                    asset_holding := AssetHolding.balance(self,
Txn.assets[i.load()]),
                    asset_holding,
                    If(Not(asset_holding.hasValue())).Then(Seq(
                        InnerTxnBuilder.Begin(),
                        InnerTxnBuilder.SetFields({
                                TxnField.type_enum: TxnType.AssetTransfer,
                                TxnField.xfer_asset: Txn.assets[i.load()],
                                TxnField.asset_receiver:
Global.current_application_address(),
                                TxnField.fee: Int(0),}),
                        InnerTxnBuilder.Submit(),
                        App.globalPut(VAR_GLOBAL_ASSETS_OPTED_IN,
                        App.globalGet(VAR_GLOBAL_ASSETS_OPTED_IN) + Int(1))
                        ))))),))
```

In the code snippet above, the affected smart contract does not have any validations for Txn.asset_close_to() and Txn.close_remainder_to() for its application call. For payment transaction or asset transfer transaction which is expected to take place afterwards, as order router app does not strictly limit the source of funding or the transaction parameters such as Txn.asset_close_to() and Txn.close_remainder_to(), it would be considered at the choice of user to close accounts or close assets to a specific Algorand address after funding the order router. It addition, interaction between participants and the order router application is expected to be largely backend-driven with minimal human interaction. It is recommended to add such details to the documentation so that such is clearly communicated to potential users or other developers.

Instance 3 - registry_app.py

Code Snippet - registry_app.py [Escrow_opt_in]

```
@router.method(opt_in=CallConfig.CALL)
def Escrow_opt_in(limit_order_app: abi.Application) -> Expr:
    hash_limit_order_approval_program =
hashlib.sha256(limit_order_approval_program_bytes).digest()
    hash_limit_order_clear_program =
hashlib.sha256(limit_order_clear_program_bytes).digest()
    app_approval_program =
AppParam.approvalProgram(limit_order_app.application_id())
    app_clear_program =
AppParam.clearStateProgram(limit_order_app.application_id())
    return Seq(
        app_approval_program,
        app_clear_program,
        Assert(app_approval_program.hasValue()),
        Assert(app_clear_program.hasValue()),
```

```
        Assert(Sha256(app_approval_program.value()) ==
Bytes(hash_limit_order_approval_program)),
        Assert(Sha256(app_clear_program.value()) ==
Bytes(hash_limit_order_clear_program)),
        Assert(App.optedIn(Txn.sender(), limit_order_app.application_id())),
    )
```

In the code snippet above, validation for Txn.close_remainder_to() and Txn.asset_close_to() were missing.

IMPACT

Once a transaction with closeRemainderTo gets approved, the remaining balance of the account after fees and amount is transferred to the address specified within the closeRemainderTo field. For assetCloseTo, all remaining ASA balance of one specific asset being transferred will be closed to the specified address. An attacker may compromise the front-end of web applications or similar to prompt for approval of self-damaging transactions to users. Users may approve the transaction based on the credibility of the applications or addresses interacting with, even though the balance of the account could be transferred to an attacker's account.

RECOMMENDATIONS

As a best practice, always validate the closeRemainderTo and assetCloseTo field value of all non-closing transactions is equal to Global zero address.

Example Code

```
Assert(Txn.close_remainder_to() == Global.zero_address()),
Assert(Txn.asset_close_to() == Global.zero_address()),
```

COMMENT

Reviewed on 8 Nov 2022

Based on following commits at https://github.com/deflex-fi/deflex-contracts/tree/audit_fork, the documentation has been updated to highlight the support of rekeying, closing of an asset and an account within funding transactions for order router application.

- ef94fe6a9dcf4c4155fa3110373234d7c9e5cd4d
- f63e7f30c3d702a4282c0cabaf9eec9e65ab24a2

Documentation - Updated

```
Send the input amount from user to order-router app.
Note: the user can fund the input in any way desired, including
closing/rekeying
```

- Payment, only if input asset is ALGO
  Sender: Swapper Account
  Receiver: Order-Router App Account
  Amount: Input Amount
- Asset Transfer, only if input asset is non-ALGO
  Sender: Swapper Account
  Receiver: Order-Router App Account
  Amount: Input Amount
  Asset: Input Asset

Reviewed on 6 Nov 2022

Based on the updated repo at https://github.com/deflex-fi/deflex-contracts with commit ID b2ddd7fa54f8657ce3dbfdcedc493e095072ba02, following updates were noted along with a new subroutine asset_safety_checks under commons.py

Code Snippet - commons.py [assert_safety_checks]

```
@Subroutine(TealType.none)
def assert_safety_checks() -> Expr:
    return Seq(
        Assert(Txn.rekey_to() == Global.zero_address()),
        Assert(Txn.close_remainder_to() == Global.zero_address()),
        Assert(Txn.asset_close_to() == Global.zero_address()),
        Return(),
    )
```

Instance 1 - logicsig.py [is_setup]

Instance1 of this issue is closed as logicsig.py is no longer in use and has been replaced with a normal Algorand account.

Instance 2 - order_router_app.py [User_opt_into_assets]

Instance 2 of this issue is closed as the User_opt_into_assets() method of order_router_app.py now makes use of subroutine assert_safety_checks() to validate RekeyTo, CloseRemainderTo and AssetCloseTo field values.

Code Snippet - order_router_app.py [User_opt_into_assets]

```
@ABIReturnSubroutine
    def User_opt_into_assets() -> Expr:
        """Opt the order-router app into all assets in the foreign assets
array."""
        i = ScratchVar(TealType.uint64)
        self = Global.current_application_address()
        return Seq(
            assert_safety_checks(),
            <REDACTED>
        )
```

Instance 3 - registry_app.py [Escrow_opt_in]

Instance 3 of this issue is closed as the Escrow_opt_in() method of registry_app.py now makes use of subroutine assert_safety_checks() to validate RekeyTo, CloseRemainderTo and AssetCloseTo field values.

Code Snippet - registry_app.py [Escrow_opt_in]

```
@ABIReturnSubroutine
    def Escrow_opt_in(limit_order_app: abi.Application) -> Expr:
        """Opt the limit order in to the registry. This function must be
called
        with OnCompletion=OptIn.
        Args:
            limit_order_app: The ID of the limit-order app.
        """
        return Seq(
            assert_safety_checks(),
            # make sure that a valid escrow is calling
            assert_escrow_opted_into_valid_limit_order_app(Txn.sender(),
limit_order_app),
            # this call opts the escrow into the current app
            Assert(Txn.on_completion() == OnComplete.OptIn),
            # update stats
```

```
            App.globalPut(VAR_GLOBAL_NR_OPEN, App.globalGet(VAR_GLOBAL_NR_OPEN)
+ Int(1)),
        )
```

REFERENCES

Algorand Developer Portal - Close an Account
https://developer.algorand.org/docs/get-details/transactions/#close-an-account

4.6. ARC-04 Smart Contract with Router Class without Pragma or pragma

Low

Closed

BACKGROUND

ARC-04 Algorand Smart Contract Transaction Calling Conventions provide standards for encoding smart contract's method calls. ARC-04 allows wallets and dapp frontends to properly encode call transactions based on the interface description and explorers to show details of invoked methods. To make it easier for an application to route across many base app calls and methods, PyTeal introduced the Router class which adheres to the ARC-4 ABI conventions. However, as the Router class is still expecting backwards-incompatible changes, it is recommended use pragma/Pragma expression to pin the version of PyTeal compiler in the source code.

DESCRIPTION

Affected Code/File

- https://github.com/deflex-fi/deflex-contracts/blob/e51855d854188b3325bee46a4ef2b9d25dbb2c73/src/alop/contracts/limit_order_app.py
- https://github.com/deflex-fi/deflex-contracts/blob/e51855d854188b3325bee46a4ef2b9d25dbb2c73/src/alop/contracts/order_router_app.py
- https://github.com/deflex-fi/deflex-contracts/blob/e51855d854188b3325bee46a4ef2b9d25dbb2c73/src/alop/contracts/registry_app.py

It was noted that smart contracts above are ARC-04 compliant and make use of PyTeal's Router class for routing methods and base application calls.

PyTeal Documentation - https://pyteal.readthedocs.io/en/stable/abi.html#creating-an-arc-4-program

```
Warning - Router usage is still taking shape and is subject to backwards
incompatible changes. Feel encouraged to use Router and expect a best-effort
attempt to minimize backwards incompatible changes along with a migration path.
For these reasons, we strongly recommend using pragma or the Pragma expression
to pin the version of PyTeal in your source code. See Version Pragmas for more
information.
As noted above, Router class is still expecting changes and therefore, it is
strongly recommended to use Pragma or pragma to pin the version of PyTeal in
the source code. However, as noted below, PyTeal compiler version pinning with
pragma or Pragma was not observed.
```

Code Snippet - limit_order_app.py - Line 825-847

```
router = Router(
        name="Limit-Order App",
        descr=("The limit order app implements the limit order protocol and
allows users to "
            "create, fill, or cancel limit orders."),
        bare_calls=BareCallActions(
```

```
            no_op=OnCompleteAction(action=do_create(),
call_config=CallConfig.CREATE),
            opt_in=OnCompleteAction(action=do_opt_in(),
call_config=CallConfig.CALL),
            close_out=OnCompleteAction(action=do_close_out(),
call_config=CallConfig.CALL),
            clear_state=OnCompleteAction(action=Approve(),
call_config=CallConfig.CALL),
            update_application=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            delete_application=OnCompleteAction(action=do_delete(),
call_config=CallConfig.CALL),
        ),
    )
    router.add_method_handler(User_initialize)
    router.add_method_handler(User_opt_into_assets)
    router.add_method_handler(User_opt_out_assets)
    router.add_method_handler(User_create_order)
    router.add_method_handler(User_cancel_order)
    router.add_method_handler(Backend_fill_order_initialize)
    router.add_method_handler(Backend_fill_order_finalize)
    return router.compile_program(version=7)
```

Code Snippet - order_router_app.py - Line 474-497

```
router = Router(
        name="Order Router App",
        descr=("The order router app is used to atomically perform multiple
swaps through Algorand DEXes."
               "The swaps are performed using contract-to-contract calls.
Assets are sent "
               "to the app address before a group of swaps. After a swap, the
app can either"
               "close out all assets to a beneficiary, or return the assets
for future swaps in"
               "the transaction group (allowing users to perform multi-hop
swaps through different"
               "trading pairs)."),
        bare_calls=BareCallActions(
            no_op=OnCompleteAction(action=Approve(),
call_config=CallConfig.CREATE),
            opt_in=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            close_out=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            clear_state=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            update_application=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            delete_application=OnCompleteAction(action=do_delete(),
call_config=CallConfig.CALL),
        ),
    )
    router.add_method_handler(User_opt_into_assets)
    router.add_method_handler(User_swap)
    router.add_method_handler(User_swap_finalize)
```

```
    return router.compile_program(version=7)
```

Code Snippet - registry_app.py - Line 24-66

```
    router = Router(
        name="Limit-Order App Registry",
        descr=("The registry app is used to index all limit orders for the
protocol, "
            "so that orders can be discovered for filling."),
        bare_calls=BareCallActions(
            no_op=OnCompleteAction(action=Approve(),
call_config=CallConfig.CREATE),
            # a bare opt-in is not supported
            opt_in=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            # closing out / clear state is irrelevant since there is no state
            close_out=OnCompleteAction(action=Approve(),
call_config=CallConfig.CALL),
            clear_state=OnCompleteAction(action=Approve(),
call_config=CallConfig.CALL),
            # updating & deleting is not allowed
            update_application=OnCompleteAction(action=Reject(),
call_config=CallConfig.CALL),
            delete_application=OnCompleteAction(action=delete,
call_config=CallConfig.CALL),
        ),
    )
        return router.compile_program(version=7)
```

IMPACT

As PyTeal's Router class may have backward-incompatible changes in the future, if the PyTeal compiler version is not pinned through the use of Pragma or pragma, the PyTeal smart contract code can be compiled using a different version of PyTeal compiler and result in an error or an unexpected behavior.

RECOMMENDATIONS

It is recommended to make use of Pragma or pragma to pin the version of PyTeal in the source code to cater for potential backward-incompatable changes to the Router class.

Sample Code

```
pragma(compiler_version="0.17.0")
```

Do note that above is an example and correct range of compiler version should be determined and specified in the code.

COMMENT

Reviewed on 13 Oct 2022

Based on the commit 7ad3622e778bfeebc1d24b485e9c4ce75505f10c, all affected smart contracts using Router class now makes use of the pragma to pin the compiler version to 0.17.0, as seen below.

Code Snippet - /src/alop/constants.py

```
ALGO_ASSET_ID = 0
ALOP_NOTE = "alop"
```

```
COMPILER_VERSION = "0.17.0"
TEAL_VERSION = 7
```

Code Snippet - /src/alop/contracts/order_router_app.py

```
 pragma(compiler_version=constants.COMPILER_VERSION)
    return router.compile_program(version=constants.TEAL_VERSION)
```

Code Snippet - /src/alop/contracts/registry_app.py

```
    pragma(compiler_version=constants.COMPILER_VERSION)
    return router.compile_program(version=constants.TEAL_VERSION)
```

Code Snippet - /src/alop/contracts/limit_order_app.py

```
pragma(compiler_version=constants.COMPILER_VERSION)
    return router.compile_program(version=constants.TEAL_VERSION)
```

REFERENCES

PyTeal - ABI Support
https://pyteal.readthedocs.io/en/stable/abi.html?highlight=Router#abi-support

PyTeal - Pragma
https://pyteal.readthedocs.io/en/stable/api.html?highlight=Router#pyteal.Pragma

4.7. Incorrect Use or Calculation of Minimum Balance

Low

Closed

BACKGROUND

All Algorand accounts require minimum balance of 100,000 microAlgo at all times. Based on the application opt-ins and ASA opt-ins, this minimum balance requirement increases. Having incorrect considerations or logic around minimum balance calculation may reject legitimate transactions for operations.

DESCRIPTION

Affected Code/File:

- https://github.com/deflex-fi/deflex-contracts/blob/e51855d854188b3325bee46a4ef2b9d25dbb2c73/src/alop/contracts/limit_order_app.py [get_algo_balance()]

It was noted that the affected smart contract had an incorrect logic for subroutine get_algo_balance() as noted in the code snippet below.

Code Snippet - limit_order_app.py - Line 195-197

```
@Subroutine(TealType.uint64)
def get_algo_balance():
    return Balance(Global.current_application_address()) - Global.min_balance()
```

Based on the context of the application, the subroutine get_algo_balance() is expected to return the current Algo balance of the smart contract minus the minimum balance required for the smart contract's address. However, as observed in the code snippet above, the value being subtracted from the current Algo balance of the smart contract is Global.min_balance(), a global variable which does not take opt-ins to Algorand Standard Assets (ASA) and applications. To provide the correct value based on the context of the subroutine get_algo_balance(), min_balance(acccount: Expr) should be used.

Global.min_balance()

- Returns a global parameter, minimum balance. Currently, set to 0.1 Algo

min_balance(account: Expr)

- Returns the minimum balance for a specific account, based on the number of ASAs and applications opted-in

IMPACT

When an incorrect logic is used for calculating the minimum balance of an account, the funds held by the smart contract may be locked as operations which allows transfer of Algos may always fail due to minimum balance requirements or return an incorrect value for any further logics in place.

RECOMMENDATIONS

Review and use the correct minimum balance calculation logic. Consider opt-ins to applications and ASAs in calculating the minimum balance for accounts and ensure the application can execute transactions normally.

COMMENT

Reviewed on 30 Oct 2022

Based on the updated repo at https://github.com/deflex-fi/deflex-contracts with commit b2ddd7fa54f8657ce3dbfdcedc493e095072ba02, it was noted that the subroutine get_algo_balance() is now defined as below within commons.py. As the smart contract now uses the correct logic for calculating the spendable Algo balance, this issue is closed.

Snippet - commons.py - Line 40-44

```
@Subroutine(TealType.uint64)
def get_algo_balance():
    """Get the amount of ALGO that this app can freely spend, i.e., that are
not
    locked by the accounts' minimum balance"""
    return           Balance(Global.current_application_address())          -
MinBalance(Global.current_application_address())
```

REFERENCES

Algorand Developer Portal - Minimum Balance for Smart Contract
https://developer.algorand.org/docs/get-details/parameter_tables/?from_query=minimum%20balance#minimum-balance-for-smart-contract

Algorand Developer Portal - Algorand Parameter Tables
https://developer.algorand.org/docs/get-details/parameter_tables/?from_query=minimum%20balance#minimum-balance

4.8. Insufficient Validation for RekeyTo Field

Low

Closed

BACKGROUND

Algorand transactions can have a *RekeyTo* field set to an address to allow future transactions to be authorized from the specified address. Rekeying is a powerful protocol feature but requires careful consideration as any transactions from logic signatures or accounts being re-keyed requires approval from the auth-addr field.

DESCRIPTION

Affected Code/File

Instance 1

- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/order_router_app.py [User_opt_into_assets, User_swap,User_swap_finalize]

Instance 2

- https://github.com/deflex-fi/deflex-contracts/blob/7ad3622e778bfeebc1d24b485e9c4ce75505f10c/src/alop/contracts/limit_order_app.py [clear_state]

It was noted that the affected smart contracts did not have sufficient validation against the *Rekey* field and allows rekeying to an arbitrary address as seen in the code snippets below. As the affected contracts are not a logic signatures, the attack vector for this finding is limited to modification of transaction details prior to signing due to front-end compromise if such transactions are generated from the front-end. The ultimate responsibility lies with the sender of the transaction to validate and ensure the transaction details are in-line with the intended goals of the transaction. Having on-chain validation only offers an additional transaction for transaction types that would not require rekeying operation under normal circumstances.

Example Code Snippet - order_router_app.py - Subroutine User_opt_into_assets

```
@ABIReturnSubroutine
def User_opt_into_assets() -> Expr:
    i = ScratchVar(TealType.uint64)
    self = Global.current_application_address()
    return Seq(
        Assert(is_txn_order_router_app_operation(Txn.group_index() + Int(2),
get_selector(OrderRouterAppApi.USER_SWAP))),
        For(i.store(Int(0)), i.load() < Txn.assets.length(), i.store(i.load()
+ Int(1))).Do(
            Seq(
                If(Txn.assets[i.load()] != ALGO_ID).Then(Seq(
                    asset_holding := AssetHolding.balance(self,
Txn.assets[i.load()]),
                    asset_holding,
                    If(Not(asset_holding.hasValue())).Then(Seq(
                        InnerTxnBuilder.Begin(),
                        InnerTxnBuilder.SetFields({
                                TxnField.type_enum: TxnType.AssetTransfer,
                                TxnField.xfer_asset: Txn.assets[i.load()],
```

```
                                TxnField.asset_receiver:
Global.current_application_address(),
                                TxnField.fee: Int(0),}),
                            InnerTxnBuilder.Submit(),
                            App.globalPut(VAR_GLOBAL_ASSETS_OPTED_IN,
                            App.globalGet(VAR_GLOBAL_ASSETS_OPTED_IN) +
Int(1)))))))),))
```

IMPACT

Once an account is re-keyed, it is no longer possible to sign transactions to the re-keyed account using the original private key. Insufficient validation of the Rekey field may allow logic signatures to be re-keyed and the assets held to be taken by an attacker.

RECOMMENDATIONS

For transactions that do not expect re-keying, validate that the Txn.RekeyTo() is set to Global.zero_address(). Otherwise, validate the Txn.RekeyTo() is set to the intended address of the new auth-addr.

COMMENT

Reviewed on 8 Nov 2022

Based on following commits to https://github.com/deflex-fi/deflex-contracts/tree/audit_fork, the documentation has been updated to highlight on the fact the funding transactions (payment or asset transfers) can have rekeying or closing address set.

- ef94fe6a9dcf4c4155fa3110373234d7c9e5cd4d
- f63e7f30c3d702a4282c0cabaf9eec9e65ab24a2

Updated Documentation - docs/order_router_app.md

```
Send the input amount from user to order-router app.
Note: the user can fund the input in any way desired, including
closing/rekeying
```

- Payment, only if input asset is ALGO
  a. Sender: Swapper Account
  b. Receiver: Order-Router App Account
  c. Amount: Input Amount
- Asset Transfer, only if input asset is non-ALGO
  a. Sender: Swapper Account
  b. Receiver: Order-Router App Account
  c. Amount: Input Amount
  d. Asset: Input Asset

Reviewed on 6 Nov 2022

Based on the updated repo at https://github.com/deflex-fi/deflex-contracts with commit ID b2ddd7fa54f8657ce3dbfdcedc493e095072ba02, following updates were noted along with a new subroutine asset_safety_checks under commons.py.

Code Snippet - commons.py [assert_safety_checks]

```
@Subroutine(TealType.none)
def assert_safety_checks() -> Expr:
    return Seq(
        Assert(Txn.rekey_to() == Global.zero_address()),
```

```
        Assert(Txn.close_remainder_to() == Global.zero_address()),
        Assert(Txn.asset_close_to() == Global.zero_address()),
        Return(),
    )
```

Instance 1 - order_router_app.py [User_opt_into_assets, User_swap, User_swap_finalize]

Instance 1 of this issue is closed as User_opt_into_assets, User_swap and User_swap_finalize methods of order_router_app.py now make use of subroutine assert_safety_checks() to validate RekeyTo, CloseRemainderTo, and AssetCloseTo field values.

Code Snippet - order_router_app.py [User_opt_into_assets]

```
@ABIReturnSubroutine
    def User_opt_into_assets() -> Expr:
        """Opt the order-router app into all assets in the foreign assets
array."""
        i = ScratchVar(TealType.uint64)
        self = Global.current_application_address()
        return Seq(
            assert_safety_checks(),
            <REDACTED>
```

Code Snippet - order_router_app.py [User_swap]

```
@ABIReturnSubroutine
    def User_swap(protocol : abi.Uint64,
                  asset_in: abi.Asset,
                  asset_out: abi.Asset,
                  app: abi.Application,
                  manager_app: abi.Application,
                  percent_bps_balance_amount_in: abi.Uint64,
                  amount_in_offset: abi.Uint64,
                  protocol_specific_arg: abi.DynamicBytes,
                  swap_note: abi.DynamicBytes) -> Expr:
        <REDACTED>
        return Seq(
            assert_safety_checks(),
<REDACTED>
```

Code Snippet - order_router_app.py [User_swap_finalize]

```
    @ABIReturnSubroutine
    def User_swap_finalize(asset_in: abi.Asset,
                           asset_out: abi.Asset,
                           min_amount_out: abi.Uint64,
                           type: abi.Uint64,
                           beneficiary: abi.Account,
                           treasury_address_param: abi.Account,
                           referrer_address: abi.Account,
                           fee_bps: abi.Uint64) -> Expr:
        <REDACTED>
        return Seq(
            assert_safety_checks(),
            <REDACTED>
```

It was also advised to highlight in the documentation that funding transactions (Payment or Asset Transfer) between User_opt_into_asset and User_swap can be of any form and allows rekeying and closing of assets or accounts, as long as the input amount of the correct assets are transferred to the order router's application account.

Instance 2 - limit_order_app.py [clear_state]

Instance 2 of this issue is closed. Based on the escrow lifecycle, the escrow is to be re-keyed to either the limit order application or the registry until Backend_close_escrow method is called successfully to close the escrow back to the user. As the clear_state application call can only be called when the escrow does not have any auth-addr set and for such scenario and the escrow is expected to just serve as a Algorand address that is used for holding local state and none of the assets or Algos, the risk to approving clear_state application calls without further validation is minimal.

REFERENCES

Algorand Developer Portal - Rekeying
https://developer.algorand.org/docs/get-details/accounts/rekey/?from_query=Rekey#create-publication-overlay

4.9. Inaccurate Comments

Observational

Closed

BACKGROUND

Code comments play a vital role in providing more human-readable context to the readers of the code so that technical specifications, transaction details and the expectations can be clearly conveyed to the users or future developer team members. Inaccurate comments could cause confusion to the developers and users who rely on the information provided in understanding the logic of smart contracts. Confusion in the business logic and technical specifications may not be critical in the short-term but for the continuity and maintainability of the smart contract projects, it is important to have the most updated details reflected in comments.

DESCRIPTION

Affected File/Code:

- https://github.com/deflex-fi/deflex-contracts/blob/e51855d854188b3325bee46a4ef2b9d25dbb2c73/src/alop/contracts/limit_order_app.py [do_opt_in]

It was noted that the smart contract code had incorrect comments for the following operation.

Code Snippet - limit_order_app.py - Line 781-785

```
    def do_opt_in() -> Expr:
        # make sure the transaction two positions after this transaction is a
call
        # to User_create_order for this escrow address
        esc_address = Txn.sender()
        return Seq(
```

Based on the provided documentation, the validation for the transaction position should be three positions after the current transaction.

IMPACT

Inaccurate comments could cause confusion to the developers and users who rely on the information provided in understanding the logic of smart contracts. Confusion in the business logic and technical specifications may not be critical in the short-term but could potentially end up with a wrong implementation which may have rooted from a wrong understanding of the existing smart contract logic.

RECOMMENDATIONS

Update comments according to the latest business logic and technical specification to ensure the understanding of the developers who read the code are coherent.

COMMENT

Reviewed on 14 Oct 2022

Based on the commit 786ac3103dc1d858294c26144d99b8d5a4f5ca84, incorrect parts of the comments have been updated with correct details per below.

Code Snippet - Limit Order

```
    def do_opt_in() -> Expr:
        # make sure the transaction three positions after this transaction is
a call
        # to User_create_order for this escrow address
        esc_address = Txn.sender()
        return Seq(
```

REFERENCES

CWE-1116 Inaccurate Comments
https://cwe.mitre.org/data/definitions/1116.html

5. Appendix

5.1. Disclaimer

The material contained in this document is confidential and only for use by the company receiving this information from Vantage Point Security Pte. Ltd. (Vantage Point). The material will be held in the strictest confidence by the recipients and will not be used, in whole or in part, for any purpose other than the purpose for which it is provided without prior written consent by Vantage Point. The recipient assumes responsibility for further distribution of this document. In no event shall Vantage Point be liable to anyone for direct, special, incidental, collateral or consequential damages arising out of the use of this material, to the maximum extent permitted under law.

The security testing team made every effort to cover the systems in the test scope as effectively and completely as possible given the time budget available. There is however no guarantee that all existing vulnerabilities have been discovered. Furthermore, the security assessment applies to a snapshot of the current state at the examination time.

5.2. Risk Rating

All vulnerabilities found by Vantage Point will receive an individual risk rating based on the following four categories.

Critical

A CRITICAL finding requires immediate attention and should be given the highest priority by the business as it will impact business interest critically.

High

A HIGH finding requires immediate attention and should be given higher priority by the business.

Medium

A MEDIUM finding has the potential to present a serious risk to the business.

Low

A LOW finding contradicts security best practices and have minimal impact on the business.

Observational

An OBSERVATIONAL finding relates primarily to non-compliance issues,security best practices or are considered an additional security feature that would increase the security stance of the environment which could be considered in the future version of smart contract.