Zellic

Limitbreak Payment Processing Layer

Smart Contract Security Assessment

May 31, 2023

Prepared for:

Nathan Gang

Limit Break

Prepared by:

Aaron Esau and Junyi Wang

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Limit Break from April 17th to April 21st, 2023. During this engagement, Zellic reviewed Limitbreak Payment Processing Layer's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could a buyer force an unconsenting third party to pay using the `designatedPurchaser`?
- Is it possible to steal assets from a buyer without rendering the goods?
- Is it possible to pay the seller with an undesired payment method?
- Can either the buyer or seller bypass the security policy?
- Can the token creator potentially raise the royalties to an unacceptable amount in the middle of a transaction?
- Will the contract always honor marketplace and royalty fees?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- The actual marketplaces, which could be a smart contract or a Web2 interface
- The token contracts, which can potentially contain unusual behavior

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

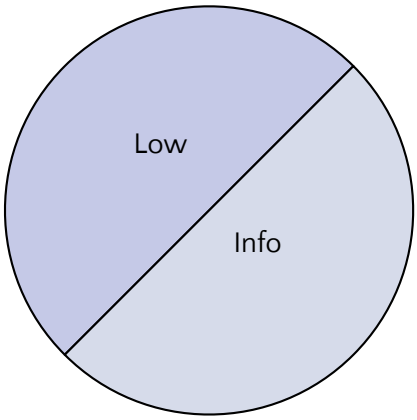
During our assessment on the scoped Limitbreak Payment Processing Layer contracts, we discovered two findings. No critical issues were found. Of the two findings, one

was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Limit Break's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	0
Low	1
Informational	1



2 Introduction

2.1 About Limitbreak Payment Processing Layer

Limit Break is introducing a new payment processing layer for NFT marketplace integrations. Combined with the whitelisted transfer and wrapper mechanics introduced in Creator Token Contracts, the Payment Processor contract can be used to securely process NFT sales with secondary market royalty enforcement built in. The Payment Processor, as implemented, enables truly enforceable programmable royalty systems to be built by NFT creators.

By default, all NFT collections fall back to the default security profile. The default profile is very open and permissive, allowing private sale, delegated purchases, and to buy and sell among EOAs and multi-sigs, and it is open to any marketplace to integrate. In case of abuse, NFT collection creators may create or use a custom security profile that is fully under their control. Thus, this contract can be considered to be a ***Creator-Defined Marketplace****.*

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Limitbreak Payment Processing Layer Contracts

Repository	https://github.com/limitbreakinc/minimum-floor-operator-zellic
Version	minimum-floor-operator-zellic: `e6277f80a7e881a3fa043518c5ccd64c8d19e4d0`
Programs	• PaymentProcessor • PaymentProcessorDataTypes
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of eight person-days. The assessment was conducted over the course of four calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau, Engineer
aaron@zellic.io

Junyi Wang, Engineer
junyi@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

April 17, 2023	Start of primary review period
April 21, 2023	End of primary review period

3 Detailed Findings

3.1 Voluntary marketplace fees

- **Target**: PaymentProcessor.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

Description

The marketplace and marketplace fees only need to be signed by the seller. If the seller does not want to pay the fee, the seller can simply remove the fee from the listing and sign. If the marketplace does not allow this listing to be transferred to the buyer if the fee is not paid in the listing, the seller and buyer can meet off the platform and perform the transaction without the fees.

Impact

The seller can choose to not pay the marketplace fees.

Recommendations

Document this fact prominently so that marketplace implementers can be aware of it. To enforce the marketplace fee, the marketplace may require both the buyer and seller to sign a transaction that includes the marketplace fee going to the marketplace.

Remediation

This issue has been acknowledged by Limit Break, and a fix was implemented in commit 911fdee4.

3.2 Token protocol is not signed

- **Target**: PaymentProcessor.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The contract is designed to support two NFT standards, namely ERC721 and ERC1155. These standards are usually not implemented at the same address, and therefore, for each transaction, the choice of standard must be explicitly specified. The transfer of the NFT from the seller to the buyer is then carried out using the appropriate API based on the passed standard. The variable that indicates the chosen standard is called the token protocol type in the code, and it is not signed by either the buyer or the seller.

Impact

In nearly every case, calling with an incorrect protocol will merely result in a revert, as the specified token address would not have the necessary functions implemented. This cannot be used to DOS, since a reverted transaction has no effects and the nonce would not be incremented. However, it is possible in extremely specific implementations where this could allow unwanted actions by a third party.

Recommendations

Consider making the seller and buyer sign the protocol type as well.

Remediation

This issue has been acknowledged by Limit Break, and a fix was implemented in commit f7ebf1f6.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Renounceable security policies

Note that a security policy must be renounced using the `PaymentProcessor.renounceSecurityPolicyOwnership` function before use to avoid centralization risk since otherwise the owner could update the security policy at any time after a `PaymentProcessor` struct has been signed.

Quoting from the `renounceSecurityPolicyOwnership` comments,

```
/**
     * @notice Allow security policy owners to transfer ownership of their
    security policy to the zero address.
     * This can be done to make a security policy permanently immutable.
```

Limit Break provided the following response to this note:

> This is intentional. Security policies should be mutable until an owner of a policy decides to lock them in. The fact that a security policy is mutable or immutable is clearly viewable on-chain. It is also noteworthy that the owner/admin of a collection can simply change from pointing to an immutable policy to a new mutable policy at will, so making policies immutable only has meaning if elevated privileges over a collection are also renounced.

4.2 Confusing function names

In PaymentProcessor, the `approveCoin` function name could be confused with token transfer approvals. We recommend renaming it to `whitelistCoin` or `allowCoin` (and, of course, consider renaming `disapproveCoin`).

Limit Break made this change in 7c37bd61.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Buyer and seller signed variables

In a given transaction, the buyer and seller must sign certain parameters of the transaction for it to be considered secure. For example, if the seller does not sign the price, the buyer could then execute an offer with a very low price.

The following fields identify the item under consideration and must be signed by both buyer and seller:

- `protocol`
- `tokenId`
- `amount`
- `tokenAddress`

The following field concerns the method of payment and must be signed by both buyer and seller:

- `paymentCoin`

This must be signed by both buyer and seller since it is used to enforce nonces:

- `marketplace`

The following fields must be signed by the seller:

- `seller`
- `listingMinPrice`
- `listingExpiration`
- `maxRoyaltyFeeNumerator`, since the payment to the marketplace comes out of the seller's share
- `privateBuyer`, since the seller tries to specify a specific buyer

The following fields must be signed by the buyer:

- `buyer`
- `offerPrice`
- `offerExpiration`

5.2 Security policies

For each transaction, all of the following security policies must be enforced:

- Exchange whitelist (if enabled)
- Payment methods whitelist (if enabled)
- Private listings allowed or not
- Delegated purchases allowed or not
- EIP1271 signatures allowed or not
- Amount of gas given when paying with native coin

5.3 Module: PaymentProcessor.sol

Function: `buySingleListing()`

Signed variables

The following fields must be signed by the seller:

- `protocol`
- `tokenId`
  - Verified at `_verifySignedItemListing:1379`
- `amount`
  - Verified at `_verifySignedItemListing:1380`
- `tokenAddress`
  - Verified at `_verifySignedItemListing:1378`
- `paymentCoin`
  - Verified at `_verifySignedItemListing:1390`
- `marketplace`
  - Verified at `_verifySignedItemListing:1371`
- `seller`
  - Verified at `_verifySignedItemListing:1377`
- `listingMinPrice`
  - Verified at `_verifySignedItemListing:1381`
- `listingExpiration`

  - Verified at `_verifySignedItemListing:1382`
- `maxRoyaltyFeeNumerator`
  - Verified at `_verifySignedItemListing:1373`
- `privateBuyer`
  - Verified at `_verifySignedItemListing:1374`

The variables take the following path:

- `saleDetails` in `buySingleListing`
  - `saleDetails` in `_executeMatchedOrderSale`
  - `saleDetails` in `_verifySignedItemListing`

The following fields must be signed by the buyer:

- `protocol`
- `tokenId`
  - Verified at `_verifySignedItemOffer:1254`
- `amount`
  - Verified at `_verifySignedItemOffer:1255`
- `tokenAddress`
  - Verified at `_verifySignedItemOffer:1253`
- `paymentCoin`
  - Verified at `_verifySignedItemOffer:1267`
- `marketplace`
  - Verified at `_verifySignedItemOffer:1250`
- `buyer`
  - Verified at `_verifySignedItemOffer:1252`
- `offerPrice`
  - Verified at `_verifySignedItemOffer:1256`
- `offerExpiration`
  - Verified at `_verifySignedItemOffer:1259`

The variables take the following path:

- `saleDetails` in `buySingleListing`
  - `saleDetails` in `_executeMatchedOrderSale`
  - `saleDetails` in `_verifySignedItemOffer`

Security policy

- Exchange whitelist (if enabled)
  - Enforced at `_executeMatchedOrderSale:992`

- Payment methods whitelist (if enabled)
  - Enforced at `_executeMatchedOrderSale:929`
- Private listings allowed or not
  - Enforced at `_executeMatchedOrderSale:967`
- Delegated purchases allowed or not
  - Enforced at `_executeMatchedOrderSale:977`
- EIP1271 signatures allowed or not
  - Enforced at `_executeMatchedOrderSale:982`
- Amount of gas given when paying with native coin
  - Used at `_executeMatchedOrderSale:1020`

Function: `buyBatchOfListings()`

Signed variables

Same as `buySingleListing()`, since it calls the same underlying function.

Security policy

Same as `buySingleListing()`, since it calls the same underlying function.

Function: `buyBundledListing()`

Signed variables

The following fields must be signed by the seller:

- `protocol`
- `tokenId`
  - Verified at `_verifySignedBundleListing:1347`
- `amount`
  - Verified at `_verifySignedBundleListing:1348`
- `tokenAddress`
  - Verified at `_verifySignedBundleListing:1335`
- `paymentCoin`
  - Verified at `_verifySignedBundleListing:1346`
- `marketplace`
  - Verified at `_verifySignedBundleListing:1331`
- `seller`
  - Verified at `_verifySignedBundleListing:1334`

- `listingMinPrice`
  - Verified at `_verifySignedBundleListing:1350`
- `listingExpiration`
  - Verified at `_verifySignedBundleListing:1338`
- `maxRoyaltyFeeNumerator`
  - Verified at `_verifySignedBundleListing:1349`
- `privateBuyer`
  - Verified at `_verifySignedBundleListing:1333`

The `marketplace`, `privateBuyer`, `seller`, `tokenAddress`, `listingExpiration`, and `paymentCoin` fields take the following path:

- `bundleDetails` in `buyBundledListing`
  - `bundleDetails` in `_validateBundledItems`
  - `bundleDetails` in `_verifySignedBundleListing`

The `tokenId`, `amount`, `maxRoyaltyFeeNumerator`, and `listingMinPrice` fields take the following path:

- `bundleItems` in `buyBundledListing`
  - `bundledOfferItems` in `_validateBundledItems`
  - `accumulatorHashes` in `_verifySignedBundleListing`

The following fields must be signed by the buyer:

- `protocol`
- `tokenId`
  - Verified at `_verifySignedOfferForBundledItems:1307`
- `amount`
  - Verified at `_verifySignedOfferForBundledItems:1308`
- `tokenAddress`
  - Verified at `_verifySignedOfferForBundledItems:1294`
- `paymentCoin`
  - Verified at `_verifySignedOfferForBundledItems:1306`
- `marketplace`
  - Verified at `_verifySignedOfferForBundledItems:1291`
- `buyer`
  - Verified at `_verifySignedOfferForBundledItems:1293`
- `offerPrice`
  - Verified at `_verifySignedOfferForBundledItems:1295`
- `offerExpiration`
  - Verified at `_verifySignedOfferForBundledItems:1298`

The `marketplace`, `buyer`, `tokenAddress`, `offerPrice`, `offerExpiration`, and `paymentCoin` fields take the following path:

- `bundleDetails` in `buyBundledListing`
  - `bundleDetails` in `_validateBundledOffer`
  - `bundledOfferDetails` in `_verifySignedOfferForBundledItems`

The `amount` and `tokenId` fields take the following path:

- `bundleItems` in `buyBundledListing`
  - `accumulator` from `_validateBundledItems`
  - `accumulator` in `_validateBundledOffer`
  - `tokenIdsKeccakHash` and `amountsKeccakHash` in `_verifySignedOfferForBundledItems`

Security policy

- Exchange whitelist (if enabled)
  - Enforced at `_validateBundledOffer:1077`
- Payment methods whitelist (if enabled)
  - Enforced at `_validateBundledOffer:1046`
- Private listings allowed or not
  - Enforced at `_validateBundledItems:1219`
- Delegated purchases allowed or not
  - Enforced at `_validateBundledOffer:1066`
- EIP1271 signatures allowed or not
  - Enforced for buyer at `_validateBundledOffer:1072`
  - Enforced for seller at `_validateBundledItems:1224`
- Amount of gas given when paying with native coin
  - Used at `buyBundledListing:622`
  - Used at `buyBundledListing:639`

Function: `sweepCollection()`

Signed variables

The following fields must be signed by the seller:

- `protocol`
- `tokenId`
  - Verified at `_verifySignedItemListing:1379`

- `amount`
  - Verified at `_verifySignedItemListing:1380`
- `tokenAddress`
  - Verified at `_verifySignedItemListing:1378`
- `paymentCoin`
  - Verified at `_verifySignedItemListing:1390`
- `marketplace`
  - Verified at `_verifySignedItemListing:1371`
- `seller`
  - Verified at `_verifySignedItemListing:1377`
- `listingMinPrice`
  - Verified at `_verifySignedItemListing:1381`
- `listingExpiration`
  - Verified at `_verifySignedItemListing:1382`
- `maxRoyaltyFeeNumerator`
  - Verified at `_verifySignedItemListing:1373`
- `privateBuyer`
  - Verified at `_verifySignedItemListing:1374`

The `marketplace`, `privateBuyer`, `tokenAddress`, and `paymentCoin` fields take the following path:

- `bundleDetails` in `buyBundledListing`
  - `bundleDetails` in `_validateBundledItems`
  - `saleDetails` in `_verifySignedItemListing`

The `tokenId`, `amount`, `maxRoyaltyFeeNumerator`, `listingMinPrice`, `seller`, and `listingExpiration` fields take the following path:

- `bundleItems` in `buyBundledListing`
  - `bundledOfferItems` in `_validateBundledItems`
  - `saleDetails` in `_verifySignedItemListing`

The following fields must be signed by the buyer:

- `protocol`
- `tokenId`
  - Verified at `_verifySignedOfferForBundledItems:1307`
- `amount`
  - Verified at `_verifySignedOfferForBundledItems:1308`
- `tokenAddress`
  - Verified at `_verifySignedOfferForBundledItems:1294`
- `paymentCoin`

  - Verified at `_verifySignedOfferForBundledItems:1306`
- `marketplace`
  - Verified at `_verifySignedOfferForBundledItems:1291`
- `buyer`
  - Verified at `_verifySignedOfferForBundledItems:1293`
- `offerPrice`
  - Verified at `_verifySignedOfferForBundledItems:1295`
- `offerExpiration`
  - Verified at `_verifySignedOfferForBundledItems:1298`

The `marketplace`, `buyer`, `tokenAddress`, `offerPrice`, `offerExpiration`, and `paymentCoin` fields take the following path:

- `bundleDetails` in `buyBundledListing`
  - `bundleDetails` in `_validateBundledOffer`
  - `bundledOfferDetails` in `_verifySignedOfferForBundledItems`

The `amount` and `tokenId` fields take the following path:

- `bundleItems` in `buyBundledListing`
  - `accumulator` from `_validateBundledItems`
  - `accumulator` in `_validateBundledOffer`
  - `tokenIdsKeccakHash` and `amountsKeccakHash` in `_verifySignedOfferForBundledItems`

Security policy

- Exchange whitelist (if enabled)
  - Enforced at `_validateBundledOffer:1077`
- Payment methods whitelist (if enabled)
  - Enforced at `_validateBundledOffer:1046`
- Private listings allowed or not
  - Enforced at `_validateBundledItems:1190`
- Delegated purchases allowed or not
  - Enforced at `_validateBundledOffer:1066`
- EIP1271 signatures allowed or not
  - Enforced for buyer at `_validateBundledOffer:1072`
  - Enforced for seller at `_validateBundledItems:1195`
- Amount of gas given when paying with native coin
  - Used at `sweepCollection:705`
  - Used at `sweepCollection:722`

6 Audit Results

At the time of our audit, the audited code was not deployed to mainnet EVM.

During our assessment on the scoped Limitbreak Payment Processing Layer contracts, we discovered two findings. No critical issues were found. One was of low impact and the remaining finding was informational in nature.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.