

Liquidity Ethereum Swap Audit

- 1 [Summary](#)
 - 1.1 [Audit Dashboard](#)
 - 1.2 [Audit Goals](#)
 - 1.3 [System Overview](#)
 - 1.4 [Key Observations/Recommendations](#)
- 2 [Issue Overview](#)
- 3 [Issue Detail](#)
 - 3.1 [Ethereum swap provider forces secrets to be 32 bytes](#)
 - [Description](#)
 - [Example](#)
 - [Remediation](#)
 - 3.2 [Lack of parameter validation](#)
 - [Description](#)
 - [Examples](#)
 - [Remediation](#)
 - 3.3 [Using zeros as a secret leads to a nonrefundable contract](#)
 - [Description](#)
 - [Example](#)
 - [Remediation](#)
 - 3.4 [Users need to trust the Liquidity tools](#)
 - [Description](#)
 - [Remediation](#)
 - 3.5 [Lack of testing](#)
 - [Description](#)
 - [Remediation](#)
- [Appendix 1 - Severity](#)
 - [A.1.1 - Minor](#)
 - [A.1.2 - Medium](#)
 - [A.1.3 - Major](#)
 - [A.1.4 - Critical](#)
- [Appendix 2 - Disclosure](#)

1 Summary

ConsenSys Diligence conducted a security audit on Liquidity's "Ethereum Swap" contract. The scope of the audit was just the output of the Ethereum provider's `createSwapScript` function, which is a hashed timelock that is used in the context of cross-chain atomic swaps.







The audit covered the file `EthereumSwapProvider.js` with the MD5 hash `395c95e0449cd60b14202c383ea858a2`.

1.1 Audit Dashboard

Audit Details

- Project Name:** Liquidity Ethereum Swap
- Client Name:** Liquidity
- Client Contact:** Matthew Black
- Auditors:** Steve Marx, Gonalo S
- GitHub:** <https://github.com/liquidity/chainabstractionlayer/blob/4a1e55e31e20206165dabbf4bd22c44ac785a34a/src/providers/ethereum/EthereumSwapProvider.js>
- Languages:** EVM bytecode
- Date:** 2018-11-02

Number of issues per severity

				
	0	0	1	0
	0	2	0	2

1.2 Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped in the following three categories:

Security: Identifying security related issues within each contract and within the system of contracts.

Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:

- Correctness
- Readability
- Sections of code with high complexity
- Improving scalability
- Quantity and quality of test coverage

1.3 System Overview

Scope

The scope of the audit was purely on the output of the Ethereum provider's `createSwapScript` function. However, other code was skimmed to understand the context in which this code is used.

Design











The contract is used as part of an atomic swap. It stores ether and transfers it to one of two addresses depending on whether a correct hash preimage is provided and whether an expiration time has been reached.

1.4 Key Observations/Recommendations

- The bytecode is well optimized for gas consumption.
- The bytecode is easy to follow (as compared to other hand-written bytecode).
- There's poor test coverage of the contract.
- Hand-written bytecode is difficult to audit and for end users to review as compared to Solidity.



2 Issue Overview

The following table contains all the issues discovered during the audit. The issues are ordered based on their severity. More detailed description on the levels of severity can be found in Appendix 1. The table also contains the GitHub status of any discovered issue.

Chapter	Issue Title	Issue Status	Severity
3.1	Ethereum swap provider forces secrets to be 32 bytes		
3.2	Lack of parameter validation		
3.3	Using zeros as a secret leads to a nonrefundable contract		
3.4	Users need to trust the Liquidity tools		
3.5	Lack of testing		

3 Issue Detail

3.1 Ethereum swap provider forces secrets to be 32 bytes

Severity	Status	Link	Remediation Comment
		issues/25	The issue has been

Description

The Ethereum swap provider truncates the user's input to 32 bytes, while the Bitcoin swap provider appears to not. (The Bitcoin provider was out of scope for this audit, but it was briefly reviewed for this issue.) This means that using a secret of a length other than 32 bytes allows an attacker to make it impossible to execute the Ethereum side of a cross-chain swap.

[EthereumSwapProvider.js:L29-L36](#)

```
// Contract
'60', '20', // PUSH1 20
```

```
// Get secret
'80', // DUP1
'60', '00', // PUSH1 00
'80', // DUP1
'37', // CALLDATACOPY
```

Example

1. Eve deploys a swap contract on Ethereum with the secret hash `sha256("short")` and Alice as the recipient.
2. Alice deploys a swap contract on Bitcoin with the the same secret hash and Eve as the recipient.
3. Eve collects the bitcoins by revealing the secret `"short"`.
4. Alice attempts to retrieve the ether using the same secret, but it is interpreted as length 32-byte (right-padded with zeros), and thus the hash doesn't match.
5. After the expiration has passed, Eve collects an ether refund.

Remediation

Be consistent across all providers in requiring a fixed size secret (and how shorter/longer secrets are interpreted) or in allowing arbitrary length secrets. (On the Ethereum side, this would mean looking at the call data size and providing the right call data and gas to the SHA256 precompile.)



Resolution (comment from the Liquidity team)

This has been addressed through including secret size limits in the contracts.

Here we have added the same limitation check on the bitcoin side:

<https://github.com/liquidity/chainabstractionlayer/blob/b32e18dd62cf76a65b51699c96020a9af8c2ae0c/packages/bitcoin-swap-provider/lib/BitcoinSwapProvider.js#L146>

3.2 Lack of parameter validation

Severity	Status	Link	Remediation Comment
		issues/23	The issue is currently under review

Description

`createSwapScript` accepts four parameters: `recipientAddress`, `refundAddress`, `secretHash`, and `expiration`. These are presumably passed in (through some other code) by an end user who is setting up one end of an atomic currency swap.

These parameters are injected directly into the constructed bytecode without any validation. This allows an attacker who can manipulate these values to inject arbitrary bytecode into the contract.

Even `verifyInitiateSwapTransaction` and `doesTransactionMatchSwapParams` perform no parameter validation, so if the same parameters are passed into those functions, it will appear that the deployed contract is indeed valid.

Examples

If an attacker is able to provide a `secretHash` that's longer than 32 bytes, all the bytes after the first 32 will be injected as bytecode that will execute quite early in the contract's execution. (For example, the injected code might transfer the contract's balance to an address controlled by the attacker regardless of what secret is provided.)

`secretHash` is only one place for injection. Injecting a long `expiration` could inject arbitrary bytecode as well as set `expirationPushOpcode` to any desired opcode greater than `0x60`. Long `recipientAddress` or `refundAddress` es could also wreak havoc.

Values that are shorter than expected could also cause serious problems by shifting existing bytecode up.

Remediation

It is highly recommended that all parameters are strictly validated for content and length as close as possible to their use (e.g. in `createSwapScript`). Addresses should be exactly 40 hex characters, and the secret hash should be exactly 64 hex characters. There are several options for validating the expiration timestamp, but we recommend requiring a fixed size for it. (5 bytes = 10 hex characters is a good choice. In 30,000 years, the new maintainer of this code can just adjust that to 6.)

A benefit of fixing the expiration size is that there's no longer a need for a dynamic `expirationPushOpcode` or dynamic computation of `dataSizeEncoded`, `redeemDestinationEncoded`, or `refundDestinationEncoded`. All of these can be hardcoded, reducing the possible attack surface further.



It should be noted that using a factory to deploy the contracts, as suggested in the issue "Users need to trust the Liquidity tools," would mean the parameter validation could happen directly in the factory (e.g. via Solidity's type safety).

Resolution (comment from the Liquidity team)

Type checks are being done here, the method used for the swap serves as self verification. The checks done in the code base are for client validation, and the chain serves as server validation. Any parameter which is not passed in terms of the agreed parameters between the two parties will render the swap invalid by default.

<https://github.com/liquality/chainabstractionlayer/blob/b32e18dd62cf76a65b51699c96020a9af8c2ae0c/packages/client/lib/Swap.js#L104>

3.3 Using zeros as a secret leads to a nonrefundable contract

Severity	Status	Link	Remediation Comment
		issues/24	The issue is currently under review

Description

Expiration is only checked *after* the secret is checked against the hash. To get a refund after the expiration has been reached, the caller needs to provide a secret that won't match the hash. In the existing JavaScript code, this is done by passing in an empty byte sequence as the secret. If the hash is actually

`sha256(0x0000...)`, then passing in an empty string will cause a transfer to the `recipientAddress` instead of the `refundAddress`.

[EthereumSwapProvider.js:L89-L92](#)

```
async refundSwap (initiationTxHash, recipientAddress, refundAddress, secretHash, expiration) {
  const initiationTransaction = await this.getMethod('getTransactionReceipt')(initiationTxHash)
  return this.getMethod('sendTransaction')(initiationTransaction.contractAddress, 0, '')
}
```

Example

1. Eve deploys a swap script on Bitcoin that expires in an hour and is locked with the hash `sha256(0x0000...)`. Alice is the recipient, and Eve is the refund address.
2. Alice deploys the corresponding script on Ethereum with the same hash and approximate expiration. Eve is the recipient, and Alice is the refund address.
3. Eve waits until the expiration has passed.
4. Alice attempts to get a refund from the Ethereum swap script, but because the empty string is actually the correct secret, the funds are transferred to Eve (instead of Alice, as expected).
5. Eve gets a refund on the Bitcoin side. (Either the Bitcoin script has the same flaw, or Eve simply knows to provide a random secret instead of an empty string when invoking the script.)

Note that if Alice is somehow savvy enough to recognize `sha256(0x0000...)`, the most she can do is execute the agreed-upon swap (collect the bitcoins but lose the ether). Thus Eve is at best stealing ether and at worst performing a fair swap.



Remediation

A few options come immediately to mind:

1. Recommend that users provide a random byte sequence when requesting a refund from an expired contract. Random bytes will never hash to the right secret hash, so that check will always fail, and the refund will be performed.
2. Jump straight to the expiration check if the message data is size 0. This way a user can safely request a refund by passing in a 0-length byte sequence.

The latter option seems better, in that it doesn't require that users do anything special (and it doesn't require that they use a certain specialized tool that does this for them).

3.4 Users need to trust the Liquidity tools

Severity	Status	Link	Remediation Comment
		issues/27	The issue is currently under review

Description

In the current model, users are given bytecode to deploy and have little recourse for reviewing that bytecode. They essentially agree to "Trust me: this contract is safe for you to deploy and use." Not only do they need to trust the author (Liquidity), but they need to rely on the provider's operations security. For example, DNS hijacking or a XSS bug may allow an attacker to modify the JavaScript that produces and verifies contracts.

Remediation



1. At a minimum, offline tools should be provided with well-known hashes so users can generate and validate contracts themselves.
2. Writing the contract in Solidity with constructor parameters would make it easy for anyone to verify the code before they use the contract and to deploy it easily using their own tools (e.g. Remix). This is a tradeoff: less trust required but higher gas costs than the current hand-optimized bytecode.
3. Using a factory pattern (a contract that deploys each swap contract) would require almost no verification by users. Once the factory is believed to be correct, it's impossible for any contract deployed by it to *not* be correct. At this point, users only need to verify that they're interacting with the correct factory contract. (They can do this by checking the address.)

Resolution (comment from the Liquality team)

On a standards level, we promote an amended version of BIP199 and have published ERC1630 for public review. The code base implements these standards and is provisioned under the MIT license which can be independently verified.

The intended means for executing the code is from the client, and we only provide hosted instances for testnet scenarios, for demonstration purposes.

3.5 Lack of testing

Severity	Status	Link	Remediation Comment
		issues/26	The issue is currently under review

Description

There are no automated tests that deploy scripts created by `createSwapScript` with various parameters and validate that the deployed contract functions properly.

For contracts that handle potentially large sums of ether, it's important to *at least* have automated tests that provide full coverage of the contracts. In the case of the script that produces these contracts, different lengths for different parameters can create quite different scripts, so it's important to get test coverage of as many of those combinations as possible too.

Remediation

Write automated tests that exercise the full range of parameter values and contract functionality.

Resolution (comment from the Liquality team)

We have introduced a set of unit, integration and code coverage within our build process.

<https://github.com/liquality/chainabstractionlayer/tree/dev/test>

Appendix 1 - Severity

A.1.1 - Minor

Minor issues are generally subjective in nature, or potentially deal with topics like "best practices" or "readability". Minor issues in general will not indicate an actual problem or bug in code.

The maintainers should use their own judgment as to whether addressing these issues improves the codebase.

A.1.2 - Medium

Medium issues are generally objective in nature but do not represent actual bugs or security problems.

These issues should be addressed unless there is a clear reason not to.

A.1.3 - Major

Major issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable, or may require a certain condition to arise in order to be exploited.

Left unaddressed these issues are highly likely to cause problems with the operation of the contract or lead to a situation which allows the system to be exploited in some way.

A.1.4 - Critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract.

Appendix 2 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") -- on its GitHub account (<https://github.com/ConsenSys>). CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.