our shielding · Your smart contracts, our shielding · Your smart

shieldify

CSX

SECURITY REVIEW

Date: 5 February 2024

CONTENTS

1. About Shieldify

We are Shieldify Security – Revolutionizing Web3 security. Elevating standards with top-tier reports and a unique subscription-based auditing model.

Learn more about us at shieldify.org or @ShieldifySec.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About CSX Protocol

CSX is a decentralized application (dApp) that utilizes smart contracts. The peer-to-peer trading system on CSX allows users to buy and sell Counter-Strike skins on-chain seamlessly for Stablecoins and Ether without the need to deposit or sign in.

The protocol is community-governed, meaning that users have a say in the direction and development of its features via voting with their CSX tokens.

Learn more about CSX's concept and the technicalities behind it here.

4. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security assessment spanned three weeks, during which four security researchers from the core Shieldify team collectively dedicated 552 hours. The code exhibits professionalism and incorporates fundamental best practices The test coverage is extensive.

Shieldify expresses their gratitude for CSX team's prompt and efficient communication, which greatly enhanced the quality of the audit report.

5.1 Protocol Summary

Project Name	**CSX Protocol**
Repository	csx-contracts
Type of Project	DEX for CS skins, P2P trading
Audit Timeline	23 days
Review Commit Hash	bc51a67ccff86f2c691375f3cc92ee8c0a9fc369
Fixes Review Commit Hash	3d1ddc42ffd920c2fef968e122d641eb31590aa9

5.2 Scope

The following smart contracts were in the scope of the security review:

contracts/Trade/CSXTrade.sol	450
src/Enigma.sol	312
contracts/Trade/BuyAssistoor.sol	26
contracts/Referrals/IReferralRegistry.sol	14
contracts/Referrals/NetValueCalculator.sol	21
contracts/Referrals/ReferralRegistry.sol	122
contracts/Keepers/Keepers.sol	91
contracts/Keepers/IKeepers.sol	3
contracts/CSX/EscrowedCSX.sol	32

contracts/CSX/VestedCSX.sol	71
contracts/CSX/StakedCSX.sol	203
contracts/CSX/VestedStaking.sol	147
contracts/CSX/CSXToken.sol	8
contracts/CSX/Interfaces.sol	16
contracts/Users/UserProfileLevel.sol	75
contracts/Users/IUsers.sol	20
contracts/TradeFactory/TradeFactoryBase.sol	61
contracts/TradeFactory/CSXTradeFactory.sol	236
contracts/TradeFactory/ITradeFactory.sol	49
contracts/TradeFactory/storage/TradeFactoryBaseStorage.sol	66
contracts/TradeFactory/storage/ITradeFactoryBaseStorage.sol	5
Total	**1928**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **3**
- **Medium** issues: **8**
- **Low** issues: **6**

ID	Title	Severity
[C-01]	Malicious Users Can Steal All Staking Rewards From `StakedCSX` Due to a Logical Error	Critical
[H-01]	Unclaimed User Staking Rewards Are Going to Get Lost on Each Call to `stake()` Function in `StakedCSX`	High
[H-02]	The `USDT` Reward Claiming Functionality of `VestedStaking` Can Be DoSed Due to a Logical Error	High
[M-01]	A Buyer Can Use Two Different Referral Codes for the Same Trade	Medium
[M-02]	Whenever There Is a Buyer Discount and a Trade Is Completed Successfully, an Amount of Tokens that Is Equal to the Discount Fee Will Get Stuck in the `CSXTrade` Contract	Medium

ID	Title	Severity
[M-03]	Wrong Address Check in `receive()` Function in `VestedStaking.sol` Contract Can Lead to Integrating Contracts Not Being Able to Claim Their Staking Rewards	Medium
[M-04]	Some Trade-Terminating Functions Do Not Call `removeAssetIdUsed()` Internally, Potentially Leading The Same Skin Not Being Tradable On CSX Again	Medium
[M-05]	Users Can Use the Referral System to Refer Themselves in Order to Buy Skins at a Discount	Medium
[M-06]	Malicious Buyers Can Get Skins Without Paying For Them If The Steam Trade Is Accepted By the Seller After More Than 24 Hours Have Passed Since the Buy Commit Placement	Medium
[M-07]	`Vested CSX` to `Regular CSX` Conversion Process Enables Potential Unauthorized Withdrawal of Staked Deposits by Malicious Council	Medium
[M-08]	Tokens with a `Fee-On-Transfer` Mechanism Could Break the Protocol	Medium
[L-01]	DoS Attack on `getTradeIndexesByStatus()` Function By Creating Listings with Large Sticker Arrays	Low
[L-02]	If a Referral Code Owner Gets Blacklisted From `USDC/USDT`, Seller Trade Rewards Will Get Locked	Low
[L-03]	Centralization Risk Due To All CSX Tokens Being Minted To The Contract Creator	Low
[L-04]	Incomplete Validation In Multiple State Changing Functions	Low
[L-05]	The `safeApprove()` Function Could Revert For Non-Standard Token Like `USDT`	Low
[L-06]	The `stake()` Function does Not Follow The `CEI` Pattern	Low

7. Findings

[C-01] Malicious Users Can Steal All Staking Rewards From `StakedCSX` Due to a Logical Error

Severity

Critical Risk

Description

The current implementation of `_claimToCredit()` function in `StakedCSX.sol` contract has a logical error in it:

```
(,,uint256 rewardWETH) = rewardOf(_to);
if (rewardWETH > 0) {
  credit[address(TOKEN_WETH)][_to] += rewardWETH;
}

(uint256 rewardUSDC,,) = rewardOf(_to);
if(rewardUSDC > 0) {
  credit[address(TOKEN_USDC)][_to] += rewardUSDC;
}

(,uint256 rewardUSDT,) = rewardOf(_to);
if(rewardUSDT > 0) {
  credit[address(TOKEN_USDT)][_to] += rewardUSDT;
}
```

As can be seen, an **addition assignment** operator is being used on all of the updates to the `credit` mapping. This is wrong since the current value of the respective credit for each asset is already being added to the reward variables that are retrieved through the `StakedCSX::rewardOf()` function. This means that on each call to `StakedCSX::_claimToCredit()`, the credit values are going to get **doubled**.

```
if(credit[address(TOKEN_WETH)][_account] > 0) {
  wethAmount += credit[address(TOKEN_WETH)][_account];
}

if(credit[address(TOKEN_USDC)][_account] > 0) {
  usdcAmount += credit[address(TOKEN_USDC)][_account];
}

if(credit[address(TOKEN_USDT)][_account] > 0) {
  usdtAmount += credit[address(TOKEN_USDT)][_account];
}
```

And because `StakedCSX::_claimToCredit()` is being called in the `StakedCSX::_beforeTokenTransfer()` callback, this error can easily be abused to infinitely double the reward credits of a given user by simply making zero value `StakedCSX` token transfers.

Impact

Malicious stakes can easily drain all of the staking rewards from the `StakedCSX.sol` contract, making the contract insolvent and leaving other stakes empty-handed.

Proof of Concept

In the following test, it can be seen how a user who is entitled to only 1/8 of the staking rewards is able to claim **all** of them, using the above-described vulnerability:

```
it("should double the reward of a given user on each token transfer",
   async () => {
  const amount1 = ethers.parseEther("500"); // 1/8 of underlying supply
  const amount2 = ethers.parseEther("3500"); // 7/8 of underlying supply
  const distributeAmount = ethers.parseUnits("500", 6);

  await CSXToken.connect(deployer).transfer(user1.getAddress(), amount1);
  await CSXToken.connect(deployer).transfer(user2.getAddress(), amount2);
  await CSXToken.connect(user1).approve(staking.target, amount1);
  await CSXToken.connect(user2).approve(staking.target, amount2);

  await staking.connect(user1).stake(amount1);
  await staking.connect(user2).stake(amount2);

  await USDCToken.connect(deployer).approve(staking.target,
     distributeAmount);
  await staking
    .connect(deployer)
    .depositDividend(USDCToken.target, distributeAmount);
  await staking.connect(keeperNode).distribute(true, true, true);

// credit[USDC][user1] = 0 => rewardOf(user1) = 500
  const initialReward = await staking.rewardOf(user1.getAddress());
  await staking
    .connect(user1)
    .transfer("0x0000000000000000000000000000000000000001", BigInt(0));
// credit[USDC][user1] = 500 => rewardOf(user1) = 500
await staking
  .connect(user1)
  .transfer("0x0000000000000000000000000000000000000001", BigInt(0));

// credit[USDC][user1] = 1000 => rewardOf(user1) = 1000
await staking
  .connect(user1)
  .transfer("0x0000000000000000000000000000000000000001", BigInt(0));

// credit[USDC][user1] = 2000 => rewardOf(user1) = 2000
await staking
  .connect(user1)
  .transfer("0x0000000000000000000000000000000000000001", BigInt(0));

// credit[USDC][user1] = 4000 => rewardOf(user1) = 4000
const postTransferReward = await staking.rewardOf(user1.getAddress());

console.log("> Initial reward: ", initialReward.usdcAmount);
console.log("> Reward after 4 transfers: ", postTransferReward.usdcAmount
   );
expect(postTransferReward.usdcAmount).to.equal(
  initialReward.usdcAmount * BigInt(8)
);

const usdcBalanceBefore = await USDCToken.balanceOf(staking.getAddress())
   ;
await staking.connect(user1).claim(true, false, false, false);

const usdcBalanceAfter = await USDCToken.balanceOf(staking.getAddress());

console.log("> USDC balalance before claim: ", usdcBalanceBefore);
console.log("> USDC balalance after claim: ", usdcBalanceAfter);

expect(usdcBalanceAfter).to.equal(0);
});
```

To run the PoC test, add it to the `Staking` describe a block of the `StakedCSX.test.ts` file and then run `npx hardhat test --grep "should double the reward of a given user on each token transfer"` in the project root directory.

Location of Affected Code

File: StakedCSX.sol#L296

File: StakedCSX.sol#L300

File: StakedCSX.sol#L304

Recommendation

To address this vulnerability, replace the addition assignment operators with plain assignment operators:

```
function _claimToCredit(address _to) private {
  if(balanceOf(_to) != 0) {
    (,,uint256 rewardWETH) = rewardOf(_to);
    if (rewardWETH > 0) {
-       credit[address(TOKEN_WETH)][_to] += rewardWETH;
+       credit[address(TOKEN_WETH)][_to] = rewardWETH;
    }
    (uint256 rewardUSDC,,) = rewardOf(_to);
    if(rewardUSDC > 0) {
-       credit[address(TOKEN_USDC)][_to] += rewardUSDC;
+       credit[address(TOKEN_USDC)][_to] = rewardUSDC;
    }
```

```diff
     (,uint256 rewardUSDT,) = rewardOf(_to);
     if(rewardUSDT > 0) {
-        credit[address(TOKEN_USDT)][_to] += rewardUSDT;
+        credit[address(TOKEN_USDT)][_to] = rewardUSDT;
     }
   }
   _updateRewardRate(_to, address(TOKEN_WETH));
   _updateRewardRate(_to, address(TOKEN_USDC));
   _updateRewardRate(_to, address(TOKEN_USDT));
}
```

Team Response

Fixed by removing the `credit` mapping altogether and modifying the `_updateRewardRate()` function.

[H-01] Unclaimed User Staking Rewards Are Going to Get Lost on Each Call to `stake()` Function in `StakedCSX` Contract

Severity

High Risk

Description

The current implementation of the `stake()` function in `StakedCSX.sol` calls `_updateRewardRate()` internally for all three reward tokens. This is incorrect as it will just reset the reward rates of the caller, leading to the loss of all of their unclaimed staking rewards.

Impact

Users are going to lose all of their unclaimed rewards whenever they stake additional `CSX` tokens.

Location of Affected Code

File: StakedCSX.sol#L92-L94

```solidity
//**
//* @notice Stakes the user's CSX tokens
//* @dev Mints sCSX & Sends the user's CSX to this contract.
//* @param _amount The amount of tokens to be staked
//*
function stake(uint256 _amount) external nonReentrant {
  if (_amount == 0) {
    revert AmountMustBeGreaterThanZero();
  }
_updateRewardRate(msg.sender, address(TOKEN_WETH));
_updateRewardRate(msg.sender, address(TOKEN_USDC));
_updateRewardRate(msg.sender, address(TOKEN_USDT));
_mint(msg.sender, _amount);
TOKEN_CSX.safeTransferFrom(msg.sender, address(this), _amount);
emit Stake(msg.sender, _amount);
}
```

Recommendation

Replace the `_updateRewardRate()` calls with a single `claimToCredit()` call, which both updates the reward rates of the user and saves their rewards to the credit mapping for later claiming:

```
function stake(uint256 _amount) external nonReentrant {
  if (_amount == 0) {
    revert AmountMustBeGreaterThanZero();
  }
- _updateRewardRate(msg.sender, address(TOKEN_WETH));
- _updateRewardRate(msg.sender, address(TOKEN_USDC));
- _updateRewardRate(msg.sender, address(TOKEN_USDT));
+ _claimToCredit(msg.sender);
  _mint(msg.sender, _amount);
  TOKEN_CSX.safeTransferFrom(msg.sender, address(this), _amount);
  emit Stake(msg.sender, _amount);
}
```

Team Response

Fixed by changing the reward logic and removing the `_claimToCredit()`.

[H-02] The `USDT` Reward Claiming Functionality of `VestedStaking` Can Be DoSed Due to a Logical Error

Severity

High Risk

Description

There is a logical error in the current implementation of the `claimRewards` function in the `VestedStaking.sol` contract, and more specifically, in its logic for claiming USDT rewards.

```
if (claimUsdt) {
  if (usdtAmount != 0) {
    uint256 beforeBalance = IUSDT_TOKEN.balanceOf(address(this));
    IUSDT_TOKEN.safeTransfer(msg.sender, usdtAmount);
    usdtAmount = IUSDT_TOKEN.balanceOf(msg.sender) - beforeBalance;
  }
}
```

As it can be seen, the balance of the current contract is being fetched into the `beforeBalance` variable. After the transfer of USDT rewards to the `msg.sender` that value gets subtracted from the `msg.sender` balance, in order to determine the actual amount of USDT that was transferred. Those operations will lead to the wrong value of `usdtAmount` being calculated.

Not only that but when the vester of the given contract has a zero or close to zero USDT balance, this issue can be abused by malicious users in order to DoS the USDT reward-claiming functionality of the vested staking contract. This can be achieved by directly transferring a USDT amount that is greater than the vester's balance to the `VestedStaking` contract - leading to the USDT reward claiming functionality always reverting on `VestedStaking.sol/L172` due to an arithmetic underflow error. The same DoS scenario is possible if the USDT transfer fees get enabled and they are greater than the vester's USDT balance.

Impact

Users won't be able to claim their USDT staking rewards from their `VestedStaking` contracts.

Additionally, the `Claim` event at the end of the function will be emitted with a wrong USDT value when the function is not DoSed.

Proof of Concept

The following test demonstrates how a user that has a X USDT balance can be prevented from claiming their USDT staking rewards.

```
it("should revert reward claiming when the vester has a X USDT balance
   and the contract has a > X USDT balance", async () => {
  const vesterUsdtBalance = ethers.parseUnits("2", 6); // Vester has a
     total of 2 USDT
  await usdt.connect(deployer).transfer(vesterAddress, vesterUsdtBalance)
     ;

  const depositAmount = ethers.parseUnits("1000", 6);
  await usdt.connect(deployer).approve(stakedCSX.getAddress(),
     depositAmount);
  await stakedCSX
  .connect(deployer)
  .depositDividend(await usdt.getAddress(), depositAmount);
  await stakedCSX.connect(council).distribute(false, false, true);
// Transfer 0.000001 USDT more than the vested balance to make the
    claimRewards() function revert
  await usdt
    .connect(deployer)
    .transfer(vestedStaking.getAddress(), vesterUsdtBalance + BigInt(1));

  await expect(
    vestedStaking.connect(vesterAddress).claimRewards(false, true, false,
        false)
  ).to.be.revertedWithPanic("0x11"); // Arithmetic under/overflow panic
      code
});
```

To run the PoC test, add it to the `VestedStaking` describe block of the `VestedStaking.test.ts` file and then run `npx hardhat test --grep "should revert reward claiming when the vester has a X USDT balance and the contract has a > X USDT balance"` in the project root directory

Location of Affected Code

File: contracts/CSX/VestedStaking.sol#L170

Recommendation

Fetch the balance of `msg.sender` in the `beforeBalance` variable instead of the balance of the current contract:

```
if (claimUsdt) {
  if (usdtAmount != 0) {
- uint256 beforeBalance = IUSDT_TOKEN.balanceOf(address(this));
+ uint256 beforeBalance = IUSDT_TOKEN.balanceOf(msg.sender);
  IUSDT_TOKEN.safeTransfer(msg.sender, usdtAmount);
  usdtAmount = IUSDT_TOKEN.balanceOf(msg.sender) - beforeBalance;
  }
}
```

Team Response

Fixed as suggested.

[M-01] A Buyer Can Use Two Different Referral Codes for the Same Trade

Severity

Medium Risk

Description

There is no validation on the referral code that is passed into `CSXTrade::commitBuy()`. The code is simply assigned to the `referralCode` storage variable and then is used to calculate the buyer discount in the function itself.

```
(uint256 buyerNetValue, , , ) = getNetValue(_affLink, weiPrice);
depositedValue = _transferToken(msg.sender, address(this), buyerNetValue)
    ;
```

However, in `CSXTrade::_distributeProceeds()` the referral code is also being used, but in that function, a check is made to see if the buyer already has a referral code attached to them in the `ReferralRegistry.sol`, and if they have one, that code is being used instead.

```
bytes32 storageRefCode = referralRegistryContract.getReferralCode(buyer);

if (storageRefCode != bytes32(0)) {
  if (storageRefCode != referralCode) {
    referralCode = storageRefCode;
  }
}
```

Because of that, a given buyer can use two different referral codes for a single trade.

IMPORTANT NOTE: Right now, this issue can't be abused for more than just receiving slightly more referral fees. However if the `M-02` issue gets fixed, this one can actually be used to DoS the `CSXTrade::_distributeProceeds` function **permanently**, leading to the sellers not being able to claim their tokens at the end of a trade. For example, the scenario described in the **Attack Scenario** section will lead to this impact, if that issue gets fixed. Because of that, it is **crucial** that if you decide to fix that vulnerability, you should fix this one as well.

Impact

Buyers will be able to extract more referral fees from a single trade than they should be able to.

Attack Scenario

This example scenario showcases how the vulnerability in question can be abused to get more fees than it would normally be possible to:

1. A user calls `ReferralRegistry::setReferralCodeAsUser()` with a referral code that has `ownerRatio = 100`
2. The same user then calls `CSXTrade::commitBuy()` while passing in a different referral code that has `buyerRatio = 100`
3. The buyer gets the maximum discount applied to their buy deposit
4. Afterwards, if the trade is completed successfully, the referral code owner will also get the max referrer fee amount sent to them

Location of Affected Code

CSXTrade.sol#L233

Recommendation

Move the logic for checking whether a buyer has a referral code set for them from `_distributeProceeds()` to the `commitBuy()` function in the `CSXTrade.sol` contract.

Team Response

Fixed as proposed.

[M-02] Whenever There Is a Buyer Discount and a Trade Is Completed Successfully, an Amount of Tokens that Is Equal to the Discount Fee Will Get Stuck in the `CSXTrade` Contract

Severity

Medium Risk

Description

Due to the way that the `CSXTrade::_distributeProceeds()` function calculates the number of payment tokens to be sent out, whenever there is a buyer discount from the referral code used by the buyer, several tokens that are equal to the discount fee will get stuck in the contract forever when that function is used to complete a given trade. That's because the `depositedValue()` (which is equal to `weiPrice - discountedFee` for non-fee-on-transfer tokens) is passed into `getNetValue()` to calculate the final output amounts.

```
(uint256 buyerNetPrice, uint256 sellerNetProceeds, uint256
   affiliatorNetReward, uint256 tokenHoldersNetReward) = getNetValue(
   referralCode, depositedValue);
```

The lost amount itself will come from the `sellerNetProceeds`, `affiliatorNetReward` and `tokenHoldersNetReward` values - i.e. the transferred out amounts will be proportionally less than they could have been, depending on the discount fee.

Impact

An amount of tokens that is equal to the discount fee will get lost on some trades.

Location of Affected Code

File: CSXTrade.sol#L558

Recommendation

To mitigate this issue, one possible approach would be to add an additional storage variable, which is calculated after `depositedValue` is assigned like so:

`weiPrice(depositedValue1e18/buyerNetValue)/1e18`. That way, the discount fee is taken into account, while the support for a fee on transfer tokens is also kept:

```
  uint256 public depositedValue;
+ uint256 public depositedValueWithFees;
  uint256 public weiPrice;
```

```
function commitBuy(
  TradeUrl memory _buyerTradeUrl,
  bytes32 _affLink,
  address _buyerAddress
) external nonReentrant {
// code
  (uint256 buyerNetValue, , , ) = getNetValue(_affLink, weiPrice);
  depositedValue = _transferToken(msg.sender, address(this),
     buyerNetValue);
+ depositedValueWithFees = weiPrice * (depositedValue * 1e18 /
   buyerNetValue) / 1e18; // Will work for all ERC20 tokens that have
   decimals <= 18
// code
}
```

```
function _distributeProceeds() private {
// code
- (uint256 buyerNetPrice, uint256 sellerNetProceeds, uint256
   affiliatorNetReward, uint256 tokenHoldersNetReward) = getNetValue(
   referralCode, depositedValue);
+ (uint256 buyerNetPrice, uint256 sellerNetProceeds, uint256
   affiliatorNetReward, uint256 tokenHoldersNetReward) = getNetValue(
   referralCode, depositedValueWithFees);
  ...
}
```

Team Response

Fixed as proposed.

[M-03] Wrong Address Check in `receive()` Function in `VestedStaking.sol` Contract Can Lead to Integrating Contracts Not Being Able to Claim Their Staking Rewards

Severity

Medium Risk

Description

The implementation of the `receive()` function of the `VestedStaking.sol` contract currently looks like this:

```
receive() external payable {
  if (address(IERC20_WETH_TOKEN) != msg.sender) {
    revert InvalidSender();
  }
}
```

As it can be seen, it only contains a single if statement that checks whether the `msg.sender` is the WETH address to protect against accidental ETH transfers. However, the check itself is incorrect, because when the `claimRewards()` function of the contract is called with `convertWethToEth` being equal to true (which is the only scenario where the contract will receive ETH in an expected manner), the ETH that is going to be sent to the contract will be sent from the `StakedCSX` contract, rather than from the `WETH` contract. Because of that, the convert WETH to ETH functionality of `claimRewards()` in `VestedStaking.sol` is practically broken.

What's worse is that since that functionality is present, the users of the protocol will assume that it works properly. And because of that, in the scenario where the vester of a given `VestedStaking.sol` contract is a smart contract itself if its logic for claiming the rewards from `VestedStaking.sol` is static and has the `convertWethToEth` argument of `claimRewards()` hardcoded to `true`, there will be no way to claim the accumulated staking rewards at all.

Impact

In the case where the vester is a smart contract, it is possible that all staking rewards accumulated by the `VestedStaking.sol` contract might be lost forever, depending on the vester contract implementation.

Location of Affected Code

File: VestedStaking.sol#L196

Recommendation

Check that the `msg.sender` in the receive function is the `sCSX` address rather than the `WETH` ' address:

```
receive() external payable {
- if (address(IERC20_WETH_TOKEN) != msg.sender) {
+ if (address(ISTAKED_CSX) != msg.sender) {
    revert InvalidSender();
  }
}
```

Team Response

Fixed as proposed.

[M-04] Some Trade-Terminating Functions Do Not Call `removeAssetIdUsed()` Internally, Potentially Leading The Same Skin Not Being Tradable On CSX Again

Severity

Medium Risk

Description

Unlike other trade-terminating functions, `CSXTrade::buyerCancel()`, `CSXTrade::sellerTradeVeridict()` and `CSXTrade::sellerConfirmsTrade()` don't call `Users::removeAssetIdUsed()` after transferring the committed tokens to the buyer/seller. And since asset IDs are the unique identifiers of Steam items, this means that the same skin won't be tradable on CSX again. Although asset IDs change, they do so only when the items are traded or changed in some way, so it is unlikely that users will know how or want to change the asset ID of their skin, in order for it to become tradable on CSX again.

This can prove to be especially problematic with the `CSXTrade::buyerCancel()` function, as it can be used to intentionally DoS the trading of concrete user skins.

Impact

Some skins will become impossible to trade on CSX for an undefined amount of time.

Location of Affected Code

File: CSXTrade.sol#L267-L283

File: CSXTrade.sol#L291-L329

File: CSXTrade.sol#L363-L381

Recommendation

Add a `Users::removeAssetIdUsed` call at the appropriate places in all the above-mentioned functions:

```
function buyerCancel() external onlyAddress(buyer) nonReentrant {
// code
  IUSERS_CONTRACT.changeUserInteractionStatus(address(this), buyer,
     status);

  _transferToken(address(this), buyer, depositedValue);

+ _rmvAId();
}
```

```
function sellerTradeVeridict(
  bool _sellerCommited
) external onlyAddress(SELLER_ADDRESS) nonReentrant {
// code
  } else {
    _changeStatus(TradeStatus.SellerCancelledAfterBuyerCommitted, "
       SE_DEFAULT");
    IUSERS_CONTRACT.changeUserInteractionStatus(address(this),
       SELLER_ADDRESS, status);
    IUSERS_CONTRACT.changeUserInteractionStatus(address(this), buyer,
       status);
    _transferToken(address(this), buyer, depositedValue);

+   _rmvAId();
  }
}
```

```
function sellerConfirmsTrade() external onlyAddress(SELLER_ADDRESS)
    nonReentrant {
// code
  _changeStatus(TradeStatus.Completed, _data);
  IUSERS_CONTRACT.endDeliveryTimer(address(this), SELLER_ADDRESS);
  IUSERS_CONTRACT.changeUserInteractionStatus(address(this),
     SELLER_ADDRESS, status);
  IUSERS_CONTRACT.changeUserInteractionStatus(address(this), buyer,
     status);

  _distributeProceeds();

+ _rmvAId();
}
```

```
+ function _rmvAId() private {
+   bool _s = IUSERS_CONTRACT.removeAssetIdUsed(itemSellerAssetId,
    SELLER_ADDRESS);

+   if (!_s) {
+       revert TradeIDNotRemoved();
+   }
+ }
```

Team Response

Fixed by using the newly added `_rmvAId()` function.

[M-05] Users Can Use the Referral System to Refer Themselves in Order to Buy Skins at a Discount

Severity

Medium Risk

Description

The current implementation of the referral system has a check that does not let buyers assign referral codes owned by their addresses on trades.

```
// If the referral code is valid, then we fetch the owner of the referral
    code.
address refOwner = referralRegistryContract.getReferralCodeOwner(
   referralCode);
// Check if the owner of the referral code is not the buyer.
// If it's the buyer, then we set the referral code to zero.
// If it's not the buyer, then we set the referral code as Primary for
   the buyer.
if (refOwner == buyer) {
  referralCode = bytes32(0);
} else {
  referralRegistryContract.setReferralCodeAsTC(referralCode, buyer);
}
```

This is fine on its own, but as we know, a given user can have a countless number of externally owned addresses. And because there is no access control on the `ReferralRegistry::registerReferralCode()` function, buyers can simply create referral codes from a different address than the one they are buying from and use those, in order to bypass the self-referral restriction.

Impact

Users can abuse the referral system in order to buy skins at a discount.

Location of Affected Code

File: ReferralRegistry.sol#L182-L206

```
//**
//* @notice Register a referral code with distribution ratios
//* @param referralCode The referral code
//* @param ownerRatio affiliator rebate ratio
//* @param buyerRatio buyer discount ratio
//*
function registerReferralCode(
  bytes32 referralCode,
  uint256 ownerRatio,
  uint256 buyerRatio
) external {
  if (referralCode == 0) revert InvalidReferralCode("Referral code cannot
      be empty");
  if (referralInfos[referralCode].owner != address(0)) revert
     InvalidReferralCode("Referral code already registered");
  if (ownerRatio + buyerRatio != 100) revert InvalidRatios("The sum of
     ownerRatio and buyerRatio must be 100");
  if (containsSpace(referralCode)) revert InvalidReferralCode("Referral
     code cannot contain spaces");

  referralInfos[referralCode] = ReferralInfo({
    owner: msg.sender,
    ownerRatio: ownerRatio,
    buyerRatio: buyerRatio
  });

  userCreatedCodes[msg.sender].push(referralCode);

  emit ReferralCodeRegistered(
    referralCode,
    msg.sender,
    ownerRatio,
    buyerRatio
  );
}
```

Recommendation

Add a whitelist mechanism that only allows users who have bought/sold X amount of skins to create referral codes.

Team Response

Acknowledged, will be factored into base fee.

[M-06] Malicious Buyers Can Get Skins Without Paying For Them If The Steam Trade Is Accepted By the Seller After More Than 24 Hours Have Passed Since the Buy Commit Placement

Severity

Medium Risk

Description

The `CSXTrade.sol` contract requires sellers to accept the Steam trade of the buyer for the skin they are selling before receiving their desired amount of tokens. After the trade is accepted by the seller, the Keeper Oracle calls the `CSXTrade::keeperNodeConfirmsTrade` function, which releases the desired token amount from the contract to the seller. However, in some scenarios, this trading approach can be exploited to get the seller's skin without paying for it.

If more than 24 hours have passed since the buy commit was made, a malicious buyer can front-run the `CSXTrade::keeperNodeConfirmsTrade` call with a `CSXTrade::buyerCancel` call. That way, the buyer will be able to get their deposited tokens back, even though the skin was already traded to them.

Impact

Skins are going to get stolen from sellers.

Attack Scenario

To illustrate the issue further, let's take a look at the following example scenario:

1. Bob creates a `CSXTrade` for his Karambit Fade
2. Alice makes a buy commit on Bob's trade and sends him a Stream trade offer for the skin
3. Bob decides to wait a little to see whether he won't be able to get a better offer for his skin from a different marketplace
4. 2 days later, he decides that the trade he received from CSX is the best one he can get, so he accepts the Steam trade Alice sent him
5. The Keeper Oracle sees that the trade has been made, so it sends a transaction that calls `CSXTrade::keeperNodeConfirmsTrade`
6. Unfortunately for Bob though, Alice has been carefully monitoring the transaction mem-pool. After she sees the transaction of the Oracle, she front runs it with her one that calls `CSXTrade::buyerCancel`
7. The transaction of Alice gets successfully executed, while the one of the Keeper Oracle gets reverted. Alice now has both her tokens back and the skin, while Bob is left empty-handed

Location of Affected Code

File: CSXTrade.sol#L267-L283

```
//**
//* @notice Cancel the trade
//* @dev Only the buyer can cancel the trade
//* @dev The listing must be in status BuyerCommitted
//* @dev The buyer can only cancel the trade if the seller has not
   veridicted the trade.
//* @dev The buyer can only cancel the trade if 24 hours have passed
   since the buyer committed to buy.
//*
function buyerCancel() external onlyAddress(buyer) nonReentrant {
  if (status != TradeStatus.BuyerCommitted) {
    evert StatusIncorrect();
  }

  if (block.timestamp < buyerCommitTimestamp + 24 hours) {
    revert TimeNotElapsed();
  }
```

```
    _changeStatus(TradeStatus.BuyerCancelled, "BU_DEFAULT");
    IUSERS_CONTRACT.changeUserInteractionStatus(
      address(this),
      SELLER_ADDRESS,
      status
    );
    IUSERS_CONTRACT.changeUserInteractionStatus(address(this), buyer,
       status);

    _transferToken(address(this), buyer, depositedValue);
}
```

Recommendation

Add an authorization mechanism to the `buyerCancel()` function in the `CSXTrade.sol` contract. For example, you can require a Keeper to authorize the buyer cancelation before the buyer can execute it.

Team Response

Acknowledged.

[M-07] `Vested CSX` to `Regular CSX` Conversion Process Enables Potential Unauthorized Withdrawal of Staked Deposits by Malicious Council

Severity

Medium Risk

Description

Some of the key privileges of the Council role:

- Ability to execute a forced withdrawal of tokens from the contract via `cliff()` function without any restrictions. In the standard `withdraw()` method there is a validation check that requires the vesting period (2 years) to pass before the `Vester` is able to withdraw.
- Ability to change factory and keeper contracts.

Additionally, the Council role can be changed through `changeCouncil()` function and mistakenly a wrong address can be given resulting in this role being set to an unexpected address.

Location of Affected Code

File: VestedStaking.sol#L191

```
/// @notice Executes a forced withdrawal of tokens from the contract.
/// @dev Can only be called by the council to mitigate against malicious
   vesters.
/// @param amount Specifies the amount of tokens to be withdrawn.
function cliff(uint256 amount) external onlyCouncil {
  if (amount > vesting.amount || amount == 0) {
    revert NotEnoughTokens();
  }
  vesting.amount -= amount;
  cliffedAmount += amount;
  sCsxToken.unStake(amount);
  csxToken.safeTransfer(msg.sender, amount);
  emit Cliff(msg.sender, amount, vesting.amount);
}
```

File: Keepers.sol#L94

```
function changeCouncil(address _newCouncil) public onlyCouncil {
  if (_newCouncil == address(0)) {
    revert NotCouncil();
  }
  council = _newCouncil;
  emit CouncilChanged(_newCouncil);
}
```

Recommendation

There are two vectors to resolve the risks:

1. Consider removing some owner privileges or using a `Timelock` contract.
2. It is recommended to implement a two-step process where the council nominates an account, and the nominated account needs to call an `acceptCouncilRole()` function for the transfer of the council role to succeed fully. This ensures the nominated `EOA` account is valid and active.

Team Response

Acknowledged and fixed as proposed.

[M-08] Tokens with a `Fee-On-Transfer` Mechanism Could Break the Protocol

Severity

Medium Risk

Description

The current transferring implementation is incompatible with tokens that have a `fee-on-transfer` mechanism. Such tokens for example is `PAXG`, while `USDT` has a built-in fee-on-transfer mechanism that is currently switched off.

This will work incorrectly if the token has a `fee-on-transfer` mechanism - the contract will cache the amount as its expected added balance, but it will actually add the amount - the fee balance. This will result in the inability to withdraw tokens out of the contract.

However, the likelihood is relatively low since there are interface checks for the supported tokens, but still `USDT` is one of them.

Location of Affected Code

File: StakedCSX.sol

```
function depositDividend(address _token, uint256 _reward) nonReentrant
   external returns (bool) {
function claim( bool claimUsdc, bool claimUsdt, bool claimWeth, bool
   convertWethToEth ) external nonReentrant {
```

File: VestedStaking.sol#L134

```
function claimRewards(bool claimUsdc, bool claimUsdt, bool claimWeth,
   bool convertWethToEth) external onlyVester {
```

File: CSXTrade.sol

```
function commitBuy(TradeUrl memory _buyerTradeUrl, bytes32 _affLink,
   address _buyerAddress) public {
function buyerCancel() external onlyAddress(buyer) {
function sellerTradeVeridict( bool _sellerCommited) public onlyAddress(
   seller) {
function buyerConfirmReceived() external onlyAddress(buyer) {
function sellerConfirmsTrade() external onlyAddress(seller) {
function keeperNodeConfirmsTrade(bool isTradeMade, string memory message)
    external onlyKeeperNode {
function resolveDispute(bool isFavourOfBuyer, bool giveWarningToSeller,
   bool giveWarningToBuyer, bool isWithValue) external onlyKeepersOrNode
   {
```

Recommendation

There are two options to resolve this:

1. Consider caching the balance before a `transferFrom()/safeTransferFrom()` to the contract and then check it after the transfer and use the difference between them as the newly added balance. This requires a `nonReentrant` modifier, as generally, `ERC777` tokens have a callback mechanism that introduces a re-entrancy vulnerability.
2. Exclude `USDT` from the whitelisted tokens or document that you might do this in the future when the fee-on-transfer mechanism is activated.

Team Response

Acknowledged and decided to update the whitepaper.

[L-01] DoS Attack on `getTradeIndexesByStatus()` Function By Creating Listings with Large Sticker Arrays

Severity

Low Risk

Description

In the current implementation of the `CSXTradeFactory::createListingContract()` function, there is no limit to the size of the stickers array being passed in.

Since `CSXTradeFactory::getTradeIndexesByStatus()` loops through each sticker array of the trades that it iterates through (by calling `getTradeDetailsByIndex()`), this can be abused to make `getTradeIndexesByStatus()` revert for some inputs, due to an OOG error.

Impact

The `getTradeIndexesByStatus()` function in the `CSXTradeFactory.sol` contract can be DoSed for some specific inputs.

Although the function is not being called anywhere within the smart contracts of the protocol itself, if it is used somewhere off-chain, this can still have adverse effects on the overall functionality of the protocol.

Location of Affected Code

File: CSXTradeFactory.sol#L102-L209

File: CSXTradeFactory.sol#L328

File: CSXTradeFactory.sol#L283-L290

Recommendation

To address this vulnerability, limit the length of the sticker arrays. It looks like the maximum amount of stickers that a single skin can have is 5 (source).

Team Response

Fixed as suggested.

[L-02] If a Referral Code Owner Gets Blacklisted From `USDC/USDT`, Seller Trade Rewards Will Get Locked

Severity

Low Risk

Description

The current implementation of `CSXTrade::_distributeProceeds` has a push mechanism for distributing referral code owner rewards.

```
uint256 actualAmountTransferredToAff = _transferToken(address(this),
    referralRegistryContract.getReferralCodeOwner(referralCode),
    affiliatorNetReward);
```

However, as it's known, both the `USDC` and `USDT` contracts have a blacklisting functionality implemented in them. This, combined with the above-mentioned push mechanism can lead to serious problems down the road. More specifically, in the scenario where a referral code with an owner ratio that is `> 0` is used for a given trade, if the owner of that code gets blacklisted from the `paymentToken`, the `CSXTrade::_distributeProceeds` will become DoSed, leading to the seller of the trade not being able to claim their rewards in any way.

Impact

Sellers won't be able to receive their rewards at the end of trades.

Location of Affected Code

File: CSXTrade.sol#L561

Recommendation

To mitigate this issue, you can either implement a pull-withdraw mechanism for the affiliation fees or you can make the referral code owner transfer fail silently, by wrapping it in a try/catch.

Team Response

Acknowledged.

[L-03] Centralization Risk Due To All CSX Tokens Being Minted To The Contract Creator

Severity

Low Risk

Description

Due to the way that the `CSXToken.sol` contract is currently implemented, when it gets deployed, the maximum supply of it (`100 million`, to be exact) is going to get minted to its deployer. After the contract is deployed, the deployer will be responsible for distributing all of the `CSX tokens` to the protocol users.

The problem with that approach is that since the CSX token is the governance token of the protocol, this means that the CSX creator will be able to single-handedly accept or reject governance proposals. In an event where the address of the creator gets compromised, this could prove to be catastrophic for the protocol.

Impact

A single authority will hold all/most of the governance power.

Location of Affected Code

File: CSXToken.sol#L19

```
constructor() ERC20("CSX Token", "CSX") {
  _mint(msg.sender, maxSupply);
}
```

Recommendation

Refactor the CSX minting mechanism, so that when the token contract is deployed it's `totalSupply` is equal to zero and tokens get minted from that point on, whenever there is a need to do so.

Team Response

Fixed as proposed.

[L-04] Incomplete Validation In Multiple State Changing Functions

Severity

Low Risk

Description

1. Missing zero-address checks

It has been detected that almost all functions of the smart contracts are missing address validation. Every input address should be checked not to be zero, especially the ones that could lead to rendering the contract unusable, lock tokens, etc. This is considered a best practice.

2. No upeer-limit on `changeBaseFee()` and `baseFee` in constructor.
3. Add validation for 3 false input parameters in the `distribute()` function, in order to prevent emitting the Distribute event unnecessarily.

Location of Affected Code

File: TradeFactoryBase.sol

```
constructor(address _keepers, address _users, address
    _tradeFactoryBaseStorage, uint256 _baseFee) {
  keepersContract = IKeepers(_keepers);
  usersContract = IUsers(_users);
  tradeFactoryBaseStorage = ITradeFactoryBaseStorage(
     _tradeFactoryBaseStorage);
  baseFee = _baseFee;
}

function changeBaseFee(uint256 _baseFee) external isCouncil {
  baseFee = _baseFee;
}
```

File: StakedCSX.sol#L119

```
function distribute(bool dWeth, bool dUsdc, bool dUsdt) external {
```

Recommendation

Consider incorporating zero-address checks into the majority of state-changing functions, introducing an upper limit for the `changeBaseFee()` function and `baseFee` setting in the constructor, and implementing appropriate input validation within the `distribute()` function.

Team Response

Acknowledged and fixed as proposed.

[L-05] The `safeApprove()` Function Could Revert For Non-Standard Token Like `USDT`

Severity

Low Risk

Description

Some non-standard tokens like `USDT` will revert when a contract or a user tries to approve an allowance when the spender allowance has already been set to a non-zero value.

We also should note that `OpenZeppelin` has officially deprecated the `safeApprove()` function, suggesting to use instead of `safeIncreaseAllowance()` and `safeDecreaseAllowance()`.

Location of Affected Code

File: VestedCSX.sol

File: VestedStaking.sol

File: CSXTrade.sol

Recommendation

Consider replacing deprecated functions, the official `OpenZeppelin` documentation recommends using `safeIncreaseAllowance()` & `safeDecreaseAllowance()`.

Team Response

Acknowledged and fixed as proposed.

[L-06] The `stake()` Function does Not Follow The CEI Pattern

Severity

Low Risk

Description

In `stake()` function, even though there is a `nonReentrant` modifier, the **Checks-Effects-Interactions** pattern is not followed. The mint is executed after the `_updateRewardRate()` function. It is recommended to always first change the state before updating the internal state - while the code is not vulnerable right now due to the `nonReentrant` modifier and the lack of a callback function, it is still a best practice to be followed.

Location of Affected Code

File: StakedCSX.sol#L69

```
function stake(uint256 _amount) external nonReentrant {
  if (_amount == 0) {
    revert AmountMustBeGreaterThanZero();
  }
  _mint(msg.sender, _amount);
  _updateRewardRate(msg.sender, address(tokenWETH));
  _updateRewardRate(msg.sender, address(tokenUSDC));
  _updateRewardRate(msg.sender, address(tokenUSDT));
  tokenCSX.safeTransferFrom(msg.sender, address(this), _amount);
  emit Stake(msg.sender, _amount);
}
```

Recommendation

Apply the `Checks-Effects-Interactions` pattern.

Team Response

Acknowledged and fixed as proposed.

our shielding · Your smart contracts, our shielding · Your smart

shieldify

Thank you!