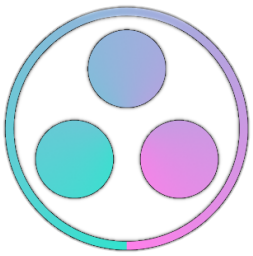
Avely Finance Audit Report

Smart Contract Security Assessment

Mar 28, 2022



ABSTRACT

Dedaub was commissioned to perform a security audit on the Avely Finance ZIL staking smart contracts, at commit hash `0fe2b1fdd5a8e54bcdd87b795130a6a8898d7a8b`. The code can be found here.

SETTING & CAVEATS

Avely Finance aZil protocol aims to provide a liquid ZIL staking service on the Zilliqa blockchain. Users who stake their ZIL using the protocol will receive some aZil tokens, which support the ZRC-2 FungibleToken transitions. As staking rewards accumulate in the protocol, the fair price (`total protocol ZIL / total token supply`) of the tokens is increasing.

When a user wants to withdraw some of his staked ZIL, he can either exchange his aZil tokens for ZIL through the protocol, or use any third-party exchange to perform the swap. The former option allows for a guaranteed pay-out at the current aZIL fair price, but with some time delay due to Zilliqa staking unbonding period limitations, while the latter allows for immediate swap of the assets, at the cost of some extra market inefficiency (slippage, potential exchange fees etc).

In order to circumvent the restrictions that SSNList imposes on staking regarding buffered staking funds and pending rewards, the aZil makes clever use of a set of 3 buffers, which are cycled through – at each point in time, there is an “active” buffer that receives any funds to be staked in the current reward cycle, while the other two “cooldown” (two reward cycles from the last time they were active). At the end of each round, any rewards and staked funds from all buffers are transferred to the Holder contract, which handles the staking of all the funds.

Most of the fund movement and buffer cycling is orchestrated with the help of some off-chain tools, which can trigger admin transitions on the aZil contract, which is owned by a multisig contract. The owner can trigger transitions that are concerned with the configuration of the aZil contract, but cannot trigger any admin actions.

We developed a “transition graph” notation to follow transitions in the contracts’ execution and reason about correctness. For instance, we can tell right away from the transition graph that there is no possibility of reentrancy: no untrusted transition (i.e., on a “*” or “**” contract) can be followed by one in aZil contract code. The transitions in the main contracts, together with an explanation of the notation appear below.

```
/*
The notation we use is:

    <contract>.<transition>
    operators are :, (..-->..), { .. ; .. ; ..}, [ .. | .. ]

    <initiator>: <contract>.<transition>   who can initiate the transaction
                                           (from the outside world)
    (..-->..) transition that causes another, i.e., the target of the first is
              initiator of the second
    { .. ; .. } sequencing of transitions, i.e., the same initiator starts the first
                transition, then the second, then...
    [ .. | .. ] initiator does either the first or the second (or ...) transition

    We use * to denote "any untrusted external client" and if we need to talk about
    two of them that are distinct, the second is **.
*/

owner: aZil.PauseIn
owner: aZil.UnPauseIn
owner: aZil.PauseOut
owner: aZil.UnPauseOut
owner: aZil.PauseZrc2
owner: aZil.UnPauseZrc2
owner: aZil.ChangeAdmin
owner: aZil.ChangeOwner
newOwner: aZil.ClaimOwner
owner: aZil.ChangeAzilSSNAddress
owner: aZil.ChangeTreasuryAddress
owner: aZil.SetHolderAddress
owner: aZil.ChangeZimplAddress
owner: aZil.ChangeBuffers


/// Mint/RecipientAcceptMint transitions can no longer arise: gZil mintin is over
admin: (aZil.DrainBuffer-->
        {(buffer.ClaimRewards-->proxy.WithdrawStakeRewards-->
          ssnlist.WithdrawStakeRewards-->
```

```
          {buffer.AddFunds; (gzil.Mint-->buffer.RecipientAcceptMint);
           (buffer.WithdrawStakeRewardsSuccessCallBack-->
            aZil.ClaimRewardsSuccessCallBack-->treasury.AddFunds)
          }
         );
         (holder.ClaimRewards-->proxy.WithdrawStakeRewards-->
          ssnlist.WithdrawStakeRewards-->
          {holder.AddFunds; (gzil.Mint-->holder.RecipientAcceptMint);
           (holder.WithdrawStakeRewardsSuccessCallBack-->
            aZil.ClaimRewardsSuccessCallBack-->treasury.AddFunds)
          }
         );
         (buffer.RequestDelegatorSwap-->proxy.RequestDelegatorSwap-->
          ssnlist.RequestDelegatorSwap);
         (holder.ConfirmDelegatorSwap-->proxy.ConfirmDelegatorSwap-->
          ssnlist.ConfirmDelegatorSwap)
        }

admin: (aZil.PerformAutoRestake-->buffer.DelegateStake-->
        proxy.DelegateStake-->ssnlist.DelegateStake-->
        buffer.DelegateStakeSuccessCallBack-->aZil.DelegateStakeSuccessCallBack)

admin: aZil.IncreaseAutoRestakeAmount
owner: aZil.ChangeRewardsFee
owner: aZil.UpdateStakingParameters

*: (aZil.DelegateStake-->buffer.DelegateStake-->proxy.DelegateStake-->
    ssnlist.DelegateStake-->buffer.DelegateStakeSuccessCallBack-->
    aZil.DelegateStakeSuccessCallBack-->*.DelegateStakeSuccessCallBack)

/// holder is allowed to start this transition, but does not
holder: (aZil.DelegateStakeSuccessCallBack-->*.DelegateStakeSuccessCallBack)

admin: (aZil.ClaimWithdrawal-->holder.CompleteWithdrawal-->
        proxy.CompleteWithdrawal-->ssnlist.CompleteWithdrawal-->
        [holder.CompleteWithdrawalNoUnbondedStakeCallBack |
         {holder.AddFunds;
          (holder.CompleteWithdrawalSuccessCallBack-->
           aZil.CompleteWithdrawalSuccessCallBack)
         }
        ]
       )

*: (aZil.WithdrawStakeAmt-->
    {(holder.WithdrawStakeAmt-->proxy.WithdrawStakeAmt-->
      ssnlist.WithdrawStakeAmt-->holder.WithdrawStakeAmtSuccessCallBack);
     *.WithdrawStakeAmtSuccessCallBack
```

```
    }
   )

*: (aZil.CompleteWithdrawal-->
    [{*.AddFunds; *.CompleteWithdrawalSuccessCallBack} |
     *.CompleteWithdrawalNoUnbondedStakeCallBack])

/// buffer is allowed to start this transition, but does not
buffer: aZil.CompleteWithdrawalSuccessCallBack

*: (aZil.ChownStakeConfirmSwap-->
    [(buffer.RejectDelegatorSwap-->proxy.RejectDelegatorSwap-->
      ssnlist.RejectDelegatorSwap) |
     (buffer.ConfirmDelegatorSwap-->proxy.ConfirmDelegatorSwap-->
      ssnlist.ConfirmDelegatorSwap)]
   )

admin: (aZil.ChownStakeReDelegate-->
        {(buffer.ClaimRewardsSsn-->proxy.WithdrawStakeRewards-->
          ssnlist.WithdrawStakeRewards-->
          {buffer.AddFunds; (gzil.Mint-->buffer.RecipientAcceptMint);
           (buffer.WithdrawStakeRewardsSuccessCallBack-->
            aZil.ClaimRewardsSuccessCallBack-->treasury.AddFunds)
          }
         );
         (buffer.ReDelegateStake-->proxy.ReDelegateStake-->
          ssnlist.ReDelegateStake-->buffer.ReDelegateStakeSuccessCallBack)
        }
       )

*: aZil.IncreaseAllowance
*: aZil.DecreaseAllowance

*: (aZil.Transfer-->{**.RecipientAcceptTransfer; *.TransferSuccessCallBack})

*: (aZil.TransferFrom-->{**.RecipientAcceptTransferFrom;
                         *.TransferFromSuccessCallBack})

/// One-time call, for initialization. Not part of regular workflow,
/// hence the unconnectedness.
*: (holder.DelegateStake-->proxy.DelegateStake-->
    ssnlist.DelegateStake-->holder.DelegateStakeSuccessCallBack)
```

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: 01) User or system funds can be lost when third party systems misbehave. 02) Limited DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming errors.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed", by the client, or "resolved", per the auditors.

CRITICAL SEVERITY

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	Denial of Service attack on withdrawals enabled by Holder.DelegateStake	**RESOLVED**

The `Holder.DelegateStake` transition is intended to be a one-time call at contract deployment time, but there is no logic that enforces this restriction. Moreover, this transition can be called by anyone:

```
(* Purpose of this transition is one-time min_deleg_stake delegate from Holder after its deploy *)
(* Else Zimpl will not know anything about Holder and will return DelegDoesNotExistAtSSN error *)
transition DelegateStake()
  (* Dedaub: Comment says this is a one-time transition, but it's not enforced *)
  accept;
  stake_amt = _amount;

  ssnaddr <- azil_ssn_address;
  var_zproxy_address <- zproxy_address;

  ProxyDelegate ssnaddr var_zproxy_address stake_amt
end
```

While this may seem innocent at first, it can be used to perform a Denial of Service attack. More specifically, suppose that `Holder.DelegateStake` has already been called once after deployment. Then an attacker can execute a DoS attack on the funds held in the holder which are registered with the SSN node by calling `Holder.DelegateStake` again.

Assuming the SSN node is active, the outline of the attack would look something like the following:

1. External caller calls `Holder.DelegateStake` with an amount of at least 10000000000000 QA (10 ZIL - size of mindelegstake field in SSNList contract)
2. Holder calls `Proxy.DelegateStake`; initiator is Holder
3. Proxy calls `SSNList.DelegateStake`
4. SSNList calls `SSNList.Delegate`

5. IsDelegstakeSufficient procedure passes successfully
6. Buffered deposit for holder is updated due to line 1078 in the SSNList contract:

```
buff_deposit_deleg[initiator][ssnaddr][lrc] := new_stake_amt_for_deleg
```

7. All funds staked by `holder` are now blocked from Withdrawal, due to the restrictions that SSNList places (no Buffered Deposits in order to Withdraw)

This attack can be repeated during future cycles and is magnified by the fact that Holder cannot be upgraded once installed in aZil.

It is highly recommended that additional logic is added at the beginning of the `Holder.DelegateStake` transition that enforces the intended single-call behavior, which will also prevent this DoS vector.

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Local SSNList address fields may become outdated	**RESOLVED**

Both Buffer and Holder have a locally "cached" version of the SSNList address, that are initialized once at contract creation and never written to again:

```
field zimpl_address : ByStr20 = init_zimpl_address
```

This field is used extensively in the two contracts, in the form of access-control logic, specifically:

```
procedure RequireZimpl()
  var_zimpl_address <- zimpl_address;
  is_zimpl = builtin eq _sender var_zimpl_address;
  match is_zimpl with
```

```
  | True  =>
  | False =>
      e = ZimplValidationFailed;
      ThrowError e
  end
end
```

While the SSNList contract is highly unlikely to be updated, it is designed to do so (implementation hidden behind a proxy which routes the calls to the SSNList contract), and when it does the locally stored addresses in Buffer and Holder will be outdated, causing them to not function correctly.

It is highly recommended that the `zimpl_address` be updated in each transition that might read it by directly querying the proxy contract, in order to prevent any issues that could arise from an outdated locally cached address.

Similar "automation" could be added to the `aZil` contract as well, although it's much less critical in that case, due to the ability to manually update the field.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend addressing them.

ID	Description	STATUS
A1	Field can be simplified from `Uint32` to `Bool`	**RESOLVED**

In `aZil`, the field `tmp_buffer_exists_at_ssn` is declared as `Uint32`:

```
field tmp_buffer_exists_at_ssn: Uint32 = uint32_zero
```

However, all writes to this field are either 0 or 1, and all reads from it are followed up by an equality check with 0 and a match statement - the field is a boolean à la C.

It is recommended that the field be declared `Bool`, in order to improve code readability and simplify the snippets that read from it.

A2	Field assignment sequence can be simplified	**RESOLVED**

In the `aZil.CalculateTotalWithdrawalBlock` procedure, can be simplified:

```
procedure CalculateTotalWithdrawalBlock(deleg_withdrawal: Pair ByStr20 Withdrawal)
  match deleg_withdrawal with
  | Pair delegator withdrawal =>
      match withdrawal with
      | Withdrawal withdraw_token_amt withdraw_stake_amt =>
        match withdrawal_unbonded_o with
        | Some (Withdrawal token stake) =>
          updated_token = builtin add token withdraw_token_amt;
          updated_stake = builtin add stake withdraw_stake_amt;
          unbonded_withdrawal = Withdrawal updated_token updated_stake;
          withdrawal_unbonded[delegator] := unbonded_withdrawal
        | None =>
        (* Dedaub: This branch can be simplified to
               withdrawal_unbonded[delegator] := withdrawal
        *)
          unbonded_withdrawal = Withdrawal withdraw_token_amt withdraw_stake_amt;
          withdrawal_unbonded[delegator] := unbonded_withdrawal
      end
    end
  end
end
```

The inner match's None case can become:

```
| None =>
    withdrawal_unbonded[delegator] := withdrawal
end
```

A3	Logic of `multisig_wallet.RevokeSignature` can be simplified	**DISMISSED**

In `multisig_wallet`, `RevokeSignature` can be simplified. The transition checks whether there are zero signatures through

```
c_is_zero = builtin eq c zero;
```

But for this line of code to execute

```
exists signatures[transactionId][_sender];
```

must have already been true. Therefore it is guaranteed that there is at least one signature, and c_is_zero cannot be 0.

Thus the following transition can be simplified:

```
(* Revoke signature of existing transaction, if it has not yet been executed. *)
transition RevokeSignature (transactionId : Uint32)
  sig <- exists signatures[transactionId][_sender];
  match sig with
  | False =>
       err = NotAlreadySigned;
       MakeError err
  | True =>
       count <- signature_counts[transactionId];
       match count with
       | None =>
       err = IncorrectSignatureCount;
       MakeError err
       | Some c =>
       c_is_zero = builtin eq c zero;
       match c_is_zero with
       | True =>
       err = IncorrectSignatureCount;
       MakeError err
       | False =>
       new_c = builtin sub c one;
       signature_counts[transactionId] := new_c;
       delete signatures[transactionId][_sender];
       e = mk_signature_revoked_event transactionId;
       event e
       end
       end
  end
end
```

By replacing the Some c branch with the following:

```
Some c =>
       new_c = builtin sub c one;
       signature_counts[transactionId] := new_c;
       delete signatures[transactionId][_sender];
       e = mk_signature_revoked_event transactionId;
       event e
```

A4	Logic of `azil.DrainBuffer` logic can be simplified	**RESOLVED**

Transition `DrainBuffer` of `aZil` also admits some simplification: a bind action can be factored out, since it occurs in both cases of a match, and another binding is redundant, both shown in comments below.

```
transition DrainBuffer(buffer_addr: ByStr20)
  RequireAdmin;
  buffers_addrs <- buffers_addresses;
  is_buffer = is_buffer_addr buffers_addrs buffer_addr;

  match is_buffer with
  | True =>
      FetchRemoteBufferExistsAtSSN buffer_addr;
      (* local_lastrewardcycle updated in FetchRemoteBufferExistsAtSSN *)
      lrc <- local_lastrewardcycle;
      RequireNotDrainedBuffer buffer_addr lrc;
      var_buffer_exists <- tmp_buffer_exists_at_ssn;
      is_exists = builtin eq var_buffer_exists uint32_one;
      match is_exists with
      | True =>
          holder_addr <- holder_address;
          ClaimRewards buffer_addr;
          ClaimRewards holder_addr;
          RequestDelegatorSwap buffer_addr holder_addr;
          ConfirmDelegatorSwap buffer_addr holder_addr
      | False =>
          holder_addr <- holder_address;
          (* Dedaub: This is also done in the True branch of the match *)
          ClaimRewards holder_addr
      end
  | False =>
      e = BufferAddrUnknown;
      ThrowError e
  end;
  lrc <- local_lastrewardcycle;
   (* Dedaub: extraneous, it was already done above in the True case,
      and the False case is irrelevant *)
  buffer_drained_cycle[buffer_addr] := lrc;
  tmp_buffer_exists_at_ssn := uint32_zero
end
```

A5	Buffer/Holder have permissions for transitions they will never execute	**DISMISSED**
As can be seen in the earlier transition graph, Buffer is allowed to initiate `aZil.CompleteWithdrawalSuccessCallBack` but never will. Holder is allowed to initiate `aZil.DelegateStakeSuccessCallBack` but never will.		

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contracts. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.