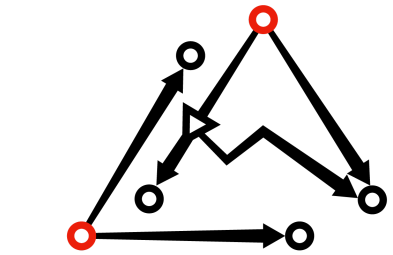

Least Authority

PRIVACY MATTERS

.bit Contracts

Security Audit Report

d.id

Final Audit Report: 10 April 2024

Table of Contents

Overview

Background

d.id has requested that Least Authority perform a security audit of their .bit Contracts.

Project Dates

- **February 6, 2024 - March 4, 2024**: Initial Code Review *(Completed)*
- **March 6, 2024**: Delivery of Initial Audit Report *(Completed)*
- **April 9, 2024:** Verification Review *(Completed)*
- **April 10, 2024:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nikolaos D. Bougalis, Security Researcher and Engineer
- Mehmet Gönen, Cryptography Researcher and Engineer
- Jasper Hepp, Security Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the .bit Contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- das-contracts: https://github.com/dotbitHQ/das-contracts-private
- das-lock: https://github.com/dotbitHQ/das-lock

Specifically, we examined the Git revisions for our initial review:

- das-contracts: `448ceed0f13b315e44e70f4cad1b12a187eb3fc0`
- das-lock: `c425807be11cd8a44ea65ec51f8e0b56965060e2`

For the verification, we examined the Git revisions:

- das-contracts: `33f299ffaf66ab12c6a2b4bd302a759a5729545b`
- das-lock: `0b96a9c75d5c2d432ecac2e2b72ee2cf0fef240c`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- das-contracts: https://github.com/LeastAuthority/das-contracts
- das-lock: https://github.com/LeastAuthority/das-lock

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Docs:
  https://github.com/dotbitHQ/das-contracts/tree/docs/docs
- User documentation:
  https://community.d.id/c/get-started
- Blog:
  https://blog.d.id
- Knowledge Base:
  https://community.d.id/c/knowledge-base-bit
- .bit Reward System:
  https://talk.did.id/t/bits-reward-system/631
- d.id Media Kit:
  https://dotbit.notion.site/d-id-Media-Kit-56bde70bcc554af0a975ad67c96c32bc

In addition, this audit report references the following documents:

- Nervos Network:
  https://www.nervos.org
- Cell Model:
  https://medium.com/nervosnetwork/https-medium-com-nervosnetwork-cell-model-7323fca57571
- CKB Transaction Structure:
  https://github.com/nervosnetwork/rfcs/blob/master/rfcs/0022-transaction-structure/0022-transaction-structure.md
- Introduction to CKB Script Programming 1:
  https://xuejie.space/2019_07_05_introduction_to_ckb_script_programming_validation_model
- NatSpec format:
  https://docs.soliditylang.org/en/latest/natspec-format.html
- Ethereum Smart Contract Security Best Practices:
  https://consensys.github.io/smart-contract-best-practices
- EIP712:
  https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md#abstract
- Top 10 classical bugs for Ethereum:
  https://immunefi.com/immunefi-top-10
- Precision Loss in Arithmetic Operations:
  https://blog.solidityscan.com/precision-loss-in-arithmetic-operations-8729aea20be9
- RFC of the CKB VM:
  https://github.com/nervosnetwork/rfcs/blob/master/rfcs/0003-ckb-vm/0003-ckb-vm.md#overview
- CERT C:
  https://wiki.sei.cmu.edu/confluence/display/c/Introduction

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation with respect to the business logic and the verification logic;
- Adversarial actions and other attacks on the network;
- Potential misuse and gaming of the scripts;

- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the smart contracts or disrupt their execution;
- Vulnerabilities in the scripts' code;
- Protection against malicious attacks and other ways to exploit the scripts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a comprehensive review of the d.id .bit Contracts. d.id is building an account system on the Common Knowledge Blockchain (CKB), a generalized bitcoin-like blockchain, founded by Nervos.

The basic unit of the CKB blockchain is called a cell, which is similar to a UTXO on Bitcoin. However, instead of storing value, it allows the storing of data. The corresponding value is called capacity and measures the size of the cell. A transaction on the CKB blockchain modifies the set of cells. A transaction contains two types of scripts, which are typically referred to as 'smart contracts' on other blockchains. The type script specifies the business logic and transforms the data in the cell. The lock script defines the verification logic and allows a user to access the cell. The multi-chain principle allows the implementation and usage of any custom verification protocols (for more details, see the transaction RFC, this series of blog posts, or this post by d.id).

We reviewed both the business logic in `das-contracts` and the verification logic in `das-lock`. In order to investigate the business logic, we reviewed the correctness of the sign up process (which involves several steps), all the actions performed for the Account cell (e.g., `transfer` or `renew`), the entire sub-account setup (configuration, update, and profit collection), and the Income and Balance cells. For `das-lock`, we reviewed the correctness of the signature verification for different signature schemes from chains such as Ethereum or Tron.

Since d.id is implemented on a novel blockchain, at the time of writing of this report, there are no best practices available that developers can refer to when writing type scripts, nor do they have access to a database of vulnerabilities that often lead to well-known security best practices — as is the case with Ethereum that has a well-defined code comment template (NatSpec) and established guidelines for security best practices. The lack of such standards makes it more difficult to review scripts on the CKB blockchain. We acknowledge that these fundamental security challenges are beyond the control of the d.id team and that security should be considered a shared responsibility with the broader community of application developers, in addition to the users.

System Design

Our team examined the design of the implementation and found that security has generally been taken into consideration. The `das-contracts` codebase and part of the `das-lock` codebase are written in Rust, a language with good performance and memory safety characteristics.

While our team did not identify any critical security vulnerabilities in the design of the system, we did identify several issues in the implementation, as detailed below.

We found several instances of missing checks in the type scripts, which can lead to unintended behavior or make the system susceptible to attacks (Issue A, Issue B, Issue C, Issue D). We recommend adding the missing checks to improve the overall security of the system.
We also found that the capacity calculations are imprecise due to rounding errors (Issue E), which can result in financial losses for d.id.

Our team investigated adherence to the EIP-712 standard and found that the domain separator for the EIP-712 does not follow best practices (Suggestion 3).
We also checked whether transactions are prone to replay attacks but did not identify any issues. We further investigated whether the impersonation of contracts or forgery of contract cells are possible but did not identify any issues due to the use of the `code_hash` and super lock mechanisms.

Additionally, we found several implementation and logic errors in the C code of the `das-lock` repository (Issue F, Issue G, Issue H, Issue I, Issue J), some of which relate to signature verification. Programming in C is notoriously difficult and can result in such issues. We recommend further improving the security and quality of the C code (Suggestion 6).

Code Quality

We performed a comprehensive review of the type and lock scripts and found that, in some instances, the code deviates from best practices. In particular, our team found inconsistency in the use of constants (Suggestion 2) as well as incomplete type matching in the `sub-account-cell-type` type script (Suggestion 5). We recommend improving code quality by deleting redundant code in the `das-contracts`, adding more code comments, and removing deprecated code (Suggestion 4).

Tests

The scripts include test coverage. Note that our team did not assess whether test coverage was sufficient, as the tests were out of the scope of this audit.

Documentation and Code Comments

Although the d.id team was very responsive and helpful in answering our questions, the documentation provided by the team was insufficient. Our team noted that more thorough documentation could have facilitated the early identification of the Issues around missing checks. We recommend improving documentation (Suggestion 1). Additionally, we found that the implementation is sparsely commented. We recommend improving code comments (Suggestion 4).

Scope

The scope of this audit included all security-critical components. However, our auditors noted that certain components were not included in their review, as the d.id team stated that they will soon be removed from the implementation. Furthermore, our team focused on the correctness and security of the business logic and the verification logic and did not review in detail all helper functions in the folders `das-contracts/libs`, `das-lock/libs`, `das-lock/deps`, and `das-lock/das-lock-lib` (Note that *some* of these functions were reviewed, but only when they were called from the type and lock scripts).

Dependencies

We examined all the dependencies implemented in the Rust codebase using `cargo audit` and did not identify any security concerns resulting from the unsafe use of dependencies.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Missing Check on is_system_off	Resolved
Issue B: Missing Check on the Approval Field in the Witness of the Account Cell	Resolved
Issue C: Missing Check on Action Field in the Approval	Resolved
Issue D: Missing Check on status_flag in verify_cell_initial_properties	Resolved
Issue E: Rounding Error Leads To Imprecise Capacity Calculation	Resolved
Issue F: Incorrect Validation of Length in Base64	Resolved
Issue G: Potential Buffer Overflow in Base64	Resolved
Issue H: Improper Input Validation	Resolved
Issue I: Signature and Public Key Duplication in Multi-Sign Verification	Resolved
Issue J: Use of Uninitialized Memory When Hashing	Resolved
Suggestion 1: Improve and Update Documentation	Unresolved
Suggestion 2: Use Constants Instead of Hard Coded Values	Resolved
Suggestion 3: Reorder Domain Separator for EIP-712	Resolved
Suggestion 4: Improve Code Quality	Unresolved
Suggestion 5: Update Incomplete Match in Subaccount Cell	Resolved
Suggestion 6: Improve C Code Security	Unresolved

Issue A: Missing Check on is_system_off

Location

`das-contracts/contracts/proposal-cell-type/src/entry.rs#L26-L42`

`das-contracts/contracts/balance-cell-type/src/entry/mod.rs#L12-L90`

`das-contracts/contracts/device-key-list-cell-type/entry.rs#L13-L33`

Synopsis

The function `is_system_off` permits checking in the type scripts of each cell whether the d.id system has been shut down, for example, for security reasons. This function call is missing in the three cells `proposal-cell-type`, `balance-cell-type`, and `device-key-list-cell-type`, thus making them vulnerable during an attack.

Impact

If the system is under attack, certain parts of the system cannot be protected. As an example, the `proposal-cell-type` script can be used by an attacker to tamper with the sign up process. It is used without any other script in the transactions `Propose`, `ExtendProposal`, and `RecycleProposal`. An attacker can potentially manage to propose false accounts, for example, to circumvent the uniqueness check. In this case, d.id would be unable to pause the usage of this type script.

Preconditions

The system would need to be shut down for security reasons; that is, when an attack is detected.

Feasibility

Low.

Remediation

We recommend adding the function `call` to the type scripts listed above.

Status

The d.id team has added the checks.

Verification

Resolved.

Issue B: Missing Check on the Approval Field in the Witness of the Account Cell

Location

[das-contracts/contracts/proposal-cell-type/src/entry.rs#L796]

[das-contracts/contracts/proposal-cell-type/src/entry.rs#L592]

Synopsis

In `verify_proposal_execution_result`, the type script `proposal-cell-type` verifies whether the Account and PreAccount cells have been converted according to the data in the Proposal cell. A check is missing here on the field `approval` in the witness of the new Account cell.

Impact

An attacker can trick a user to sign up with a non-empty `approval` field. This can be potentially used to insert an approved transfer to the attacker's address. However, the attacker would still need to pass the signature verification. A second potential scenario is that the user can block any further usage of the `approval` field for the user by inserting specific data (such as a timestamp far in the future) for the field `input_protected_until`.

Preconditions

The attacker would need to be able to tamper with the `approval` field of the attacked account.

Feasibility

Low.

Technical Details

The witness field `approval` stores authorization-related data. At the moment, the only authorized action that is implemented is a `transfer`, which allows the transfer of an account to another user. During sign up, it is necessary to check that this field is empty in the type script `proposal-cell-type`.

Remediation

We recommend adding the check.

Status

The d.id team has added the check by implementing the function `verify_witness_initial_approval`.

Verification

Resolved.

Issue C: Missing Check on Action Field in the Approval

Location

das-contracts/contracts/account-cell-type/src/approval.rs#L15

Synopsis

In the function `transfer_approval_create`, there is no check verifying that the `action` field equals `transfer`.

Impact

Passing in a different type leads to undefined behavior in the code. Since only one `approval` type exists currently, the impact is low. However, if new types of approvals are introduced, the missing check could lead to severe consequences (depending on the type of approvals being introduced).

Technical Details

The `approval` struct is used for storing information on authorization. The field `action` within this struct stores the specific authorization type. The only implemented type so far is account transfers, with the primary use case being third-party platforms.

Remediation

We recommend adding the check.

Status

The d.id team has added the check, which verifies that the `action` field equals `transfer`, prior to calling the `transfer_approval_create` function.

Verification

Resolved.

Issue D: Missing Check on status_flag in verify_cell_initial_properties

Location

das-contracts/libs/das-core/src/verifiers/sub_account_cell.rs#L22

das-contracts/contracts/account-cell-type/src/entry.rs#L840

Synopsis

When the type script `account-cell-type` runs the action `EnableSubAccount`, the initialization of the SubAccount cell is verified by the function `verify_cell_initial_properties`. However, this function does not check whether the `status_flag` equals zero.

Impact

A malicious attacker can trick a user to set the `status_flag` to the value one when the user enables the SubAccount cells. This activates the rules stored in the fields `price_rules_hash` and `preserved_rules_hash`. Since the code verifies whether the rule is initially set to zero, the impact of the attack is low. The attacker would need to modify these rules in a second attempt to successfully exploit the user. Nevertheless, refraining from checking the field leads to unintended behavior since the rest of the code assumes an initial value of zero. In addition, if the `status_flag` is set to a value different than zero or one, it could lead to unexpected system states or consequences.

Technical Details

The `status_flag` implements the rules stored in the fields `price_rules_hash` and `preserved_rules_hash`. These allow the user to specify custom pricing rules for the SubAccount cell.

Remediation

We recommend adding the check.

Status

The d.id team has added checks to the `verify_cell_initial_properties` function.

Verification

Resolved.

Issue E: Rounding Error Leads To Imprecise Capacity Calculation

Location

das-contracts/libs/das-core/src/util.rs#L716

das-contracts/libs/das-core/src/util.rs#L727

Link to Rust playground: https://play.rust-lang.org

Synopsis

The functions `calc_yearly_capacity` and `calc_duration_from_paid` perform division before multiplication for u64 types. Consequently, this results in imprecisions in the calculations, which could lead to financial losses for d.id.

Impact

The imprecision in `calc_yearly_capacity` could potentially lead to less income for the d.id team since customers benefit from the current calculation – at least in the numerical examples our team reviewed (see the Rust playground for more details). For this function, our numerical analysis identified a deviation of up to 99%. We did not identify a problem with the imprecision found in `calc_duration_from_paid`.

Preconditions

The prices would need to be in a range where division differs significantly between the two approaches, and `yearly_price` would have to be larger than `quote`.

Technical Details

For demonstration purposes, we implemented this rust playground and varied the yearly price. Below, we describe the extreme case that we were able to detect. Note that we did not perform a thorough investigation of the complete input space of `yearly_capacity` and `quote`. The example is rather used to demonstrate that the deviation is non-negligible and needs to be addressed.

The function `calc_yearly_capacity` behaves differently depending on the relation between `yearly_price` and `quote`. If `yearly_price` is less than `quote`, then `yearly_price` is first multiplied by 100000000 and then divided by `quote`. This is necessary to prevent the rounding behavior from yielding a value of zero for the yearly capacity. To avoid this, the function uses the `if-loop` specified below:

```
Python
pub fn calc_yearly_capacity(yearly_price: u64, quote: u64, discount: u32) ->
u64

{

	let total u64;

	if yearly_price < quote {

		total = yearly_price * 100_000_000 / quote;

	} else {

		total = yearly_price / quote * 100_000_000;

	}

	...

}
```

However, the `else-case` is still imprecise for the case where `yearly_price` is greater than or equal to `quote`, as in this case, the code performs division before multiplication. To illustrate, consider the following values:

```
Python
yearly_price= 199;

quote = 100;


// based on total = yearly_price / quote * 100_000_000;

total_1 =  100000000;


// based on total = yearly_price * 100_000_000 / quote;

total_2 = 199000000;
```

The deviation between the two cases reaches 99% in this extreme case.

Mitigation

The recommended remediation below might be not practicable since overflows can occur and hence lead to an unfavorable user experience. In this case, we recommend monitoring the situation closely. Note that the size of the deviation, and hence the resulting loss of income, highly depend on the exact numbers used in the calculations.Therefore, it is possible that the actual losses might be lower than expected.

Remediation

We recommend performing multiplication before division, especially for `calc_yearly_capacity`. This recommendation is a standard for other smart-contract-based blockchains like Ethereum (see here), where rounding errors are among the top 10 classical bugs. In addition, if possible, we recommend implementing "*softfloat implementation into the binary,*" as described in the RFC of the CKB VM.

Status

The d.id team has resolved the Issue by using the `primitive-types` crate to handle u256.

Verification

Resolved.

Issue F: Incorrect Validation of Length in Base64

Location

`das-lock/c/base64url.h#L231-L234`

Synopsis

A check for undersized or oversized inputs is incorrectly structured, using logical conjunction (Boolean AND) instead of logical disjunction (Boolean OR). Since an input cannot, at the same time, be both oversized and undersized, the check does not capture either condition.

Impact

An attacker can pass oversized data that can result in buffer overflows and the overwriting of arbitrary memory.

Feasibility

Straightforward.

Technical Details

```C/C++
if (*len < 1 && *len > 256) {

  debug_print("decode_base64url_to_string: invalid input, length out of range");

  return ERROR_ARGUMENTS_LEN;
}
```

Remediation

We recommend using logical disjunction (i.e., Boolean OR) by changing && to `||` instead.

Status

The d.id team has implemented the remediation as recommended.

Verification

Resolved.

Issue G: Potential Buffer Overflow in Base64

Location

[das-lock/c/base64url.h#L235C1-L236C46]

Synopsis

A potential buffer overflow might occur when converting Base64URL encoded strings to Base64 encoded strings as a result of the additional padding characters present in Base64.

Feasibility

Straightforward.

Technical Details

Base64URL encoded string does not contain padding characters, which are included in Base64 encoded strings. As a result, when converting from Base64URL to Base64, the resulting string may be longer than the input.

The code allocates a 256 byte buffer in which the converted string is stored. It checks whether the length of the input string is, at most, 256 bytes (see Issue F) and then attempts the conversion.

If the conversion requires the insertion of padding bytes (1 or 2) it might be possible to overflow the buffer.

Remediation

We recommend expanding the buffer in which the conversion is performed to allow for the insertion of up to two padding characters from the conversion.

Status

The d.id team has remediated this Issue following a discussion with our team.

Verification

Resolved.

Issue H: Improper Input Validation

Location

[das-lock/c/webauthn_sign.c#L52-L62]

Synopsis

The 'challenge string' retrieved from the JSON data passed to the function is assumed to have a fixed length of 86 characters in a comment, but no validation of the length is performed.

The challenge string is subsequently decoded and assumed to have a length of precisely 64 bytes, but it is possible for the input to be much longer, resulting in out-of-bounds writes.

Impact

This Issue could result in buffer overflowing and out-of-bounds writes, affecting the stack.

Technical Details

If the JSON document contains the following *example* challenge string, the code will overflow the buffer:

```
RnJvbSAuYml0OiBUaGlzIGlzIGEgc2ltcGxlIHN0cmluZyBlbmNvZGVkIHdpdGggYmFzZTY0dXJsLi
BCbGFoIGJsYWggYmxhaC4
```

This is the `Base64URL` encoding of the string *"From .bit: This is a simple string encoded with base64url. Blah blah blah."* However, any string would lead to the same outcome. By having the expected common prefix present, the function may appear to succeed, but, depending on the stack layout, it may corrupt the state of the program in unpredictable ways.

Remediation

We recommend converting the comment to an actual check to ensure that the challenge has the correct fixed length, and generating an error otherwise. Additionally, we recommend increasing the size of the buffer into which the challenge is decoded to ensure that it cannot be overflowed.

Status

The d.id team has implemented the remediation as recommended. The team additionally noted that there are already protocol-level safeguards in place to prevent this Issue from being exploited.

Verification

Resolved.

Issue I: Signature and Public Key Duplication in Multi-Sign Verification

Location

das-lock/c/ckb_multi_sign.c#L52

Synopsis

The code checks signatures against an array of public keys. No checking is performed to ensure that a public key is not listed multiple times. As a result, it is possible to construct scenarios where an attacker can reuse a single signature.

Impact

An attacker may be able to bypass threshold checking by reusing a single valid signature.

Preconditions

The `lock_bytes` buffer would have to be under the control of the attacker.

Technical Details

The code determines the number of public keys present (storing them in the `pubkeys_cnt` variable) and the required threshold (storing it in the `threshold` variable) and then iterates by checking the signature and comparing it against a list of public keys.

The code does not include any checks to ensure that the signature at index `i` is not duplicated, or that the public key used in the signature has not already been used. As long as the signature is valid, it will set `used_signatures[i]` to one.

If a threshold is specified, an attacker can either duplicate a single signature or, if in possession of one private key, generate `threshold` valid signatures with the same key and place them in the first `threshold` slots to meet the simple check.

Remediation

We recommend checking that all listed signatures and their corresponding public keys are unique. If possible, we recommend imposing a canonical format (e.g., one where public keys are listed in ascending order).

Status

The d.id team stated they investigated this Issue and confirmed that this risk does exist. Our team agrees and thus considers this resolved.

Verification

Resolved.

Issue J: Use of Uninitialized Memory When Hashing

Location

das-lock/c/doge_sign.c#L23

Synopsis

The code collates several inputs to form a message, which it then hashes. One of the inputs is encoded using a variable-length encoding (either 1 or 2 bytes). For messages between 256 and 65536 bytes, one byte is not explicitly initialized and may have unpredictable values, causing non-deterministic behavior.

Feasibility

Straightforward.

Technical Details

The `message _vi_len` specifies the number of bytes needed to encode the length of the `message`, which will then be set to either one or two.

The code leverages the compiler's variable length array support to allocate a properly-sized buffer. The C standard imposes no requirements on the contents of such allocated buffers. While it is possible that a given compiler zero-initializes variable length arrays, this behavior is, at best, compiler-dependent.

For messages greater than 255 bytes (and less than 65536 bytes) the 28th byte of the buffer (i.e., `total_message[27]`) will be left uninitialized.

As a result, the `magic_hash` function may return unpredictable values for messages greater than 255 bytes.

Remediation

We recommend properly initializing the contents of the buffer to some known value using `memset`. The value zero might be appropriate, depending on existing behavior.

Note that it appears that the variably-encoded length is never written to the buffer. It might be possible to simply adjust the size of the `total_message` array to not include `message_vi_len`. Care should be taken to ensure that this is not a breaking change and that the resulting buffer is still appropriately sized to hold the data written to it.

Status

The d.id team has implemented the remediation as recommended. However, our team noted that the code is still complex and difficult to understand and therefore presents a risk that the d.id team should further address.

Verification

Resolved.

Suggestions

Suggestion 1: Improve and Update Documentation

Location

[das-contracts/tree/docs/docs/en]

Synopsis

Due to the high diversification of roles (Users, Keepers, Registrars, Resolvers, etc.) within the d.id system, the flow of information within the network needs to be well-documented to be examined. During the audit,

our team discovered imprecisions in the implementation as well as missing documentation, which reduce the readability of the code and, thus, makes reasoning about the security of the system more difficult. For example, we identified:

- Missing cells in deps for transactions (e.g., the Preaccount cell should have the Account cell in deps);
- Several outdated parts in the documentation (e.g., the auction) and dead links (e.g., to 'anti-robbery'); and
- Missing documentation on the usage of the `approval` logic for the Subaccount cell. Even though the behavior is the same as that of the Account cell, we recommend explicitly stating this in the documentation.

Mitigation

We recommend updating the available documentation to include clear and precise specifications, as well as providing additional, detailed documentation to allow both future developers and auditors to easily understand the different components of the system.

We understand that outdated parts of the documentation are kept because some cells might still have access to part of the functionality. However, we recommend marking these outdated parts more clearly as outdated.

Status

The d.id team acknowledged the importance of this suggestion but stated that the mitigation cannot be implemented in a short time frame. Hence, the team noted that they will continue making improvements in the future, as they consider this mitigation to be part of an ongoing and incremental process.

Verification

Unresolved.

Suggestion 2: Use Constants Instead of Hard Coded Values

Location

`das-contracts/libs/das-core/src/constants.rs#L40-L71`

Synopsis

The implementation relies on constants in the Rust code and follows best practices. However, the constants are used inconsistently across the codebase. For example, the constants `DAY_SEC` and `ONE_CKB` are not used in parts of the code (the search for 86400 yields only 11 results).

Mitigation

We recommend using the constants consistently. This can be done by checking all currently used constants to ensure consistent usage across the codebase.

Status

The d.id team has replaced all the constants as recommended.

Verification

Resolved.

Suggestion 3: Reorder Domain Separator for EIP-712

Location

das-lock/eip712-lib/src/eip712.rs#L348

Synopsis

d.id employs the EIP-712 standard to hash and sign typed structured data. A domain separator is used in the standard to prevent collisions across dApps. As stated in the EIP-712, "*The `EIP712Domain` fields should be the order as above, skipping any absent fields.*" However, d.id has a different order and hence does not follow the standard security recommendations. In particular, the standard recommends the order (`name`, `version`, `chainId`, `verifyingContract`), while d.id implements the order (`chainId`, `name`, `verifyingContract`, `version`). The standard also specifies that "*User-agents should accept fields in any order as specified by the EIP712Domain type.*" Consequently, MetaMask is currently accepting the transaction, despite the incorrect order.

Mitigation

We recommend following the standard and reordering the EIP-712 domain in the specified order.

Status

The d.id team has reordered the EIP-712 domain accordingly.

Verification

Resolved.

Suggestion 4: Improve Code Quality

Location

das-contracts/contracts/sub-account-cell-type/src/entry.rs#L304

das-contracts/contracts/sub-account-cell-type/src/sub_action.rs#L189

das-contracts/libs/das-types/rust/src/constants.rs

Synopsis

During our extensive review of the codebase, our team identified practices that impact the quality, readability, and maintainability of the codebase. To illustrate, the following is a non-exhaustive list of examples:

- We found several instances of redundant code in the `das-contracts`, which reduces the simplicity and efficiency of the code; for example:
  - The call to `verify_sub_account_enabled` in `action_update_sub_account` is unnecessary; and
  - The code in `if-loop` in the function `create` cannot be reached because `manual_mint_list_smt_root` is always None.
- There are several empty files in `das-lock`, whose purpose is not clear (e.g., `das-lock/c/protocol.h` and `das-lock/c/utils.rs`);
- There are several deprecated `Enum` fields (e.g., in the `constants` file), which should be removed;
- There are several large commented-out sections of code marked as abandoned or otherwise not used (e.g., in the `das-lock/c/json_operator.h` file), which should be removed;
- Code that appears to perform useful error or 'belt-and-suspenders' checking is often commented out (e.g., in the `get_challenge_from_json` function in `das-lock/c/json_operator.h`);

- There are unresolved TODOs (e.g., here) in the codebase, which should be resolved; and
- There are very few code comments, which decreases the readability of the code and, as a result, makes reasoning about the security of the system more difficult.

Mitigation

We recommend improving code quality by addressing the items listed above.

Status

The d.id team acknowledged the importance of this suggestion but stated that the mitigation cannot be implemented in a short time frame. Hence, the team noted that they will continue making improvements in the future, as they consider this mitigation to be part of an ongoing and incremental process.

Verification

Unresolved.

Suggestion 5: Update Incomplete Match in Subaccount Cell

Location

`das-contracts/contracts/sub-account-cell-type/src/entry.rs`

Synopsis

There are three cases in the type script `sub-account-cell-type` where the field `flag` is matched with the corresponding enum `SubAccountConfigFlag`. In one case, the field `flag` is matched with either `CustomRule` or `Manual`, or with `_` for errors. The other two cases only match the field `flag` with either CustomRule or `_`, excluding `Manual`. This is not recommended practice for type scripts, as it fails to include the possible match `Manual`. In particular, for the cases here and here, the code implicitly assumes that `_` equals `Manual` and calls `verify_sub_account_cell_is_consistent`. Consequently, a false value is not detected for the field `flag`.

Mitigation

We recommend matching the `flag` with all possible cases and throwing an error for undefined cases.

Status

The d.id team has, for each of the three cases, either implemented the recommended mitigation or argued convincingly to justify why it was not replaced (for more details, see this commit).

Verification

Resolved.

Suggestion 6: Improve C Code Security

Synopsis

The C programming language offers few, if any, safety features commonly found in more modern languages, such as Rust. Developers are responsible for manually managing memory and object lifetime, performing complicated handling of strings and buffers, managing locking, and avoiding race conditions.

Mitigation

We recommend adopting one of the restricted styles of C (e.g., CERT C), leveraging functions from the C standard library, enabling all compiler warnings available and, finally, considering the use of a static analysis tool as part of the development process to help identify potential bugs early.

Status

The d.id team acknowledged the importance of this suggestion but stated that the mitigation cannot be implemented in a short time frame. Hence, the team noted that they will continue making improvements in the future, as they consider this mitigation to be part of an ongoing and incremental process.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.