verichains

SECURITY AUDIT OF

KRYSTAL

SMART CONTRACTS

PUBLIC REPORT

JULY 19, 2021

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ACRONYMS AND ABBREVIATIONS

NAME	DESCRIPTION
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
EVM	Ethereum Virtual Machine
LP	Liquidity Provider

EXECUTIVE SUMMARY

This Security Audit Report was initially prepared by Verichains Lab on July 02, 2021, last updated on July 19, 2021. We would like to thank the Krystal team for trusting Verichains Lab in audit smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Krystal core smart contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified several vulnerable issues in the first version of smart contracts code. All these issues were resolved in the updated version of smart contracts.

Overall, the code reviewed is of good quality, written with the awareness of smart contract development best practices.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Krystal

Krystal (https://krystal.app) is an all-in-one DeFi platform that has brought together the best DeFi services under one roof. Krystal will save users time, money, and effort by consolidating the best DeFi services, including: seamless token swaps, lending tokens for interest/yield, hassle-free portfolio management, and solid security, with more features coming in the future.

More information at https://krystal.app/.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the core smart contracts of Krystal. It was conducted on commit `4f9b12268669ba6b9dc33941ac3a14-6f38a69614` of branch `audit-1` from GitHub repository of Krystal Core.

Repository URL of the commit to be audited: https://github.com/KYRDTeam/krystal-core/commit/4f9b12268669ba6b9dc33941ac3a146f38a69614.

The updated version was conducted on commit `2d7d8f2d41e5d1d04d1b8f3399-e988b3d6fe6ffb` of branch `audit-1`, on URL: https://github.com/KYRDTeam/krystal-core/commit/2d7d8f2d41e5d1d04d1b8f3399e988b3d6fe6ffb.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference

- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in Table 1, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1: Vulnerability severity levels

1.4. Result summary

Table 2 shows the discovered vulnerabilities and fixing status.

#	Summary	Severity	Status	Updated At
1	Transferred amount calculation can be wrong if token is ERC777	MEDIUM	✓ MITIGATED	09-July-2021
2	Old spenders' allowances are not revoked when approving new spenders	LOW	✓ NO-FIX	09-July-2021
3	Unlimited token allowances	LOW	✓ NO-FIX	09-July-2021

Table 2: Vulnerabilities summary

1.5. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Contract overview

Krystal is a DeFi platform that allows users to interact with other popular DeFi protocols, currently supports Uniswap, Aave, Kyber DMM and Compound. It is a simplified wrapper layer to communicate with DApps and these DeFi protocols with fee handlings logic built-in for each transaction. The collected fees will be transferred to the platform wallets configured by the participated DApps.

The following subsections contain the summary of the Krystal platform's logic (as of the writing of this audit report, some functions have not been implemented yet).

2.1.1. SmartWalletProxy contract

The `SmartWalletProxy` contract is the proxy contract which delegates all the calls to the implementation contract (`SmartWalletImplementation`). Proxing allows the contract's implementation to be upgraded later.

2.1.2. SmartWalletImplementation contract

This is the implementation contract for the above proxy. This contract has following important functions:

- *`claimPlatformFees`*: send the collected fees to the specified platform wallets.
- *`approveAllowances`*: approve/revoke LPs usage on particular tokens balances of platform contract.
- *`getExpectedReturn`*: calculates the expected return including the fee before swap.
- *`swap`*: perform swapping by calling `swapInternal` function.
- *`swapInternal`*: actual handler for swapping tokens and collecting fees. It calls the appropriate swap contract (including `UniSwap`, `UniSwapV3, KyberProxy` and `KyberDmm`) to perform the swap.
- *`swapAndDeposit`*: swap tokens if necessary and deposit the result token to lending pools by calling the appropriate lending contract (including `AaveV1Lending`, `AaveV2Lending` and `CompoundLending`).
- *`withdrawFromLendingPlatform`*: withdraw token from lending platforms by calling the above lending contracts.
- *`swapAndRepay`*: swap tokens if necessary and repay the debt to the lending platforms.
- *`safeTransferWithFee`*: transfer tokens including fee, the fees will then be transferred to the platform wallet. Note that all the platform fees and platforms wallet are specified by the caller (DApps).

2.1.3. Lending gateway contracts

The following contracts are platform's contracts which act as gateways to the external contracts to perform swap logics and called by `SmartWalletImplementation`:

- *UniSwap*: gateway to interact with Uniswap v2 protocol.
- *UniSwapV3*: gateway to interact with Uniswap v3 protocol.
- *KyberDmm*: gateway to interact with Kyber DMM protocol.
- *KyberProxy*: gateway to interact with Kyber Network protocol.

All these contracts are extended from the `BaseSwap` contract, which implements the `ISwap` interface.

2.1.4. Swap gateway contracts

The following contracts are platform's contracts which act as gateways to the external contracts to perform lending logics and called by `SmartWalletImplementation`:

- *AaveV1Lending*: gateway to interact with Aave v1 lending protocol.
- *AaveV2Lending*: gateway to interact with Aave v2 lending protocol.
- *CompoundLending*: gateway to interact with Compound lending protocol.

All these contracts are extended from the `BaseLending` contract, which implements the `ILending` interface.

2.2. Discovered vulnerabilities

This section contains a detailed analysis of all the vulnerabilities that were discovered by the audit team during the audit process.

2.2.1. Transferred amount calculation can be wrong if token is ERC777

✓ MITIGATED [MEDIUM]

The *safeTransferWithFee* function in the SmartWalletImplementation contract can lead to wrong calculation of the amountTransferred variable if the token transferred is an ERC777 token[1]. Consider the following implementation:

SmartWalletImplementation.sol#L350:379

```
function safeTransferWithFee(
    address payable from,
    address payable to,
    address token,
    uint256 amount,
    uint256 platformFeeBps,
    address payable platformWallet
) internal returns (uint256 amountTransferred) {
    uint256 fee = amount.mul(platformFeeBps).div(BPS);
    uint256 amountAfterFee = amount.sub(fee);
    IERC20Ext tokenErc = IERC20Ext(token);

    if (tokenErc == ETH_TOKEN_ADDRESS) {
        (bool success, ) = to.call{value: amountAfterFee}("");
        require(success, "transfer failed");
        amountTransferred = amountAfterFee;
    } else {
        uint256 balanceBefore = tokenErc.balanceOf(to);
        if (from != address(this)) {
            // case transfer from another address, need to transfer fee to this proxy contract
            tokenErc.safeTransferFrom(from, to, amountAfterFee);
            tokenErc.safeTransferFrom(from, address(this), fee);
        } else {
            tokenErc.safeTransfer(to, amountAfterFee);
        }
        amountTransferred = tokenErc.balanceOf(to).sub(balanceBefore); // <- VULNERABLE
    }
```

[1] https://eips.ethereum.org/EIPS/eip-777

```
        addFeeToPlatform(platformWallet, tokenErc, fee);
    }
```

Since ERC777 token standard supports *tokensToSend* and *tokensReceived* hooks[2], so that the attacker can deploy a smart contract with these hooks as a receiver. After this address receives the token, the attacker can modify the balance himself and abuse the value of the *amountTransferred* variable. EIP1820[3] registry can prevent malicious contracts from hooking but that is not enough because the contract just needs to receive/send tokens from another source to change their balance which is a normal behavior in their own context.

An example case where the attacker exploits the ERC777 hook vulnerability can be found here: https://defirate.com/imbtc-uniswap-hack.

RECOMMENDED FIXES

Use the *amountAfterFee* value for the *amountTransfererred* as the following code suggestion:

SmartWalletImplementation.sol#L350:379

```
    function safeTransferWithFee(
        address payable from,
        address payable to,
        address token,
        uint256 amount,
        uint256 platformFeeBps,
        address payable platformWallet
    ) internal returns (uint256 amountTransferred) {
        uint256 fee = amount.mul(platformFeeBps).div(BPS);
        uint256 amountAfterFee = amount.sub(fee);
        IERC20Ext tokenErc = IERC20Ext(token);

        if (tokenErc == ETH_TOKEN_ADDRESS) {
            (bool success, ) = to.call{value: amountAfterFee}("");
            require(success, "transfer failed");
            amountTransferred = amountAfterFee;
        } else {
            uint256 balanceBefore = tokenErc.balanceOf(to);
            if (from != address(this)) {
                // case transfer from another address, need to transfer fee to this proxy contract
```

[2] https://docs.openzeppelin.com/contracts/4.x/api/token/erc777#hooks
[3] https://eips.ethereum.org/EIPS/eip-1820

```
                tokenErc.safeTransferFrom(from, to, amountAfterFee);
                tokenErc.safeTransferFrom(from, address(this), fee);
            } else {
                tokenErc.safeTransfer(to, amountAfterFee);
            }
        }
        amountTransferred = amountAfterFee; // <- FIXED HERE

        addFeeToPlatform(platformWallet, tokenErc, fee);
    }
```

MITIGATIONS

The audit team have confirmed that attacker has no benefit from attacking using ERC777 hooks. In some external functions of Krystal's smart contracts, the return value of *safeTransferWithFee* function is used to emit events and returned value to external callers, so there must be no benefit for the attacker in the Krystal platform. The Krystal team has added some validations to ensure that the 3rd-parties code will function properly.

However, if we look at the bigger picture, this can be used as a chaining bug if the platform is chained to some other services that read the return value of these functions, which would cause damage to their system. That's outside of Krystal platform's scope, but as a security auditor, the audit team still considers returning the wrong value as a security risk.

For above reasons, the audit team decided to reduce this issue's severity to MEDIUM.

2.2.2. Old spenders' allowances are not revoked when approving new spenders

✓ NO-FIX [LOW]

Affected functions:

- *updateAaveData* in the AaveV1Lending and AaveV2Lending contract.
- *updateCompoundData* in the CompoundLending contract.
- *swap* in the KyberDmm, KyberProxy, UniSwap, UniSwapV3 contracts.

For example, consider the current implementation of *updateAaveData* function in the AaveV1Lending contract. When updating the Aave data, old lending pool allowances are not revoked before approving to new lending pool. As the old lending pool are no longer used in the platform, leaving the allowance amount unhandled can leads to future attacks that were unforeseen to the developers.

SmartWalletImplementation.sol#L25:51

```
    function updateAaveData(
        IAaveLendingPoolV1 poolV1,
```

```
        address lendingPoolCoreV1,
        uint16 referalCode,
        IERC20Ext[] calldata tokens
    ) external onlyAdmin {
        ...
        for (uint256 i = 0; i < tokens.length; i++) {
            try
                ILendingPoolCore(poolV1.core()).getReserveATokenAddress(address(tokens[i]))
            returns (address aToken) {
                if (aaveData.aTokensV1[tokens[i]] != aToken) {
                    aaveData.aTokensV1[tokens[i]] = aToken;
                    safeApproveAllowance(lendingPoolCoreV1, tokens[i]);
                }
            } catch {}
        } // <- OLD TOKENS ALLOWANCE WERE NOT REVOKED
    }
```

RECOMMENDED FIXES

There are two recommend ways to fix this issue:

- Instead of approving to *MAX_ALLOWANCE* at the first time, the contract should only approve a sufficient allowance for each transaction.
- The contract should revoke the allowances of old spenders before approving to new ones.

Each way has its own trade-off, such as execution gas cost increasing, code complexity increasing, etc.

UPDATES

From the Krystal team's feedback:

"Actually, the swap and lending contract don't keep fund. Hence overall it's safe and gas-friendly to just set MAX_ALLOWANCE on the 3rd parties (uni, kyber, aave etc)."

The audit team agreed that as long as the platform contracts are non-reentrant and swap and lending contracts don't keep funds, attackers won't be able to take advance of this. But the audit team also considering this problem as a possible helper to chain with some other bugs so that attackers can steal the balance in current transaction if there's a bug in old 3rd-party contracts - because they're still approved to use swap or lending contract's tokens.

For above reasons, the audit team decided to set this issue's severity to **LOW**.

2.2.3. Unlimited token allowances

✓ **NO-FIX** **[LOW]**

Consider the following code pattern from the *safeApproveAllowance* function in the BaseLending and BaseSwap contracts:

```
function safeApproveAllowance(address spender, IERC20Ext token) internal {
    if (token != ETH_TOKEN_ADDRESS && token.allowance(address(this), spender) == 0) {
        token.safeApprove(spender, MAX_ALLOWANCE);
    }
}
```

Theese functions are used in the following functions:

- *updateAaveData* in the AaveV1Lending and AaveV2Lending contract.
- *updateCompoundData* in the CompoundLending contract.
- *swap* in the KyberDmm, KyberProxy, UniSwap and UniSwapV3 contracts.

We can see that all the tokens from our contracts are approved for the external contracts. This behaviour may help in saving the gas cost but it's still a bad security practice. An example case of exploiting this type of vulnerability has been mentioned here https://cointelegraph.com/news/bancors-bug-exposes-dangerously-common-practice-in-ethereum-defi.

Additionally, these functions may lead to reverted transaction if the current allowance is not enough. Practically this can not occur within current Ethereum landscape, but theorically this may occur if the token's volume is big enough or the transaction volume may grow bigger in future Ethereum versions which are all possible. A proper way to fix should be reapproving to *MAX_ALLOWANCE* if current allowance less than *MAX_ALLOWANCE/2*, which still require the same gas as current implementation most of the time (or all in current practical conditions).

RECOMMEND FIXES

Consider approving a sufficient allowance for each transaction instead of approving to *MAX_ALLOWANCE* at the first time.

UPDATES

With the same update as in issue 2.2.2, the audit team decided to set this issue's severity to **LOW**.

2.3. Recommended gas optimizations

2.3.1. Improper usage of enum variables which increase gas cost

In Solidity compiler, every time an enum variable's value is used, its current value (stored as a *word* type - *uint* in case of Ethereum) will be checked for the enum's value range. In short, a single value-usage of an enum variable will be compiled to 3 additional opcodes:

- push the variable's value to stack.
- duplicate that value (for enum value check comparision).
- check if that value larger than maximum possible value of the enum type.
  - in that case, revert.

That's an increasing in final byte code size and runtime gas. To optimize this behavior, we can cast the enum variable to *uint256* for later usage whenever safe to stop the value range checking. The optimization helps with the gas overhead problem but will result in an uglier code and the one that do the optimization must know when it's safe to do it.

For example, consider the following contract:

```
pragma solidity 0.7.6;

contract Test {
    enum TestEnum {
        A,
        B,
        C
    }

    function test(TestEnum p) external returns (uint256 r) {
        r = 4;
        if (p == TestEnum.A) r = 1;
        if (p == TestEnum.B) r = 2;
        if (p == TestEnum.C) r = 3;
    }
}
```

In this contract, we can optimize the usage of enum values in function *test* as below:

```
function test(TestEnum p) external returns (uint256 r) {
    uint256 puint = uint256(p);
    r = 4;
    if (puint == uint256(TestEnum.A)) r = 1;
```

```
        if (puint == uint256(TestEnum.B)) r = 2;
        if (puint == uint256(TestEnum.C)) r = 3;
    }
```

In case of Krystal smart contracts, the *FeeMode* enum variable is read-only everytime it is used in the code, so it's safe to do this optimization. But it is not easy to perform refactor globally for this optimization as the code currently wrap that variable into a struct that is sent from external and reuse across function calls. We can still do this optimization locally, for example, in *getExpectedReturn* function:

```
function getExpectedReturn(
    ISmartWalletImplementation.GetExpectedReturnParams calldata params
) external view override returns (uint256 destAmount, uint256 expectedRate) {
    if (params.feeBps >= BPS) return (0, 0); // platform fee is too high

    uint256 actualSrc = (params.feeMode == FeeMode.FROM_SOURCE)
        ? params.srcAmount * (BPS - params.feeBps) / BPS : params.srcAmount;

    destAmount = ISwap(params.swapContract).getExpectedReturn(
        ISwap.GetExpectedReturnParams({
            srcAmount: actualSrc,
            tradePath: params.tradePath,
            feeBps: params.feeMode == FeeMode.BY_PROTOCOL ? params.feeBps : 0,
            extraArgs: params.extraArgs
        })
    );

    if (params.feeMode == FeeMode.FROM_DEST) {
        destAmount = destAmount * (BPS - params.feeBps) / BPS;
    }

    expectedRate = calcRateFromQty(
        params.srcAmount,
        destAmount,
        getDecimals(IERC20Ext(params.tradePath[0])),
        getDecimals(IERC20Ext(params.tradePath[params.tradePath.length - 1]))
    );
}
```

Which can be optimized into:

```
function getExpectedReturn(
    ISmartWalletImplementation.GetExpectedReturnParams calldata params
) external view override returns (uint256 destAmount, uint256 expectedRate) {
    uint256 feeMode = uint256(params.feeMode);
    ...
    uint256 actualSrc = (feeMode == uint256(FeeMode.FROM_SOURCE))
    ...
            feeBps: feeMode == uint256(FeeMode.BY_PROTOCOL) ? params.feeBps : 0,
    ...
    if (feeMode == uint256(FeeMode.FROM_DEST)) {
    ...
}
```

Note that the enum constant value usage does not have gas overhead.

2.3.2. Comparison with unilateral outcome in a loop

When a comparison is executed in each iteration of a loop but the result is the same each time, it should be removed from the loop.

For example, consider the following code from the `SmartWalletProxy` contract:

`SmartWalletProxy.sol#L76:86`

```
function updateSupportedSwaps(address[] calldata addresses, bool isSupported)
    external onlyAdmin
{
    for (uint256 i = 0; i < addresses.length; i++) {
        if (isSupported) {
            supportedSwaps.add(addresses[i]);
        } else {
            supportedSwaps.remove(addresses[i]);
        }
    }
}
```

The comparison in each iteration of the for loop has the same value in each loop, so it should be move outside the loop for gas optimization as below:

SmartWalletProxy.sol#L76:86

```
function updateSupportedSwaps(address[] calldata addresses, bool isSupported)
    external onlyAdmin
{
    if (isSupported) {
        for (uint256 i = 0; i < addresses.length; i++) {
            supportedSwaps.add(addresses[i]);
        }
    } else {
        for (uint256 i = 0; i < addresses.length; i++) {
            supportedSwaps.remove(addresses[i]);
        }
    }
}
```

Other functions in Krystal smart contracts which contain this for-loop pattern (such as `updateSupportedLendings`, `updateSupportedPlatformWallets`, etc.) can be optimized with the same technique.

3. VERSION HISTORY

Version	Date	Status/Changes	Created by
0.9	July 02, 2021	Initial report	Verichains Lab
1.0	July 09, 2021	Private report	Verichains Lab
1.1	July 19, 2021	Public report	Verichains Lab

APPENDIX A: OVERVIEW FUNCTION CALL GRAPH

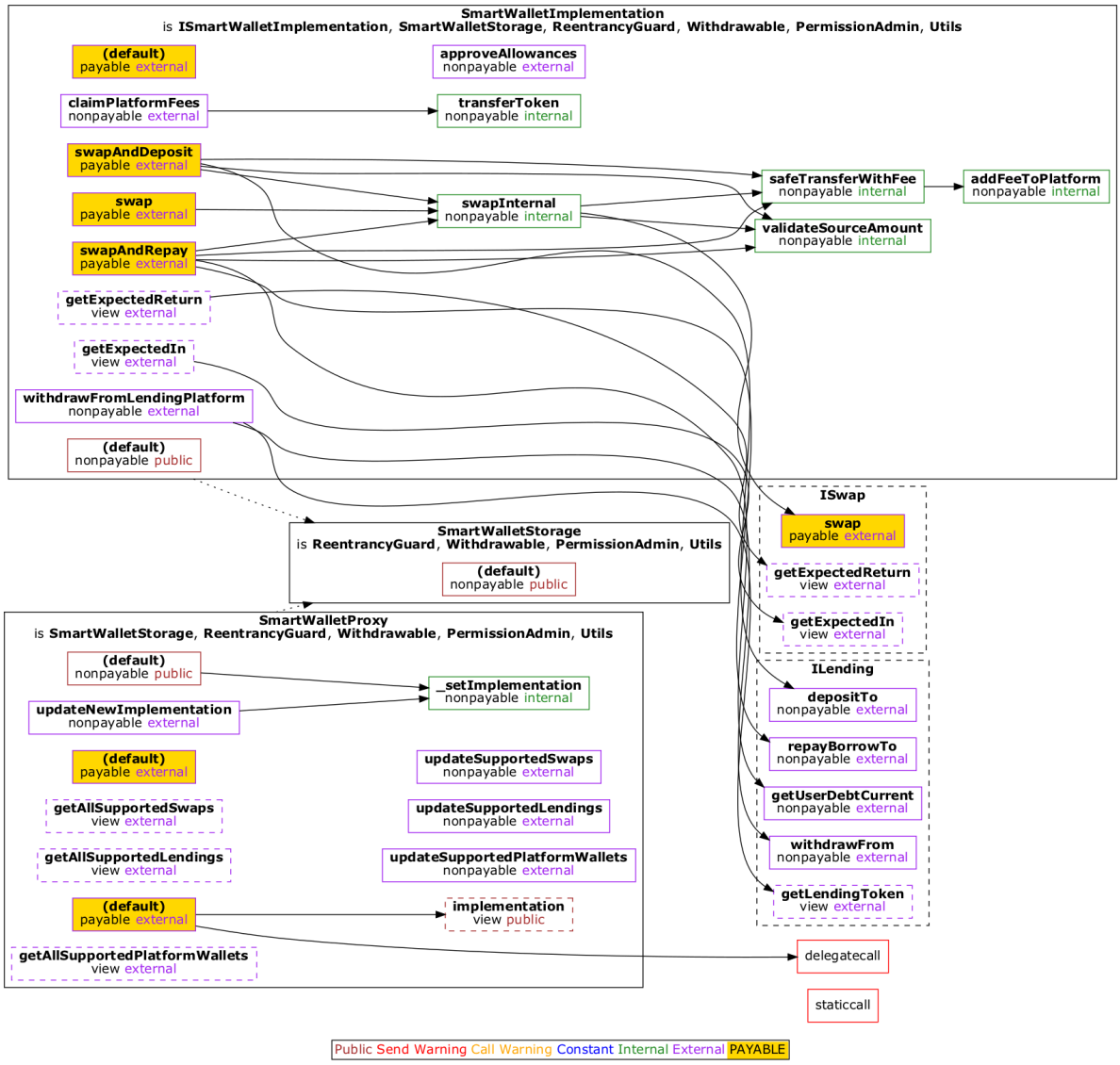

Figure 1: The overview function call graph of Krystal smart contracts

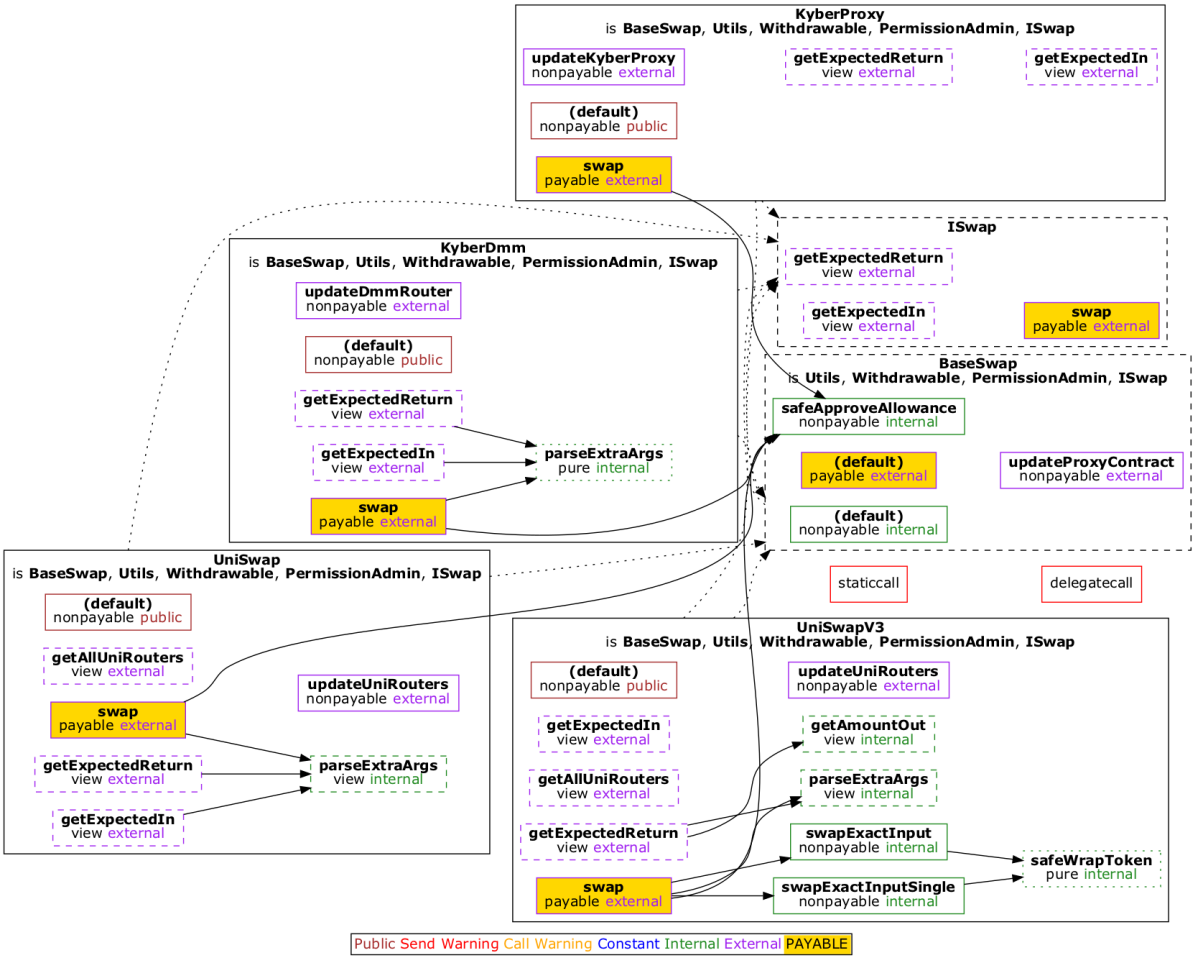

Figure 2: The overview function call graph of swap contracts

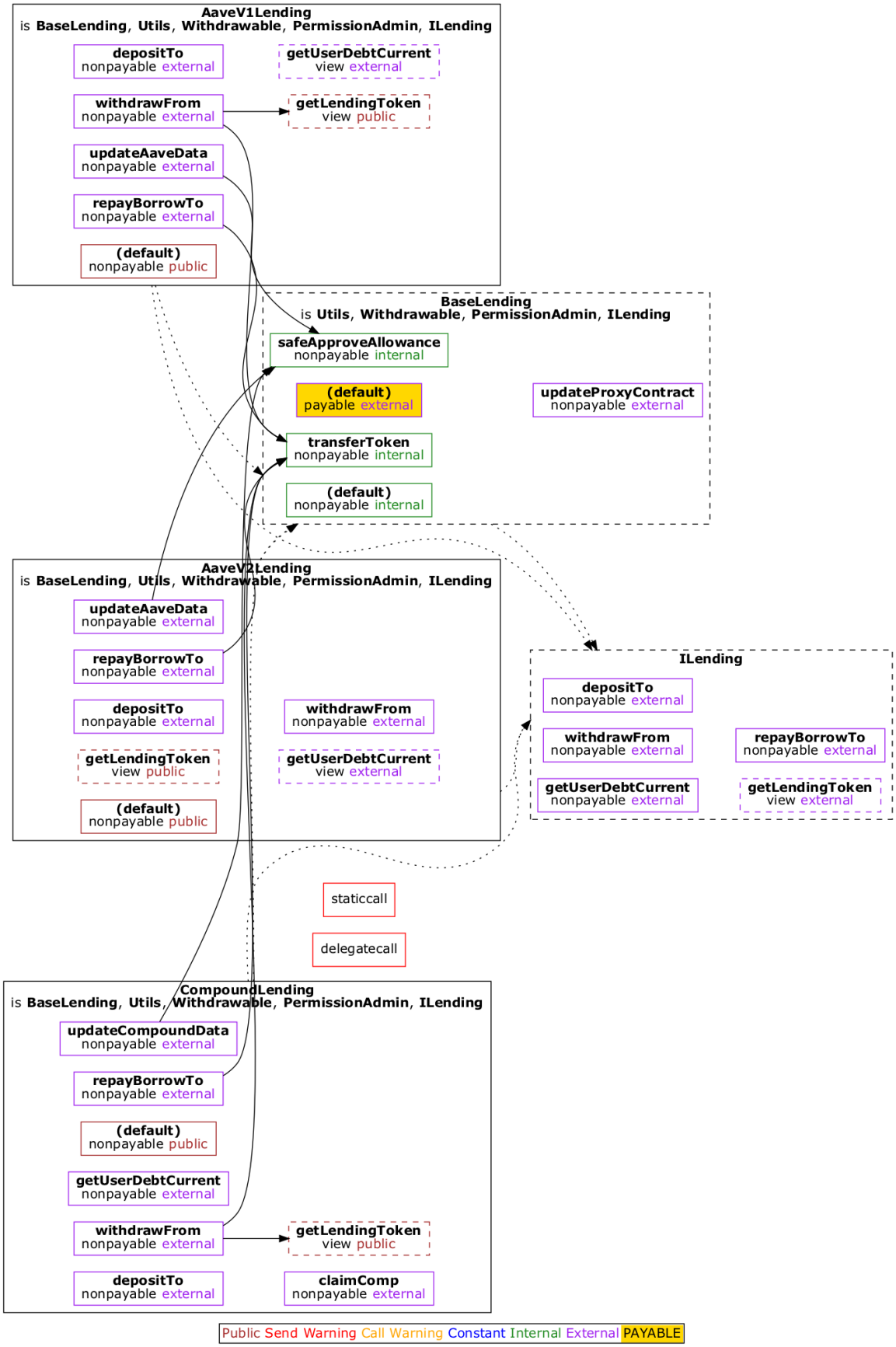

Figure 3: The overview function call graph of lending contracts