vSlice Contracts Test Report

The vSlice contract architecture is backed by a set of comprehensive unit tests made to work in conjunction with the testRPC environment, which examine all the different contracts and their various functions to provide assurance that the contracts function in unison and as expected. It encompasses 41 tests currently that the contracts must conform to. The tests range from simple modifier tests, to time-based testing which asserts the different periods work correctly. An overview of the cases and scenarios the tests cover will be provided in the following list.

- **Initialization Tests**
  - **Wallet Variable checks:** Ensuring owners are set as they were declared in deployment. That public variables within the contract have their expected values.
  - **Wallet Modifier checks:** Ensuring that the contract throws on invalid calls such as sending values on functions without payable or non-multisig owners attempting to call functions for which they do not have appropriate rights. Also making sure that those functions which are supposed to receive funds do so.
  - **Wallet Withdrawal:** Checks that owners can withdraw funds from the contract.
- **Token Tests**
  - **buyTokens and fallback test:** Verifies that both the fallback test can be used to purchase tokens, and buyTokens function; for which one can designate a beneficiary, and this specific test does, and checks their balances to ensure they received the expected token amounts they should.
  - **Token Limit tests:** Creates a scenario whereby the ICO ends within the first week of creation, ensuring the sale halts appropriately, that the variables across the Token contract and Wallet are in their correct post-sale state, and that Token are successfully transferable thereafter.
- **Unit Test Method**
  - **Internal Test Method:** This is a method that is used to initialize some variables to be used within the unit tests, for the time-based testing with testRPC.

raised by the token swap, we considered this a non-issue since owners have no economic incentives to act in this way. Moreover, accessing the funds could reduce the security risks of having a large amount of ether in only one contract, since owners may want to spread it between multiple accounts/wallet contracts.

Extra Notes during Testing/Development

Profit Container Calculation: During development, a couple of formulas were engineered for calculating the amount to be remitted by the withdrawalFunction. The 2 that were deliberated upon include the one that the contract currently uses, which is a simple fractional calculation of a user's proportion of the profits. This solution comes with the caveat that depending on the balance in the contract at the time of locking, and the user's token balance, there may be some wei leaking in certain combinations. From our testing, it has always been within single digits, and to be precise. This occurs due to how division is handled, and the fact that any remainders are truncated.
The other solution we had seemed leak-proof, as it did not leak any wei in some of the cases the previous version did. However, the formula was more complex and required 2 additional storage variables, to be updated on every call of the function, and this also entailed the use of underflow checks. In the end, to save the end-user from some wei leaking, was magnitudes more expensive due to the much higher gas required. We concluded, that it would only save the end-user funds, in the case of gas approaching a 0 wei cost, which is quite unlikely. In addition, any leaked wei would be carried over to the next profit distribution period, so it's not lost.
However, another added benefit of the Profit Container contract is that it is easily replaceable, so if there ever comes a time where 0 wei gas becomes a reality, the extra complexity required would be worthwhile and could be implemented. The same goes if solidity becomes more flexible with division as well.

Last Investor:
Due to how Ethereum currently works, there is always an external actor that has to pay the transaction fees (in gas) to perform executions and update the state of a contract. This means that to close the token swap period, there needs to be a transaction which will pay the gas fees to lock the wallet down. The investor making this last transaction won't receive any tokens, since we are after the end of the token swap period, but also he won't receive any refunds.
After discussing internally, we came to the conclusion that the best way, if the client deems it necessary, is to operate a manual refund toward that last investor because adding a refund logic would increase the wallet contract attack surface for a really small gain in term of functionality.
This case only applies if the token swap reaches its natural end period (4 weeks). If the max token swap cap is reached before that, than the last investors will correctly receive his fair share of tokens.

Wallet Lockdown
Another issue which we raised internally, it is if we considered appropriate for the wallet owners to be able to have complete access to the funds during the token swap period. If owners are able to withdraw the funds, they could use them to mint token for themselves for free. But due to the fact that the blockchain is transparent, so any strange behavior would be surely be noted and reputational damage would ensue, and that this would lower the overall amount of ether

vSlice Gas Profile: Testing done on Morden post hard fork.

Contract	*Wallet.sol*	*Token.sol*	*ProfitContainer.sol*
Deployment	1,717,094	770,465	590,779
*1st Fallback**	76,787		21,037
*Fallback initial***	61,787		21,037
*Fallback again****	47,259		
buyTokens initial	61,787		
buyTokens again	49,507		
setTokenContract	44,184		
startTokenSwap	27,122		
stopTokenSwap	27,060		
transfer initial		53,053	
transfer again		38,053	
transferFrom initial		47,130	
transferFrom again		30,112	
approve initial		46,701	
approve again		31,701	
1st withdrawalProfit			107,094
withdrawalProfit			63,643
changeTokenContract			29,526
transferOwnership			28,521

Notes: The Wallet contract was deployed using 2 declared owners plus the deploying address. Therefore its cost is based on a total of 3 owners.

*: The 1st indicates the first time in the contract the function is called by anyone. Gas costs tend to be higher in these cases, since some variable is being initialized for the first time. Once called, its cost drops thereafter for everyone.

**: Initial defines a function for which a certain variable is provided the first-time. In the above cases, this certain variable will usually be an address. Such as for the tokens, if a non-token holder is provided in the address field, the cost will be the initial function, as the address needs to be added to the contract's state for the first time, and therefore costs more.

***: Again indicates the function is being called upon a variable already in the contract's state, such as an existing token holder. Therefore gas costs tend to be cheaper, as no extra initialization needs to take place.

```
Contract: Wallet & Token
  Week 4 - Token Test
    √ Check Modifiers (77ms)
    √ Ensure vars are correct for week 4 (129ms)
  Week 4 - Payable Tests
    √ Check that payable functions accept value (765ms)
  Week 4 - Withdrawal Tests
    √ Check that owners can withdraw funds (1294ms)

Contract: Wallet & Token
  10 mins before end of Week 4 - before ICO Ending
    10 minutes before Week 4 End : Thu, 17 Nov 2016 18:16:44 GMT
    √ Ensure correct total of tokens are minted (2614ms)
    √ Ensure vars are correct near sale end (79ms)
  Start Week 5 - ICO End
    Week 5 Date : Thu, 17 Nov 2016 18:26:44 GMT
    √ Ensure no more tokens are minted (1087ms)
    Epoch 2 date : Sat, 19 Nov 2016 18:26:44 GMT
    √ Start Epoch 2 and ensure vars are correct and ready for token transfers (650ms)
    √ Ensure token transfers work after ICO (224ms)
    √ Ensure vars are correct after sale end (71ms)

Contract: ProfitContainer + Token
  Test profit deposit and withdrawal with tokens
    Epoch 2 date : Sat, 19 Nov 2016 18:26:44 GMT
    Epoch 2 lock date : Wed, 14 Dec 2016 18:26:44 GMT
    √ Ensure proper amounts are withdrawn from profitContainer for single tokenholder (1950ms)
    Epoch 3 date : Mon, 19 Dec 2016 18:26:44 GMT
    Epoch 3 lock date : Fri, 13 Jan 2017 18:26:44 GMT
    √ Ensure proper multiple tokenholder withdrawal, proper division among them, and epoch reset (4165ms)

Contract: Token
  Long-term Epoch & Lock tests
    Epoch 4 date : Wed, 18 Jan 2017 18:26:44 GMT
    Epoch 3 Locked phase : Sun, 12 Feb 2017 18:28:46 GMT
    Epoch 90 date : Sun, 11 Feb 2024 18:26:44 GMT
    Epoch 90 Locked phase : Thu, 07 Mar 2024 18:28:45 GMT
    √ Ensure epochs progress and lock/unlock as expected (3234ms)


41 passing (38s)
```

Raw Unit Test Output: All 41 Tests are passing with the latest contract iteration.

```
$ truffle test
Compiling Migrations.sol...
Compiling ProfitContainer.sol...
Compiling Token.sol...
Compiling Wallet.sol...


  Contract: Wallet
    Variable checks
      √ Ensure correct owners are set (708ms)
      √ Ensure other public variables have expected values on contract initilization (554ms)
    Modifier Tests
      √ Check that externally accessible functions without payable throw on value (216ms)
      √ Check that only authenticated owners can change state with onlyowners modifier (164ms)

  Contract: Wallet
    Payable Tests
      √ Check that payable functions accept value (783ms)
    Withdrawal Tests
      √ Check that owners can withdraw funds (1315ms)

  Contract: Wallet & Token
    Token Tests
      √ Ensure beneficiary & self token balances are correct using buyTokens (289ms)

  Contract: Wallet & Token
    Token Limit Tests
      √ Ensure that only the specified max can be minted, and funding stops when hit (576ms)
      √ Ensure vars are correct after maxing out (64ms)
      √ Ensure token transfers work after maxing out at start (80ms)

  Contract: Internal Test Method
    Calculate contract deployment date, and expected end date
      Contracts deployed on: Thu, 20 Oct 2016 18:26:44 GMT
      ICO token sale end: Thu, 17 Nov 2016 18:26:44 GMT
      √ Log values (177ms)

  Contract: Token
    Start Week 1 - Token Test
      √ Check Modifiers (82ms)
      √ Ensure vars are properly set at start (71ms)

  Contract: Wallet & Token
    Week 1 - Mint Test
      √ Ensure correct total of tokens are minted (2089ms)
    Start Week 2 - Mint Test
      Week 2 Date : Thu, 27 Oct 2016 18:26:44 GMT
      √ Week 2 - Ensure correct total of tokens are minted (2560ms)

  Contract: Wallet & Token
    Week 2 - Token Test
      √ Check Modifiers (58ms)
      √ Ensure vars are correct for week 2 (78ms)
    Week 2 - Payable Tests
      √ Check that payable functions accept value (769ms)
    Week 2 - Withdrawal Tests
      √ Check that owners can withdraw funds (1314ms)

  Contract: Wallet & Token
    Week 2 - Token Limit Tests
      √ Ensure that only the specified max can be minted, and funding stops when hit (452ms)
      √ Ensure vars are correct after maxing out (64ms)
      √ Ensure token transfers work after maxing out at Week 2 (75ms)

  Contract: Wallet & Token
    Start Week 3 - Mint Test
      Week 3 Date : Thu, 03 Nov 2016 18:26:44 GMT
      √ Ensure correct total of tokens are minted (2604ms)

  Contract: Wallet & Token
    Week 3 - Token Test
      √ Check Modifiers (71ms)
      √ Ensure vars are correct for week 3 (98ms)
    Week 3 - Payable Tests
      √ Check that payable functions accept value (769ms)
    Week 3 - Withdrawal Tests
      √ Check that owners can withdraw funds (1295ms)

  Contract: Wallet & Token
    Start Week 4 - Mint Test
      Week 4 Date : Thu, 10 Nov 2016 18:26:44 GMT
      √ Ensure correct total of tokens are minted (2605ms)
```