Skillet

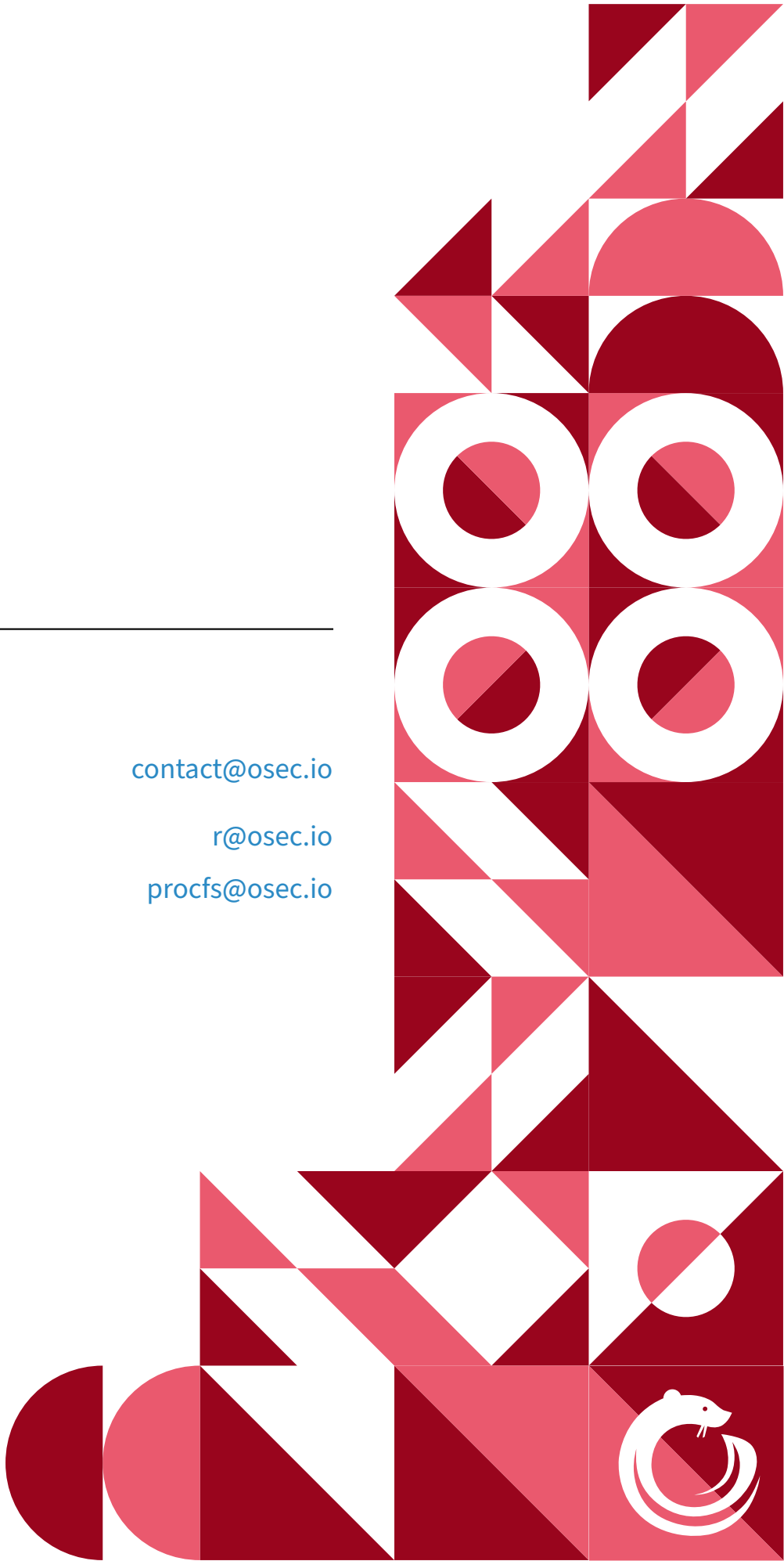
Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Woosun Song procfs@osec.io



Contents

Appendices

01 | Executive Summary

Overview

Skillet engaged OtterSec to perform an assessment of the `skillet-contract` and `fee-manager` programs. This assessment was conducted between February 27th and March 4th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 6 findings total.

In particular, we discovered an issue allowing the takeover of the implementation contract by invoking `initialize` (OS-SKT-ADV-01) and regarding unchecked return values potentially leading to the loss of user assets (OS-SKT-ADV-01).

Additionally, we made recommendations around uninitialized variables in `initialize` that may potentially escalate security concerns if upgrades are introduced (OS-SKT-SUG-00) and regarding code size and gas consumption optimization (OS-SKT-SUG-01, OS-SKT-SUG-02).

02 | Scope

The source code was delivered to us in a git repository at github.com/Skillet-Capital/skillet-library/tree/master. This audit was performed against commit b0f73db.

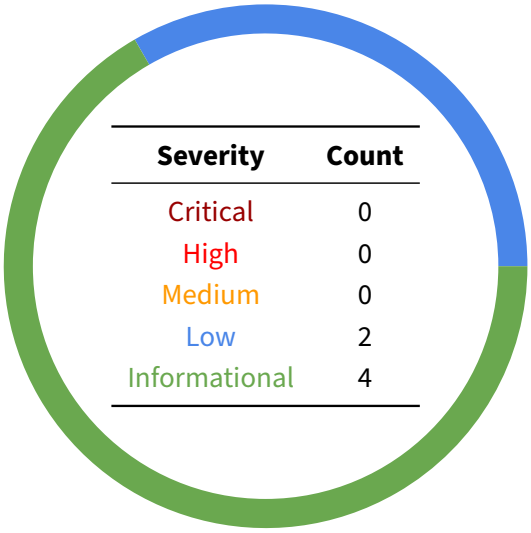
A brief description of the programs is as follows.

Name	Description
skillet-contract	An upgradable contract that implements an aggregator for selling ERC−721 and ERC−1155 NFTs.
fee-manager	This contract manages the fee rate and receiver for the protocol fees generated in `skillet-contract`.

03 | Findings

Overall, we reported 6 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	0
Low	2
Informational	4

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-SKT-ADV-00	Low	Resolved	`initialize` may be called for the implementation contract `SkilletContract`, resulting in a takeover.
OS-SKT-ADV-01	Low	Resolved	The return values of ERC20 `approve` and `transfer` are left unchecked in `ProxyApprovable` and `PaymentManager`.

OS-SKT-ADV-00 [low] | Implementation Contract Takeover

Description

Under the current implementation of `Skilletcontract`, it is possible to call `initialize` on the implementation address. Calling `initialize` on the implementation contract sets the caller to `owner`, allowing the execution of methods with the `onlyOwner` modifier. This poses two security implications.

First, the takeover may be exploited if `SkilletContract` starts to implement `UUPSUpgradable`. In this case, an attacker may exploit this takeover to `selfdestruct` the implementation contract and brick the proxy permanently. At present, such attacks are infeasible since `SkilletContract` does not yet implement `UUPSUpgradable`.

Second, the takeover of the implementation contract may be used for an intricate scam. The implementation contract will be annotated as credible on block explorers and wallets, and users may be deceived into depositing funds and NFTs into the overtaken contract.

Remediation

Prevent `initialize()` from being invoked on the implementation contract by adding the following constructor to `SkilletContract`.

SOLIDITY

```
constructor() {
    _disableInitializers();
}
```

Patch

Resolved in commit a5e8eae.

OS-SKT-ADV-01 [low] | Unchecked ERC20 Return Values

Description

The vulnerability pertains to `checkAndSetProxyApprovalERC20` of the `ProxyApprovable` contract, wherein `proxyAddress` is approved with the `MAX_UINT256` value by the `tokenContract`. However, the function fails to check the return value of `approve`.

Another instance of vulnerability is in the `PaymentManager` contract, wherein `calculateAndTakeFee` and `transferPaymentToSeller` use `transfer` without checking its return value.

ProxyApprovable.sol SOLIDITY

```
// checkAndSetProxyApprovalERC20()
if (!(allowance == MAX_UINT256)) {
    tokenContract.approve(proxyAddress, MAX_UINT256);
}
```

PaymentManager.sol SOLIDITY

```
// transferPaymentToSeller()
IERC20 paymentToken = IERC20(paymentTokenAddress);
paymentToken.transfer(msg.sender, netAmount);
return;

// calculateAndTakeFee()
IERC20 paymentToken = IERC20(paymentTokenAddress);
paymentToken.transfer(protocolFeeRecipient, feeAmount);
return feeAmount;
```

The ERC20's standards `approve` and `transfer` return a bool confirming whether the operation was successful. While Openzeppelin's implementation never returns a false value by returning either true or reverting, it does not guarantee that other implementations also never return a false value.

Moreover as specified in EIP-20:
"Callers MUST handle false from returns (bool success). Callers MUST NOT assume that false is never returned!"

A possible consequence may be:

A token returns false on a `transfer` and in such cases, the contract would never know about it as it does not check the return value. Hence, no error is raised although no transfer of tokens occurred, resulting in the loss of user assets, which in this case would be NFTs. These NFTs were sold however, the seller did not get paid due to some error while transferring of the tokens.

Remediation

Utilize the `SafeERC20` contract, which serves as a wrapper for `ERC20` operations and throws an exception when the token contract returns a false value. Additionally, the contract supports tokens that do not return a value and revert or throw on failure, with non-reverting calls being assumed to be successful.

ProxyApprovable.sol PaymentManager.sol SOLIDITY

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract ProxyApprovable{
    using SafeERC20 for IERC20;

    // checkAndSetProxyApprovalERC20()
    if (!(allowance == MAX_UINT256)) {
        tokenContract.safeApprove(proxyAddress, MAX_UINT256);
    }
}

contract PaymentManager{
    using SafeERC20 for IERC20;

    // transferPaymentToSeller()
    IERC20 paymentToken = IERC20(paymentTokenAddress);
    paymentToken.safeTransfer(msg.sender, netAmount);
    return;

    // calculateAndTakeFee()
    IERC20 paymentToken = IERC20(paymentTokenAddress);
    paymentToken.safeTransfer(protocolFeeRecipient, feeAmount);
    return feeAmount;
}
```

Patch

Fixed in c612d33

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-SKT-SUG-00	`SkilletContract.initialize` fails to initialize multiple storage variables.
OS-SKT-SUG-01	`MAX_UINT256` and `MAX_PROTOCOL_FEE` may be declared as constant.
OS-SKT-SUG-02	The `sender` argument is unused in `calculateFee`.
OS-SKT-SUG-03	`PaymentManager.setFeeManager` fails to check for zero addresses and allows the setting of unintended addresses to be relatively easier.

OS-SKT-SUG-00 | Uninitialized Storage Variables

Description

`initialize` does not initialize storage variables with initial values of zero. As of now, this does not pose any security hazards.

SkilletContract.sol SOLIDITY

```
function initialize() public initializer {
    WETH_ADDRESS = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    ETH_ADDRESS = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
    MAX_UINT256 = 2**256 - 1;

    __Ownable_init();
}
```

Nonetheless, explicitly initializing all variables in `initialize` improves code readability, and if the initializer functions for upgraded contracts, such as `initializeV2()`, are implemented in the same manner, this matter may manifest into a security hazard.

This arises because the zero-initialized assumption is invalid for upgradable contracts, and accessing stale values may result in undefined behaviour.

Remediation

Initialize all storage variables and inheritance chains within `initialize`.

SOLIDITY

```
function initialize() public initializer {
    WETH_ADDRESS = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    ETH_ADDRESS = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
    MAX_UINT256 = 2**256 - 1;

    // initialize storage variables
    feeManager = address(0);
    alwaysWithdrawWeth = false;

    // initialize inheritance chains
    __Ownable_init();
    __Pausable_init();
}
```

OS-SKT-SUG-01 | Convert Variables To Constants

Description

`MAX_UINT256` and `MAX_PROTOCOL_FEE` are declared as mutable member variables, but they may be declared as constant. This change may save code size and gas consumption.

Remediation

Declare the two variables with the `constant` specifier and remove all code that initializes said variables.

SOLIDITY

```
contract ProxyApprovable {
    uint256 public constant MAX_UINT256 = type(uint256).max;
}
```

OS-SKT-SUG-02 | Dead Argument

The `sender` argument is unused in `calculateFee`. Omitting unused arguments in Solidity may contribute to a reduction in code size and gas consumption.

FeeManager.sol SOLIDITY

```
function calculateFee(address sender, uint256 amount)
    public
    view
    returns (uint256 feeAmount)
{
    feeAmount = (amount * protocolFee) / 10000;
}
```

Remediation

Remove the `sender` argument as follows.

FeeManager.sol SOLIDITY

```
function calculateFee(uint256 amount)
    public
    view
    returns (uint256 feeAmount)
{
    feeAmount = (amount * protocolFee) / 10000;
}
```

OS-SKT-SUG-03 | Possible Inadvertent Address Modification

Description

`PaymentManager.setFeeManager` initializes the address for `IFeeManager`. Therefore, it is crucial to exercise caution to prevent the owner from mistakenly setting the zero address or an unintended address by calling the function.

PaymentManager.sol SOLIDITY

```
function setFeeManager(address feeManagerAddress) public onlyOwner {
    feeManager = IFeeManager(feeManagerAddress);
}
```

Remediation

Implement a two-step process wherein an initial function records the address change, and the owner calls a second function to finalize and set the address. This approach may help prevent inadvertent calls to set the fee manager by requiring the owner to call two separate functions.

PaymentManager.sol SOLIDITY

```
// initiate the set function
function initiateSetManager(address feeManagerAddress) public onlyOwner {

    require(feeManagerAddress != address(0), "Invalid address");
    tempManager = feeManagerAddress;
}

// finalize the change
function finalizeSetManager() public onlyOwner {

    require(tempManager != address(0), "No new admin address set");
    feeManager = IFeeManager(feeManagerAddress);

    tempManager = address(0);
}
```

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.