VTVL contest
Findings & Analysis Report

2022-11-01

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the VTVL smart contract system written in Solidity. The audit contest took place between September 20—September 23 2022.

Wardens

208 Wardens contributed reports to the VTVL contest:

1. pashov
2. sorrynotsorry
3. Respx
4. fatherOfBlocks
5. m9800

6. wagmi
7. CertoraInc (egjlmn1, OriDabush, ItayG, shakedwinder, and RoiEvenHaim)
8. csanuragjain
9. TomJ
10. hansfriese
11. ayeslick
12. Lambda
13. rbserver
14. KIntern_NA (TrungOre and duc)
15. neko_nyaa
16. rokinot
17. 0xSky
18. neumo
19. bin2chen
20. Trust
21. wastewa
22. datapunk
23. 0xhunter
24. dipp
25. wuwe1
26. lllllll
27. Ruhum
28. obront
29. RustyRabbit
30. 0xSmartContract
31. 0xA5DF

32. pedroais
33. pcarranzav
34. ElKu
35. Czar102
36. sashik_eth
37. pauliax
38. 0x52
39. 0xdapper
40. eierina
41. AkshaySrivastav
42. JohnSmith
43. __141345__
44. djxploit
45. 0xDecorativePineapple
46. zzzitron
47. hyh
48. MiloTruck
49. rotcivegaf
50. JLevick
51. Aymen0909
52. supernova
53. 0x4non
54. Chom
55. ak1
56. 0xNazgul
57. 0x1f8b
58. rajatbeladiya

59. joestakey
60. berndartmueller
61. c3phas
62. lukris02
63. pfapostol
64. ajtra
65. imare
66. cryptostellar5
67. Deivitto
68. gogo
69. Bnke0x0
70. oyc_109
71. JC
72. durianSausage
73. Diana
74. brgltd
75. ladboy233
76. Tomo
77. Rolezn
78. seyni
79. 0xbepresent
80. peanuts
81. OptimismSec (sseefried and tofunmi)
82. d3e4
83. RockingMiles (robee and pants)
84. Waze
85. tnevler

86. Funen
87. a12jmx
88. prasantgupta52
89. Sm4rty
90. martin
91. delfin454000
92. leosathya
93. RaymondFam
94. Rohan16
95. erictee
96. millersplanet
97. aysha
98. ChristianKuri
99. V_B (Barichek and vlad_bochok)
100. CodingNameKiki
101. karanctf
102. ret2basic
103. medikko
104. slowmoses
105. ReyAdmirado
106. B2
107. peiw
108. 0x040
109. carrotsmuggler
110. ikbkln
111. async
112. sach1r0

113. rvierdiiev
114. eighty
115. ignacio
116. bobirichman
117. got_targ
118. nalus
119. cryptphi
120. SooYa
121. tibthecat
122. natzuu
123. indijanc
124. 2997ms
125. exd0tpy
126. MasterCookie
127. StevenL
128. bulej93
129. Diraco
130. 0v3rf10w
131. 0x85102
132. Yiko
133. Bahurum
134. chatch
135. 0xmatt
136. cccz
137. innertia
138. reassor
139. zzykxx

140. 0x5rings
141. ubermensch
142. 0xf15ers (remora and twojoy)
143. Dravee
144. JohnnyTime
145. Aeros
146. yongskiws
147. romand
148. dic0de
149. peritoflores
150. sikorico
151. Margaret
152. pedr02b2
153. ch13fd357r0y3r
154. Junnon
155. Atarpara
156. jag
157. DimitarDimitrov
158. adriro
159. zishansami
160. ch0bu
161. SnowMan
162. Saintcode_
163. 0xsam
164. gianganhnguyen
165. WilliamAmbrozic
166. Tomio

167. samruna
168. yaemsobak
169. emrekocak
170. Tadashi
171. tgolding55
172. Ocean_Sky
173. caventa
174. beardofginger
175. dharma09
176. malinariy
177. lucacez
178. subtle77
179. 0xDanielC
180. mics
181. w0Lfrum
182. hxzy
183. Amithuddar
184. Tagir2003
185. 0xc0ffEE
186. Satyam_Sharma
187. Noah3o6
188. jpserrat
189. Matin
190. Sta1400
191. mrpathfindr
192. francoHacker
193. cRat1st0s

194. cryptonue
195. Franfran
196. GimelSec (rayn and sces60107)
197. JGcarv
198. Soosh

This contest was judged by Oxean.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 12 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 10 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 135 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 141 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 VTVL contest repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 239 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] Loss of vested amounts

Submitted by eierina, also found by 0x52, 0xA5DF, 0xdapper, ElKu, obront, pauliax, pcarranzav, pedroais, rbserver, Ruhum, RustyRabbit, and TomJ

VTVLVesting.sol#L418
VTVLVesting.sol#L147-L151
VTVLVesting.sol#L364

Vesting is a legal term that means the point in time where property is earned or gained by some person.
The VTVLVesting contract defines:

- a start time (`Claim::startTimestamp`) and an end time (`Claim::endTimestamp`) at which vesting starts and ends for a entitled user
- the calculated points in time when the fractions of the total amount are released and therefore can be withdrawn (which are defined by `Claim::releaseIntervalSecs`).

The entitled user can either withdraw after each interval elapses, or after the whole vesting period is over or any variant of the two options.

The administrator of the contract can revoke the claim for a user at any time, which for vesting assets is expected. For example an employee with a vesting stock allocation of 1000 shares vesting at each quarter over a period of 4 years, may resign after 2 years and therefore the only half of the shares would be vested and therefore sold by the employee. The employee can either sell them at each quarter, or before, or after resigning, in any case the half of the shares have vested and are by legal right owned by the employee.

The VTVLContract revoke has the following defects:

- it ignores the amount already vested and now yet withdrawn
- if called, say half-way the total period, just after claimer withdraws the already vested amount, it revokes only the right to vest the remaining part in future.
- if called, say half-way the total period, right before the claimer withdraws the already vested amount, it revokes both the already vested amount and the right to vest the remaining part in future.

Raising as high impact because it actually causes:

- loss of already vested amounts of a user with a valid claim that has already righteously vested a part but not withdrawn
- different outcomes depending on the order in which withdraw and revokeClaim functions are called which means that one of the two behavoiurs is certainly in conflict with the other causing a loss on one of the two sides, contract or claimer (by definition of Vesting rights, the claimer).
- lack of trust by the potential claimers/users whch can be at any time deprived of righteously vested amounts.

Proof of Concept

The following two tests prove the behaviour difference when the order by which revokeClaim vs withdraw are called, whch shows that the vesting right is not guaranteed.

```
// NOTE: USES ORIGINAL REVOKE BEHAVIOUR
it('sample revoke use case USER LOSE: employee withdra
  const {tokenContract, vestingContract} = await creat

  const startTimestamp = await getLastBlockTs() + 100;
  const endTimestamp = startTimestamp + 2000;
  const terminationTimestamp = startTimestamp + 1000 +
  const releaseIntervalSecs = 100;

  await vestingContract.createClaim(owner2.address, st

  // move clock to termination timestamp (half-way the
  await ethers.provider.send("evm_mine", [terminationT

  let availableAmt = await vestingContract.claimableAm
  // revoke the claim preserving the "already vested bu
  await (await vestingContract.revokeClaim(owner2.addr

  let userBalanceBefore = await tokenContract.balanceO
  await expect(vestingContract.connect(owner2).withdra
  let userBalanceAfter = await tokenContract.balanceOf

  // move the clock to the programmed end of vesting pe
  await ethers.provider.send("evm_mine", [endTimestamp

  // cliffTimestamp < startTimestamp < terminationTime
  let expectedVestedAmount = cliffAmount.add(linearVes

  // RESIGNING EMPLOYEE LOSES HIS VESTED AMOUNT BECAUSE
  expect(userBalanceAfter.sub(userBalanceBefore)).to.be
  // VTVLVesting CONTRACT TOOK ALREADY VESTED AMOUNT FR
  expect(await vestingContract.finalClaimableAmount(ow
});

// NOTE: USES ORIGINAL REVOKE BEHAVIOUR
it('sample revoke use case USER WIN: employee withdraw
  const {tokenContract, vestingContract} = await creat
```

```
    const startTimestamp = await getLastBlockTs() + 100;
    const endTimestamp = startTimestamp + 2000;
    const terminationTimestamp = startTimestamp + 1000 +
    const releaseIntervalSecs = 100;

    await vestingContract.createClaim(owner2.address, sta

    // move clock to termination timestamp (half-way the
    await ethers.provider.send("evm_mine", [terminationT

    let userBalanceBefore = await tokenContract.balanceO
    await (await vestingContract.connect(owner2).withdra
    let userBalanceAfter = await tokenContract.balanceOf

    // revoke the claim preserving the "already vested bu
    await (await vestingContract.revokeClaim(owner2.addr

    // move the clock to the programmed end of vesting pe
    await ethers.provider.send("evm_mine", [endTimestamp

    console.log(userBalanceAfter.sub(userBalanceBefore))
    // RESIGNING EMPLOYEE RECEIVES HIS VESTED AMOUNT BY W
    expect(userBalanceAfter.sub(userBalanceBefore)).to.be
    expect(await vestingContract.finalClaimableAmount(own
});
```

Recommended Mitigation Steps

Below are, in order, a test and a diff/patch for a proposed fix. The proposed fix is just an idea at how to fix, or in other words, a way to preserve the already vested amount when claim is revoked.

The diff/patch add a deactivationTimestamp to claim, and a new revokeClaimProper that shall replace the revokeClaim function to correct the behaviour. The deactivationTimestamp is used to track the deactivation time for the claim in order to preserve the amount vested so far and allow the user to withdraw the amount righteously earned so far. The

_baseVestedAmount and hasActiveClaim have been updated to do proper math when isActive is false but deactivationTimestamp is greater than 0.

The finalVestedAmount has been update to show the “what would be” amount if the vesting would have reached the claim endTimestamp while the finalClaimableAmount takes into consideration the deactivationTimestamp if the claim has been revoked.

The test shows that the already vested amount (cliff + half way linear vesting) is preserved.

```
diff --git a/contracts/VTVLVesting.sol b/contracts/VTVLVe
index 133f19f..7ab955c 100644
--- a/contracts/VTVLVesting.sol
+++ b/contracts/VTVLVesting.sol
@@ -34,6 +34,7 @@ contract VTVLVesting is Context, Acces
         // Gives us a range from 1 Jan 1970 (Unix epoch)
         uint40 startTimestamp; // When does the vesting
         uint40 endTimestamp; // When does the vesting en
+        uint40 deactivationTimestamp;
         uint40 cliffReleaseTimestamp; // At which times
         uint40 releaseIntervalSecs; // Every how many se

@@ -108,7 +109,7 @@ contract VTVLVesting is Context, Acce

         // We however still need the active check, since
         // we want to only allow active claims
-        require(_claim.isActive == true, "NO_ACTIVE_CLA
+        require(_claim.isActive == true || _claim.deact

         // Save gas, omit further checks
         // require(_claim.linearVestAmount + _claim.cli
@@ -144,20 +145,20 @@ contract VTVLVesting is Context, A
     @param _claim The claim in question
     @param _referenceTs Timestamp for which we're calcu
      */
-    function _baseVestedAmount(Claim memory _claim, uin
+    function _baseVestedAmount(Claim memory _claim, uin
         uint112 vestAmt = 0;
-
```

```
-         // the condition to have anything vested is to h
-         if(_claim.isActive) {
+
+         if(_claim.isActive || _claim.deactivationTimesta
              // no point of looking past the endTimestamp
              // So if we're past the end, just get the re
-             if(_referenceTs > _claim.endTimestamp) {
-                 _referenceTs = _claim.endTimestamp;
+             if(_referenceTs > vestEndTimestamp) {
+                 _referenceTs = vestEndTimestamp;
              }

              // If we're past the cliffReleaseTimestamp,
              // We don't check here that cliffReleaseTime
-             if(_referenceTs >= _claim.cliffReleaseTimest
+             if(_referenceTs >= _claim.cliffReleaseTimest
+                 // @audit NOTE: (cliffReleaseTimestamp :
                  vestAmt += _claim.cliffAmount;
              }

@@ -195,7 +196,8 @@ contract VTVLVesting is Context, Acce
      */
     function vestedAmount(address _recipient, uint40 _re
         Claim storage _claim = claims[_recipient];
-         return _baseVestedAmount(_claim, _referenceTs);
+         uint40 vestEndTimestamp = _claim.isActive ? _cla
+         return _baseVestedAmount(_claim, _referenceTs, v
     }

     /**
@@ -205,7 +207,18 @@ contract VTVLVesting is Context, Acc
      */
     function finalVestedAmount(address _recipient) publi
         Claim storage _claim = claims[_recipient];
-         return _baseVestedAmount(_claim, _claim.endTimes
+         return _baseVestedAmount(_claim, _claim.endTimes
+     }
+
+     /**
+     @notice Calculates how much wil be possible to clai
+             amount from the vestedAmount at this moment.
+     @param _recipient - The address for whom we're calcu
+     */
```

```
+    function finalClaimableAmount(address _recipient) ex
+        Claim storage _claim = claims[_recipient];
+        uint40 vestEndTimestamp = _claim.isActive ? _cla
+        return _baseVestedAmount(_claim, vestEndTimestar
     }

     /**
@@ -214,7 +227,8 @@ contract VTVLVesting is Context, Acce
     */
     function claimableAmount(address _recipient) externa
         Claim storage _claim = claims[_recipient];
-        return _baseVestedAmount(_claim, uint40(block.t
+        uint40 vestEndTimestamp = _claim.isActive ? _cla
+        return _baseVestedAmount(_claim, uint40(block.t
     }

     /**
@@ -280,6 +294,7 @@ contract VTVLVesting is Context, Acce
         Claim memory _claim = Claim({
             startTimestamp: _startTimestamp,
             endTimestamp: _endTimestamp,
+            deactivationTimestamp: 0,
             cliffReleaseTimestamp: _cliffReleaseTimestar
             releaseIntervalSecs: _releaseIntervalSecs,
             cliffAmount: _cliffAmount,
@@ -436,6 +451,30 @@ contract VTVLVesting is Context, Ac
         emit ClaimRevoked(_recipient, amountRemaining, u
     }

+    function revokeClaimProper(address _recipient) exte
+        // Fetch the claim
+        Claim storage _claim = claims[_recipient];
+        // Calculate what the claim should finally vest
+        uint112 finalVestAmt = finalVestedAmount(_recip
+
+        // No point in revoking something that has been
+        // so require that there be unconsumed amount
+        require( _claim.amountWithdrawn < finalVestAmt,
+
+        _claim.isActive = false;
+        _claim.deactivationTimestamp = uint40(block.time
+
+        uint112 vestedSoFarAmt = vestedAmount(_recipien
```

```
+        // The amount that is "reclaimed" is equal to tl
+        // vested without the part that was already witl
+        uint112 amountRemaining = finalVestAmt - (veste
+
+        numTokensReservedForVesting -= amountRemaining;
+
+        // Tell everyone a claim has been revoked.
+        emit ClaimRevoked(_recipient, amountRemaining, u
+    }
+
     /**
     @notice Withdraw a token which isn't controlled by t
     @dev This contract controls/vests token at "tokenAdd
```

lawrencehui (VTVL) confirmed and commented:

> Thank you warden for the findings. We did think about adding a grace withdrawing period to further strengthen the users' trust to the admin. I would argue that severity is medium in the case as in practical sense we would assume admin will inform the receivers upon revocation and therefore withdrawAdmin was designed to be separated from revokeClaim.

> I acknowledge that some malicious admin might abuse this right and to claimed the receiver's already earned token before they claimed (as described in the scenario in this findings) and therefore we will consider adding the grace period to restrict admin to act maliciously.

Oxean (judge) commented:

> I am going to stick with High, even a non malicious admin would have no choice but to kindly ask a user to claim before they revoked all their other tokens. If the user didn't comply, the admin has no option but to either "steal" their tokens or allow them to keep vesting.

[H-02] Permanent freeze of vested tokens due to overflow in `_baseVestedAmount`

Submitted by Trust, also found by 0xSky, bin2chen, CertoraInc, hansfriese, KIntern_NA, neko_nyaa, neumo, rokinot, and wastewa

[VTVLVesting.sol#L176]

The _baseVestedAmount() function calculates vested amount for some (claim, timestamp) pair. It is wrapped by several functions, like vestedAmount, which is used in withdraw() to calculate how much a user can retrieve from their claim. Therefore, it is critical that this function will calculate correctly for users to receive their funds.

Below is the calculation of the linear vest amount:

```
    uint112 linearVestAmount = _claim.linearVestAmount * trui
```

Importantly, _claim.linearVestAmount is of type uint112 and truncatedCurrentVestingDurationSecs is of type uint40. Using compiler >= 0.8.0, the product cannot exceed uint112 or else the function reverts due to overflow. In fact, we can show that uint112 is an inadequate size for this calculation.

The max value for uint112 is 5192296858534827628530496329220096.
Seconds in year = 3600 * 24 * 365 = 31536000
Tokens that inherit from ERC20 like the ones used in VTVL have 18 decimal places -> 10000000000000000
This means the maximum number of tokens that are safe to vest for one year is 2**112 / 10e18 / (3600 * 24 * 365) = just 16,464,665 tokens. This is definitely not a very large amount and it is expected that some projects will mint a similar or larger amount for vesting for founders / early employees. For 4 year vesting, the safe amount drops to 4,116,166.
Projects that are not forewarned about this size limit are likely to suffer from

freeze of funds of employees, which will require very patchy manual revocation and restructuring of the vesting to not overflow.

Impact

Employees/founders do not have access to their vested tokens.

Proof of Concept

Below is a test that demonstrates the overflow issue, 1 year after 17,000,000 tokens have matured.

```
describe('Long vest fail', async () => {
  let vestingContract: VestingContractType;
  // Default params
  // linearly Vest 10000, every 1s, between TS 1000 and 7
  // additionally, cliff vests another 5000, at TS = 900
  const recipientAddress = await randomAddress();
  const startTimestamp = BigNumber.from(1000);
  const endTimestamp = BigNumber.from(1000 + 3600 * 24 *
  const midTimestamp = BigNumber.from(1000 + (3600 * 24 *
  const cliffReleaseTimestamp = BigNumber.from(0);
  const linearVestAmount = BigNumber.from('17000000000000
  const cliffAmount = BigNumber.from(0);
  const releaseIntervalSecs = BigNumber.from(5);

  before(async () => {
    const {vestingContract: _vc} = await createPrefundedV
    vestingContract = _vc;
    await vestingContract.createClaim(recipientAddress, s
  });

  it('half term works', async() => {
    expect(await vestingContract.vestedAmount(recipientAd
  });

  it('full term fails', async() => {
    // Note: at exactly the cliff time, linear vested amo
    await expect(vestingContract.vestedAmount(recipientAd
    );
  });
```

```
    });
```

Tools Used

Manual audit, hardhat / chai.

Recommended Mitigation Steps

Perform the intermediate calculation of linearVestAmount using the uint256 type.

```
uint112 linearVestAmount = uint112( uint256(_claim.linea
```

lawrencehui (VTVL) confirmed and commented:

> This finding is very useful and appreciate all wardens that flagged the potential risk of overflowing.

Medium Risk Findings (10)

[M-01] Supply cap of `VariableSupplyERC20Token` is not properly enforced

Submitted by Czar102, also found by __141345__, 0xbepresent, 0xDecorativePineapple, 0xmatt, 0xNazgul, 0xSky, adriro, ajtra, Atarpara, Bahurum, bin2chen, cccz, cRat1st0s, cryptonue, d3e4, DimitarDimitrov, Franfran, GimelSec, innertia, jag, JGcarv, JLevick, joestakey, Junnon, neumo, obront, OptimismSec, pashov, pauliax, pcarranzav, peanuts, rajatbeladiya, rbserver, reassor, Rolezn, Ruhum, seyni, Soosh, Tomo, Trust, wagmi, zzykxx, and zzzitron

VariableSupplyERC20Token.sol#L36-L46

The admin of the token is not constrained to minting `maxSupply_`, they can mint any number of tokens.

Proof of Concept

```
// If we're using maxSupply, we need to make sure we res
// mintableSupply = 0 means mint at will
if(mintableSupply > 0) {
        require(amount <= mintableSupply, "INVALID_AMOUN
        // We need to reduce the amount only if we're us
        mintableSupply -= amount;
}
```

The logic is as follows: if the amount that can be minted is zero, treat this as an infinite mint. Else require that the minted amount is not larger than mintable supply.

One can note that it is possible to mint all mintable supply. Then the mintable supply will be `0` which will be interpreted as infinity and any number of tokens will be possible to be minted.

Recommended Mitigation Steps

Treat `2 ** 256 - 1` as infinity instead of `0`.

Oxean (judge) decreased severity to Medium and commented:

> The warden's logic is correct, but given that this is behind an admin only flag, there are some external factors that would need to come into play for this to be realized. Downgrading to Medium severity.

lawrencehui (VTVL) confirmed

[M-02] `_baseVestedAmount()` and `vestedAmount()` Return Incorrect Historical Values

Submitted by Respx, also found by m9800

VTVLVesting.sol#L183-L187
VTVLVesting.sol#L198

As the comments in `_baseVestedAmount()` explain, once there is any `_claim.amountWithdrawn`, it will be returned if it is greater than the calculated value `vestAmt`. However, `vestAmt` takes account of time, `_referenceTs`, whereas `_claim.amountWithdrawn` is always the amount withdrawn to date. Therefore, for all historical values below `_claim.amountWithdrawn`, including timestamps before `_claim.startTimestamp` and before `_claim.cliffReleaseTimestamp`, `_claim.amountWithdrawn` will be returned.

Impact

Given that VTVL is intended to be an accessible platform for use by a wide variety of users, this behaviour does create a security risk. Consider these scenarios:

- A protocol relies on VTVL as an off-the-shelf solution for vesting, but builds other systems (escrow, NFT grants, access, airdrops) that work by checking the value of `vestedAmount()`. Airdrops are especially likely to be interested in historical values. These values would be distorted by how much users have claimed and so would result in an undesirable distribution of resources.
- Even if the above does not occur, consider that VTVL might be passed over as a vesting solution precisely because its historical data is inaccurate.
- A contract could be built that inherits from `VTVLVesting` and attempts to use `_baseVestedAmount()` (which is `internal` and so can be

used by inheriting contracts). The inheriting contract might apportion rewards based on historical usage.

- VTVL itself might wish to inherit from `VTVLVesting` in future.

Proof of Concept

```
diff --git a/test/VTVLVesting.ts b/test/VTVLVestingPOC.ts
index bb609fb..073e53f 100644
--- a/test/VTVLVesting.ts
+++ b/test/VTVLVestingPOC.ts
@@ -500,14 +500,37 @@ describe('Revoke Claim', async () =
   const recipientAddress = await randomAddress();
   const [owner, owner2] = await ethers.getSigners();

-  it('allows admin to revoke a valid claim', async () =>
+  it('POC: WITHDRAWN DATA IS UNRELIABLE', async () => {
     const {vestingContract} = await createPrefundedVest
-    await vestingContract.createClaim(recipientAddress,
+    const startTimestamp2 = startTimestamp.add(releaseI
+    const endTimestamp2 = endTimestamp.add(releaseInter
+    const cliffReleaseTimestamp2 = cliffReleaseTimestam
+    await vestingContract.createClaim(owner2.address, s
+
+    // Fast forward to middle of claim
+    const halfWay = startTimestamp2.toNumber() + (endTi
+    await ethers.provider.send("evm_mine", [halfWay]);
+
+    let vestAmt = await vestingContract.vestedAmount(ow
+    console.log("NO WITHDRAWAL, BEFORE VEST START: ",ve
+    vestAmt = await vestingContract.vestedAmount(owner2
+    console.log("NO WITHDRAWAL, AT VEST START: ",vestAm
+    vestAmt = await vestingContract.vestedAmount(owner2
+    console.log("NO WITHDRAWAL, HALF WAY THROUGH VEST: "
+    vestAmt = await vestingContract.vestedAmount(owner2
+    console.log("NO WITHDRAWAL, AT VEST END: ",vestAmt.
+
+    await (await vestingContract.connect(owner2).withdra

-    (await vestingContract.revokeClaim(recipientAddress
+    vestAmt = await vestingContract.vestedAmount(owner2
```

```
+    console.log("WITHDRAWAL, BEFORE VEST START: ",vestAr
+    vestAmt = await vestingContract.vestedAmount(owner2.
+    console.log("WITHDRAWAL, AT VEST START: ",vestAmt.to
+    vestAmt = await vestingContract.vestedAmount(owner2.
+    console.log("WITHDRAWAL, HALF WAY THROUGH VEST: ",ve
+    vestAmt = await vestingContract.vestedAmount(owner2.
+    console.log("WITHDRAWAL, AT VEST END: ",vestAmt.toSt

-    // Make sure it gets reverted
-    expect(await (await vestingContract.getClaim(recipie
  });

  it('prohibits a random user from revoking a valid cla
```

Recommended Mitigation Steps

For active claims, there is no reason to consider `_claim.amountWithdrawn`, as it will always have been below or equal to `vestAmt` at any point in time. So only consider `vestAmt` for inactive claims. For them, return the lowest of `vestAmt` and `_claim.amountWithdrawn`. This will keep the values monotonic with time without distorting the historical values. It will act as though `_claim.amountWithdrawn` was withdrawn and the claim was revoked in the block when `vestAmt` reached `_claim.amountWithdrawn`. That is a distortion,0xean but it is required to provide monotonicity.

Oxean (judge) commented:

> Good find.

lawrencehui (VTVL) confirmed

[M-03] Possible DoS on `vestingRecipients` due to lack of disposal mechanism

Submitted by fatherOfBlocks, also found by wagmi

VTVLVesting.sol#L224
VTVLVesting.sol#L245
VTVLVesting.sol#L302
VTVLVesting.sol#L317

When the smart contracts start to be used, the variable in storage vestingRecipients will start to be filled with addresses, as there is no mechanism to eliminate elements, this will cause the `allVestingRecipients()` function to generate a DoS yes has many addressess.

Recommended Mitigation Steps

In the `withdraw()` function you could remove the element from `vestingRecipients` that no longer has vesting. This would make the variable not grow without reducing elements.

Oxean (judge) commented:

> On the fence on this one. I agree with the warden, but in the current implementation `allVestingRecipients` is unused and assumed to be for external, off chain uses so the impact is hard to determine. Going to leave as Medium, pending sponsor review.

lawrencehui (VTVL) confirmed and commented:

> I would agree with the warden on the lack of control for an ever-growing array size could be an issue. I will tag this as an enhancement.

> On the side, I want to check what is the allowed max size of the array in this case? `2**256 -1` ? but theoretically calling a large array would exceed the the block gas limit when retrieving it?

Oxean (judge) commented:

> @lawrencehui - Yes retrieval will eventually fail, long before you populate the array fully. You could pass in an index range to retrieve portions of the array to avoid this failure mode.

> And yes `2**256 -1` is my understanding of the theoretical limit.

[M-04] not able to create claim

Submitted by rajatbeladiya, also found by 0x4non, ak1, berndartmueller, CertoraInc, Chom, imare, JLevick, joestakey, JohnSmith, KIntern_NA, obront, rbserver, rotcivegaf, Ruhum, RustyRabbit, and supernova

If admin revoked any recipient's claim, admin can not create claim for the same recipient because `startTimestamp` is not updated to initial value on revoke claim.
There will be a need to create a claim again for any reason like: 1) mistakenly revoked claim, 2) wrong info provided to claim, 3) new vesting period starts, etc.

Proof of Concept

1. Alice creates claim for Bob
2. Alice revokes claim of Bob
    - On `revokeClaim()`, claim's `isActive` will be false, but `startTimestamp` will remain as it is
    - VTVLVesting.sol#L418-L437
3. Alice tries to create claim for Bob but claim will not create because it has modifier `hasNoClaim()` which is checked for claim should not active and it checks for `require(_claim.startTimestamp == 0, "CLAIM_ALREADY_EXISTS");`
4. VTVLVesting.sol#L245-L253
5. VTVLVesting.sol#L123-L140

Recommended Mitigation Steps

Update `startTimestamp to 0` on `revokeClaim()`.

Oxean (judge) decreased severity to Low and commented:

> Downgrading to low severity. While true, why wouldn't the employee just use a different address? There is no residual benefit to using the old address (unless it was a smart contract, which the warden doesn't mention as part of their POC). The sponsor may want to fix this, since the fix is simple, but it poses very little risk and certainly no direct loss of funds.

Oxean (judge) increased severity to Medium and commented:

> Spent a bit more time thinking about this one and do think that it qualifies as Medium severity since it does affect the availability of the protocol in a number of ways. Going to go ahead and revise to Medium.

lawrencehui (VTVL) acknowledged, but disagreed with severity and commented:

> The vesting contract is designed to be created and used in a one-off manner and the revoke function is to prevent any mistakes made upon creation (wrong address / amount / timestamp etc.). In practical sense, if a claim (or the recipient address) is revoked, one (the admin) can always create a new vesting contract with the correct claim parameters.

> I therefore think that it is by design that the address is only able be claimable once per vesting contract, in all circumstance, the admin can re-create a new vesting contract to mitigate this issue and therefore this is a low risk / non-critical issue.

Oxean (judge) commented:

> I don't think the tactic of deploying a new contract is the correct one here simply to be able to set up vesting for one botched person or someone whose vesting token amount changes for example. I am going to stick with the Medium severity on this one, but do appreciate the response and thoughts on possible mitigations.

ak1 (warden) commented:

> I have explained one of the real use case scenarios where this protocol will fail to serve many. Refer to issue 384.
> It is not always contract address or EOA which will decide the identity of a person. Each one will have unique ID. That id is going to be used in all the places.

[M-05] Tokens with lower number of decimals can result in postponed linear vesting for user

Submitted by pashov

VTVLVesting.sol#L174

In the `_baseVestedAmount` of `VTVLVesting.sol` we see the following code

```
uint40 finalVestingDurationSecs = _claim.endTimestamp - _
uint112 linearVestAmount = _claim.linearVestAmount * tru
```

Let's look at `truncatedCurrentVestingDurationSecs` as just the duration passed from the start of the vesting period for the PoC (this doesn't omit important data in this context).

Now think of the following scenario:

We have a token `$TKN` that has 6 decimals (those are the decimals of both USDT & USDC). We want to distribute 10,000 of those tokens to a user vested over a 10 year period.

10 years in seconds is 315360000 ****(this is `finalVestingDurationSecs`)

This means that we will distribute 10,000 * 10^6 = 10 000 000 000 fractions of a token for 315360000 seconds, meaning we will distribute 310 fractions of a token each second - this is `linearVestAmount`

Now, since `finalVestingDurationSecs` is so big (315360000) it will almost always round `linearVestAmount` to zero when dividing by it, up until

`_claim.linearVestAmount * truncatedCurrentVestingDurationSecs` becomes a bigger number than 315360000, but since `_claim.linearVestAmount` is 310 we will need the current vesting duration to be at least 1 017 290 seconds which is **12** days postponed vesting. 12 days in a moving market can make a big difference if the user was expecting the tokens to start vesting from the first day.

Impact

Unexpected postponing of vesting can result in waiting times for users to receive their must-be-vested tokens. This period can be used by other token holders to dump the token and decrease the price of it, resulting in a loss of capital for the vesting receiver.

Recommended Mitigation Steps

Enforce the contract to work with only 18 decimal tokens with a `require` statement in the constructor.

Oxean (judge) decreased severity to Medium and commented:

> Downgrading to Medium, there are a lot of external factors presented here by the warden to line up to a loss of funds.

lawrencehui (VTVL) acknowledged, but disagreed with severity and commented:

> I acknowledge the warden's concern of the rounding, but I think the result of loss of funds is one of the extreme edge cases. I would suggest instead of restricting only to 18 decimal tokens (which is impractical as we would also want to include USDC and USDT for vesting too!), I would implement the rounding checking in the frontend UI and prompt user of potential delay caused by rounding / truncation as described in this issue.

0xean (judge) commented:

> Given that smart contracts can be interacted with in any number of ways (etherscan, programmatically, etc), I don't think the mitigation negates the risk entirely and am going to stick with the Medium severity here. The wardens demonstrates clearly the way in which this can happen. While it may be a bit outside of the normal vesting schedule expected, I do think it's valuable to understand the bounds of the math you have employed here.

[M-06] Variable balance token causing fund lock and loss

Submitted by __141345__, also found by 0xDecorativePineapple, CertoraInc, djxploit, hyh, lllllll, JohnSmith, MiloTruck, rbserver, and zzzitron

VTVLVesting.sol#L295
VTVLVesting.sol#L388

Some ERC20 token's balance could change, one example is stETH. The balance could become insufficient at the time of `withdraw()`. User's fund will be locked due to DoS. The way to take the fund out is to send more

token into the contract, causing fund loss to the protocol. And there is no guarantee that until the end time the balance would stay above the needed amount, the lock and loss issue persist.

Proof of Concept

For stETH like tokens, the `balanceOf()` value might go up or down, even without transfer.

```
// stETH
    function balanceOf(address who) external override vi
        return _shareBalances[who].div(_sharesPerToken);
    }
```

In `VTVLVesting`, the `require` check for the spot `balanceOf()` value will pass, but it is possible that as time goes on, the value become smaller and fail the transfer. As a result, the `withdraw()` call will revert, causing DoS, and lock user's fund.

```
// contracts/VTVLVesting.sol
    function _createClaimUnchecked() private  hasNoClaim
        // ...
        require(tokenAddress.balanceOf(address(this)) >=
        // ...
    }

    function withdraw() hasActiveClaim(_msgSender()) ext
        // ...
        tokenAddress.safeTransfer(_msgSender(), amountRe
        // ...
    }
```

Reference

https://etherscan.io/address/0x312ca0592a39a5fa5c87bb4f1da7b77544a91b87#code

Recommended Mitigation Steps

Disallow such kind of variable balance token.

Oxean (judge) decreased severity to Medium and commented:

> stETH only rebases up, not down. So that is a poor example.
>
> The sponsor's README does say they will support any ERC20 token, so that could include Fee on Transfer or downward rebasing tokens which could lead to less tokens in the contract than expected and transfers to revert due to balances being lower than expected.
>
> Downgrading to Medium as the external requirement is using these contracts on tokens that are known to have variable supply.

lawrencehui (VTVL) confirmed and commented:

> Thanks for reporting this and I will add this as a feature enhancement to cater / avoid for tokens with rebasing supplies.
>
> Question: Is there a straight forward way to detect rebasing tokens? Or on the flip side, restricting erc20 tokens that do not exhibit rebasing behaviour?

Oxean (judge) commented:

> @lawrencehui - Great question. There isn't a great way to detect this functionality in any generic manner unfortunately.
>
> Most contracts that want to handle FOT tokens will do something like (in pseudocode)

```
uint256 balBefore = ERC20.balanceOf(address(this));
ERC20.transferFrom(...);
uint256 balAfter= ERC20.balanceOf(address(this));
```

```
uint256 actualBalChange = balAfter - balBefore;
```

> Rebasing tokens are different again, and the easiest way to handle them is to create shares to track the internal math. The shares track the % ownership of the entire balance of the contract. Probably more than I can explain here, but would be happy to work with you if this is something you are interested in.

[M-07] Vesting Schedule Start and End Time can be Set in the Past

Submitted by TomJ, also found by ayeslick, csanuragjain, and pashov

[VTVLVesting.sol#L245-L304]

There is no check whether `_startTimestamp` and `_endTimestamp` is greater than `block.timestamp` at `VTVLVesting.sol` `_createClaimUnchecked` function. Therefore it is possible for administrators to accidentally create vesting schedule that starts and ends in the past without noticing it. When administrators does this and this transaction goes through, then the vesting recipients can withdraw their entire vest amount which is not what administrators intended to do. Add require check that force `_startTimestamp` to be greater than `block.timestamp`.

Team comments as below on line 260

```
// -> Conclusion: we want to allow this, for founders tha
transactions not going through because of discoordinatio
```

However this is not an issue by adding `require(_startTimestamp > uint40(block.timestamp))` since this will revert transaction if _startTimestamp is less than block.timestamp so administrators can simply try again with correct time. On the other hand, it is more dangerous to not

include this check because transaction will simply succeed even though _startTimestamp is set to past which means that there is a chance of administrators not noticing this.

Proof of Concept

1. Admin creates new vesting schedule using `createClaim` function. However admin mistakenly set `_startTimestamp` and `_endTimestamp` in the past.
2. Since there is no check of `require(_startTimestamp > uint40(block.timestamp))`, this transaction is valid and claim is created.
3. Vesting recipients calls the `withdraw` function and receive entire vest amount.

Recommended Mitigation Steps

Add following check in `VTVLVesting.sol:_createClaimUnchecked` function.

```
require(_startTimestamp > uint40(block.timestamp), "INVAI
```

Oxean (judge) commented:

> Going to use this issue for encompassing a few different reports that all revolve around adding some better validation around timestamps. These include a few different potential fixes that the sponsor can review, but ultimately point to the same underlying issues.

lawrencehui (VTVL) acknowledged, but disagreed with severity and commented:

> As described in the documentation, this back dated (`startTimestamp < block.timestamp`) feature is indeed intended as there are many real life

> cases that founders want to reward their employees in the way the vesting period starts well before Token Generation Event (TGE).

> We appreciate wardens' feedback on additional checking (both start and end time) and in our actual application, we would include multiple layer of checking / approval processes in front and backend before the transaction signing happens and therefore the risk is low in our opinion.

[M-08] Two address tokens can be withdrawn by the admin even if they are vested

Submitted by CertoraInc, also found by 0xhunter, datapunk, dipp, Lambda, and wuwe1

[VTVLVesting.sol#L446-L451]

Two address tokens exist in the blockchain. For example, Synthetix's `ProxyERC20` contract is such a token which exists in many forms (sUSD, sBTC...). Tokens as such can be vested, but the admin can withdraw them even if they are vested by providing the other address to the `withdrawOtherToken` function. The only check in this function is that `_otherTokenAddress != tokenAddress`, which is irrelevant in the case of two address tokens.

This can make the admin be able to withdraw the vested funds and break the system, because the balance of the contract can be less than the vested amount.

Proof of Concept

1. The `VTVLVesting` is deployed with the `sUSD` contract, using its main (proxy) address - `0x57Ab1ec28D129707052df4dF418D58a2D46d5f51`.
2. A claim is created for Alice, vesting 1000 sUSD in linear vesting. Assuming this is the only claim currently, the balance of the contract is

1000 sUSD and the value of `numTokensReservedForVesting` is `1000e18`.

3. The admin calls the `withdrawOtherToken` for 1000e18 sUSD, providing its second address - `0x57Ab1ec28D129707052df4dF418D58a2D46d5f51`. The value of `numTokensReservedForVesting` is still `1000e18`, but the balance of the contract is now 0 sUSD.
4. Alice waits for her vest to end, calls the withdraw function, but the function reverts on the call to `safeTransfer()` because there is insufficient balance of sUSD. Alice can't receive her funds.

Recommended Mitigation Steps

Replace the address check with a balance check - record the vesting token balance of the contract before and after the transfer and assert that they are equal.

Oxean (judge) decreased severity to Medium and commented:

> Downgrading to Medium. The fix is a good idea, but this is a pretty rare token implementation and definitely qualifies as an external factor.

lawrencehui (VTVL) acknowledged and commented:

> Yes, agreed with @0xean that this is very rare and appreciate warden's suggestion on the fix on balance checking.

[M-09] `_releaseIntervalSecs` is not validated

Submitted by sorrynotsorry

VTVLVesting.sol has `_createClaimUnchecked` function to create the claims internally while validating parameters with the users' allocations. However, `_releaseIntervalSecs` is not validated comparing to user's

`_linearVestAmount` and `_startTimestamp` `_endTimestamp`.
Theoretically, `_linearVestAmount` should be equal to `((_endTimestamp - _startTimestamp) * _releaseIntervalSecs)` so the `_releaseIntervalSecs = _linearVestAmount / ((_endTimestamp - _startTimestamp)`.
But this check was never done.

If the `_releaseIntervalSecs` is validated either to a higher or to a lower amount, it will create unfair distributions amongst the users during withdrawals due to being higher/lower than it should be. And also it may end up with the last withdrawals can be reverted due to the calculation board not matching.

Proof of Concept

```
        function _createClaimUnchecked(
                address _recipient,
                uint40 _startTimestamp,
                uint40 _endTimestamp,
                uint40 _cliffReleaseTimestamp,
                uint40 _releaseIntervalSecs,
                uint112 _linearVestAmount,
                uint112 _cliffAmount
                    ) private  hasNoClaim(_recipient) {


            require(_recipient != address(0), "INVALID_ADDRE
            require(_linearVestAmount + _cliffAmount > 0, "IN
            require(_startTimestamp > 0, "INVALID_START_TIME
            // Do we need to check whether _startTimestamp i
            // Or do we allow schedules that started in the
            // -> Conclusion: we want to allow this, for fou
            // require(_endTimestamp > 0, "_endTimestamp mus
            require(_startTimestamp < _endTimestamp, "INVALI
            require(_releaseIntervalSecs > 0, "INVALID_RELEA
            require((_endTimestamp - _startTimestamp) % _rel
```

```
// Potential TODO: sanity check, if _linearVestA

// No point in allowing cliff TS without the cli
// Both or neither of _cliffReleaseTimestamp and
require(
    (
        _cliffReleaseTimestamp > 0 &&
        _cliffAmount > 0 &&
        _cliffReleaseTimestamp <= _startTimestam
    ) || (
        _cliffReleaseTimestamp == 0 &&
        _cliffAmount == 0
), "INVALID_CLIFF");

Claim memory _claim = Claim({
    startTimestamp: _startTimestamp,
    endTimestamp: _endTimestamp,
    cliffReleaseTimestamp: _cliffReleaseTimestam
    releaseIntervalSecs: _releaseIntervalSecs,
    cliffAmount: _cliffAmount,
    linearVestAmount: _linearVestAmount,
    amountWithdrawn: 0,
    isActive: true
});
// Our total allocation is simply the full sum o
// Not necessary to use the more complex logic f
uint112 allocatedAmount = _cliffAmount + _linear

// Still no effects up to this point (and tokenA
require(tokenAddress.balanceOf(address(this)) >=

// Done with checks

// Effects limited to lines below
claims[_recipient] = _claim; // store the claim
numTokensReservedForVesting += allocatedAmount;
vestingRecipients.push(_recipient); // add the v
emit ClaimCreated(_recipient, _claim); // let ev
```

```
}
```

[Permalink]

Recommended Mitigation Steps

The `_releaseIntervalSecs` should be validated comparing to user's `_linearVestAmount` and `_startTimestamp` `_endTimestamp`.

[Oxean (judge) decreased severity to Medium and commented]:

> This is fair, but due to it being behind only admin functionality and coming down to input sanitization, going to downgrade to Medium.

[lawrencehui (VTVL) acknowledged, but disagreed with severity and commented]:

> I agree with @0xean that the risk in this case is low given the onlyAdmin modifier and the input will be validated from the frontend anyway. Appreciate the finding and we will take consideration of adding additional checking of `_releaseIntervalSecs`.

[M-10] Reentrancy may allow an admin to steal funds

Submitted by Czar102, also found by 0xSmartContract, csanuragjain, hansfriese, Lambda, Respx, and sashik_eth

[VTVLVesting.sol#L394-L411]

If the token is reentrant, an admin can steal all tokens locked in the `VTVLVesting` contract while having active locks.

In other words, due to this exploit possibility, the contract may be insolvent with respect to *active* vestings. Note that revoking claim doesn't break this

invariant since the vesting is closed in that case.

Proof of Concept

The reentrancy in the vested token can be used by an admin if the execution can be hijacked before the balance change occurs.

```
/**
@notice Admin withdrawal of the unallocated tokens.
@param _amountRequested - the amount that we want to witl
        */
function withdrawAdmin(uint112 _amountRequested) public {
        // Allow the owner to withdraw any balance not cu
        uint256 amountRemaining = tokenAddress.balanceOf(

        require(amountRemaining >= _amountRequested, "INS

        // Actually withdraw the tokens
        // Reentrancy note - this operation doesn't touch
        // Also following Checks-effects-interactions pat
        tokenAddress.safeTransfer(_msgSender(), _amountRe

        // Let the withdrawal known to everyone
        emit AdminWithdrawn(_msgSender(), _amountRequeste
}
```

Let's consider function `withdrawAdmin`. Firstly, the balance is checked and then if there is enough token surplus to withdraw, the withdrawal is allowed. The surplus is based on two values: `numTokensReservedForVesting` which isn't changed by this function and the balance of the contract.

If the owner hijacks the execution before the balance change in the token transfer (which is possible in, for example, ERC777), an admin can call this function again and it will allow for an invocation of another transfer since the token balance hasn't changed yet.

For example, if there is `$1m` in vestings in the contract, an admin can send `$100k` to it in tokens and recursively invoke `withdrawalAdmin` with the amount of `$100k` eleven times so that the whole contract balance will be drained.

Recommended Mitigation Steps

Add `ReentrancyGuard`'s `nonReentrant` to the `withdrawAdmin` function.

Oxean (judge) commented:

> This would require a number of assumptions to be the case including a malicious admin which the sponsors called out of scope. Because it is obviously not intended functionality, I am going to leave as Medium pending sponsor review. I think the non-reentrant modifier is worth adding.

lawrencehui (VTVL) acknowledged and commented:

> Will consider adding ReentrancyGuard as suggested.

Low Risk and Non-Critical Issues

For this contest, 135 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **AkshaySrivastav** received the top score from the judge.

The following wardens also submitted reports: rbserver, IIIIIII, 0xNazgul, 0x1f8b, lukris02, rotcivegaf, ajtra, cryptostellar5, 0xSmartContract, Deivitto, Bahurum, brgltd, Diana, c3phas, chatch, ladboy233, supernova, RockingMiles, Waze, tnevler, a12jmx, Funen, pcarranzav, 0x4non, KIntern_NA, __141345__, delfin454000, 0x5rings, Aymen0909, ubermensch, Rolezn, leosathya, 0xf15ers, gogo, CodingNameKiki, V_B, aysha, seyni, Dravee, ChristianKuri, JLevick, CertoraInc, JohnnyTime, BnkeOx0, Lambda, Respx, RaymondFam, rajatbeladiya, ikbkln, neumo,

TomJ, 0xA5DF, 0xSky, Aeros, sorrynotsorry, async, prasantgupta52, 0xDecorativePineapple, rvierdiiev, sach1r0, ElKu, slowmoses, neko_nyaa, Tomo, bin2chen, innertia, yongskiws, ignacio, djxploit, JohnSmith, got_targ, joestakey, csanuragjain, rokinot, cryptphi, ayeslick, romand, peanuts, RustyRabbit, 0xbepresent, hansfriese, Chom, berndartmueller, dic0de, peritoflores, zzzitron, cccz, obront, reassor, bobirichman, sikorico, Margaret, datapunk, karanctf, fatherOfBlocks, 0xmatt, nalus, eighty, ret2basic, Ruhum, Sm4rty, Rohan16, pedr02b2, ReyAdmirado, indijanc, SooYa, pedroais, d3e4, ak1, zzykxx, erictee, oyc_109, ch13fd357r0y3r, millersplanet, martin, 2997ms, B2, tibthecat, OptimismSec, exd0tpy, medikko, peiw, JC, StevenL, durianSausage, 0v3rf10w, 0x040, natzuu, Yiko, carrotsmuggler, 0x85102, MasterCookie, bulej93, *and* Diraco.

[01]

The `setAdmin()` function in AccessProtected.sol can be used to revoke all admins. This could be a feature to completely renounce ownership of the contract after all claims are set or could be a bug in which one admin either intentionally or unintentionally removes all admin (or all other admins except himself).

[02]

Line 161 in `VTVLVesting._baseVestedAmount()` function should not get executed when `cliffAmount` is 0. In the case of no cliff amount, i.e. where `cliffReleaseTimestamp` and `cliffAmount` are both set as 0, the program execution should not enter the `if` block.

```
160     if(_referenceTs >= _claim.cliffReleaseTimestamp) {
161         vestAmt += _claim.cliffAmount;
162     }
```

[03]

Solidity pragma versioning should be upgraded to latest available version. Currently the solidity version in contracts is ^0.8.14 which was found to possess some bugs.

[04]

Solidity pragma versioning should be exactly same in all contracts. Currently some contracts use ^0.8.14 but some are fixed to 0.8.14.

[05]

No need to re-inherit Context contract in **VTVLVesting** smart contract as Context is already inherited by **AccessProtected** contract.

[06]

Ownable smart contract is unnecessarily imported in **AccessProtected.sol** while it is never used. Unnecessary imports decreases the readability of smart contract code.

[07]

Unnecessary imports are also present in **VTVLVesting.sol**. The compilation works completely fine with just importing SafeERC20.sol and AccessProtected.sol.

[08]

AccessProtected - contract docs do not match implementaion. The implementation only has multiple equal rights admins and no `owner` field is present while the docs states something else.

```
7   /**
8       @title Access Limiter to multiple owner-specifi
9       @dev Exposes the onlyAdmin modifier, which will
```

```
10    */
```

[09]

`VariableSupplyERC20Token.constructor()` has an empty `@dev` tag.

[10]

VariableSupplyERC20Token contract mentions an incorrect comment

```
48    // We can't really have burn, because that could ma
49    // Example: if the user can burn tokens already as
```

Token can be made burnable in which users can be allowed to burn their own tokens.

[11]

Line 159 in `VTVLVesting._baseVestedAmount()` contains a misleading comment

```
159         // We don't check here that cliffReleaseTimes
160         if(_referenceTs >= _claim.cliffReleaseTimestar
161             vestAmt += _claim.cliffAmount;
162         }
```

`cliffReleaseTimestamp` can never be after `startTimestamp` as per the `require` statements of `_createClaimUnchecked()`.

[12]

As per the implementation of vesting contract, Line 21 in VTVLVesting.sol should mention *greater than or equal* instead of just *greater than*.

```
21    /// @dev Our balance of the token must always be g
```

[13]

`VTVLVesting.ClaimCreated` and `VTVLVesting.ClaimRevoked` events should also log the admin's address so it can be easily queried which admin created and revoked the claim.

[14]

In VTVLVesting contract, before revoking a claim the contract should transfer all the pending/partially vested rewards. Otherwise the entire vesting amount will get revoked.

```
It is at the discretion of protocol development team to
```

[15]

At Line 82 of `VTVLVesting.constructor()`, a better check would be to do `_tokenAddress.totalSupply()`. As this will also ensure that the input address in indeed a token's address and perform the zero address check as well.

```
82    require(address(_tokenAddress) != address(0), "INV
```

[16]

The `tokenAddress` state variable of VTVLVesting should be renamed to `token` as this variable represents an `IERC20` interface rather that just an address. Renaming it to `token` aligns better with its usage.

```
17    IERC20 public immutable tokenAddress;
```

[17]

There should be a factory contract for VTVLVesting contract which can keep track of all vesting contracts deployed by different founders. The Factory contract aligns better with the business usecase of VTVL protocol owners.

```
From the spec, "The core function of VTVL is to allow use
```

[18]

In `VariableSupplyERC20Token.mint()` function, non-zero input validation check should be done similar to `FullPremintERC20Token.constructor()`.

[19]

In all solidity files, license keyword should be mentioned as `// SPDX-License-Identifier: UNLICENSED`.

[20]

All the actors interacting with a VTVLVesting contract need to fully trust all of its admins. Any one of the potentially infinite admins of VTVLVesting contract has the power to (either intentionally or unintentionally): * revoke claims of all recipients and withdraw all tokens, resulting in a rugpull attack.

* give or take back the admin rights to or from any ethereum address. * withdraw any other ERC20 token from the vesting contract.

Gas Optimizations

For this contest, 141 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: Aymen0909, pfapostol, c3phas, JLevick, Deivitto, gogo, CertoraInc, JC, Bnke0x0, oyc_109, durianSausage, __141345__, rotcivegaf, 0xSmartContract, ajtra, Sm4rty, cryptostellar5, Diana, JohnSmith, Tomo, zishansami, martin, ch0bu, SnowMan, prasantgupta52, erictee, millersplanet, djxploit, Rohan16, 0x1f8b, RaymondFam, 0x4non, rbserver, Rolezn, TomJ, brgltd, 0xNazgul, Saintcode_, karanctf, medikko, ret2basic, 0xsam, ReyAdmirado, seyni, gianganhnguyen, Ruhum, carrotsmuggler, slowmoses, WilliamAmbrozic, B2, peiw, 0x040, leosathya, delfin454000, Tomio, samruna, lukris02, aysha, yaemsobak, Junnon, imare, eighty, 0xA5DF, ladboy233, emrekocak, tnevler, pauliax, ikbkln, neko_nyaa, jag, Tadashi, Atarpara, tgolding55, 0xbepresent, Ocean_Sky, peanuts, caventa, RockingMiles, supernova, SooYa, beardofginger, natzuu, pedroais, bobirichman, dharma09, DimitarDimitrov, sach1r0, Waze, ignacio, async, tibthecat, OptimismSec, AkshaySrivastav, malinariy, lucacez, ChristianKuri, Chom, Funen, d3e4, subtle77, fatherOfBlocks, 0xDanielC, indijanc, ak1, got_targ, mics, Lambda, KIntern_NA, w0Lfrum, hxzy, Amithuddar, V_B, Tagir2003, 0xc0ffEE, Respx, MasterCookie, Satyam_Sharma, Noah3o6, rokinot, nalus, jpserrat, CodingNameKiki, Matin, rvierdiiev, adriro, StevenL, bulej93, 2997ms, Diraco, csanuragjain, Sta1400, 0v3rf10w, 0x85102, mrpathfindr, exd0tpy, cryptphi, a12jmx, francoHacker, m9800, *and* Yiko.

Summary

	Issue	Instances	Total Gas Saved
[G-01]	Save gas by not requring non-zero interval if no linear amount	1	17100
[G-02]	Results of calls to `_msgSender()` not cached	4	64
[G-03]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	7	840
[G-04]	State variables should be cached in stack variables rather than re-reading them from storage	1	97
[G-05]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	4	452
[G-06]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	4	340
[G-07]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	1	60
[G-08]	Optimize names to save gas	3	66
[G-09]	Using `bool` s for storage incurs overhead	1	20000
[G-10]	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	1	10
[G-11]	Splitting `require()` statements that use `&&` saves gas	1	3
[G-12]	Don't compare boolean expressions to boolean literals	1	9
[G-13]	Use custom errors rather than `revert()` / `require()` strings to save gas	24	-
[G-14]	Functions guaranteed to revert when called by normal users can be marked `payable`	7	147
[G-15]	Don't use `_msgSender()` if not supporting EIP-2771	13	208

Total: 73 instances over 15 issues with **39396 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values.

[G-01] Save gas by not requring non-zero interval if no linear amount

If there is no linear amount, a Gsset for the claim's interval can be converted to a Gsreset, saving **17100 gas**.

There is 1 instance of this issue:

```
File: /contracts/VTVLVesting.sol

263           require(_releaseIntervalSecs > 0, "INVALID_I
264:          require((_endTimestamp - _startTimestamp) %
```

[VTVLVesting.sol#L263-L264]

[G-02] Results of calls to `_msgSender()` not cached

Saves at least **16 gas** per call skipped.

There are 4 instances of this issue:

```
File: /contracts/VTVLVesting.sol

371:          uint112 allowance = vestedAmount(_msgSender

388:          tokenAddress.safeTransfer(_msgSender(), amou

391:          emit Claimed(_msgSender(), amountRemaining)
```

```
410:         emit AdminWithdrawn(_msgSender(), _amountRe
```

VTVLVesting.sol#L371

[G-03] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one.

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved.

There are 7 instances of this issue:

```
File: contracts/VTVLVesting.sol

/// @audit _recipients
/// @audit _startTimestamps
/// @audit _endTimestamps
```

```
/// @audit _cliffReleaseTimestamps
/// @audit _releaseIntervalsSecs
/// @audit _linearVestAmounts
/// @audit _cliffAmounts
333       function createClaimsBatch(
334           address[] memory _recipients,
335           uint40[] memory _startTimestamps,
336           uint40[] memory _endTimestamps,
337           uint40[] memory _cliffReleaseTimestamps,
338           uint40[] memory _releaseIntervalsSecs,
339           uint112[] memory _linearVestAmounts,
340           uint112[] memory _cliffAmounts)
341:          external onlyAdmin {
```

VTVLVesting.sol#L333-L341

[G-04] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There is 1 instance of this issue:

```
File: contracts/token/VariableSupplyERC20Token.sol

/// @audit mintableSupply on line 40
41:               require(amount <= mintableSupply, "INV
```

VariableSupplyERC20Token.sol#L41

[G-05] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves [113 gas](#).

There are 4 instances of this issue:

```
File: contracts/token/VariableSupplyERC20Token.sol

43:               mintableSupply -= amount;
```

[VariableSupplyERC20Token.sol#L43](#)

```
File: contracts/VTVLVesting.sol

301:          numTokensReservedForVesting += allocatedAm

383:          numTokensReservedForVesting -= amountRemai

433:          numTokensReservedForVesting -= amountRemai
```

[VTVLVesting.sol#L301](#)

[G-06] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if`-statement

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

There are 4 instances of this issue:

```
File: contracts/VTVLVesting.sol
```

```
/// @audit require() on line 262
264:          require((_endTimestamp - _startTimestamp) %

/// @audit require() on line 374
377:          uint112 amountRemaining = allowance - usrC

/// @audit require() on line 426
429:          uint112 amountRemaining = finalVestAmt - _

/// @audit if-condition on line 166
167:                  uint40 currentVestingDurationSecs =
```

VTVLVesting.sol#L264

[G-07] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** per loop.

There is 1 instance of this issue:

```
File: contracts/VTVLVesting.sol

353:          for (uint256 i = 0; i < length; i++) {
```

VTVLVesting.sol#L353

[G-08] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works.

Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, per sorted position shifted.

There are 3 instances of this issue:

```
File: contracts/AccessProtected.sol

/// @audit isAdmin(), setAdmin()
11:   abstract contract AccessProtected is Context {
```

AccessProtected.sol#L11

```
File: contracts/token/VariableSupplyERC20Token.sol

/// @audit mint()
10:   contract VariableSupplyERC20Token is ERC20, AccessI
```

VariableSupplyERC20Token.sol#L10

```
File: contracts/VTVLVesting.sol

/// @audit getClaim(), vestedAmount(), finalVestedAmount
11:   contract VTVLVesting is Context, AccessProtected {
```

VTVLVesting.sol#L11

[G-09] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any ty
```

```
        // word because each write operation emits an extra S
        // slot's contents, replace the bits taken up by the
        // back. This is the compiler's defense against conti
        // pointer aliasing, and it cannot be disabled.
```

OpenZeppelin/ReentrancyGuard.sol#L25-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past.

There is 1 instance of this issue:

```
File: contracts/AccessProtected.sol

12:       mapping(address => bool) private _admins; // us
```

AccessProtected.sol#L12

[G-10] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop**.

There is 1 instance of this issue:

```
File: contracts/VTVLVesting.sol

353:           for (uint256 i = 0; i < length; i++) {
```

VTVLVesting.sol#L353

[G-11] Splitting `require()` statements that use `&&` saves gas

See [this issue] which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by **3 gas.**

There is 1 instance of this issue:

```
File: contracts/VTVLVesting.sol

344           require(_startTimestamps.length == length &
345                   _endTimestamps.length == length &&
346                   _cliffReleaseTimestamps.length ==
347                   _releaseIntervalsSecs.length == le
348                   _linearVestAmounts.length == lengt
349                   _cliffAmounts.length == length,
350                   "ARRAY_LENGTH_MISMATCH"
351:          );
```

[VTVLVesting.sol#L344-L351]

[G-12] Don't compare boolean expressions to boolean literals

`if (<x> == true) => if (<x>), if (<x> == false) => if (!<x>)`

There is 1 instance of this issue:

```
File: contracts/VTVLVesting.sol

111:          require(_claim.isActive == true, "NO_ACTIVI
```

[VTVLVesting.sol#L111]

[G-13] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 24 instances of this issue:

```
File: contracts/AccessProtected.sol

25:           require(_admins[_msgSender()], "ADMIN_ACCES

40:           require(admin != address(0), "INVALID_ADDRI
```

AccessProtected.sol#L25

```
File: contracts/token/FullPremintERC20Token.sol

11:           require(supply_ > 0, "NO_ZERO_MINT");
```

FullPremintERC20Token.sol#L11

```
File: contracts/token/VariableSupplyERC20Token.sol

27:           require(initialSupply_ > 0 || maxSupply_ >

37:           require(account != address(0), "INVALID_ADI

41:               require(amount <= mintableSupply, "INV/
```

VariableSupplyERC20Token.sol#L27

File: contracts/VTVLVesting.sol

```
82:           require(address(_tokenAddress) != address(0
107:          require(_claim.startTimestamp > 0, "NO_ACTI
111:          require(_claim.isActive == true, "NO_ACTIVI
129:          require(_claim.startTimestamp == 0, "CLAIM_
255:          require(_recipient != address(0), "INVALID_
256:          require(_linearVestAmount + _cliffAmount >
257:          require(_startTimestamp > 0, "INVALID_START
262:          require(_startTimestamp < _endTimestamp, "I
263:          require(_releaseIntervalSecs > 0, "INVALID_
264:          require((_endTimestamp - _startTimestamp) %

270           require(
271               (
272                   _cliffReleaseTimestamp > 0 &&
273                   _cliffAmount > 0 &&
274                   _cliffReleaseTimestamp <= _startTi
275               ) || (
276                   _cliffReleaseTimestamp == 0 &&
277                   _cliffAmount == 0
278:          ), "INVALID_CLIFF");

295:          require(tokenAddress.balanceOf(address(thi

344           require(_startTimestamps.length == length &
345                   _endTimestamps.length == length &&
346                   _cliffReleaseTimestamps.length ==
347                   _releaseIntervalsSecs.length == le
348                   _linearVestAmounts.length == lengt
349                   _cliffAmounts.length == length,
350                   "ARRAY_LENGTH_MISMATCH"
351:          );
```

```
374:          require(allowance > usrClaim.amountWithdra
402:          require(amountRemaining >= _amountRequeste
426:          require( _claim.amountWithdrawn < finalVes
447:          require(_otherTokenAddress != tokenAddress
449:          require(bal > 0, "INSUFFICIENT_BALANCE");
```

VTVLVesting.sol#L82

[G-14] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are
`CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 7 instances of this issue:

```
File: contracts/AccessProtected.sol
39:       function setAdmin(address admin, bool isEnable
```

AccessProtected.sol#L39

```
File: contracts/token/VariableSupplyERC20Token.sol
```

```
36:       function mint(address account, uint256 amount)
```

VariableSupplyERC20Token.sol#L36

```
File: contracts/VTVLVesting.sol

317       function createClaim(
318               address _recipient,
319               uint40 _startTimestamp,
320               uint40 _endTimestamp,
321               uint40 _cliffReleaseTimestamp,
322               uint40 _releaseIntervalSecs,
323               uint112 _linearVestAmount,
324               uint112 _cliffAmount
325:                  ) external onlyAdmin {

333       function createClaimsBatch(
334           address[] memory _recipients,
335           uint40[] memory _startTimestamps,
336           uint40[] memory _endTimestamps,
337           uint40[] memory _cliffReleaseTimestamps,
338           uint40[] memory _releaseIntervalsSecs,
339           uint112[] memory _linearVestAmounts,
340           uint112[] memory _cliffAmounts)
341:          external onlyAdmin {

398:      function withdrawAdmin(uint112 _amountRequeste

418:      function revokeClaim(address _recipient) exter

446:      function withdrawOtherToken(IERC20 _otherToken
```

VTVLVesting.sol#L317-L325

[G-15] Don't use `_msgSender()` if not supporting EIP-2771

Use `msg.sender` if the code does not implement [EIP-2771 trusted forwarder](#) support.

There are 13 instances of this issue:

```
File: contracts/AccessProtected.sol
17:             _admins[_msgSender()] = true;
18:             emit AdminAccessSet(_msgSender(), true);
25:             require(_admins[_msgSender()], "ADMIN_ACCE
```

[AccessProtected.sol#L17]

```
File: contracts/token/FullPremintERC20Token.sol
12:             _mint(_msgSender(), supply_);
```

[FullPremintERC20Token.sol#L12]

```
File: contracts/token/VariableSupplyERC20Token.sol
32:                 mint(_msgSender(), initialSupply_);
```

[VariableSupplyERC20Token.sol#L32]

```
File: contracts/VTVLVesting.sol
367:             Claim storage usrClaim = claims[_msgSender
371:             uint112 allowance = vestedAmount(_msgSende
388:             tokenAddress.safeTransfer(_msgSender(), am
```

```
391:          emit Claimed(_msgSender(), amountRemaining
364:      function withdraw() hasActiveClaim(_msgSender(
407:          tokenAddress.safeTransfer(_msgSender(), _ar
410:          emit AdminWithdrawn(_msgSender(), _amountRe
450:          _otherTokenAddress.safeTransfer(_msgSender
```

[VTVLVesting.sol#L367]

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top