KLEVA Protocol Post-Mortem Analysis

Written by Theori
2022.01.27

Summary

Vault의 Token(KUSDT)과 Interest Model간 denominator 차이로 인해 Vault의 debt이 비정상적으로 커짐으로써 deposit 시 mint 오류가 발생하거나 withdraw 시 과도한 금액 출금 현상이 발생했습니다.

Background

Alpaca Finance, Compound와 같은 DeFi Lending 시스템은 Interest Bearing 모델 (예: ibToken, cToken)[1]을 통해 이용자의 이자를 관리합니다. 예치한 토큰의 이자는 이율 모델 컨트랙트에 의해 매 초 또는 매 블록마다 계산됩니다. 이율 모델 컨트랙트는 Utilization Level 마다 세 단계로 나눈 후, 대출이 많이 발생하면 이율을 높여 대출 상환을 장려하고, 대출이 적다면 이율을 낮춤으로써 대출을 장려하는 Triple Slope 모델[2]을 예로 들 수 있습니다.

Root Cause Analysis

KLEVA Protocol의 Lending 시스템이 fork한 Alpaca Finance[3]가 사용하는 매 초별 이율은 `1e18`을 denominator로 사용하고 있습니다. 따라서 해당 이율을 사용하는 곳에서는 아래의 식을 통해 마지막 갱신 시점 이후부터 지금까지의 이자를 계산할 수 있습니다. `1e18` denominator를 기준으로 매 초별 이율을 구했으니 마지막에 `1e18`로 나누어야 합니다.

$$debt\ *\ (interestRatePerSecond\ *\ delta)\ /\ 1e18$$

여기서, $debt = X * 1e18$. 즉, 임의의 X Token을 표현하면 $X * 1e18$이 됩니다.

Alpaca Finance가 사용하는 BSC의 스테이블 토큰은 모두 18 decimal을 사용하고 있습니다. 만약 10,000 토큰의 10%를 계산한다면 다음과 같이 계산할 수 있습니다:

---

[1] https://docs.alpacafinance.org/v/ko/tokenomics/ibtokens
[2] https://docs.kleva.io/protocol-details/parameters#interest-rate-model
[3] https://github.com/alpaca-finance/bsc-alpaca-contract/blob/c5491769be3d5481572e15db40d4831416055bc6/contracts/6/protocol/Vault.sol#L167-L178

$$(10\_000 * 1e18) * 0.1e18 / 1e18 = 1\_000 * 1e18$$

하지만, KLEVA에서 사용되는 Klaytn 체인의 스테이블 토큰은 (KUSDT) 6 decimal을 사용하고 있습니다. 만약 6 decimal을 사용하는 토큰에 18 decimal을 가정하는 이율 모델을 그대로 적용하면, 항상 0 또는 0에 매우 가까운 값이 반환됩니다.

$$(10\_000 * 1e6) * 0.1e18 / 1e18 \approx 0$$

이 때문에, KLEVA를 통해 KUSDT 대출을 해도 이율이 사실상 증가하지 않습니다. 아마도 KLEVA 측에서는 해당 현상을 인지하고 $interestRatePerSecond$의 denominator를 고려하지 않은 채 `pendingInterest` 에서 계산에 사용되는 `1e18` 값을 `1e6`로 바꾸면서 더 큰 문제로 이어졌습니다.

이로써 의도된 이율보다 1조(1,000,000,000,000) 배의 차이가 발생하게 되면서 0.00001 ibKUSDT를 상환했을 때 35,723,755.563132 KUSDT를 받게 됩니다.

$$(10\_000 * 1e6) * 0.1e18 / 1e6 \neq 0$$

다음은 KLEVA 측에서 패치하기 전과 패치한 후의 컨트랙트 코드를 디컴파일하여 비교한 것입니다.

```
function 0x1775(uint256 varg0) private {
    if (block.timestamp > _lastAccrueTime) {
        require(block.timestamp >= _lastAccrueTime, 17);
        v0 = 0xcc9();
        require(v0 >= varg0, 17);
        v1 = v0 - varg0;
        require(_setConfig.code.size);
        v2, v3 = _setConfig.staticcall(0xd4a99bbb, stor_44, v1).gas(msg.gas);
        require(v2); // checks call status, propagates error data on error
        require(MEM[64] + RETURNDATASIZE() - MEM[64] >= 32);
        require(!(v3 & (stor_44 > ~0 / v3)), 17);
        require(!(v3 * stor_44 & (block.timestamp - _lastAccrueTime > ~0 / (v3 * stor_44))), 17);
        require(0xde0b6b3a7640000, 18);
        return v3 * stor_44 * (block.timestamp - _lastAccrueTime) / 0xde0b6b3a7640000;
    } else {
}
```

1e18

```
function 0x17bb(uint256 varg0) private {
    if (block.timestamp > _lastAccrueTime) {
        require(block.timestamp >= _lastAccrueTime, 17);
        v0 = 0xce9();
        require(v0 >= varg0, 17);
        v1 = v0 - varg0;
        require(_setConfig.code.size);
        v2, v3 = _setConfig.staticcall(0xd4a99bbb, stor_44, v1).gas(msg.gas);
        require(v2); // checks call status, propagates error data on error
        require(MEM[64] + RETURNDATASIZE() - MEM[64] >= 32);
        require(!(v3 & (stor_44 > ~0 / v3)), 17);
        require(!(v3 * stor_44 & (block.timestamp - _lastAccrueTime > ~0 / (v3 * stor_44))), 17);
        require(0xf4240, 18);
        return v3 * stor_44 * (block.timestamp - _lastAccrueTime) / 0xf4240;
    } else {
        return 0;
    }
```

1e6

```
function deposit(uint256 amountToken)
  external
  payable
  override
  transferTokenToVault(amountToken)
  accrue(amountToken)
  nonReentrant
{
  _deposit(amountToken);
}

function _deposit(uint256 amountToken) internal {
  uint256 total = totalToken().sub(amountToken);
  uint256 share = total == 0 ? amountToken :
amountToken.mul(totalSupply()).div(total);
  _mint(msg.sender, share);
  require(totalSupply() > 1e17, "no tiny shares");
}
```

deposit 메소드의 accrue modifier 및 totalToken() 메소드 호출
total의 값이 비정상적으로 커져 _mint(msg.sender, 0); 상황 발생

```
function withdraw(uint256 share) external override accrue(0) nonReentrant {
  uint256 amount = share.mul(totalToken()).div(totalSupply());
  _burn(msg.sender, share);
  _safeUnwrap(msg.sender, amount);
  require(totalSupply() > 1e17, "no tiny shares");
}
```

withdraw 메소드의 accrue modifier 및 totalToken() 메소드 호출
작은 share로 큰 amount를 만들 수 있음

```
modifier accrue(uint256 value) {
  if (now > lastAccrueTime) {
    uint256 interest = pendingInterest(value);
    uint256 toReserve = interest.mul(config.getReservePoolBps()).div(10000);
    reservePool = reservePool.add(toReserve);
    vaultDebtVal = vaultDebtVal.add(interest);
    lastAccrueTime = now;
  }
  _;
}
```

accrue modifier 내 pendingInterest() 호출을 통한 interest 계산

```
function pendingInterest(uint256 value) public view returns (uint256) {
  if (now > lastAccrueTime) {
    uint256 timePast = now.sub(lastAccrueTime);
    uint256 balance = SafeToken.myBalance(token).sub(value);
    uint256 ratePerSec = config.getInterestRate(vaultDebtVal, balance);
    return ratePerSec.mul(vaultDebtVal).mul(timePast).div(1e6);
  } else {
    return 0;
  }
}
```

pendingInterest 메소드 내 InterestModel과 일치하지 않는 denominator 사용

```
interface InterestModel {
  /// @dev Return the interest rate per second, using 1e18 as denom.
  function getInterestRate(uint256 debt, uint256 floating) external view
returns (uint256);
}
```

InterestModel 인터페이스 내 getInterestRate 메소드의 주석 참고

```
function totalToken() public view override returns (uint256) {
  return SafeToken.myBalance(token).add(vaultDebtVal).sub(reservePool);
}
```

totalToken 메소드 내 비정상적으로 커진 vaultDebtVal 사용

Incident Timeline

Block #	Tx	Notes
Denom is 1e18 (Contract: 0x89d6f3b45f5b78dbec1cb77bb6141ad7c23091b9)		
81433228	0x69254801	Contract upgraded to 0x1aad9ec2...
Denom is 1e6 (Contract: 0x1aad9ec213086c2246c96cecbb42b935b995a19f)		
81433403	0xe6e01b42	At this moment, the vault's debt is 0
81433459	0x65fd8978	Borrow (openPosition)
81433479	0xcd5531db	Debt before accrue interest: 0x162426b8
81433503	0xf8b7821d	Debt before accrue interest: 0x24723afb
81433533	0xecd364d2	Debt before accrue interest: 0x52f5786d
81433577	0x90907115	Debt before accrue interest: 0x1802d3d25
81433623	0xcd604375	Debt before accrue interest: 0x26822261cf
81433817	0x9e5775bf	Debt before accrue interest: 0x6388fd99f156
81433923	0x444dacd3	Debt before accrue interest: 0x5c38f2d4fb61e93c3
Loss of fund / Emergency Notice		
Notice #1	2022.01.17 20:02 KST	현재 KUSDT 풀 점검 중에 있습니다. 점검 완료되는대로 바로 안내드리겠습니다 🙂
Notice #2	2022.01.27 20:07 KST	< 긴급 공지 : KUSDT Lending Pool 점검 > 안녕하세요? 현재 KLEVA Protocol에서 KUSDT 풀을 점검하고 있습니다. 빠른 시일내에 점검 완료 후 안내를 드릴 예정이니 조금만 기다려주시면 감사하겠습니다. 감사합니다. KLEVA TEAM 드림
Notice #3	2022.01.27 20:39 KST	< 공지 사항 > KLEVA 프로토콜의 Farm 서비스 오픈에 앞서 일부 프론트 표기에 오류가 있어 수정중에 있습니다.

		사용자들의 모든 자산은 안전하게 운용중에 있습니다 표기 오류 수정을 위해 KUSDT의 랜딩 및 스테이킹을 잠시 막아둔 상태입니다. 완료시 추가 공지 드리겠습니다. 감사합니다.
Notice #4	2022.01.27 22:28 KST	< 긴급 공지 > 안녕하세요 클래바 커뮤니티 여러분, 일부 트랜잭션에서 비정상적인 활동이 포착되어 점검 중에 있습니다. 현재 정상화를 위해 클레이튼, 오지스, 바이낸스를 포함한 다수의 거래소와 긴밀하게 협업을 진행하고 있으니 조금만 기다려 주시기 바랍니다.
Notice #5	2022.01.27 23:07 KST	<긴급 공지> 안녕하세요 클레바 커뮤니티 여러분, 일부 트랜잭션에서 비정상적인 활동이 포착되어 점검 중에 있습니다. 현재 정상화를 위해 클레이튼, 오지스, 바이낸스를 포함한 다수의 거래소와 긴밀하게 협업을 진행하고 있습니다. 빠르게 원인 분석을 완료하였으며, 많이들 우려하고 계신 해킹은 아닌것으로 확인 됩니다. 곧 원인 분석 내용을 포함한 향후 진행 상황을 공유하겠습니다. 감사합니다.
Notice #6	2022.01.27 23:35 KST	<긴급 공지> 안녕하세요 클레바 커뮤니티 여러분, 일부 트랜잭션에서 비정상적인 활동이 포착되어 점검 중에 있습니다. 현재 원인 분석을 완료했고 원인 분석 보고서를 작성 중에 있습니다. 아울러 오류 트랜잭션 리스트를 기반으로 모든 참여자들의 손실 등을 파악하고 있습니다. 사용상에 있어서 오류 트랜잭션이 의심되는 사용자께서는 제게 DM으로 해당 트랜잭션의 해시값 등 관련 정보를 모두 보내주시기 바랍니다. 감사합니다. KLEVA TEAM 드림
Notice #7	2022.01.28 05:12 KST	[공지사항] 안녕하세요 KLEVAR 여러분. 먼저 사용자 여러분들에게 큰 심려를 끼쳐드린 점 죄송합니다. 비정상적인 트랜잭션으로 인하여 과도하게 인출된 자산의 99% 이상이 반환처리되었으며, 반환되지 않은 자산은 외부 거래소와의 프로세스가 정리되는 대로 반환이 완료될 예정입니다.

		또한 과정상에서 개별적으로 손실을 본 사용자들 역시 TXID 등 확인을 진행 후, 해당 손실 자산 역시 반환됩니다. 금번 비정상적인 트랜잭션 발생은 해킹은 아니며 사용자들의 자산은 안전하게 보호고 되고 있습니다. 외부 거래소와의 관련 프로세스가 마무리되는 대로 현재 중단된 KUSDT 렌딩 / 스테이킹 역시 정상화 조치 될 예정입니다. 더불어, 비정상적인 트랜잭션의 발생 사유 및 조치 사항에 대한 리포트를 금일 내로 공유드리겠습니다. 다시 한번 이용에 불편을 겪으신 사용자 여러분들에게 진심으로 사과드립니다. 감사합니다 KLEVA Team 드림

Acknowledgement

Special thanks to @rkm0959

Interested in Smart Contract Security Audit?

✉ contact@theori.io

representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.