DLC-Link Audit

October 2023

By CoinFabrik

v202307

Executive Summary

CoinFabrik was asked to audit the files for the DLC-Link project. The audit is based on the commit `d02c87eccca7eb016631344b29377fdf3d78d9a9` of the repository `https://github.com/DLC-link/dlc-stack`. Fixes were checked on commit f16ee8b15a2ce8b2ada45bd7f2e0b64c66a10302 of the same repo.

DLC.link is developed to address the absence of smart contract capabilities in the Bitcoin ecosystem. Despite Bitcoin's position as the foremost digital asset, it lacks native support for smart contracts. This has necessitated users to transfer Bitcoin to either a custodian or a bridge when seeking to use their Bitcoin assets in financial applications. As a result, custodian failures and bridge hacks have accounted for over $140Bn in losses.

To mitigate these risks, DLC.link has implemented Discreet Log Contracts, an invention from MIT. This technology allows for the integration of native Bitcoin into applications by placing the Bitcoin into an on-chain escrow. From this escrowed position, Bitcoin can be interfaced with and controlled by Stacks, Ethereum and other smart contract platforms. By doing so, DLC.link creates a "trustless bridge" that facilitates financial transactions.

ID	Title	Severity	Status
CR-01	Remote Code Execution with Callback Contract	Critical	Resolved
CR-02	Unauthenticated storage service API	Critical	Resolved
CR-03	DoS Via Infinity Loops At Protocol Wallet	Critical	Resolved
CR-04	Arbitrary URL requests at Protocol Wallet	Critical	Resolved
HI-01	Weak Source of Randomness in Attestor Selection	High	Acknowledged
HI-02	Credential leak via log mechanism	High	Resolved
HI-03	Unencrypted API exposure	High	Resolved
ME-01	Insecure Attestor Key Management	Medium	Acknowledged

ID	Title	Severity	Status
MI-01	Floating Solidity Pragma	Minor	Resolved
MI-02	Authentication via tx-sender	Minor	Resolved
MI-03	Service panic instead of returning a proper error code	Minor	Resolved

Scope

The audited files are from the git repository located at the following repos and commits:

- https://github.com/DLC-link/dlc-stack/:
`d02c87eccca7eb016631344b29377fdf3d78d9a9`.
    - `/attestor/`: numeric DLC Attestor service implementation
    - `/observer/: blockchain observer`
    - `/attestor-client/: Attestor client implementation`
    - `/wallet/: protocol wallet`
    - `/wallet/wallet-blochain-interface/: Blockchain interface service`
    - `/storage/: Storage service`
- https://github.com/DLC-link/dlc-solidity/:
`414509500c1a46954853e9dc87094a813bb4928e`.
    - `contracts/DLCManagerV1.sol`: Ethereum link contract.
- https://github.com/DLC-link/dlc-clarity/:
`a0e37cb4d6200452c43826967eb88f76c0846fda`.
    - `contracts/dlc-manager-v1.clar`: Stacks link contract.

Methodology

CoinFabrik was provided with the source code, including automated tests that define the expected behavior, and general documentation about the project. Our auditors spent five weeks auditing the source code provided, which includes understanding the context of use, analyzing the boundaries of the expected behavior of each program and function, understanding the implementation by the development team (including dependencies beyond the scope to be audited) and identifying possible situations in which the code allows the caller to reach a state that exposes some vulnerability. Without being limited to them, the audit process included the following analyses.

- Arithmetic errors
- Outdated version of Solidity compiler
- Race conditions
- Reentrancy attacks
- Misuse of block timestamps
- Denial of service attacks
- Excessive gas usage
- Missing or misused function qualifiers
- Needlessly complex code and program interactions
- Poor or nonexistent error handling
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

- Centralization and upgradeability

Findings

In the following table we summarize the security issues we found in this audit. The severity classification criteria and the status meaning are explained below. This table does not include the enhancements we suggest to implement, which are described in a specific section after the security issues.
Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. Blocking bugs are also included in this category. They must be fixed **immediately**.
- **High:** These refer to a vulnerability that, if exploited, could have a substantial impact, but requires a more extensive setup or effort compared to critical issues. These pose a significant risk and demand immediate attention.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of, but might be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.

Issues Status

An issue detected by this audit has one of the following statuses:

- **Unresolved**: The issue has not been resolved.
- **Acknowledged**: The issue remains in the code, but is a result of an intentional decision. The reported risk is accepted by the development team.
- **Resolved**: Adjusted program implementation to eliminate the risk.
- **Partially resolved**: Adjusted program implementation to eliminate part of the risk. The other part remains in the code, but is a result of an intentional decision.
- **Mitigated**: Implemented actions to minimize the impact or likelihood of the risk.

Critical Severity Issues

CR-01 Remote Code Execution with Callback Contract

Location:

- `dlc-clarity/contracts/dlc-manager-v1.clar:95`

Classification:

- CWE-20: Improper Input Validation[1]

The `dlc-manager-v1.clar` contract is susceptible to a Remote Code Execution (RCE) through the callback contract. The issue arises due to the absence of validation for the callback address provided in the `set-status-funded()` and `post-close()` functions. Without this validation, malicious actors could potentially exploit this vulnerability to execute arbitrary code on the contract.

Steps to Replicate

1. Initiate the creation of a DLC.
2. During DLC creation, store a valid callback address in the contract's state.
3. Later, invoke either the `set-status-funded()` or `post-close()` functions with a different, potentially malicious callback address as the argument.

Recommendation

Implement callback address validation in the `set-status-funded()` and `post-close()` functions. The contract should verify that the provided callback address matches the stored address associated with the corresponding DLC's UUID.

Status

Resolved. Added the two checks that restrict callback-contract to that connected to the given DLC.

CR-02 Unauthenticated storage service API

Location:

- `storage/api/src/main.rs:54`

Classification:

- CWE-306: Missing Authentication for Critical Function[2]

The storage service provides an open API to read, store and modify events and contracts. The API does not provide any authentication so any external user can modify, delete or add any contract and event, just by having the id and corresponding key. These values can be

[1] https://cwe.mitre.org/data/definitions/20.html
[2] https://cwe.mitre.org/data/definitions/306.html

found in several ways, for example sniffing connections (as they are not encrypted) or by using the public API `get_contracts()` of the same service.

The list of services exposed by the storage API is:

- `get_contracts(), create_contract(), update_contract(), delete_contract(), delete_contracts(),`
- `get_events(), create_event(), update_event(), delete_event(), delete_events()`

But most APIs except `get_contracts()` and `get_events()` do modify the database without authentication.

Steps to Replicate

1. Get the uuid and key of any contract or event.
2. Using curl or javascript, craft an API request to any sensitive api, for example, `delete_contracts()`.
3. No authentication is requested to delete all contracts.

Recommendation

Ideally, never expose APIs that modify the internal state without authentication. If needed, you can implement a challenge/response authentication protocol (I.E. auto generating a secret) so only clients that created the event or the contract, are allowed to modify/delete it. Note: this requires that the API connection is encrypted. For unencrypted authentication, a public key signature authorization might be needed.

Status

Resolved. A complete authentication system was implemented for the storage API. Replay attacks are avoided using whitelisted nonces.

CR-03 DoS Via Infinity Loops At Protocol Wallet

Location:

- `wallet/src/main.rs:117`

Classification:

- CWE-835: Loop with unreachable exit condition[3]

The `retry!` macro used in the Protocol Wallet checks for return values and retries indefinitely if there is an error.

As there is no exit condition, this can be used to lock the service in an infinite loop, for example, specifying an inexistent attestor using the `offer()` API endpoint. For example,

[3] https://cwe.mitre.org/data/definitions/835.html

this in `main.rs:117`

```
    let p2p_client: AttestorClient = retry!(
        AttestorClient::new(url).await,
        10,
        "attestor client creation"
    );
```

If the url does not exist, the service will loop indefinitely. This can be used to lock the service up by repeatedly asking for non-existent attestor URLs until the TCP queue is filled and the protocol wallet cannot accept more API requests until restart. The `retry!` macro is used in other places (for example, retrieving blockchain parameters at line 266 and 273) but those are not as critical as the DoS in the `offer()` API.

Steps to Replicate

1. Start the protocol wallet service
2. Do several `offer()` API call with the following parameters, until the server stop responding:

```
curl -X POST http://localhost:8085/offer -H "Content-Type:
application/json" -d '{"uuid": "79", "acceptCollateral":
1234,"offerCollateral":0,"totalOutcomes":100,"attestorList":
"[\"nonexistant\"]" }'
```

Recommendation

Implement a reasonable retry limit on the macro. The service will still lock up temporarily if too many requests are made, but not indefinitely.

Status

Resolved. Added a retry limit to the 'retry' macro. Updated wallet code such that there are limits on failing API calls

CR-04 Arbitrary URL requests at Protocol Wallet

Location:

- `wallet/src/main.rs:118`

Classification:

- CWE-601: URL Redirection to Untrusted Site[4]

[4] https://cwe.mitre.org/data/definitions/601.html

When specifying an attestor using the `offer()` API call, the protocol wallet will download the public key via an http request (Code at AttestorClient:new() attestor-client/src/lib.rs:151):

```
    let attestor_key = client_builder
         .build()
         .map_err(|e| {
             DlcManagerError::IOError(std::io::Error::new(
                 std::io::ErrorKind::NotConnected,
                 e.to_string(),
             ))
         })?
         .get(path)
         .send()
         .await
```

And then, the contents of this file is logged directly into the console:

```
     info!("Attestor Pub Key: {}", attestor_key.to_string());
```

For example: if the specified attestor is in http://www.dlc-link.org/u1, the protocol wallet will make a http request to http://www.dlc-link.org/u1/public_key.

As this URL can be arbitrary, and the `offer()` API is public, any external user can cause the protocol wallet to create any http request to any URL with any parameters.

Steps to Replicate

1. Start the protocol wallet service
2. Execute this command, that causes the protocol wallet to issue a query to google.com:

```
curl -X POST http://localhost:8085/offer -H "Content-Type:
application/json" -d '{"uuid": "79", "acceptCollateral":
1234,"offerCollateral":0,"totalOutcomes":100,"attestorList":
"[\"http://www.google.com/search?q=hello\",\"a\"]" }'
```

It is also possible to download files with arbitrarily large size, causing a denial-of-service via resource exhaustion (Depending on debug options, the whole file may be logged to the console).

This can also be used by malicious external users to use the DLC-Link services as a pivot for malicious activities, by using the service as a relay.

Recommendation

Perform input validation by an attestor URL whitelist so you cannot create an offer with an arbitrary URL.

Status

Resolved. The Protocol Wallet performs a check on the incoming request, and only proceeds if the provided attestors are in a whitelist.

High Severity Issues

HI-01 Weak Source of Randomness in Attestor Selection

Location:

- `dlc-solidity/contracts/AttestorManager.sol:95`

Classification:

- SWC-120: Weak Sources of Randomness from Chain Attributes[5]
- CWE-330: Use of Insufficiently Random Values[6]

The `randomNumber()` function is used to select attestors for the DLC based on the block timestamp. This weak randomness source enables potential exploitation, wherein a malicious block builder can manipulate the timestamp and therefore the attestor selection.

If successfully exploited, a malicious actor can repeatedly be selected as an attestor or ensure specific attestors are consistently selected, compromising the integrity and fairness of the DLC attestor selection process. This can lead to centralization or collusion risks in the DLC.

Steps to Replicate

1. Become a block builder or miner.
2. Wait for a transaction creating a DLC.
3. Manipulate block's timestamp to influence the random number generation. Try with as many timestamps as it is possible until you get the best result where attestors under your control are selected.
4. Use the attestors for signing a false outcome which is beneficial for you.

[5] https://swcregistry.io/docs/SWC-120/
[6] https://cwe.mitre.org/data/definitions/330.html

Block headers, specifically the timestamp, are not entirely secure as a source of randomness because they can be influenced by the block builder (or miner). Although Ethereum has some mechanisms to discourage drastic deviations from the true time, small manipulations are feasible. This allows a block builder with knowledge of how the randomness is being derived to have an advantage in anticipating or manipulating the outcome.

Recommendation

Use a Verifiable Random Function (VRF) for random number generation. A commit-reveal scheme would also be an option. However, for the given scenario, a VRF would be a more suitable solution, since they provide immediate, unpredictable randomness on-chain without relying on external parties. While commit-reveal offers randomness, it requires multiple participants and multiple transactions, introducing complexity and potential points of failure. The DLC creation's single-participant nature further complicates this method. Chainlink's VRF on Ethereum provides a robust and efficient means to secure the attestor selection process.

Status

Acknowledged. On-chain attestor management will be deprecated in the future. Also, this attack was considered unlikely because an attacker needs to be in control of the majority of attestors.

HI-02 Credential leak via log mechanism

Location:

- `attestor/src/lib.rs:54`

Classification:

- CWE-532: Insertion of Sensitive Information into Log File

When creating an attestor, the initialization code logs the private key contents as a debug message. Log files often have less stringent security measures in place during creation, such as backup procedures or debugging processes. As a result, if these log files fall into the hands of unauthorized individuals, they could gain access to the private keys contained within.

Steps to Replicate

1. Create an attestor
2. Inspect the log file looking for the message “`[WASM-ATTESTOR]: Creating new attestor with storage_api_enabled:`” where the secret key is written.

Recommendation

Remove the private key from the log file, or replace it with a representation of it like `‘PRIVKEY’`.

Status

Resolved. Removed the logging of the secret key.

HI-03 Unencrypted API

Location:

- `wallet/src/main.rs:393`
- `wallet/src/main.rs:279`
- `storage/api/src/main.rs:65`
- `wallet-blockchain-interface/src/http/server.ts:11`

Classification:

- CWE-319: Cleartext Transmission of Sensitive Information

The protocol wallet employs plain http to export its API. By default, the service binds to `0.0.0.0:8085` which exposes the API without encryption on every network interface, leaving communication vulnerable to interception by malicious actors.

The same happens in the storage service, but binding to address `0.0.0.0:8100` and a similar problem is in the `wallet-blochain-interface` service.

Recommendation

To ensure the security of data transmission over public networks, it is recommended to utilize the Transport Layer Security (TLS) protocol when exposing the API. A commonly employed approach involves binding the API to a local port and then making it accessible through a reverse HTTPS proxy tool such as Nginx.

If the API is only used locally, bind the service to a local interface like `127.0.0.1`.

Status

Resolved. The Protocol Wallet no longer has a public API, it is restricted to the local interface only. Also, the services run behind an nginx reverse proxy, providing additional security.

Medium Severity Issues

ME-01 Insecure Attestor Key Management

Location:

- `attestor/src/lib.rs:47`
- `attestor/observer/dist/src/services/attestor.service.js:26`

Classification:

- CWE-922: Insecure Storage of Sensitive Information

The attestor service code retrieves the secret key from an environment variable.

Recommendation

For such sensitive credentials it's desirable to use a safer storage mechanism, like a Vault or HSMs. If possible, never store private key material in the service memory, always using an external module to realize cryptographic primitives. In this way, a malicious attacker won't be able to access the credentials even if he can access the server or cloud environment.

Status

Acknowledged. As this requires coordination with third-party attestors, it was decided to address this later, moving to an off-app solution.

Minor Severity Issues

MI-01 Floating Solidity Pragma

Location:

- `dlc-solidity/contracts/AttestorManager.sol`
- `dlc-solidity/contracts/DLCLinkCompatibleV1.sol`
- `dlc-solidity/contracts/DLCManagerV1.sol.sol`

Classification:

- SWC-103: Floating Pragma[7]
- CWE-664: Improper Control of a Resource Through its Lifetime[8]

The smart contracts use a floating Solidity pragma. This implies that these contracts are not bound to a specific compiler version. Although this can be advantageous for flexibility and compatibility, especially for libraries, it poses potential risks for contracts that are deployed in live environments.

If contracts get deployed using an unintended compiler version, it might inadvertently introduce bugs or vulnerabilities that can negatively affect the system's integrity. It is paramount that contracts be deployed with the same compiler version and flags they've been rigorously tested with to ensure predictable and secure behavior.

Recommendation

Lock the pragma version in the smart contracts. This ensures that the contracts are always compiled with the intended compiler version.

Status

Resolved. All contracts updated to the latest solidity version.

[7] https://swcregistry.io/docs/SWC-103/
[8] https://cwe.mitre.org/data/definitions/664.html

MI-02 Authentication via tx-sender

Location:

- `dlc-clarity/contracts/dlc-manager-v1.clar:151, 176, 201, 225, 252, 274, 284`

The system utilizes `tx-sender` for its authentication processes. This method, while functional, presents latent vulnerabilities, particularly exposing actors within the system to threats known as phishing[9].

Actors could inadvertently activate a malicious contract. Once activated, the deceptive contract can access and initiate certain functions, presenting actions as if they were done by the original actor. This impersonation potential poses risks, depending on the specific function being accessed.

Recommendation

It is advisable to switch from using `tx-sender` to `contract-caller` for a more reliable and secure authentication method. Furthermore, introducing a mapping for trusted callers can add an extra layer of security, particularly if the system needs to interact with specific intermediary contracts.

Status

Resolved. Changed authentication from tx-sender to contract-caller.

MI-03 Service panic instead of returning a proper error code

Location:

- `wallet/src/main.rs:198`
- `wallet/src/main.rs:240`

When parsing input JSON, the Protocol Wallet occasionally experiences a `panic()` rather than catching the exception and returning an error message. While this does not directly affect the service since only the child thread is terminated, unhandled exceptions could potentially bring down the entire service if they occur in the main thread.

Steps to Replicate

1. Start the protocol wallet service
2. Execute this command, that causes the Protocol Wallet to panic and return an empty reply:

```
curl -X POST http://localhost:8085/offer -H "Content-Type:
application/json" -d '{"uuid": "79", "acceptCollaterl":
1234,"offerCollateral":123,"totalOutcomes":23333,"attestorList":"test"}'
```

[9] https://www.coinfabrik.com/blog/tx-sender-in-clarity-smart-contracts/

(Notice that "`acceptCollaterl`" is the field that causes the parser to panic)

Recommendation

You can replace `unwrap()` with `?` operator to avoid panics in Rust.

Status

Resolved. Every instance of panic has been replaced by better error handling and error message passthrough.

Centralization

The provided `DLCManagerV1` solidity contract has the following roles:

`DLC_ADMIN_ROLE`: This role can pause and unpause the contract. It is assigned to the contract creator.

`DEFAULT_ADMIN_ROLE`: Default admin role. Also assigned to the contract creator.

`WHITELISTED_CONTRACT`: This role can create a DLC,

`WHITELISTED_WALLET`: This role can set the status as funded, and close a DLC.

Other Considerations

The considerations stated in this section are not right or wrong. We do not suggest any action to fix them. But we consider that they may be of interest to other stakeholders of the project, including users of the audited programs, token holders or project investors.

1. Consider adding authentication to all exposed APIs
2. Consider validation of all API inputs, even between internal services (Like Protocol Wallet and `wallet-blockchain-interface`).
3. Consider exporting all API and doing all requests using TLS (https protocol).

Upgrades

The Clarity and Solidity contracts lack any upgrade mechanism.

Changelog

- 2023-09-29 – Preview report based on commit `1727a9cd89ab58ee3fb51c761f0d3bda9b7710e8`.
- 2023-10-06 – Final version based on commit `d02c87eccca7eb016631344b29377fdf3d78d9a9`.

- 2023-11-29 – Update based on fixes at commit f16ee8b15a2ce8b2ada45bd7f2e0b64c66a10302.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the DLC-Link project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a code faultlessness guarantee.