dAMM Finance Audit

Smart Contract Security Assessment

Jun 13, 2022

ABSTRACT

Dedaub was commissioned to perform a security audit on System9's dAMM smart contracts at commit hash `3d59c5c055884122ecc5e7bf6f446205912ba09c`. The audit scope was limited in size and included the following two files:

- contracts/ComptrollerG7.sol
- contracts/ComptrollerStorage.sol

Two auditors worked on the audit for 3 days. Given that the protocol contracts are a fork of Compound, the subject of the audit were the introduced changes (delta) in the files listed above and most of the audit effort was expended on the security of the new whitelisting feature. As such, this audit was fairly localized and was not concerned about potential issues in the rest of the protocol.

SETTING & CAVEATS

System9's dAMM protocol is a lending protocol, which allows borrowers to have under-collateralized loans. The protocol is a fork of Compound, and builds on top of it by adding a whitelisting layer to control who and how much one can borrow. Unlike borrowing, liquidity provision is entirely permissionless, and lending pool participants enjoy yield rewards.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds

	may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: 01) User or system funds can be lost when third party systems misbehave. 02) DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming error.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed”, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY

[No critical severity issues]

HIGH SEVERITY:

ID	Description	STATUS
H1	CToken `liquidateBorrow` will always fail in the context of a CToken liquidation	**RESOLVED**
The external entry point for liquidations in either `CEther` or `CErc20` is the `liquidateBorrow`. The control flow eventually reaches `ComptrollerG7::liquidateBorrowAllowed`, which has the following check at the top:		

```
function liquidateBorrowAllowed(
    address cTokenBorrowed,
    address cTokenCollateral,
    address liquidator,
    address borrower,
    uint repayAmount) override external returns (uint) {
    // Shh - currently unused
    liquidator;

    // Dedaub: This will always fail when called from the CToken liquidation logic
    require(msg.sender == admin, "only dAMM Foundation can liquidate borrowers");
    if (!markets[cTokenBorrowed].isListed || !markets[cTokenCollateral].isListed) {
        return uint(Error.MARKET_NOT_LISTED);
    }
     ...
}
```

However, this check will always fail in the context of a `CToken` liquidation, as `msg.sender` will not be the admin.

The check should instead be moved in `CToken::liquidateBorrowInternal`.

MEDIUM SEVERITY:

ID	Description	STATUS
M1	Shortfall check in `redeemAllowedInternal` should be removed	**RESOLVED**

In `ComptrollerG7::redeemAllowedInternal`, the following check takes place:

```
function redeemAllowedInternal(address cToken, address redeemer, uint redeemTokens)
internal view returns (uint) {
    /* Otherwise, perform a hypothetical liquidity check to guard against shortfall */
    (Error err, , uint shortfall) = getHypotheticalAccountLiquidityInternal(redeemer,
CToken(cToken), redeemTokens, 0);
    if (err != Error.NO_ERROR) {
        return uint(err);
    }

    // Dedaub: This can be problematic for a borrower
    if (shortfall > 0) {
```

```
        return uint(Error.INSUFFICIENT_LIQUIDITY);
    }
    ...
}
```

In its essence, this check ensures that the position of the redeemer is not under-collateralized after the redemption.

While this check makes sense from the point of view of an over-collateralized lending protocol such as Compound, it can be problematic in the case where a borrower wants to redeem some of his cTokens as it's quite likely that the on-chain calculation will report a non-zero shortfall. It is recommended that this check be removed.

LOW SEVERITY:

ID	Description	STATUS
L1	Optimization: Iterate over only relevant markets in `getNotionalBorrowsInternal`	**RESOLVED**

In `ComptrollerG7::getNotionalBorrowsInternal` the code loops over all supported assets, in order to calculate the value of a users borrowed assets:

```
function getNotionalBorrowsInternal(address borrower) internal view returns (uint256) {
    // Dedaub: Gas-intensive; goes over all supported assets
    CToken[] memory cTokens = getAllMarkets();
    uint numMarkets = cTokens.length;
    uint balance = 0;

    for(uint i = 0; i < numMarkets; i++) {
        CToken cToken = cTokens[i];
        balance += oracle.getUnderlyingPrice(cToken) *
                                        cToken.borrowBalanceStored(borrower);
    }
    return balance;
}
```

This is very gas inefficient, as the borrower will only have borrowed a small subset of the supported tokens in most cases.

It is highly recommended that the code be refactored to use the `getAssetsIn` method instead of `getAllMarkets()`, especially since the protocol aims to support a significant amount of assets.

L2	Incorrect governance token address	**OPEN**

In ComptrollerG7::getCompAddress, the governance token address should be returned. However, being a fork of Compound, the current code erroneously returns the Compound governance token (COMP) instead of the System9 one.

The code should be updated to return the correct address.

L3	Shortfall logic should be removed from `ComptrollerG7::liquidateBorrowAllowed`	**RESOLVED**

In `ComptrollerG7::liquidateBorrowAllowed`, the following check on shortfall takes place:

```
function liquidateBorrowAllowed(...){
    ...
    (Error err, , uint shortfall) = getAccountLiquidityInternal(borrower);
    if (err != Error.NO_ERROR) {
        return uint(err);
    }

    // Dedaub: Shortfall checks don't really apply as they do in Compound
    if (shortfall == 0) {
        return uint(Error.INSUFFICIENT_SHORTFALL);
    }
    ...
}
```

The dAMM protocol aims to support on-chain collateral for some of their borrowers. However, as discussed with the developers, in the case of a default, the shortfall will occur off-chain. This means that a borrower does not need a shortfall in order to be liquidated.

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend addressing them.

ID	Description	STATUS
A1	Whitelisting an already whitelisted will reset his borrowing limit	**OPEN**

In `ComptrollerG7::whitelistBorrowerAdd`, the relevant storage field whitelisting storage fields are getting initialized:

```
borrowerArray[borrower] = true;
borrowLimit[borrower] = 0;
```

While this is entirely logical when borrower is not already whitelisted, this acts as a borrow limit reset when `borrowerArray[borrower]` is already true and has a non-zero borrow limit. While this is not a bug, it is a weird edge case and it makes sense to add a check in `ComptrollerG7::whitelistBorrowerAdd` that guards against the above scenario.

ID	Description	STATUS
A2	Optimization: Guards can be merged	**RESOLVED**

In `ComptrollerG7::setBorrowerLimits`, the following guards are in place:

```
function setBorrowerLimits(address borrower, uint256 _borrowLimit) public override returns
(uint) {
    if (msg.sender != admin) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_BORROWER_LIMIT_CHECK);
    }

    // Dedaub: This check can be merged with the above guard, to save some gas
    //         (both in deployment and in runtime)
    if (borrowerArray[borrower] != true) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_BORROWER_LIMIT_CHECK);
    }
    ...
}
```

As the failure handling is identical in both cases, these two guards can be merged, into a single one where the condition is the two conditions are combined with a logical-or operator:

```
function setBorrowerLimits(address borrower, uint256 _borrowLimit) public override returns
(uint) {
    if (msg.sender != admin || borrowerAarray[borrower] != true) {
        return fail(Error.UNAUTHORIZED, FailureInfo.SET_BORROWER_LIMIT_CHECK);
    }
    ...
}
```

This will help save some gas during normal execution, as well as make deployment slightly cheaper due to more compact bytecode.

A3	Compiler known issues	**INFO**

The contracts were compiled with the Solidity compiler v0.8.14 which, at the time of writing, have some known bugs. We inspected the bugs listed for version 0.8.13 and concluded that the subject code is unaffected.

PROTOCOL CENTRALIZATION ELEMENTS

The protocol has significant centralization elements. More specifically, the whitelisting of the borrowers and the definition of their borrow limits is entirely determined by the protocol owners/admins. Both of these elements are entirely reasonable, as the protocol aims to provide under-collateralized loans to trusted borrowers.

It should also be noted that a lot of the protocol monitoring and the locking of the collateral is done off-chain, and borrowing terms are enforced via contracts.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contracts. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

The resolution of report items is determined by local inspection of changes, not a full re-audit. Since there was a significant time elapsed between the conclusion of the audit and the addressing of some issues, the development team is advised to be especially vigilant with testing the consequences of fixes performed after the initial audit.

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the

contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.