Zellic

Arpeggi V2

Smart Contract Security Assessment

April 19, 2022

Prepared for:

Alec Papierniak

Arpeggi Labs

Prepared by:

Chad McDonald and Konstantin Nikolayev

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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About Arpeggi V2

Arpeggi is a Web3 music creation suite that enables musicians to create and mint samples, stems and songs on-chain using a browser-based digital audio workstation (DAW). Arpeggi plans to launch Arpeggi Studio v2 on the Polygon network.

We were approached to audit Arpeggi's on-chain contracts, which are centered around two core pieces of technology: the Web3 music creation tool of Arpeggi Studio and the underlying sound library of the Audio Registry Protocol (ARP), which allows all music to be a part of this ecosystem.

1.2 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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of

the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

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

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than 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 numerous soft factors, such as our clients' threat models, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Arpeggi Contracts

Repository	https://github.com/Arpeggi-Labs/arpeggi-contracts
Versions	`93f9a9fe20e26eec308d16eaf3a9b62fe6d762c3`
Contracts	• ArpeggiSong • ArpeggiSound • AudioRegistryProtocol
Type	Solidity
Platform	Polygon

1.4 Project Overview

Zellic was approached to perform an assessment with two consultants, for a total of 3 person-days.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Chad McDonald, Engineer
chad@zellic.io

Konstantin Nikolayev, Engineer
nyanko@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

Arpil 6, 2022	Kick-off call
April 6, 2022	Start of primary review period
April 9, 2022	End of primary review period
April 19, 2022	Closing call

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

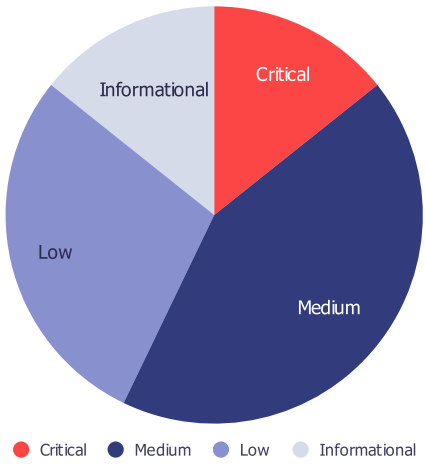
Zellic conducted an audit for Arpeggi Labs from April 6th to April 9th, 2022 on the scoped contracts and discovered 7 findings. The audit uncovered 1 finding of critical impact, 3 of medium impact, and 2 of low impact. The rest of the findings were informational in nature. The most critical issue was the ability to break minting.

Arpeggi Labs is developing version 2 of their on-chain digital audio workstation: Arpeggi Studio. Arpeggi Studio enables users to compose with others' music, while still providing attribution and value to the original creators. This functionality is centered around two core pieces of technology: the Arpeggi Studio, a browser-based DAW, and the Arpeggi Registry Protocol—an open protocol to give proper attribution for samples, stems and songs.

Zellic was approached to audit ArpeggiSong, ArpeggiSound, AudioRegistryProtocol and the accompanying auxiliary code. Our general overview of the code is that the codebase is still in active development and as such there are improvements in documentation and readability that have yet to be implemented. However, the core functionality of the contracts is generally easy to follow and reason about.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	3
Low	2
Informational	1



3 Detailed Findings

3.1 An attacker can break minting of `ArpeggiSound` and `ArpeggiSong` tokens

- **Target**: ArpeggiSound, ArpeggiSong, AudioRegistryProtocol
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

`ArpeggiSound.mintSample`, `ArpeggiSound.mintStem` and `ArpeggiSong.mintSong` are vulnerable.

We will use `ArpeggiSong.mintSong` as an example to demonstrate the issue, but everything below applies to `ArpeggiSound.mintSample` and `ArpeggiSound.mintStem` as well.

When a new song NFT is minted, the following occurs.

First, `mintSong` mints a new NFT by calling `_safeMint`. Second, `mintSong` creates an origin token[1]:

```
AudioRegistryTypes.OriginToken memory originToken = AudioRegistryTypes.
    OriginToken({
    tokenId: numSongs,
    chainId: block.chainid,
    contractAddress: address(this),
    originType: AudioRegistryTypes.OriginType.PRIMARY // primary
});
```

Third, `mintSong` passes the origin token to the `AudioRegistryProtocol.registerMedia` function. Fourth, `registerMedia` creates a new media ID and attempts to tie the newly minted NFT to it.

If the newly minted NFT is already tied to a media ID, the attempt fails and the transaction is reverted.[2]

Anyone can register a new media ID and tie an **unminted** NFT to it, simply by calling

[1] The origin token is simply a wrapper around the newly minted NFT.

[2] Unless the caller of `registerMedia` can pass the checks in the `enforceOnlyOverwriteAuthorized` function. The contract that issued the already tied NFT fails to pass the checks.

`registerMedia` with appropriate parameters. There are **no checks** that prevent that.

Therefore, anyone can break minting by registering a new media ID and tying a next-to-be-minted unminted NFT to it.

Impact

An attacker can break minting of `ArpeggiSound` and `ArpeggiSong` tokens.

Recommendations

Consider disallowing the registration of unminted NFTs.

Remediation

We provided a proof-of-concept to Arpeggi Labs. This issue was fixed by Arpeggi Labs in commit cc29275.

3.2 Potentially unsafe reentrancy in the minting functions

- **Target**: ArpeggiSong, ArpeggiSound
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

Arpeggi Studio allows users to mint samples, stems and songs and use them in the digital audio workstation. Samples are the smallest "units" of sound (think of a hand-clap sound effect in a song) in the Arpeggi ecosystem. A stem is a single track of a song. It is created by sequencing one or more samples into a pattern. A song is composed of multiple stems.

When a user creates music in Arpeggi Studio and is ready to mint a song, the Arpeggi Studio webapp processes the music and mints to the contract via various functions:

- `ArpeggiSound.mintSample`
- `ArpeggiSound.mintStem`
- `ArpeggiSong.mintSong`

There is a reentrancy issue in all of the 3 functions above. We will focus on `ArpeggiSound.mintSample` for the rest of this example. Below is a code snippet from `mintSample`:

```
function mintSample(
    uint version,
    address artistAddress,
    address tokenOwner,
    string calldata dataUri,
    string calldata metadataUri
)
    external payable whenNotPaused
    returns (uint256)
{
    _numSounds++;
    uint numSounds = _numSounds;

    _safeMint(tokenOwner, numSounds);

    // registration logic is below
```

In `mintSample`, the state variable `_numSounds` is incremented each time before a new

ERC721 token is minted. After `_numSounds` is incremented, the token is minted through `_safeMint` and a call is made to `AudioRegistryProtocol.registerMedia` to register the token's metadata in the `AudioRegistryProtocol` contract.

A reentrancy attack is potentially possible because the increment of `_numSounds` happens without checking if `_numSounds` has already been minted. Furthermore, the call to `_safeMint` happens before any of the registration logic is executed.

Impact

This reentrancy issue allows an arbitrary amount of tokens to be minted in a way that breaks the expected `mediaId`-to-`tokenId` metadata storage schema for sample, stem and song tokens.

For example, using reentrancy to mint 10 tokens results in this:

```
token.contractAddress = 0xcf7 ... , mediaId = 1, token.tokenId = 10
token.contractAddress = 0xcf7 ... , mediaId = 2, token.tokenId = 9
token.contractAddress = 0xcf7 ... , mediaId = 3, token.tokenId = 8
token.contractAddress = 0xcf7 ... , mediaId = 4, token.tokenId = 7
token.contractAddress = 0xcf7 ... , mediaId = 5, token.tokenId = 6
token.contractAddress = 0xcf7 ... , mediaId = 6, token.tokenId = 5
token.contractAddress = 0xcf7 ... , mediaId = 7, token.tokenId = 4
token.contractAddress = 0xcf7 ... , mediaId = 8, token.tokenId = 3
token.contractAddress = 0xcf7 ... , mediaId = 9, token.tokenId = 2
token.contractAddress = 0xcf7 ... , mediaId = 10, token.tokenId = 1
```

Here, the minted tokens have their `mediaId` and `tokenId` values out of sync.

Recommendations

We recommend that Arpeggi follows the checks-effects-interactions pattern by moving the increment of `_numSounds` and the call to `_safeMint` after the registration logic at the end of the function. This will ensure that `_numSounds` is accurate and that the associated metadata is correct if `mintSample` is reentered.

In addition to this, Arpeggi can make use of OpenZeppelin's `ReentrancyGuard` contract to add a `nonReentrant` modifier to all of the minting functions.

Remediation

We provided a proof-of-concept to Arpeggi Labs. This issue was fixed by Arpeggi Labs in commit 52cef08.

3.3 Payable functions exist with no way to withdraw funds

- **Target**: ArpeggiSong, ArpeggiSound
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

The mint functions: `mintSample`, `mintStem` and `mintSong` are declared payable, but there is no function to withdraw funds.

Recommendations

The Arpeggi team stated that users will not pay for minting, so we recommend removing the `payable` modifier from these functions.

Remediation

This issue was fixed by Arpeggi Labs in commit 996c882.

3.4 Origin token registration may result in a collision

- **Target**: AudioRegistryProtocol
- **Category**: Business Logic
- **Severity**: Medium
- **Likelihood**: n/a
- **Impact**: Medium

Description

If an origin token `t1` is registered and there is an attempt to register another origin token `t2`, such that `t1.contractAddress == t2.contractAddress` and `t1.tokenId == t2.tokenId`, a collision happens: `t1` gets overwritten by `t2` (in case the caller of `registerMedia` passes the checks in `enforceOnlyOverwriteAuthorized`) or the entire transaction gets reverted (otherwise).

Impact

It is impossible to register 2 or more origin tokens with identical `contractAddresses` and `tokenIds`, but different `chainIds` or `originTypes`.

Recommendations

Consider replacing the `_contractTokensToArpIndex` mapping with an "origin token"-to-"media ID" mapping and reorganizing the code accordingly.

Remediation

This issue was fixed by Arpeggi Labs in commit bd3a6ec.

3.5 The access control list for the Arpeggi admin role cannot be changed

- **Target**: ArpeggiSound, ArpeggiSong
- **Category**: Business Logic
- **Likelihood**: n/a
- **Severity**: Low
- **Impact**: Low

Description

The `ArpeggiSound` and `ArpeggiSong` contracts do not set an admin role for `ARPEGGI_ADMIN_ROLE`.

Impact

It is impossible to change the access control list for `ARPEGGI_ADMIN_ROLE`.

Recommendations

Consider adding the following code to the constructors of `ArpeggiSound` and `ArpeggiSong`:

```
_setRoleAdmin(Roles.ARPEGGI_ADMIN_ROLE, Roles.ARPEGGI_ADMIN_ROLE);
```

Remediation

This issue was fixed by Arpeggi Labs in commit 67f8be0.

3.6 The UPGRADER_ROLE role is defined, but never used

- **Target**: AudioRegistryProtocol
- **Category**: Business Logic
- **Likelihood**: n/a
- **Severity**: Low
- **Impact**: Low

Description

`UPGRADER_ROLE` is defined in `AudioRegistryProtocol.sol` at L12, but this role is never used anywhere. We assume that `UPGRADER_ROLE` was intended to be used in the `_authorizeUpgrade` function, but `_authorizeUpgrade` uses `DEFAULT_ADMIN_ROLE` instead:

```
function _authorizeUpgrade(address newImplementation)
    internal
    onlyRole(DEFAULT_ADMIN_ROLE)
    override
{}
```

Impact

The members of `UPGRADER_ROLE` are not given permission to upgrade the `AudioRegistryProtocol` contract.

Recommendations

Consider modifying `_authorizeUpgrade` to replace `DEFAULT_ADMIN_ROLE` with `UPGRADER_ROLE`:

```
function _authorizeUpgrade(address newImplementation)
    internal
    onlyRole(UPGRADER_ROLE)
    override
{}
```

Remediation

This issue was fixed by Arpeggi Labs in commit 00524c4.

3.7 Unbounded for-loop can lead to out-of-gas

- **Target**: AudioRegistryProtocol
- **Category**: Code Maturity
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Informational

Description

There is an unbounded for-loop in `AudioRegistryProtocol.registerMedia`:

```
if(subComponents.length > 0){
    for(uint i = 0; i < subComponents.length; i++){
        require(subComponents[i] ≤ _numMedia, “ARP: Invalid
    subcomponent.”);
    }
}
```

If `subComponents.length` is sufficiently large, this for-loop runs out of gas.

Our tests show that the current upper limit for the length of the `subComponents` array is around 1,315. The current 30,000,000 per-block gas limit of the Polygon network is reached after that.

Impact

The unbounded for-loop can raise an out-of-gas exception if the `subComponents` array contains more than one thousand elements.

The impact of this finding is mitigated by the fact that it is unlikely for a piece of media to be comprised of such a large number of subcomponents.

Recommendations

This issue can be remedied by forcing the frontend to impose a limit on the length of the `subComponents` array.

Remediation

This issue was fixed by Arpeggi Labs in commit a7fa6de.

4 Discussion

In this section, we discuss miscellaneous interesting observations discovered during the audit that are noteworthy and merit some consideration.

The core functionality of the Arpeggi contracts is centered around the 3 minting functions (`mintSong()`, `mintStem()`, `mintSample()`) and the metadata registration process (implemented in the `registerMedia()` function). Since minting is a major part of the core functionality, it makes sense to optimize the minting functions for maximum gas efficiency. Currently, each token is minted using a separate message call. This is fine for a song with only a few samples, but it may become a nuisance for users who want to mint a large number of tokens. It may make sense to consider modifying the minting functions so that it is possible to mint multiple tokens in a single message call.

ERC721A by the Azuki NFT development team is an implementation of `IERC721` that enables minting multiple NFTs in a single transaction. Arpeggi expressed interest in integrating the ERC721A standard into their contracts in order to be able to capitalize on the increased gas efficiency of minting. Since the default `ERC721A` contract is not upgradeable, it is unsuitable as a drop-in replacement for OpenZeppelin's `ERC721Upgradeable` contract. However, there is an open pull request in the chiru-labs GitHub repo that contains a proposal to merge an upgradeable version of the `ERC721A` contract. This pull request may be of interest to the Arpeggi team.