Zellic

SPL Token 2022

Smart Contract Security Assessment

December 5, 2022

Prepared for:

Solana Foundation

Prepared by:

Filippo Cremonese and Jasraj Bedi

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 Executive Summary

Zellic conducted an audit for Solana Foundation from September 19th to October 7th, 2022.

Our general overview of the codebase is that the migration of the code to support both legacy token accounts as well as the new, extensible form is relatively robust. The implementation appears to be devoid of major issues in the core of the token program, though some critical issues were identified in the extensions themselves. It appears that a robust platform on which to enhance the functionality of Solana token accounts has been built, though care should be taken in the implementation of the extension itself.

Zellic thoroughly reviewed the SPL Token 2022 codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

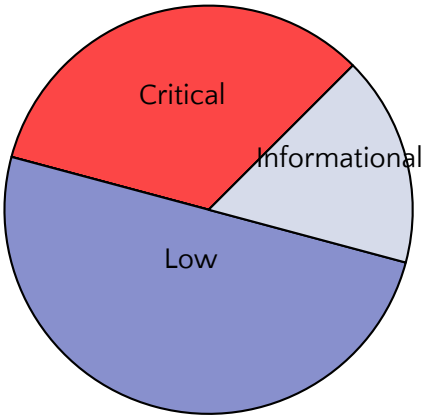
Specifically, taking into account SPL Token 2022's threat model, we focused heavily on issues that would break core invariants such as proper account serialization and deserialization, mint accounting, and violations in the guarantees offered by the new extensions.

During our assessment on the scoped SPL Token 2022 contracts, we discovered seven findings. Critical issues were identified. Of the six findings, two were critical, three were of low severity, and the remaining finding was informational in nature.

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

Breakdown of Finding Impacts

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



2 Introduction

2.1 About SPL Token 2022

SPL Token 2022 is a collection of on-chain programs targeting the Sealevel parallel runtime. These programs are tested against Solana's implementation of Sealevel, Solana runtime, and are deployed to its mainnet.

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 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, and so forth 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, and so forth.

Complex integration risks. Several high-profile 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. 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, and so forth.

Code maturity. We review for possible improvements in the codebase 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 forth.

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 being 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 numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

SPL Token 2022 Contracts

Repository	https://github.com/solana-labs/solana-program-library
Versions	`54695b233484722458b18c0e26ebb8334f98422c`
Programs	• SPL Token 2022
Type	Rust
Platform	Solana

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of five person-weeks. The assessment was conducted over the course of three calendar weeks.

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:

Filippo Cremonese, Engineer
fcremo@zellic.io

Jasraj Bedi, Co-founder, Engineer
jazzy@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 19, 2022	Start of primary review period
October 7, 2022	End of primary review period

3 Detailed Findings

3.1 Missing check in `process_transfer` leading to inflationary bug

- **Target**: Confidential Transfer Extension
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

Transfers between confidential accounts require a zero knowledge (ZK) argument proving that the source account balance is greater than the transferred amount and that the transferred amount is not negative.

Confidential token transfer transactions consist of two instructions. The first required instruction contains a cryptographic ZK argument that proves the validity of the transfer without disclosing any information about the balances involved or the transferred amount. The other instruction performs the computations and updates the account state to actually perform the transfer.

The ZK argument instruction is processed by a special built-in program that verifies its validity, reverting if validation fails. More specifically, the ZK argument is an equation in which some variables have values that correspond to the state of the accounts involved in the transaction.

The other instruction is processed by the token program. The program verifies that the instruction containing the ZK argument exists and that its inputs are consistent with the state of the involved accounts, tying the ZK argument to the state of the blockchain.

The token program does not correctly verify all the ZK argument inputs. One of the fields associated with the ZK argument, `new_source_ciphertext`, is ignored. This field contains the expected value of the source account encrypted balance after the transfer is performed. The lack of this check implies that the source account encrypted balance is not validated. This effectively decouples the ZK argument from the balance of the source account.

Impact

A malicious transaction constructed to exploit the issue allows to perform repeated transfers, totalling an amount bigger than the source account encrypted balance.

We created a proof-of-concept exploit by constructing a transaction with multiple

instructions performing a transfer, all referencing the same instruction containing the ZK argument.

The source account encrypted balance underflows and becomes invalid, but the destination account encrypted pending balance is credited multiple times, creating tokens out of nothing and inflating the supply. The supply inflation will not be reflected by the information stored in the mint account associated with the token. The destination account is able to apply the pending balance and make use of the unfairly obtained amount normally.

The PoC would perform the following operations:

```
[!] Starting double transfer PoC
[!] Current balances:
Alice:
    - available balance: 42
    - pending balance: 0
Bob:
    - available balance: 0
    - pending balance: 0

[!] Running malicious transaction. Instructions:
- Instruction 0: TransferWithFeeData instruction
    - amount: 42
- Instruction 1: ConfidentialTransferInstruction::Transfer instruction
- Instruction 2: ConfidentialTransferInstruction::Transfer
    instruction (repeated)

[!] Current balances:
Alice: could not decrypt balances
Bob:
    - available balance: 0
    - pending balance: 84

[!] Applying Bob pending balance
[!] Current balances:
Alice: could not decrypt balances
Bob:
    - available balance: 84
    - pending balance: 0
```

Recommendations

Ensure that the source account encrypted balance corresponds to the expected amount contained in the ZK argument (the `new_source_ciphertext` field of the `TransferData` struct).

Remediation

The Solana Foundation team was alerted of this finding while the audit was ongoing. The team quickly confirmed the issue and submitted a remediation patch for our review. The patch correctly implements the suggested remediation.

Pull request #3867 fixes the issue following our recommendation. The PR head commit `c7fbd4b` was merged in the `master` branch on December 3, 2022.

The confidential token transfer extension was not used at the time the audit was conducted; therefore, no funds were at risk.

3.2 Missing check in `process_withdraw` potentially leading to inflationary bug

- **Target**: Confidential Transfer Extension
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

Withdrawals from a token account confidential balance to its cleartext balance require a zero knowledge (ZK) argument that proves that the account encrypted balance is greater than the withdrawn amount.

Confidential withdraw transactions consist of two instructions. One contains the aforementioned ZK argument and is processed by a special built-in program that verifies its validity, reverting the transaction in case of failure. The other instruction, processed by SPL Token 2022, performs the operations on the balances to actually accomplish the withdrawal. The token program verifies that the instruction containing the ZK argument exists and that its inputs are consistent with the state of the involved accounts, tying the ZK argument to the state of the blockchain.

The token program does not correctly verify that the public key associated with the ZK argument corresponds to the public key associated to the source account encrypted balance. This potentially allows an attacker to forge a ZK argument asserting the validity of any desired withdrawal amount, regardless of the actual encrypted balance of the source account.

Refer to 5 for more information on the equations implementing the ZK argument.

Impact

An attacker might be able to exploit this issue and withdraw an arbitrary amount of tokens to their cleartext balance, creating tokens from nothing and inflating the supply. Note that the supply inflation will not be reflected by the information stored in the mint account associated with the token. The plaintext balance is spendable, exactly like any other regular plaintext balance on a legitimate account.

We did not fully confirm exploitability of this issue, but the team agreed that it is likely possible to forge a malicious ZK equality argument.

Recommendations

Ensure that the public key associated with the source account corresponds to the public key associated with the ZK argument (the `pubkey` field of the `WithdrawData` struct).

Remediation

The Solana Foundation team was alerted of this finding while the audit was ongoing. The team quickly helped confirm the issue.

Pull request #3768 fixes the issue following our recommendation. The PR head commit `94b912a` was merged in the `master` branch on October 27, 2022.

The confidential token transfer extension was not used at the time the audit was conducted; therefore, funds were not at risk.

3.3 Missing public key check in `EmptyAccount` leading to deflationary bug

- **Target**: Confidential Transfer Extension
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: High
- **Impact**: Low

Description

A token account can only be closed if it has a zero balance. This applies to the regular cleartext balance as well as to the balances managed by the confidential transfer extension. Since the latter balances are encrypted, a special instruction called `EmptyAccount` has to be executed before closing the account, which enables closing the account after verifying a zero knowledge (ZK) argument that proves the account balance is zero.

Similarly to other confidential token operations, a ZK argument has to be embedded in an instruction in the same transaction that invokes `EmptyAccount`. The processor for `EmptyAccount` verifies that the ZK argument exists and that it is correctly tied to the current state of the blockchain.

The function processing the `EmptyAccount` instruction does not check that the public key associated with the ZK argument corresponds to the public key of the token account to be closed. This might allow an attacker to forge a ZK argument, falsely showing the account balance to be zero.

Impact

By closing an account with a nonzero balance, an attacker would be able to decrease the circulating supply without causing an update to the supply information stored in the mint account.

The attacker would have to give up their balance; therefore, it is difficult to imagine an incentive to perform such an attack. Furthermore, the same effect could be obtained by simply keeping the tokens in the attacker's account. For this reason, this issue is classified as low likelihood and low impact.

Recommendations

Ensure that the public key associated with the proof corresponds to the public key of the account being closed.

Remediation

Pull request #3767 fixes the issue following our recommendation. The PR head commit `d6a72eb` was merged in the `master` branch on October 27, 2022.

3.4 Confidential transfer amounts information leak via transfer fees

- **Target**: Confidential Transfer Extension
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Low
- **Impact**: Low

Description

Tokens managed by SPL Token 2022 can be configured to require a transfer fee consisting of a percentage of the transferred amount (with the possibility to cap the maximum fee at a fixed amount). This configuration also applies to confidential transfers, relying on zero-knowledge cryptographic arguments to prove the validity of the encrypted balances being manipulated.

Information about the value of every transfer is leaked to the owner of the keys controlling the transfer fees for the mint.

Impact

The owner of the private key associated with management of the transfer fees can gather information on the value of confidential transfers. Since the key is able to decrypt the fee balance before and after the transfer has occurred, the fee amount for every transfer can be obtained. If the fee is lower than the cap amount, then the exact transferred amount can be inferred. Otherwise, the transferred amount is guaranteed to be at least as big as the minimum amount that would require the maximum fee.

Recommendations

Completely blinding the transfer fee amounts appears to be challenging and likely to require a significant engineering effort. If this information leak is accepted, we suggest to inform SPL token developers and users of this privacy pitfall of confidential transfers involving fees.

Remediation

Pull request #3773 addresses the issue by adding more documentation on the confidential transfer extension code, acknowledging the potential information leak if a confidential transfer with fees is performed. The PR head commit `1c3af5e` was merged in the `master` branch on October 28, 2022.

3.5 Confidential transfer fees' withdrawal instructions ignore constraints

- **Target**: Confidential Transfer Extension
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

The functions handling the confidential transfer instructions `WithdrawWithheldTokensFromAccounts` and `WithdrawWithheldTokensFromMint` ignore some of the restrictions that can be applied to confidential token accounts:

- `allow_balance_credits`: An account can be configured to deny credits to its pending balance.
- `pending_balance_credit_counter`: This value should be checked not to be greater than `maximum_pending_balance_credit_counter`. The instructions also do not increment `pending_balance_credit_counter`.

We note that these instructions directly add the entire value of the withheld balance to the `pending_balance_lo` of the destination account. This could potentially cause the pending balance to become bigger than 2^{16} or even 2^{32}, making decryption of the balance difficult.

Impact

An attacker with control of the keys trusted with managing transfer fees could credit the encrypted pending balance of an account bypassing the configuration applied by the account owner and potentially make it difficult for the victim to decrypt the encrypted balance.

Recommendations

- Revert the transaction if `allow_balance_credits` is set on the destination account.
- Revert the transaction if `pending_balance_credit_counter` is not less than `maximum_pending_balance_credit_counter`. Increment `pending_balance_credit_counter` after the transfer taken place.

Since the value of the transferred balances is encrypted, limiting the transferred value to avoid overflowing the soft amount of 2^{32} is challenging and would require extensive modifications.

Remediation

Pull request #3774 fixes the issue following our recommendation. The PR head commit `16384e2` was merged in the `master` branch on October 28, 2022.

The confidential token transfer extension was not used at the time the audit was conducted; therefore, funds were not at risk.

3.6 Confidential token account public key is not validated

- **Target**: Confidential Transfer Extension
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: High
- **Impact**: Informational

Description

A confidential token account can be initialized with an arbitrary 32-byte buffer as the public key used to encrypt the account balances. Proving knowledge of the associated private key is not required for initializing a confidential transfer account. Legitimate public keys are derived as $P = kH$, meaning they are an element of the group generated by H. The lack of validation allows to set a potentially invalid public key that can be generically written as $P = aH + bG$, opening an opportunity for attacks leveraging the invalid pubkey.

Impact

Due to the limited amount of time available for the audit, we could not fully evaluate the exploitability of this issue. It appears that controlling an account public key is not enough to forge a ciphertext-commitment equality argument (refer to 5). However, we consider not validating the public key a risk, potentially opening the opportunity for critical exploits.

Recommendations

Require proof of knowledge of the secret key associated with the public key when initializing a confidential token account.

Remediation

Pull request #28392 on Solana's zk-token-sdk adds a public key validity proof to the ZK token SDK.

Pull request #3784 integrates the public key validity proof with the instruction that initializes the confidential transfer extension for an account. The PR head commit `abc77af` was merged in the `master` branch on October 30, 2022.

The confidential token transfer extension was not used at the time the audit was conducted; therefore, funds were not at risk.

4 Discussion

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

4.1 Behavior of multisig accounts with repeated signers

During our audit we observed that it is possible to initialize a multisig account with the same signer repeated multiple times.

The function used for validating the signers of a transaction, `validate_owner`, counts a signer of a transaction as having signed it the exact number of times the account is repeated, regardless of how many times the account appears in the list of signers. Despite this being reasonable behavior, it is unclear if this specific edge case was considered.

The Solana Foundation engineering team acknowledged this discussion point.

4.2 Unclear purpose for some confidential account fields

The `actual_pending_balance_credit_counter` and `expected_pending_balance_credit_counter` fields contained in the `ConfidentialTransferAccount` struct seem largely unused, and their purpose is unclear.

The Solana Foundation engineering team acknowledged this discussion point.

4.3 Unnecessary initialization check in `_process_initialize_mint`

When `_process_initialize_mint` calls `StateWithExtensionMut::<Mint>::unpack_uninitialized( ... )`, a check is performed internally that validates that the unpacked account is not initialized. This specifically validates that the base field is not initialized. As this condition is validated, the immediately proceding check is unnecessary and can never be true.

```
if mint.base.is_initialized {
    return Err(TokenError::AlreadyInUse.into());
}
```

The Solana Foundation engineering team addressed this discussion point and removed the redundant check in pull request #3763. The head commit of the PR `d331f47` was merged in the `master` branch on October 26, 2022.

4.4 Confusing account owner check

Account ownership checks are performed using a couple of different patterns throughout the codebase. The main two checks are via `validate_owner( ... )` and `check_program_account( ... )`, which work well. One area of confusion, though, is that calling `get_required_account_extensions_from_unpacked_mint( ... )` performs an account ownership check. It is suggested that this check be migrated into `StateWithExtensions::unpack` and `StateWithExtensionsMut::unpack`. The SWE and SWEM unpack methods are only used with accounts that the program should own and would increase the resiliency of the codebase by mitigating a potential future situation where an account or mint is unpacked and data are used for some decision but program ownership failed to be checked.

The Solana Foundation engineering team acknowledged this discussion point.

4.5 Suggested alternative extension dedup

The implementation of `get_total_tlv_len( ... )` takes the slice of provided `extension_types` and deduplicates the collection before summing each of their `tlv_lens`. The implementation uses a for loop that calls `Vec::contains( ... )`. For small number of extensions this may be okay, but in calls to realloc, this could result in slow code if callers are inefficient with passed up extension (offering duplicates, for example).

A suggested alternative would be to use a `HashSet` here. For performance, `FxHashSet` may be used if solving for this situation is desired.

The Solana Foundation engineering team acknowledged this discussion point; their testing shows that for vectors of size 76 and below, `Vec::contains` is more efficient than `HashSet`.

5 Appendix A: ZK equality argument equation

The following paragraph complements finding 3.2 and describes the equation that implements one of the two ZK arguments involved in withdrawal transactions. This ZK argument asserts that the plaintext value of a given ciphertext (consisting of a Pedersen commitment and decryption handle) is the same as that of a different Pedersen commitment.

Prerequisite definitions:

- O is the point at infinity
- G is the Ristretto base point
- H is the auxiliary point used for construction of the Pedersen commitment scheme
- public keys P_i are represented as $P_i = a_i H + b_i G$
  - legitimate public keys are derived from a secret k_i as $P_i = k_i^{-1} H$, so $a_i = k_i^{-1}$ and $b_i = 0$
  - rogue public keys could have a nonzero $b_i G$ term
- encrypted values are represented as a tuple $E_i = (C_i; D_i)$, where
  - $C_i = m_i G + r_i H$
  - $D_i = r_i P$
  - m_i is the plaintext
  - r_i is an arbitrary (normally random) scalar
  - P is the public key of the recipient of the ciphertext

The equality argument is constructed, referring to

- the legitimate public key $P_L = a_L H + b_L G$
  - this is the pubkey associated with the confidential account
  - b_L might not be zero when leveraging issue 3.6
- the rogue public key $P_R = a_R H + b_R G$
  - this public key can differ from the legitimate public key when leveraging issue 3.2
- the encrypted balance $E_S = (C_S; D_S) = (m_S G + r_S H; r_S P_1)$
- the Pedersen commitment $C_D = m_D G + r_D H$

The values provided must satisfy the following relationship:

$$z_S P_R - cH - Y_0 + wz_X G + wz_S D_S - wcC_S - wY_1 + w^2 z_X G + w^2 z_R H - w^2 cC_D - w^2 Y_2 = O$$

The following additional values have been introduced:

- witness variables c and w
  - the relationship must hold for any possible value of these variables
- scalars z_S, z_X and z_R
  - these scalars are completely free and can be set arbitrarily
- points $Y_i = p_i H + q_i G$, for $i \in 0, 1, 2$
  - these points cannot depend on the value of any witness variable

The relationship can be expanded by expressing all the points as a sum of their two components xH and yG:

$$(z_S a_R - c - p_0 + w z_S r_S a_L - w c r_S - w p_1 + w^2 z_R - w^2 c r_D - w^2 p_2) H +$$
$$(z_S b_R - q_0 + w z_X + w z_S r_S b_L - w c m_S - w q_1 + w^2 z_X - w^2 c m_D - w^2 q_2) G = O$$

We can set up the following system of equations from the two scalar components of the relationship:

$$\begin{cases} z_S a_R - c - p_0 + w z_S r_S a_L - w c r_S - w p_1 + w^2 z_R - w^2 c r_D - w^2 p_2 \equiv 0 \pmod{ord(H)} \\ z_S b_R - q_0 + w z_X + w z_S r_S b_L - w c m_S - w q_1 + w^2 z_X - w^2 c m_D - w^2 q_2 \equiv 0 \pmod{ord(G)} \end{cases}$$

The system can be manipulated to obtain the following,

$$\begin{cases} (z_R - c r_D - p_2) w^2 + (z_S r_S a_L - c r_S - p_1) w + (z_S a_R - c - p_0) \equiv 0 \pmod{ord(H)} \\ (z_X - c m_D - q_2) w^2 + (z_S r_S b_L - c m_S - q_1 + z_X) w + (z_S b_R - q_0) \equiv 0 \pmod{ord(G)} \end{cases}$$

which is satisfied if the following system of equations is satisfied:

$$\begin{cases} z_R - c r_D - p_2 \equiv 0 \pmod{ord(H)} \\ z_S r_S a_L - c r_S - p_1 \equiv 0 \pmod{ord(H)} \\ z_S a_R - c - p_0 \equiv 0 \pmod{ord(H)} \\ z_X - c m_D - q_2 \equiv 0 \pmod{ord(G)} \\ z_S r_S b_L - c m_S - q_1 + z_X \equiv 0 \pmod{ord(G)} \\ z_S b_R - q_0 \equiv 0 \pmod{ord(G)} \end{cases}$$

6 Audit Results

At the time of our audit, the code was partially deployed to mainnet Solana.

During our audit, we discovered six findings. Of these, two were critical, three were low risk, and one was informational. Solana Foundation acknowledged all findings and implemented fixes.

6.1 Disclaimers

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 additional code added to the assessed 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 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.