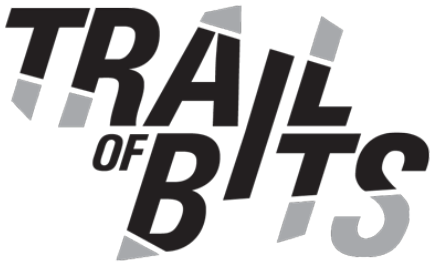

Dexter

Security Assessment

June 22, 2020

Prepared For:
Tyler Clark | *camlCase*
tyler@camlcase.io

Prepared By:
Josselin Feist | *Trail of Bits*
josselin.feist@trailofbits.com

Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Changelog:

June 22, 2020:	Initial report delivered
August 13, 2020:	Added Appendix E with retest results
September 8, 2020:	Update Appendix E with new results

Executive Summary

Project Dashboard

Code Maturity Evaluation

Engagement Goals

Coverage

Recommendations Summary
- Short Term
- Long Term

Findings Summary
- 1. updateTokenPool can be abused to drain Dexter's assets
- 2. tokenToXtz sends the tokens to the user
- 3. The allowance is incorrectly updated after removing liquidity
- 4. Morley contracts are not properly tested
- 5. dexter.Approve is not compatible with FA1.2.Approve
- 6. Discrepancy between the informal specification and the Morley contract on approve
- 7. Improper use of Haskell type system to enforce type correctness in the stack
- 8. Call injection allows price corruption
- 9. Arithmetic rounding allows minting of liquidity tokens without payment of tokens
- 10. Arithmetic rounding might allow funds to be drained
- 11. Lack of "amount sent" protection can lead to trapped tezos
- 12. User-provided inputs are not properly validated in the frontend
- 13. Users can be tricked into adding liquidity for a baker that immediately changes
- 14. Deadline for transactions are fixed at two hours from now

A. Vulnerability Classifications

B. Code Maturity Classifications

C. Code Quality

D. Call Injection vulnerability

E. Fix Log
- Detailed Fix Log

Executive Summary

From June 8 through June 19 2020, Tezos foundation engaged with Trail of Bits to review the security of the Dexter exchange. Trail of Bits conducted this assessment over the course of four person-weeks with two engineers working from 8c39498a. We focused on the Morley implementation of the contract, as the Dexter's team mentioned it was of high importance.

In the first week, we gained an understanding of the codebase and its different components. We checked the access control and reviewed the contract for the most common smart contract flaws. In the final week, we reviewed the contract's external interaction, and the arithmetics. We also partially reviewed the frontend.

In total, Trail of Bits identified 14 issues, ranging from high to informational severity. Two critical issues (high severity, low difficulty) are the direct result of the Tezos' message passing design (see TOB-DEXTER-001, TOB-DEXTER-008, and Appendix D). These issues are likely to occur in other Tezos contracts and might create systematic flaws in the Tezos ecosystem. Two other critical issues stem from the low-level aspect of Morley: using an incorrect variable as the token's receiver (TOB-DEXTER-002), and updates to allowances (TOB-DEXTER-003).

Overall, the codebase represents a significant work in progress. The fact that some of the high-severity issues are triggered without attacker intervention signals that the codebase would benefit from more testing. Morley has an underlying complexity, and does not currently provide the safeguards that are needed for a stack-based language.

As it stands, we do not recommend deploying the codebase in production. We do recommend:

- Fixing all the reported vulnerabilities.
- Adding tests for every function.
- Improving Morley's type safety system or using another smart contract language.
- Discussing the call injection vulnerability (TOB-DEXTER-008 and Appendix D) and the callback authorization bypass (TOB-DEXTER-001) with Tezos development stakeholders. Tezos should mitigate these issues at the protocol level, or consider moving from message passing architecture to direct call execution.

Update - September 8, 2020
As a result of the assessment, camlCase addressed 10 of the 14 reported issues. Additionally, camlCase significantly increased unit test coverage. Appendix E presents the review of the fixes done by Trail of Bits on August 11. One new call injection vulnerability was found in the fixes, which was then mitigated by camlCase. Trail of Bits reviewed the new mitigation on September 8.

Project Dashboard

Application Summary

Name	Dexter
Version	8c39498a
Type	Morley
Platforms	Tezos

Engagement Summary

Dates	June 8 through June 19, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	5	■ ■ ■ ■ ■
Total Medium-Severity Issues	3	■ ■ ■
Total Low-Severity Issues	3	■ ■ ■
Total Informational-Severity Issues	2	■ ■
Total Undetermined-Severity Issues	1	■
Total	14	

Category Breakdown

Dava Validation	6	■ ■ ■ ■ ■ ■
Access Controls	1	■
Error Reporting	1	■
Undefined Behavior	3	■ ■ ■
Patching	1	■
Auditing and Logging	1	■
Timing	1	■
Total	14	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Weak.** Two high-severity issues lead users to have more privileges than they should.
Arithmetic	**Further Investigation Required.** One rounding issue was found, and more may be present. Additionally, the current Tezos ecosystem does not facilitate testing and verification of arithmetic properties.
Centralization	**Satisfactory.** The owner has limited control over the exchange. However, users must be careful to interact with exchanges that were properly initialized.
Upgradeability	**Not Applicable.**
Function Composition	**Moderate.** Some components are written multiple times, and the codebase would benefit from code reuse.
Front-Running	**Satisfactory.** The exchange functions have bounds on their outcome, limiting the impact of front-running.
Monitoring	**Not Considered.** Tezos has no inbuilt events for smart contracts. We were not aware of monitoring tools developed for Dexter.
Specification	**Moderate.** The informal specification has several mismatches with the implementation.
Testing & Verification	**Weak.** Several functions were not tested. The existing tests have a low coverage. Additionally there was no use of a CI.

Engagement Goals

The engagement was scoped to provide a security assessment of the Dexter smart contract, its informal specification, and its frontend.

Specifically, we sought to answer the following questions:

- Are appropriate access controls set for the manager/baker/user roles?
- Does arithmetic regarding token exchanges hold?
- Is there any arithmetic overflow or underflow affecting the code?
- Does the use of arbitrary precision numbers introduce any security or correctness issues?
- Can race conditions or denial-of-service attacks occur to block other users from using the contract?
- Does the code prevent common issues such as:
    - Incorrect minting, burning, locking, and allocation of tokens?
    - Authorization issues or accidentally centralized features?
    - Incorrect dust collection and arithmetic calculations?
- Does the contract implementation match its informal specification?
- Does the specification provide enough detail, and is it internally consistent?
- Does the frontend successfully execute transactions?
- Does the frontend properly validate inputs from users?

Coverage

Trail of Bits manually reviewed the Dexter contract, its informal specification, and its frontend.

The Dexter contract was written in Morley, a low-level stack based language, based on Haskell. Unfortunately, there were no tools available for this language, so Trail of Bits manually reviewed the code for adherence to the informal specification. Review focused on:

- **Tezos-specific flaws:** The Tezos blockchain has certain properties and architecture that differentiate it from others like Ethereum or Hyperledger. Certain types of attacks like reentrancy are mitigated, but at the cost of a more complex way of functioning. We reviewed the Dexter contract for specific Tezos flaws and we found several ways of breaking important invariants (TOB-DEXTER-001, TOB-DEXTER-008).
- **Access controls.** Many parts of the system expose privileged functionality, such as setting the baker address or syncing the token balance. We reviewed these functions to ensure they can only be triggered by the intended actors and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the exchange.
- **Proper Testing:** Testing a smart contract is key to detect issues in the early stages of its development. We reviewed contract code coverage as well as how unit tests and properly-based testing are implemented, to avoid any blind-spots.
- **Informal specification:** The Dexter contract is detailed in its informal specification. We reviewed it to make sure it is consistent and does not leave any important detail unspecified.
- **Frontend:** Some users will prefer to use the UI in the frontend to sign transactions that interact with the exchange contract. We reviewed the frontend input validation to make sure transactions are correctly crafted. We also checked that the internal calculation performed by the fronted matches the ones in the contracts as well as the interaction with third-party components such as Beacon and TezBridge.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Remove `updateTokenPool`.** This function is vulnerable to a callback authorization bypass (TOB-DEXTER-001) and increases the likelihood of call injection exploits (TOB-DEXTER-008).

❑ **Send the tokens to the Dexter contract in `TokenToXtz`.** The tokens are incorrectly sent to the user-controlled to field, allowing anyone to drain the contract's funds (TOB-DEXTER-002).

❑ **Decrease the allowance by its expected value in `RemoveLiquidity`.** The allowance is updated with the account's balance, leading an allowed user to drain all the account's funds (TOB-DEXTER-003).

❑ **Increase the test coverage and follow all the testing practices detailed in TOB-DEXTER-004.** Several of the findings, including high-severity ones, would have been caught had testing followed the best practices available..

❑ **Remove `currentAllowance` from `approve` or clearly document that the function does not follow the FA12 standard.** The additional parameter does not follow the standard and can confuse third-party integration (TOB-DEXTER-005).

❑ **Allow `self-approve` in the ligo contract and the informal specification.** The ligo contract, the informal specification, and the Morley contract have different behavior on `self-approve`, which can lead to unexpected behavior for users (TOB-DEXTER-006).

❑ **Avoid the use of `forcedCoerce_`. Carefully implement specific coercion instances if needed.** This will keep the code's type inference strict and prevent security or correctness issues caused by type confusion. (TOB-DEXTER-007)

❑ **Use the Haskell type to ensure that values cannot be confused in the stack. Consider using `newtype` instead of `type`, and implement all the explicit conversions and type instances.** Morley's current type system does not provide the expected guarantees (TOB-DEXTER-007).

❑ **Evaluate all the functions against injection call vulnerability.** Consider the mitigations highlighted in TOB-DEXTER-008.

❑ **Prevent `tokensDeposited` from being zero in `AddLiquidity`.** Users can receive liquidity without sending tokens due to arithmetic rounding (TOB-DEXTER-009).

❑ **Revert if the transaction's amount is non-zero for all functions that do not use the amount.** Allowing amounts to be non-zero in those functions can make users lose funds (TOB-DEXTER-011).

❑ **Properly validate input values in the frontend:**

- **Enable checksum validation when parsing Tezos addresses.**
- **Split the price in two inputs, one for the integer part, one for the fractional part.**
- **Ask the user to validate the parameters before sending the transaction.**

This will mitigate the risk of users applying unexpected values in their transactions when using the frontend. (TOB-DEXTER-012).

❑ **Add a visible warning to show the user that the current baker can be changed at any time**. This will avoid to confuse the user about who is the baker that will receive the rewards of the contracts. (TOB-DEXTER-013).

❑ **Allow users to set the `deadline` parameter in the frontend.** Users might require a smaller or larger `deadline` (TOB-DEXTER-014).

Long Term

❑ **Start a discussion with the Tezos development team and stakeholders regarding the message passing design.** Two high-severity issues were due to Tezos' message passing design of (TOB-DEXTER-001, TOB-DEXTER-008, Appendix D).

❑ **Create thorough unit tests and follow testing best practices. Consider using QuickCheck for every feature and adding a CI.** Several of the findings, including high-severity ones, would have been caught with a complete suite of unit tests (TOB-DEXTER-002, TOB-DEXTER-003, TOB-DEXTER-004).

❑ **Carefully read any standard you interact with and follow. Thoroughly test the standard's specification.** If the contract is meant to follow a standard, it should follow it correctly to prevent misuse (TOB-DEXTER-005, TOB-DEXTER-006).

❑ **Use fuzzing or formal verification to make sure arithmetic rounding is bounded and cannot be used to take advantage of the exchange.** The imprecisions of Dexter's arithmetic operations may allow users to profit from the system (TOB-DEXTER-009, TOB-DEXTER-010).

❑ **Carefully review the known attack vectors of the existing blockchain and ensure Dexter and Morley have the proper mitigations.** The Tezos ecosystem should take advantage of the existing documentation on smart contract vulnerabilities and build proper mitigations (TOB-DEXTER-011).

❑ **Design input validation of user-provided values to be bullet-proof: Fail early if values are invalid and ask users to re-confirm them, especially if there is ambiguity (e.g., using dots or commas for decimal separators).** This will minimize the risk of users misusing the on-chain or off-chain components (TOB-DEXTER-012).

❑ **Make sure users have all the relevant information in the UI before confirming a blockchain transaction.** This will avoid to confuse users regarding the operation they are performing (TOB-DEXTER-013).

❑ **Do not hardcode parameters in off-chain components unless it is absolutely necessary to avoid security or correctness issues.** This will allow users to modify these parameters if needed (TOB-DEXTER-014).

Findings Summary

#	Title	Type	Severity
1	`updateTokenPool` can be abused to drain Dexter's assets	Access Control	High
2	`tokenToXtz` sends the tokens to the user	Data Validation	High
3	The allowance is incorrectly updated after removing liquidity	Undefined Behavior	High
4	Morley contracts are not properly tested	Error Reporting	Medium
5	`dexter.Approve` is not compatible with FA1.2.`Approve`	Undefined Behavior	Low
6	Discrepancy between the informal specification and the morley contract on `approve`	Undefined Behavior	Low
7	Improper use of Haskell type system to enforce type correctness in the stack	Patching	Informational
8	Call injection allows price corruption	Data Validation	High
9	Arithmetic rounding allows minting of liquidity tokens without payment of tokens	Data Validation	Medium
10	Arithmetic rounding might allow funds to be drained	Data Validation	Undetermined
11	Lack of amount sent protection can lead to trapped Tezos	Data Validation	Medium
12	User-provided inputs are not properly validated in the front-end	Data validation	High
13	Users can be tricked into adding liquidity for a baker that immediately changes	Auditing and Logging	Low

14	Deadline for transactions are fixed at two hours from now	Timing	Informational

1. `updateTokenPool` can be abused to drain Dexter's assets

Severity: High
Type: Access Control
Target: `Contract.hs`

Difficulty: Low
Finding ID: TOB-DEXTER-001

Description
The lack of caller verification context in `updateTokenPool` allows anyone to set an arbitrary `tokenTool` value and control the exchange's price.

`updateTokenPool` is supposed to be called through `getBalance`'s callback from a FA1.2 token:

```
-- | The token pool value in Dexter may become out of sync with its true value
-- in FA1.2. For gas saving purposes, this has been abstracted out into its own
-- entrypoint. To use this, call the getBalance entrypoint of FA1.2 and pass the
-- dexter contract in with the updateTokenPool entrypoint.
updateTokenPool :: Entrypoint UpdateTokenPoolParams Storage
updateTokenPool = do
  -- assert that the sender is the tokenAddress
  duupX @2; stToField #tokenAddress; sender; assertEq (mkMTextUnsafe "Only the
tokenAddress can update the tokenPool.")
  stackType @[UpdateTokenPoolParams, Storage]
  -- set the tokenPool
  stSetField #tokenPool
  nil; pair
```

Figure 1.1: `Contract.hs#L98-L109`.

```
updateTokenPool(new_value):
     require(sender == tokenAddress)
     tokenPool = new_value
```

Figure 1.2: High-level representation of Figure 1.1.

The FA1.2 standard specifies three view functions:

- `(view (address :owner, address :spender) nat) %getAllowance`
- `(view (address :owner) nat) %getBalance`
- `(view unit nat) %getTotalSupply`

Any of these functions can callback to `updateTokenPool`. `getAllowance` will return a user-controlled value. As a result, an attacker can set an arbitrary value to `tokenPool` and gain direct control over the exchange's price.

This issue also affects the LIGO implementation.

Exploit Scenario
Dexter holds $1,000,000 worth of assets. Eve uses `getAllowance` to callback to `updateTokenPool`, compromise the pool's price, and drain Dexter's funds.

Recommendation
Short term, remove `updateTokenPool`.

This issue is likely to affect any contract interacting with a FA1.2 token. We recommend starting a discussion with Tezos developers and major stakeholders to update the current calling mechanism in Tezos and mitigate this type of vulnerability (see Appendix D).

2. `tokenToXtz` sends the tokens to the user

Severity: High
Type: Data Validation
Target: `TokenToXtz.hs`

Difficulty: Low
Finding ID: TOB-DEXTER-002

Description
`tokenToXtz` sends the received tokens to a user-controlled address, which allows an attacker to drain all the assets from `Dexter`.

`tokenToXtz` allows the conversion from tokens to tezos. The user specifies how many tokens he wants to convert. The tokens are supposed to be sent to the contract, but they are actually sent to the user-controlled `to` field:

```
dip (do dip (do getField #tokensSold; dip (do getField #owner; dip (toField #to))); transferOwnerTokensTo)
```

Figure 2.1: `TokenToXtz.hs#L58.`

```
transferOwnerTokensTo(tokensSold, owner, to)
```

Figure 2.2: High-level representation of Figure 2.1.

As a result, anyone can call `tokenToXtz` and receive tezos while keeping its tokens.

Exploit Scenario
Dexter holds $1,000,000 worth of assets. Eve uses `tokenToXtz` to drain the exchange's assets.

Recommendation
Short term, send the tokens to the Dexter contract.

Long term, create thorough unit tests and test each function. Consider using QuickCheck for every feature.

3. The allowance is incorrectly updated after removing liquidity

Severity: High
Type: Undefined Behavior
Target: `RemoveLiquidity.hs`

Difficulty: Low
Finding ID: TOB-DEXTER-003

Description
`RemoveLiquidity` sets the sender's allowance to the owner's balance. As a result, an attacker can empty any account for which they have any allowance amount.

The allowance is the mechanism whereby one contract (the allower) gives permission to another contract (the allowee) to consume the allower's liquidity in the Dexter contract.

According to the informal specification of the Dexter contract, after removing liquidity, the allowance left should be updated according to the following formula:

```
accounts[owner].allowance[sender] -= lqt_burned
```

Figure 1.1: Informal specification of the allowance computation after executing `RemoveLiquidity`.

However, the implementation in the Morley contract is incorrect. The allowance is updated using the current liquidity balance:

```
  -- | removeLiquidity without operations
removeLiquidity :: RemoveLiquidityParams : Storage : s :-> Storage : TokensWithdrawn :
XtzWithdrawn : RemoveLiquidityParams : s
removeLiquidity = do
 ...
 stackType @(Storage : LqtBalance : TokensWithdrawn : XtzWithdrawn : RemoveLiquidityParams
: _)
 -- update allowance
 -- setAllowance :: Owner : Sender : Natural : Storage : s :-> Storage : s
 swap; sender; duupX @6; toField #owner; setAllowance
 ...
```

Figure 2.2: `RemoveLiquidity.hs#L55-L98`.

```
removeLiquidity(owner, to, lqtBurner, ...)
  ...
```

```
lqtBalance = balance[owner]
...
setAllowance(owner, sender, lqtBalance)
```

Figure 2.3: High-level representation of Figure 2.2.

Here, the call to `setAllowance` sets the caller's allowance to the owner's balance. As a result, after a successful call to `setAllowance`, the caller will be allowed to remove the owner's entire balance.

Exploit Scenario
Alice holds $1,000,000 worth of assets in the Dexter contract. She allows Eve to remove $1,000. Eve calls `RemoveLiquidity` for $1,000, and now has an allowance of $1,000,000. Eve calls `RemoveLiquidity` again and drains all of Alice's assets.

Recommendation
Short term, fix the computation of the allowance to decrease it as detailed in the informal specification.

Long term, create thorough unit tests and test each function. Consider using QuickCheck for every feature.

4. Morley contracts are not properly tested

Severity: Medium
Type: Error Reporting
Target: `tests`

Difficulty: Low
Finding ID: TOB-DEXTER-004

Description
The Dexter contracts implemented in Morley have only a few tests, so many features are not tested and the likelihood of bugs is high.

Using unit tests in smart contracts is a very extensive practice among developers. Despite its simplicity and limitations, this technique effectively detects early inconsistencies in code behavior before deployment.

The contracts written in Morley are tested with a combination of unit tests and property-based testing using the hspec library.

```
main :: IO ()
main = do
 hspec AddLiquidity.spec
 hspec RemoveLiquidity.spec
 hspec XtzToToken.spec
 hspec TokenToXtz.spec
```

Figure 4.1: `Spec.hs#L16-L21`.

However, we noticed several flaws in the implementation of the tests. Some of them may be related to other issues in this audit, including TOB-DEXTER-002 and TOB-DEXTER-003. Most importantly, the testing scripts do not cover all contract functionality, and some code is not covered at all. For instance, the `setAllowance` function is never tested:

```
setAllowance :: Owner : Sender : Natural : Storage : s :-> Storage : s
setAllowance = do
 dup; duupX @3; eq;
 if_
   (do drop; drop; drop)
   (do
       dup; duupX @5; swap; toAccount
       stackType @( Account : Owner : Sender : Natural : Storage  : _)
       getField #approvals; dig @4; some; dig @4; update; setField #approvals
       stackType @( Account : Owner : Storage  : _)
       some; swap; dip (dip (getField #accounts)); update; setField #accounts
```

```
    )
```

Figure 4.2: `Core.hs#L206-L217`.

```
setAllowance(owner, sender, allowance, ...)
  if sender == owner
    return

  approvals[sender][owner] = allowance
```

Figure 4.3: High-level representation of Figure 4.2.

The testing of the contract relies on the simulation of Michelson instructions using the `interpretLorentzInstr` function. However, this is not precise enough. It can be improved to include side effects produced in the contracts tested, such as the current balance after execution:

```
-- | Interpret a Lorentz instruction, for test purposes.
interpretLorentzInstr
  :: (IsoValuesStack inp, IsoValuesStack out)
  => ContractEnv
  -> inp :-> out
  -> Rec Identity inp
  -> Either MichelsonFailed (Rec Identity out)
interpretLorentzInstr env (compileLorentz -> instr) inp =
  fromValStack <$> interpretInstr env instr (toValStack inp)
```

Figure 4.4: `interpretLorentzInstr` *function in* `Lorentz.Run`.

Another issue with the current testing approach is the lack of checks when transactions are tested for failures:

```
-- | Represents `[FAILED]` state of a Michelson program. Contains
-- value that was on top of the stack when `FAILWITH` was called.
data MichelsonFailed where
  MichelsonFailedWith :: Val Instr t -> MichelsonFailed
  MichelsonArithError :: ArithError (CVal n) (CVal m) -> MichelsonFailed
  MichelsonGasExhaustion :: MichelsonFailed
  MichelsonFailedOther :: Text -> MichelsonFailed
```

Figure 4.5: `MichelsonFailed` *data type in* `Michelson.Interpret`.

The testing code ensures that some transactions fail in certain tests, but they don't check why. This may cause the operation to fail as expected, but not for the reason expected (e.g., lack of gas).

Additionally, the use of property-based testing is very incomplete; it only covers a few functions. Since QuickCheck is already used in the tests, it should be extended with more properties.

Finally, we could not verify the use of continuous integration tests.

Exploit Scenario
The Dexter contracts are developed without proper testing, allowing, the code to be deployed with a critical bug in it.

Recommendation
Short term:

- Extend the unit tests to cover all the contract code and its functionalities.
- Make sure the testing of simulated Michelson code is as faithful as possible.
- Verify the cause of the failed transactions in the tests.
- Extend the property-based testing to cover more of the contract functionality.
- If they are not already used, set up continuous integration tests to run the tests at every commit.

Long term, follow standard testing practice for smart contracts to minimize the amount of issues during development.

5. `dexter.Approve` is not compatible with FA1.2.`Approve`

Severity: Low
Type: Undefined Behavior
Target: `Core.hs`

Difficulty: Low
Finding ID: TOB-DEXTER-005

Description
Dexter's documentation mentions FA1.2's approve function, but the contract's implementation does not follow the standard. As a result, third-party integration might not work as expected.

`Core.hs` defines the `approve` function as a `FA12 data`:

```
data FA12
   = Transfer  TransferParams
   | Approve_  ApproveParams
   deriving (Generic, IsValue)
```

Figure 5.1: `Core.hs#L155-L158.`

Dexter's `approve` takes three parameters: `spender`, `allowance`, `currentAllowance`:

```
type ApproveParams =
  ( "spender"          :! Address -- the third party address that will be granted liquidity
burn rights
  , "allowance"        :! Natural -- the amount the third party will be allowed to burn
  , "currentAllowance" :! Natural -- a confirmation of the spender's current amount, a
security measure to
  )
```

Figure 5.2: `Core.hs#L42-L46.`

The FA1.2 standard defines `approve` as taking two parameters:

```
(address :spender, nat :value) %approve
```

Figure 5.3: `Approve`*'s interface.*

As a result, Dexter's `approve` is not compatible with the FA1.2 standard.

Exploit Scenario
Bob develops a contract that is meant to interact with Dexter, but Bob's contract follows the FA12 standard, so itt does not work with Dexter.

Recommendation
Short term, either:

- Remove `currentAllowance` and follow the non-zero to non-zero specification of FA1.2.approve (preventing the known front-running issue), or
- Clearly document that Dexter's `approve` is not compatible with FA1.2.

Long term, carefully read any standard you interact with and follow. Thoroughly test the standard's specification.

6. Discrepancy between the informal specification and the Morley contract on `approve`

Severity: Low
Type: Undefined Behavior
Target: `RemoveLiquidity.hs`

Difficulty: Low
Finding ID: TOB-DEXTER-006

Description
In `approve`'s informal specification, the function reverts if the spender is the caller. The Morley contract does not follow this behavior, which might result in unexpected behavior for users and third-party integration.

According to the informal specification, if the sender is equal to the spender in `approve`, the operation should fail:

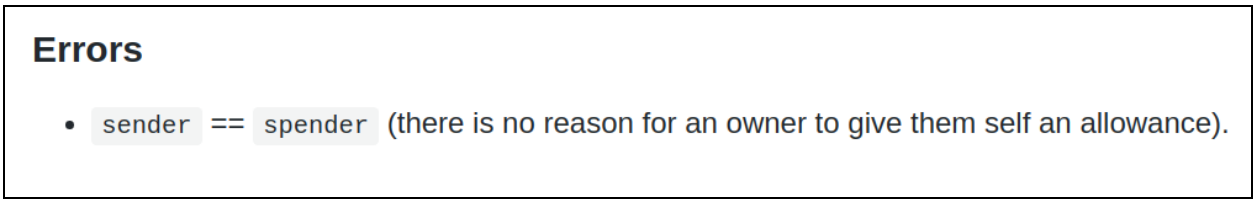

Figure 6.1: `Dexter-informal-specification.md#errors`.

The implementation in the Morley contract does not follow this behavior. Rather, owners can set an allowance for themselves:

```
-- | Give approval for a third party to burn liquidity tokens () owned by
approve :: Entrypoint ApproveParams Storage
approve = do
 duupX @2; toField #accounts; sender; get;
 ifNone (constructT @("balance" :! Natural, "approvals" :! Map Address Natural) (
fieldCtor $ push 0 >> toNamed #balance, fieldCtor $ emptyMap >> toNamed #approvals)) nop
 stackType @[Account, ApproveParams, Storage]

 ...
```

Figure 6.2: `Approve.hs#L38-L50`.

```
approve(spender, allowance, ..):
   ..
   approvals[sender][spender] = allowance
```

Figure 6.3: High-level representation of Figure 6.1.

The Ligo implementation matches the informal specification:

```
function approve(const spender  : address;
                 const allowance: nat;
                 const current_allowance: nat;
                 var storage      : storage):
                 return is
 block {
   if (spender =/= sender) then {
     ...
   } else {
     failwith("approve: the spender must not be the sender. The owner already has rights
to all of the LQT.");
   }
 } with (empty_ops, storage);
```

Figure 6.4: `Dexter.ligo#L90-L121`.

As a result, there is a discrepancy between the informal specification, the ligo, and the Morley implementation.

In this corner case, the Morley implementation follows the FA1.2.approve standard.

Exploit Scenario
Alice implements an on-chain or off-chain component to interact with the Dexter contract relying on the contract specification. As a result, her code does not work.

Recommendation
Short term, modify the Dexter specification and the Ligo contract to match the FA1.2 standard.

Long term, create thorough unit tests and test each function. Consider using QuickCheck for every feature.

7. Improper use of Haskell type system to enforce type correctness in the stack

Severity: Informational
Type: Patching
Target: Dexter implementation in Morley

Difficulty: Medium
Finding ID: TOB-DEXTER-007

Description
The use of type synonyms and forced coercion is error-prone and can give a false sense of security, leading to issues that are very difficult to detect.

The codebase uses type synonyms, but these do not enforce a strong type consistency in the stack. Haskell type synonyms are completely equivalent to the underlying type and can be easily confused:

```
import GHC.Natural

type A = Natural
type B = Natural

f :: A -> A
f = id

g :: B -> B
g = id

h :: A -> B
h = g . f
```

Figure 7.1: Type synonyms in action.

In the Morley implementation of the Dexter contract, type synonyms are widely used. For instance, the `setAllowance` function uses `Owner` and `Sender`, which are just synonyms for `Address`:

```
type Owner  = Address
type Sender = Address

setAllowance :: Owner : Sender : Natural : Storage : s :-> Storage : s
setAllowance = do
 dup; duupX @3; eq;
```

```
if_
  (do drop; drop; drop)
  (do
      dup; duupX @5; swap; toAccount
      stackType @( Account : Owner : Sender : Natural : Storage  : _)
      getField #approvals; dig @4; some; dig @4; update; setField #approvals
      stackType @( Account : Owner : Storage  : _)
      some; swap; dip (dip (getField #accounts)); update; setField #accounts
  )
```

Figure 7.2: `Core.hs#L203-L217`.

The `Owner` and `Sender` types can be confused in the stack, and the allowance value to set is just an untyped `Natural`. Additionally, there is one use of type coercion in the `tokenToToken` function:

```
tokenToToken :: Entrypoint TokenToTokenParams Storage
tokenToToken = do
 …
 -- send xtzBought to to address
 duupX @3; toField #outputDexterContract; contractCalling @Parameter (Call @"XtzToToken");
ifNone failWith nop
 dig @2
 stackType @[Mutez, ContractRef XtzToTokenParams, Storage, TokenToTokenParams]
 duupX @4; getField #to; dip (do getField #minTokensBought; dip (toField #deadline);
pair); pair; forcedCoerce_
 stackType @[XtzToTokenParams, Mutez, ContractRef XtzToTokenParams, Storage,
TokenToTokenParams]
 transferTokens
 stackType @[Operation, Storage, TokenToTokenParams]
```

Figure 7.3: Tail of the `tokenToToken` *function*.

The `forcedCoerce_` function will force matching of the argument types. Its documentation states that use of this function is unsafe.

Exploit Scenario
Alice writes code to extend the Dexter contract using type synonyms and coercion. As a result, her code is more difficult to audit and formally verify.

Recommendation
Short term:

- Avoid the use of `forcedCoerce_`. Carefully Implement specific coercion instances if needed.
- Use the Haskell type to ensure that values cannot be confused in the stack. Consider using `newtype`, instead of `type`, and implement all the explicit conversions and type instances.

Long term, review the features of the Haskell language to make sure you are taking advantage of them when writing smart contracts for Tezos.

8. Call injection allows price corruption

Severity: High
Type: Data Validation
Target: Dexter

Difficulty: Low
Finding ID: TOB-DEXTER-008

Description
Dexter is vulnerable to call injection vulnerability—anyone can manipulate the price and make a profit from the exchange.

Tezos' call order allows call injection vulnerability (see Appendix D). As a result, anyone can execute arbitrary code between the end of a function's execution and the execution of the external call generated by this function.

`tokenToXtz` swaps the tokens sent for tezos:

```
tokenToXtz :: Entrypoint TokenToXtzParams Storage
tokenToXtz = do
  getField #deadline; now; assertLe [mt|tokenToXtz: the current time must be less
than the deadline.|]

  calculateXtzBought

  -- update tokenPool
  -- storage.s.token_pool      := storage.s.token_pool + tokens_sold;
  dip (do swap; stGetField #tokenPool; duupX @3; toField #tokensSold; add;
stSetField #tokenPool)
  stackType @[XtzBought, Storage, TokenToXtzParams]

  -- send xtzBought
  duupX @3; toField #to; Lorentz.contract @(); ifNone (failWith) nop; dig @1; push
(); transferTokens
  nil; swap; cons
  stackType @[[Operation], Storage, TokenToXtzParams]

  -- transfer tokens
  dip (do dip (do getField #tokensSold; dip (do getField #owner; dip (toField
#to))); transferOwnerTokensTo)
  swap; cons; pair
```

Figure 8.1: `TokenToXtz.hs#L41-L59`.

```
tokenToXtz():
    xtzBought = calculateXtzBought()
    tokenPool += token_sent
    send_xtzBought(xtzBought, to)
```

```
receive_token(token_sent)
```

Figure 8.2: High-level representation of Figure 8.1.

An attacker can inject a call after `tokenToXtz`'s execution but before the tezos are sent or the token is received. During the execution of the injected call:

- The contract's balance will contain the tezos to be sent by `tokenToXtz`
- The attacker can execute `updateTokenPool` to the value it had before the initial call to `tokenToXtz`

Through the injected calls, the attacker can profit from an incorrect exchange rate.

This issue can affect any code with external calls that change the contract's invariant (sending of tezos or tokens). This includes other functions like `tokenToToken`.

Exploit Scenario
Dexter's exchange has a lot of liquidity and a favorable price change that allows its user to exchange $1 worth of tezos for $2 worth of the underlying tokens. Eve exploits the call injection to profit from the price over the entire pool's liquidity. As a result, Eve gains thousands of dollars.

Recommendation
Short term, evaluate all the functions against injection call vulnerability. Consider tracking the contract's balance as an internal variable and removing `updateTokenPool` to mitigate `tokenToXtz` and `tokenToToken`. This solution will cause the arithmetic imprecision to accumulate and require modeling of accumulation growth over time.

Long term, start a discussion with the Tezos development team and stakeholders to evaluate what built-in mitigation against injection calls the platform can offer.

9. Arithmetic rounding allows minting of liquidity tokens without payment of tokens

Severity: Medium
Type: Data Validation
Target: `AddLiquidity.hs`

Difficulty: Medium
Finding ID: TOB-DEXTER-009

Description
An arithmetic imprecision allows minting of liquidity tokens without transferring any underlying tokens.

To mint tokens, the user needs to send half of the value in tezos and half in the underlying token. `AddLiquidity` computes the amount of tokens to be deposited based on the amount of tezos sent:

```
-- tokensDeposited = amount * tokenPool / xtzPool
dup;duupX @5; stToField #tokenPool -- get tokenPool from storage
mul; duupX @3 -- get a copy of xtzPool
swap; ediv; ifNone (failWith) car
stackType @(TokensDeposited : Amount : XtzPool : AddLiquidityParams : Storage : _)

-- lqtMinted = amount * lqtTotal / xtzPool
swap; duupX @5; stToField #lqtTotal; mul
dip swap; ediv; ifNone (failWith) car
stackType @(LqtMinted : TokensDeposited : AddLiquidityParams : Storage : _)
```

Figure 9.1: `AddLiquidity.hs#L89-L98`.

```
tokensDeposited = amount * tokenPool / xtzPool
lqtMinted = amount * lqtTotal / xtzPool
```

Figure 9.2: High-level representation of Figure 9.1.

Due to the arithmetic imprecision, `tokensDeposited` can be zero while `lqtMinted` is positive. This can happen if:

- `mount * tokenPool < xtzPool`
- `amount * lqtTotal > xtzPool`

For example, if pools have a number of tokens significantly lower than the number of tezos (e.g., tezos with a low price and tokens with a high price).

Exploit Scenario
The Dexter exchange has the following parameters:

- `lqtTotal` = 100,000 * 10**6

- `tokenPool` = 10**4
- `xtzPool` = 100,000 * 10**6

Eve deposits 10**6 tezos. As a result, she receives liquidity without paying any underlying tokens.

Recommendation
Short term, prevent `tokensDeposited` from being zero in `AddLiquidity`.

Long term, use fuzzing or a formal method to make sure arithmetic rounding is bounded and cannot be used to take advantage of the exchange.

10. Arithmetic rounding might allow funds to be drained

Severity: Undetermined
Type: Data Validation
Target: Dexter contract

Difficulty: Undetermined
Finding ID: TOB-DEXTER-010

Description
The codebase doesn't handle arithmetic dust, so rounding might allow users to gain more than anticipated.

Arithmetic divisions lead to loss precision. Several operations do not account for the division remainder, which creates arithmetic imprecision.

For example, Figure 10.1 and 10.2 show the operations performed to compute the underlying tokens deposited when adding liquidity and the underlying tokens received when removing liquidity:

```
addLiquidity(amount):
  token_deposit = (amount * token_pool_0) / balance_0
  liquidity_received = (amount * liquidity_total_0) balance_0
  token_pool_1 = token_pool_1 + token_deposit
  liquidity_total_1 = liquidity_total_0 + pool_token
```

Figure 10.1: High-level representation of the underlying tokens deposit.

```
removeLiquidity(liquidityBurned):
  token_receive = (liquidityBurned * token_pool_1) / (liquidity_total_1)
```

Figure 10.2: High-level representation of the underlying tokens withdrawal.

According to the value of `token_pool_0`, `balance_0`, and `liquidity_total_0`, the user may be able to sell only a portion of the liquidity tokens received when `addLiquidity` was called, while receiving all the deposited tokens.

For example, with these conditions...

- `balance_0 = 50010001`,
- `token_pool_0 = 10001`
- `Liquidity_total_0 = 20001`

...a user calling `addLiquidity` with `amount == 10001` will be able to withdraw all the initial tokens deposited while keeping 1 `xtz` of liquidity tokens.

Due to assessment time constraints and the difficulty of testing and running verification tools on Morley and Michelson code, we were not able to determine an upper bound of the rounding.

Exploit Scenario
Eve finds a way to profit by leveraging rounding errors. By adding and removing liquidity, she is able to drain funds from the exchange.

Recommendation
Use fuzzing or formal verification to make sure arithmetic rounding is bounded and cannot be used to take advantage of the exchange.

11. Lack of “amount sent” protection can lead to trapped tezos

Severity: Medium
Type: Data Validation
Target: Dexter contract

Difficulty: Medium
Finding ID: TOB-DEXTER-011

Description
All the functions accept tezos when called. This behavior is error-prone and can make users lose funds.

These functions are:

- `approve`
- `removeLiquidity`
- `tokenToXtz`
- `tokenToToken`
- `updateTokenPool`
- `setBaker`

The default function also has the same behavior. It is not clear if this is intended or not.

Exploit Scenario
Bob calls `approve`, but incorrectly sends 10 tezos with the transaction. As a result, Bob loses the funds.

Recommendation
Short term, revert if the transaction’s amount is non-zero for all functions that do not use the amount.

Long term, carefully review the known attack vectors of the existing blockchain and ensure Dexter and Morley have the proper mitigations.

12. User-provided inputs are not properly validated in the frontend

Severity: High
Type: Data validation
Target: Input parsing functions in Dexter frontend

Difficulty: Medium
Finding ID: TOB-DEXTER-012

Description
The Dexter frontend allows users to sign blockchain transactions for the contract. Two important inputs of these transactions are the fund amount and the destination address. The Dexter frontend does not properly validate the amount of funds to be sent or the destination address for sending the funds.

The amount value is converted from `string` using the following function:

```
let ofTezString = (string: string): Belt.Result.t(t, string) =>
 switch (Tezos_Util.floatOfString(Common.removeCommas(string))) {
 | Some(float) =>
   let pos = Tezos_Util.getPositionOfLastDecimalDigit(string);
   if (pos <= Some(6)) {
     let mutezFloat = float *. 1000000.;


     let int64 = Int64.of_float(mutezFloat);
     switch (ofInt64(int64)) {
     | Belt.Result.Ok(mutez) => Belt.Result.Ok(mutez)
     | Belt.Result.Error(_error) =>
       Belt.Result.Error("ofString: expected a non-negative int64 value.")
     };
   } else {
     Belt.Result.Error(
       "ofTezString: expected a non-negative float with precision up to 10^-6, received: "
       ++ string,
     );
   };
 | None =>
   Belt.Result.Error(
     "ofTezString: expected a non-negative float with precision up to 10^-6",
   )
 };
```

Figure 12.1: Function to convert from `string` *to a* `tezos amount` *in* `Tezos_Mutez.re#L68-L91`.

This code invokes a function to remove commas from the input string:

```
let removeCommas: string => string = [%bs.raw
 {|
   function (s) {
     return s.replace(/,/g, "");
   }
  |}
];
```

Figure 12.2: `RemoveComma` *definition in* `dexter-frontend/src/Common/Common.re#L43-L49`.

However, in several countries, the official decimal separator is the comma, not the dot, so some users will perform transfers with incorrect values.

The validation of the tezos' destination address is also incomplete. While the frontend has some code to verify checksums of it...

```
  function (input) {
    const bs58check = require('bs58check');
    const elliptic  = require('elliptic');
    var prefix;
    var tz_prefix;
    if (input.startsWith("tz1")) { // ed25519_public_key_hash
      prefix = new Uint8Array([6, 161, 159]);
      tz_prefix = '00';
    } else if (input.startsWith("tz2")) { // secp256k1_public_key_hash
      prefix = new Uint8Array([6, 161, 161]);
      tz_prefix = '01';
    } else if (input.startsWith("tz3")) { // p256_public_key_hash
      prefix = new Uint8Array([6, 161, 164]);
      tz_prefix = '02';
    };
    const bytes = '00' + tz_prefix +
elliptic.utils.toHex(bs58check.decode(input).slice(prefix.length));
    const len = bytes.length / 2;
    const result = [];
    result.push('050a');
    result.push(len.toString(16).padStart(8, '0'));
    result.push(bytes);
    return result.join('');
  }
```

Figure 12.3: `Address validation` function in `dexter-frontend/src/Tezos/Tezos_Address.re#L37-L60`.

...it is never used in the actual validation:

```
let ofString = (candidate: string): Belt.Result.t(t, string) =>
 if ((
       Js.String2.startsWith(candidate, "tz1")
       || Js.String2.startsWith(candidate, "tz2")
       || Js.String2.startsWith(candidate, "tz3")
     )
     && String.length(candidate) == 36) {
   Belt.Result.Ok(Address(candidate));
 } else {
   Belt.Result.Error(
     "Address.mk: unexpected candidate string: " ++ candidate,
   );
 };
```

Figure 12.4: Function to convert from `string` to `tezos address` in `Tezos_Address.re#L16-L28`.

Exploit Scenario
Alice wants to use the Dexter frontend to swap a small amount of her coins. She inputs the amount as `100,25`. The frontend interprets her amount as `10025`. As a result, Alice swaps significantly more assets than expected and loses funds.

Recommendation
Short term,

- Enable checksum validation when parsing tezos addresses.
- Split the price in two inputs, one for the integer part, one for the fractional part
- Ask the user to validate the parameters before sending the transaction

Long term, design input validation of user-provided values to be bullet-proof: fail early if values are invalid and ask users to re-confirm them, especially if there is ambiguity (e.g., using dots or commas for decimal separators). This will minimize the risk of users misusing the on-chain or off-chain components.

13. Users can be tricked into adding liquidity for a baker that immediately changes

Severity: Low
Type: Auditing and Logging
Target: `DexterUi_AddLiquidity.re`

Difficulty: High
Finding ID: TOB-DEXTER-013

Description
Users cannot see whether the baker can be changed or not in the Dexter frontend, so they can be tricked into contributing to one baker when they intended to support a different one.

Users are motivated to add liquidity because they will gain rewards via exchanges, and also through the baker. The baker may help incentivize a particular Dexter contract, so the user should have easy access to baker's identity.

```
let baker =
   style([
     marginBottom(`px(10)),
     width(`px(400)),
     color(DexterUi_Style.Color.grey),
     DexterUi_Style.Font.serif,
     DexterUi_Style.Font.semibold,
     fontSize(`px(18)),
   ]);


 let bakerHeader = style([marginBottom(`px(10))]);


 let bakerDetails =
   style([
     borderWidth(`px(1)),
     borderStyle(`solid),
     borderColor(DexterUi_Style.Color.lightGrey),
     borderRadius(`px(10)),
     paddingLeft(`px(24)),
     paddingTop(`px(16)),
     paddingBottom(`px(16)),
     color(DexterUi_Style.Color.black),
     DexterUi_Style.Font.semibold,
```

```
      fontSize(`px(14)),
    ]);
```

Figure 13.1: How baker details are shown in the frontend.

However, the baker can be changed at any time if the flag `freezeBaker` is not set.

```
setBaker :: Entrypoint SetBakerParams Storage
setBaker = do
 -- assert that the sender is the manager
 duupX @2; stToField #manager; sender; assertEq (mkMTextUnsafe "Only the manager can set
the baker.")
 stackType @[SetBakerParams, Storage]


 -- assert that changing the baker is not frozen
 duupX @2; stToField #freezeBaker; not; assert (mkMTextUnsafe "Cannot change the baker
while freezeBaker is set to True.")
 stackType @[SetBakerParams, Storage]


 -- set the delegate
 unpair; setDelegate
 stackType @[Operation, Bool, Storage]

 -- set the freezeBaker value
 dip (do stSetField #freezeBaker; nil)
 stackType @[Operation, [Operation], Storage]
 cons; pair
```

Figure 13.2: The `setBaker` function.

```
setBaker(newBaker, newFreezeBaker)
  ...

  if freezeBaker
    return

  setDelegate(newBaker)
  freezeBaker = newFreezeBaker
```

Figure 13.3: High-level representation of Figure 13.2

Exploit Scenario
Alice wants to provide liquidity to an exchange. Eve creates an exchange, knowing that Alice will be tempted to provide liquidity if a certain baker is used. Alice sees the baker in Eve's exchange and adds her funds to it. After the blockchain confirms the transaction, Eve can change the baker to someone else. As a result, Alice is tricked into contributing to the wrong baker. To mitigate this risk, she should actively monitor who the baker is and retrieve her funds.

Recommendation
Short term, add a visible warning to show the user that the current baker can be changed at any time.

Long term, make sure users have all the relevant information in the UI before confirming a blockchain transaction.

14. Deadline for transactions are fixed at two hours from now

Severity: Informational
Type: Timing
Target: Dexter frontend

Difficulty: Low
Finding ID: TOB-DEXTER-014

Description
Deadlines for Dexter transactions are fixed at two hours from `now`, so users can't set lower or larger bounds on a transaction's validity.

Several of the Dexter functions include a timestamp parameter. This value is used as a deadline to process transactions to avoid having transactions confirmed unexpectedly late.

```
type TokenToXtzParams =
 ( "owner"        :! Address   -- the address which is selling their token to Dexter.
 , "to"          :! Address   -- the address which will receive the purchased XTZ.
 , "tokensSold"   :! Natural   -- the amount of Tokens sold.
 , "minXtzBought" :! Mutez     -- minimum amount of XTZ the sender wants to purchase, if not
met the transaction will fail.
 , "deadline"     :! Timestamp -- the time after which this transaction can no longer be
executed.
 )
```

Figure 14.1: Parameters needed to call `TokenToXtzParams` *in* `Core.hs#L70-L76`.

However, all calls to the Dexter contract from the frontend hardcode the timestamp parameter. For instance:

```
let hourFromNow = Tezos.Timestamp.hourFromNow();
...

Dexter_Query.tokenToXtz(
 settings.node.url,
 dexterContract,
 inputToken.tokenContract,
 inputToken.tokenBigMapId,
 owner,
 destination,
 tokensSold,
 minXtzBought,
 hourFromNow,
```

```
 account.wallet,
)
```

Figure 14.2: `TokenToXtz` call in `DexterUi_Exchange.re#L1380-L1402`.

Despite the variable's name, the value `hourFromNow` is equal to *two* hours from now:

```
let hourFromNow = () => {
 let now = MomentRe.momentNow();
 mk(
   MomentRe.Moment.add(~duration=MomentRe.duration(2., `hours), now),
 );
};
```

Figure 14.3: The `hourFromNow` *definition in* `Tezos_Timestamp.re#L32-L37`.

Exploit Scenario
Alice notices a particularly good price for certain tokens and wants to take advantage of it. She uses Dexter's fronted to quickly submit a transaction to trade them, but the network is highly congested. As a result, her transaction is confirmed too late and she does not make a profit.

Recommendation
Short term, allow users to set the `deadline` parameter.

Long term, do not hardcode parameters in off-chain components unless it is absolutely necessary to prevent security or correctness issues.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Consider using `SafeHaskell`.** Certain compiler configurations or extensions could be unsafe when compiling critical code. `SafeHaskell` is a subset of Haskell that only enables safe extensions and libraries to provide strict type safety.
- **Use a Haskell linter to improve code readability.** Using a Haskell linter such as hlint will help improve code readability. It currently detects small issues related to:
  - Unused LANGUAGE pragmas
  - Redundant brackets
  - Redundant do
  - Code duplication

- **Consider using `getAccout` in `approve` to improve code readability.** The `approve` code re-implements a code similar to the one used in `getAccount` instead of reusing it.
- **Avoid using unsafe functions in production code.** `UnsafeMkMutez` and `mkMTextUnsafe` are used in this codebase. Use their safe versions instead (`mkMutez` and `mkMText`).
- **Update the comment at `AddLiquidity.hs#L134`.** The comment states that `amount > 0` when it should be `amount >= 10**6`.

D. Call Injection vulnerability

Tezos's message passing design causes a type of vulnerability that is likely to be present in many Tezos contracts.

Description
In Tezos, a call to an external contract in a function is not done during the function's execution, but is queued in a list of calls to be executed. The ordering of the calls follows breadth-first search (BFS).

Consider Figure D1.1. In a traditional smart contract, the functions would be executed in the following order:

- start `a()`
  - start `b()`
    - start `c()`
    - end `c()`
  - end `b()`
  - start `d()`
  - end `d()`
- end `a()`

In a Tezos contract, the order would be:

- Execute `a()` # Next calls: [`b`, `d`]
- Execute `b()` # Next calls: [`d`, `c`]
- Execute `d()` # Next calls: [`c`]
- Execute `c()` # Next calls: []

In Tezos, the code of `d()` is executed before the code of `c()`.

```
function a():
     call b()
     call d()
     return

function b():
     call c()
     return

function c():
     return
```

```
function d():
    return
```

Figure D.1: Pseudo-code of function calls.

This unusual order of execution allows an attacker to execute arbitrary code between the end of a function's execution and the calls generated by the function. In particular, the contract's balance or the memory of the called contracts, might be in an invalid state due to the execution of the attacker's call. We call this attack **call injection.**

Exploit Scenario

Consider the `Receiver` contract in Figure D2.2 and its high-level representation in Figure D2.3:

```
type parameter is
    Receive of unit
  | Set of unit

function entry_Receive (const current_balance : tez) : (list(operation) * tez) is
        block {
            const alice : address = ("some_address": address);
            const to_contract: contract(unit) = get_contract(alice);
            const op : operation = transaction(unit, amount, to_contract);
        } with ((list [op]), current_balance)

function entry_Set (const current_balance : tez) : (list(operation) * tez) is
        block {
            if(amount > 0mutez) then {
                failwith("Do not send tezos")
            } else{
                current_balance := balance
            }
        } with ((nil : list(operation)), current_balance)

function main (const action : parameter; const current_balance : tez) :
(list(operation) * tez) is
        case action of
            Receive(x) -> entry_Receive(current_balance)
```

```
            | Set(x) -> entry_Set(current_balance)
        end
```

Figure D.2: `Receiver` contract.

```
storage:
    current_balance

function receive():
    transaction("some_addess", amount)

function set():
    require(amount == 0)
    current_balance = balance
```

Figure D.3: High-level representation of Figure D.2.

`Receiver` has two functions:

- `receive`, which forwards the funds sent to a fixed address
- `set`, which stores the contract's balance to `current_balance`

If we do not consider call injection, an invariant of `Receiver` would be that `current_balance` is always zero as:

- All the tezos sent to `receive` are sent to a fixed address
- `Set` cannot receive tezos

But this invariant can be broken if an attacker injects a call to `set` after the end of `receive`'s execution and before its external call is executed (`transaction("some_addess", amount)`). Figure D.4 shows an example of such an attack:

```
function main (const param : unit; const current_balance : tez) : (list(operation)
* tez) is
    block {
      const dst : address = ("receiver_address": address);
      const receive: contract(unit) = get_entrypoint("%receive", dst);
      const op1: operation = transaction(unit, amount, receive);

      const set_balance: contract(unit) = get_entrypoint("%set", dst);
      const op2: operation = transaction(unit, 0mutez, set_balance);

      op_list := list op1; op2; end;
    }
```

```
    with (op_list, current_balance)
```

Figure D.4: Exploit of Figure D2.2.

```
function main():
      call receive()
      call set()
```

Figure D.5: High-level representation of the exploit in Figure D.4.

In the exploit, `set()` is executed before the tezos are transferred from the `Receiver` contract to the static address. As a result, `current_balance` will be a non-zero value.

Recommendation
There is no straightforward fix to prevent this type of attack.

One of the reasons Tezos has this unusual call order is to prevent reentrancy. However, the current solution brings a new type of vulnerability that is likely to be more difficult to prevent than reentrancy. Additionally, the call order is unusual and difficult to reason with, which is likely to confuse developers. Two critical issues found in Dexter were the results of this design (TOB-DEXTER-001, TOB-DEXTER-008).

Reentrancies can be prevented while keeping a traditional call order. For example, the VM can have a contract-level reentrancy lock that can be enabled by default. Function-level locks can also be used for more granularity, such as those in Vyper.

We recommend the Tezos developers consider using direct calls instead of message passing and implement reentrancy mitigation in the VM.

E. Fix Log

camlCase addressed issues 1 to 11 in their codebase as a result of the assessment. Each of the fixes present in Gitlab was verified by Trail of Bits on August 11.

ID	Title	Severity	Status
1	`updateTokenPool` can be abused to drain Dexter's assets	High	Mitigated
2	`tokenToXtz` sends the tokens to the user	High	Fixed
3	The allowance is incorrectly updated after removing liquidity	High	Fixed
4	Morley contracts are not properly tested	Medium	Fixed
5	`dexter.Approve` is not compatible with FA1.2.`Approve`	Low	Fixed
6	Discrepancy between the informal specification and the Morley contract on `approve`	Low	Fixed
7	Improper use of Haskell type system to enforce type correctness in the stack	Informational	Fixed
8	Call injection allows price corruption	High	Mitigated
9	Arithmetic rounding allows minting of liquidity tokens without payment of tokens	Medium	Fixed
10	Arithmetic rounding might allow funds to be drained	Undetermined	Not fixed
11	Lack of "amount sent" protection can lead to trapped tezos	Medium	Fixed
12	User-provided inputs are not properly validated in the frontend	High	Not fixed
13	Users can be tricked into adding liquidity for a baker that immediately changes	Low	Not fixed
14	Deadline for transactions are fixed at two hours from `now`	Informational	Not fixed

Detailed Fix Log

TOB-DEXTER-001: `updateTokenPool` can be abused to drain Dexter's assets
Mitigated (159, 195).

Original review - August 11
The original fix (159) introduced a call injection vulnerability. camlCase provided a fix for it in 195.

The PRs:

- Split the update of the pool in two functions `updateTokenPool` and `updateTokenPoolInternal`, where `updateTokenPool` must be called before `updateTokenPoolInternal`
- Creates a lock on the Dexter contract to prevent any call to the contract to be executed between `updateTokenPool` and `updateTokenPoolInternal`.

The current fix will prevent an attacker exploiting an injection call on the balance's update for simple tokens. However tokens can be implemented using multiple contracts, which can allow the exploitation of call injections. In particular, if the update of the balance is done through a call, an attacker will be able to inject a call between `updateTokenPool` and the token's update, leading to an incorrect update to dexter's balance.

Consider the following token specification, which splits the right to update the balance into a separate contract:

```
contract Token
    balances: address => uint

    function transfer(to: address, amount: uint):
        call updater.decrease_balance(sender, amount)
        call updater.increase_balance(to, amount)

contract Updater
    function increase_balance(addr: address, amount: uint):
         # Has the access controls to update Token's balance

    function decrease_balance(addr: address, amount: uint):
         # Has the access controls to update Token's balance
```

The following shows an example of exploit:

- The attacker calls `xtzToToken` and `updateTokenPool`. The calls to be executed are `[dexter.xtzToToken, dexter.updateTokenPool]`

- `xtzToToken` increases the token's pool, and calls `token.transfer`. The calls to be executed are `[dexter.updateTokenPool, token.transfer]`
- `dexter.updateTokenPool` calls `token.getBalance`. The calls to be executed are `[token.transfer, token.getBalance]`
- `token.transfer` calls `token_storage.updateBalance`. The calls to be executed are `[token.getBalance, token_storage.updateBalance]`
- `token.getBalance` is executed and gets the token's balance that does not take into account the upcoming balance update.

As a result, the update of the token's balance in Dexter will be incorrect.

If no generic fix is found, we recommend to remove `updateTokenPool` and `updateTokenPoolInternal`. If these functions are not removed, we recommend to:
- Disallow the usage of arbitrary tokens
- Write a checklist of items to verify when adding new tokens, including
  - The token must update the balance directly on call to `transfer`
  - The token must correctly implement `getBalance`'s callback
- Consider every item from our Ethereum token integration checklist
- Manually review every token to be added

Additionally, we recommend preventing `updateTokenPool` from being called if `selfIsUpdatingTokenPool` is true.

Update - September 8
camlCase mitigated the new issue by:
- Disabling arbitrary tokens from Dexter's UI
- Writing a token integration checklist

TOB-DEXTER-002: `tokenToXtz` sends the tokens to the user
Fixed (PR 160).

TOB-DEXTER-003: The allowance is incorrectly updated after removing liquidity
Fixed (PR 162)

TOB-DEXTER-004: Morley contracts are not properly tested
Fixed (PRs 183, 184, 186, 187, 188, 189, 190, 191, 192)

TOB-DEXTER-005: `dexter.Approve` is not compatible with FA1.2.`Approve`
Fixed (PR 57, Approve.hs#L40-44).

Original review - August 11
While 57 states that `Approve` will be removed, neither `Dexter.Approve`, or its informal specification were updated following our recommendations.

To remove any confusion from third parties, consider either:

- Removing `currentAllowance` and follow the non-zero to non-zero specification of FA1.2.approve (preventing the known front-running issue), or
- Clearly documenting that Dexter's `approve` is not compatible with FA1.2.

Update - September 8
camlCase improved the documentation of the `approve` function (Approve.hs#L40-44) to remove the ambiguities with the FA1.2 function.

TOB-DEXTER-006: `dexter.Approve` is not compatible with FA1.2.`Approve`
Fixed (3e02a9)

TOB-DEXTER-007: Improper use of Haskell type system to enforce type correctness in the stack
Fixed (PRs 177, 179)

TOB-DEXTER-008: Call injection allows price corruption
Mitigated (PR 178).

Dexter tracks its balance in an internal variable `xtzPool`. While this prevents the exploit highlighted in TOB-DEXTER-008, other call injections opportunities might be present and the PR does not fix the issue at its core. For example, the fix for TOB-DEXTER-001 (159, 195) introduced a new call injection vulnerability.

Additionally, the impact of the arithmetic imprecision accumulated in the balance's variable is unclear.

Consider:

- Using formal verification to evaluate the impact of the arithmetic imprecision on `xtzPool`
- Evaluating the ability for an attacker to perform calls injection exploit, in particular based on the tokens's balances update.

`TOB-DEXTER-009`: Arithmetic rounding allows minting of liquidity tokens without payment of tokens
Fixed (83)

`TOB-DEXTER-010`: Arithmetic rounding might allow funds to be drained
Not fixed.

`TOB-DEXTER-011`: Lack of "amount sent" protection can lead to trapped tezos
Fixed (Core.hs#L405-411) .

TOB-DEXTER-012: User-provided inputs are not properly validated in the frontend
Not Fixed

TOB-DEXTER-013: Users can be tricked into adding liquidity for a baker that immediately changes
Not Fixed

TOB-DEXTER-014: Deadline for transactions are fixed at two hours from `now`
Not Fixed