Quantstamp DRAFT

Clanker

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	ERC20 Token Launchpad
Timeline	2025-01-08 through 2025-01-13
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	clanker-devco/DOCS
Source Code	• clanker-devco/contracts #3d82549
Auditors	• Ruben Koch Senior Auditing Engineer • Nikita Belenkov Auditing Engineer • Hytham Farah Auditing Engineer

Documentation quality	Low
Test quality	Medium
Total Findings	7 Unresolved: 7
High severity findings	0
Medium severity findings	1 Unresolved: 1
Low severity findings	3 Unresolved: 3
Undetermined severity findings	0
Informational findings	3 Unresolved: 3

Summary of Findings

In this audit, we reviewed the V2 version of the Clanker contracts. The contracts provide a platform where new ERC20 tokens can be deployed and the entire token supply is trustlessly locked into a single-sided liquidity position in a newly deployed UniswapV3 pool. The incurring trading fees in the pool are split between the Clanker team and the address on behalf of which the token was originally deployed.

The main findings are an unprotected swap potentially vulnerable to sandwich attacks (CLA-1), and a logical bug in case of updates to the reward recipient data structure (CLA-2). We also recommend that the project make use of private relayers to avoid some front-running issues on pool initialization (CLA-3).

While no high-severity issues were uncovered, the codebase would benefit from improved documentation. Some inconsistencies in the documentation were pointed out in the Auditor Suggestion section.

ID	DESCRIPTION	SEVERITY	STATUS
CLA-1	Unprotected Initial Eth Swap Vulnerable to Front-Running	Medium	Unresolved
CLA-2	`userTokenIds` Not Properly Updated in Case of Recipient-Replacement	Low	Unresolved
CLA-3	DoS Attack by Pre-Deploying a Pool for an Upcoming Token	Low	Unresolved
CLA-4	The Return Value of `transfer()` Is Not Checked	Low	Unresolved
CLA-5	Use of `transfer()` for Native Token Transfers	Informational	Unresolved
CLA-6	Missing `deadline` in Swap Requests	Informational	Unresolved
CLA-7	Advantageous First Buy of Newly Deployed Tokens	Informational	Unresolved

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Disclaimer

Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
    1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
    2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
    3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

- `Clanker.sol`
- `ClankerToken.sol`
- `LpLockerv2.sol`

Operational Considerations

- We assume that only the contracts in scope of the audit are not replaced with other implementations for deployment. If e.g. the `LiquidityLockverv2` contract were to be replaced with a contract that enables adjusting the liquidity position again instead of permanently locking it, the trust assumptions change significantly.
- As part of the `deployToken()` flow, the UniswapV3 pool is initialized at some `tick` price. This value is not input sanitized, but as it comes from a trusted source, we assume that it will be set at a reasonable value. The documentation mention a FDV of $30k.
- We assume that no fee-on-transfer ERC20 tokens are used for the token pairs in deployed UniswapV3 pools.
- The `Clanker` contract provides the option for the specified `_deployer` to perform an initial buy of tokens (CLA-7). The amount is currently unrestricted, we therefore assume that this value will remain within a fair range.
- The `updateImage()` function in the `ClankerToken` contract allows the deployer to set an arbitrary string as the token's image URL without validation. This introduces risks if a malicious or compromised deployer sets URLs leading to phishing sites, embedded scripts (e.g., XSS payloads), or SSRF vectors. Frontend applications displaying this URL should implement robust sanitation and validation to mitigate these risks. It is assumed that the system relies on external safeguards or frontend precautions for secure URL handling, and no on-chain mechanisms enforce URL format or domain restrictions.

Key Actors And Their Capabilities

Note, when a new token is created, the total supply of the token is locked into a UniswapV3 liquidity position and no privileged role can withdraw the liquidity again.

Owner

Responsibilities

- Manage contract administration, including adding/removing admins and setting fee recipients.
- Deploy and manage liquidity lockers.
- Collect fees, manage protocol parameters, and update rewards.
- Set and override fees and rewards for specific tokens.

Trust Assumptions

- Will not arbitrarily change the reward recipient for users.
- Will give fair warning and notice if they change the team portion of the fees (currently 60%)
- `updateLiquidityLocker()` will only be called with replacing contracts that also lock the liquidity position NFT indefinitely in the contract.

Exclusive Functions

1. **Clanker.sol**
    - `setAdmin()` : Assigns or removes admin roles.
    - `toggleAllowedPairedToken()` : Toggles allowed paired tokens for the Uniswap pools.
    - `setDeprecated()` : Toggles the contract deprecated or active. A deprecated contract can no longer deploy tokens.
    - `updateLiquidityLocker()` : Updates the liquidity locker address, which is the recipient of to-be-minted NFT of liquidity positions of pools of newly deployed tokens.
    - `deployToken()` : Deploys a new token, creates the pool and locks the total supply as liquidity in the pool.
2. **LpLockerV2.sol**
    - `setOverrideTeamRewardsForToken()` : Sets team rewards for a token. This can override the team's address and their share of fees for existing tokens.
    - `updateClankerFactory()` : Updates the Clanker factory, which is designed to be the `Clanker` contract.
    - `updateClankerTeamReward()` : Updates team reward percentages.
    - `updateClankerTeamRecipient()` : Updates the team reward recipient.
    - `addUserRewardRecipient()` : Adds a user reward recipient for accruing LP rewards.
    - `replaceUserRewardRecipient()` : Replaces a user reward recipient for the accruing LP rewards of the position.

Admin

Responsibilities

- Deploy new tokens for protocol use.

Trust Assumptions

- Will deploy the tokens with the correct user-specified parameters.

Exclusive Functions

- `Clanker.deployToken()` : Deploys a new token.

Findings

CLA-1

Unprotected Initial Eth Swap Vulnerable to Front-Running

● Medium ⓘ Unresolved

File(s) affected: `src/Clanker.sol`

Description: The `Clanker.deployToken()` function includes an unprotected ETH-to-paired-token swap when ETH is provided during deployment. This swap lacks slippage protection (`amountOutMinimum: 0`) and price limits (`sqrtPriceLimitX96: 0`), making it vulnerable to sandwich attacks.

This attack may be executed whenever `_poolConfig.pairedToken != weth`.

Exploit Scenario:

1. An attacker monitors the mempool for `deployToken()` transactions with ETH.
2. They front-run the transaction by executing a large ETH-to-paired-token swap, inflating the paired token price.
3. The victim's unprotected swap executes at the manipulated high price, yielding fewer tokens.
4. The attacker back-runs the transaction by reversing their position at the original or a more favorable price, profiting from the price difference.

Recommendation: Add slippage protection to the initial swap by setting a reasonable `amountOutMinimum` and enforce price limits with `sqrtPriceLimitX96` to mitigate sandwich attacks.

CLA-2
`userTokenIds` Not Properly Updated in Case of Recipient-Replacement

• Low ⓘ Unresolved

File(s) affected: `src/LpLockerv2.sol`

Description: When replacing the reward recipient for an existing `tokenId`, the wrong `_userTokenIds` array is traversed to delete the link between the old recipient and the `tokenId`. The `_userTokenIds` array of the new recipient, `recipient.recipient` is traversed and deleted from in case the tokenId is found. However, the oldRecipient.recipient should be checked and deleted from instead.

Therefore, currently, the `for` loop never deletes an element and the old recipient continues to hold the `tokenId` in its `_userTokenIds` array, though it was supposed to be deleted and solely held within the new recipient's array.

Recommendation: Traverse and delete from `_userTokenIds[oldRecipient.recipient]` array, not within the replacing recipient's `_userTokenIds` -array.

CLA-3
DoS Attack by Pre-Deploying a Pool for an Upcoming Token

• Low ⓘ Unresolved

File(s) affected: `src/Clanker.sol`

Description: When `Clanker.deployToken()` is called, a token is deployed via `CREATE2`, and a corresponding Uniswap V3 pool is created. What an attacker can do is monitor the mempool and front-run this `deployToken()` call with a `UniswapV3Factory.createPool()` and `UniswapV3Pool.initialize()` calls with the simulated address of the `CREATE2` call of the token deployment. This would create an empty pool for the token about to be deployed, initialized at some price, which would lead the original `deployToken()` call to revert, as the pool now already exists. However, simply using `createAndInitializePoolIfNecessary()` is also not a solution, as an initialization at an arbitrary price by some malicious third party could also impose significant problems.

Therefore, the pool must be initialized by the `Clanker` contract. The front-running issue can only be solved with either a fork of UniswapV3, hardcoding the `Clanker` contract as the only initializer or by making usage of private relayers for the `deployToken()` transaction to ensure the transaction cannot be simulated and front-run.

Recommendation: We recommend using a private relayers like flashbots to deploy tokens to avoid being front-ran.

CLA-4 The Return Value of `transfer()` Is Not Checked

• Low ⓘ Unresolved

File(s) affected: `src/LpLockerv2.sol`

Description: The `LpLockerv2.collectRewards()` and `withdrawERC20()` functions transfer tokens to a recipient, but do not check the return value of the transfer fails. This could lead to unexpected behavior and function succeeding without tokens being transferred.

Recommendation: Consider using `SafeTransfer()` from OpenZeppelin to check the return value for both standard and non-standard ERC-20 tokens.

CLA-5 Use of `transfer()` for Native Token Transfers

• Informational ⓘ Unresolved

File(s) affected: `src/LpLockerv2.sol`

Description: `LpLockerv2.withdraw()` performs native token transfers with the `transfer()` function. The functions `address.transfer()` and `address.send()` assume a fixed amount of gas to execute the call. The use of these functions protects against reentrancy attacks but comes with some caveats. With the uptake in smart contract accounts, there could be complex logic behind this call on the receiving end, which would use up all the gas, and, in the worst case, it could lead to failed transfers.

Recommendation: We recommend using the following pattern for native ETH transfer:

```
(bool success, bytes memory data) = _to.call{value: msg.value}(""); // Returns false on failure
require(success, "Failed to send Ether");
```

CLA-6 Missing `deadline` in Swap Requests

• Informational ⓘ Unresolved

File(s) affected: `src/Clanker.sol`

Description: The `ExactInputSingleParams` parameter initiated as part of swaps within the `Clanker.deployToken()` flow currently does not include the `deadline` field. This is particularly important for the ETH/WETH swap to the paired token, as the absence of a deadline could lead to unexpected pricing, in case the transaction execution is significantly delayed.

Recommendation: Add a `deadline` parameter to the `deployToken()` function and insert that value into the `ExactInputSingleParams` structs.

CLA-7 Advantageous First Buy of Newly Deployed Tokens

● Informational ⓘ Unresolved

Description: In a single-sided liquidity pool setup where the total supply of a newly created ERC20 token is locked, the price in the pool cannot fall below the initialization price. This is due to the constant product formula governing Uniswap V3 pools, which ensures that the price can only move as much to the right (during a buy) as it can move back to the left (during a sell). Since all liquidity is locked in the pool, the initialization price effectively becomes the minimum price for the token.

As part of the `deployToken()` function, the deployer performs an optional initial token swap during the same transaction in which the token is deployed and the pool is created. This initial buy is executed at the lowest possible price, guaranteed by the pool's configuration.

This setup introduces a noteworthy aspect: the initial token purchase by the deployer carries minimal risk. The only potential cost is the swap fee, a portion of which is returned to the deployer as part of the pool's fee-sharing mechanism. Furthermore, the purchased tokens can be sold back into the pool, restoring the price close to the initialization level and allowing the deployer to recoup nearly all paired tokens used in the initial buy (minus fees). This creates an advantageous situation for the deployer (or any initial buyer) to make a low-risk purchase while retaining full upside potential.

Recommendation: It would be reasonable to put an upper limit on the `msg.value` that can be used for the initial token buy on behalf of the `deployer` address.

Auditor Suggestions

S1 Remarks on Documentation

Unresolved

Description: 1. As clarified by the client, the provided documentation was mainly based on prior versions of the contracts. Some inconsistencies can be found between the code and the documentation, such as the documentation specifying hard-coded swap fees and revenue division between the Clanker team and deployer and a liquidity (which is however parameter-based and adjustable).

1. In `deployToken()` function, where a swap between `pairedToken` and `token` occurs, a comment says `The token we are exchanging from (ETH wrapped as WETH)`, which is not always the case, as the `pairedToken` doesn't have to be ETH/WETH, but can be any token. Similarly `The amount of ETH (WETH) to be swapped` is also not accurate.
2. In `LpLockerv2.onERC721Received`, the comment states `Only clanker team EOA can send the NFT here`, which is not accurate as only the token factory can send the LP token.

Recommendation: Adjust the comments to reflect the logic in the code.

S2 Incorrect Order of Fields in `ClaimedRewards` Event

Unresolved

File(s) affected: `src/LpLockerv2.sol`

Description: The `LpLockerV2.ClaimedRewards` event has the last two fields `totalAmount1` and `totalAmount0` mixed in order. Currently, the event emitted as part of `collectRewards()` would return the amount of `token1` as `amountToken0` and vice versa.

Recommendation: Reverse the ordering of the last two fields in the `ClaimedRewards` event.

S3 Ownership Can Be Renounced

Unresolved

Description: If the owner renounces their ownership, all ownable contracts will be left without an owner. Consequently, any function guarded by the `onlyOwner` modifier will no longer be able to be executed. Furthermore, we recommend the usage of the `Ownable2Step` library to assure that ownership has to explicitly accepted by the granted address, in order to avoid accidentally transferring to an incorrect address.

Recommendation: Confirm that this is the intended behavior. If not, override and disable the `renounceOwnership()` function in the affected contracts. For extra security, consider using a two-step process when transferring the ownership of the contract (e.g. `Ownable2Step` from OpenZeppelin).

S4 Approvals Are Not Reset After Use

Unresolved

File(s) affected: `src/Clanker.sol`

Description: When the protocol interacts with Uniswap, an approval for tokens about to be used is granted to the Uniswap router. It is a good practice to set approvals to 0 once they are no longer needed, which is not done in the following cases:

1. In `deployToken()` approval for `_poolConfig.pairedToken` is not removed in the case where `_poolConfig.pairedToken != WETH`
2. In `deployToken()` approval for `token` is not removed in the case where not all the liquidity is placed in the LP.

Recommendation: Make sure the approvals are reset in the cases above.

S5 Limitations Should Be Added to `updateClankerTeamReward()` and `setOverrideTeamRewardsForToken()`

Unresolved

File(s) affected: `src/LpLockerv2.sol`

Description: When `protocolReward0` and `protocolReward1` is calculated, `_clankerTeamReward` or, if set, the `_teamOverrideRewardRecipientForToken` variable is used to work out the percentage cut of the Clanker team. This variable can be set via the admin-controlled `updateClankerTeamReward()` and `setOverrideTeamRewardsForToken()`, but there is input sanitization being performed on the value of this `_clankerTeamReward` variable, which means it can be above `100` and lead to the transaction to revert due to mathematical error. Furthermore, a value of `100` would indicate that the Clanker team receives 100% of the accruing fees of the pool, overriding the share of the `deployer` address.

Recommendation: Set a reasonable upper limit of max `100` for the `_clankerTeamReward` and `_teamOverrideRewardRecipientForToken` variables.

S6 Gas Optimizations

Unresolved

File(s) affected: `src/LpLockerv2.sol`, `src/Clanker.sol`

Description: * In `LpLockerv2.replaceUserRewardRecipient()`, the `for` loop can be breaked out of if the array entry has been deleted.

- Functions that are marked as `public`, but are not called within the contract, can be marked as `external`.
- Variables that are only initially defined and never updated can be marked as `constant`, such as `weth` in `Clanker.sol`. Furthermore, variables that are only defined once within the constructor can be marked as `immutable`, such as `uniswapV3Factory` in `Clanker.sol`.

Recommendation: Consider implementing these gas optimizations.

S7 Duplicate Reward Recipient Registration in `addUserRewardRecipient()`

Unresolved

File(s) affected: `src/LpLockerv2.sol`

Description: The `LpLockerv2.addUserRewardRecipient()` can be used to initially set the reward recipient. However, it can still be called after a position has been registered. If that were to happen, there would multiple users registered in the `_userTokenIds` array as owning the same token.

Recommendation: The `addUserRewardRecipient()` function should revert if `_userRewardRecipientForToken[recipient.lpTokenId] != address(0x0)`. In such cases, the `replaceUserRewardRecipient()` function should be used instead.

S8 Incorrect Interface Used

Unresolved

File(s) affected: `src/Clanker.sol`, `src/Interface.sol`

Description: A pool is created when the `configurePool()` function is called and a `initialize()` function is called on that pool. However, the pool address is casted to an `IUniswapV3Factory`, whilst the `IUniswapV3Pool`, should be used. The used `IUniswapV3Factory` interface currently includes a definition for the `initialize()` function, which is incorrect, as the actual contract does not have that function.

Recommendation: Use the `IUniswapV3Pool` interface instead of `IUniswapV3Factory` for the `pool` address. Remove the `initialize()` from the `IUniswapV3Factory` interface.

S9 Code Best Practices

Unresolved

Description: * When a globally address-variable can be updated, it is a good best practice to assure that it is not the zero address, especially if it later on receives funds, e.g. `LPLockerV2.updateClankerTeamRecipient()`.

- In `LPLockerV2`, the `SafeERC721` and `e721Token` variables are unused, consider removing them.
- In `Clanker.sol.InvalidConfig` and `LPLockerV2.LockId` events are unused, consider removing them.
- The `maxUsableTick()` function in the `Clanker.sol` file should be defined within the `Clanker` contract, not outside of it.

- The `Clanker.setDeprecated()` function seems to be misnamed, as the value can be toggled, not just set once. Consider making it either callable once or call the function `toggleDeprecated()`.

Recommendation: Consider implementing these suggestions.

S10 Application Monitoring Can Be Improved by Emitting More Events

Unresolved

File(s) affected: `src/Clanker.sol`, `src/ClankerToken.sol`, `src/LpLockerv2.sol`

Description: The following functions update critical state variables without emitting events, reducing transparency and making changes harder to track.

Affected Functions

- `Clanker.sol`:
  - `setAdmin(address admin, bool isAdmin)`: Updates `admins[admin]`.
  - `toggleAllowedPairedToken(address token, bool allowed)`: Updates `allowedPairedTokens[token]`.
  - `setDeprecated(bool _deprecated)`: Updates `deprecated`.
  - `updateLiquidityLocker(address newLocker)`: Updates `liquidityLocker`.
- `ClankerToken.sol`:
  - `updateImage(string memory image_)`: Updates `_image`.
- `LpLockerv2.sol`:
  - `updateClankerFactory(address newFactory)`: Updates `_factory`.
  - `updateClankerTeamReward(uint256 newReward)`: Updates `_clankerTeamReward`.
  - `updateClankerTeamRecipient(address newRecipient)`: Updates `_clankerTeamRecipient`.
  - `setOverrideTeamRewardsForToken(uint256 tokenId, address newTeamRecipient, uint256 newTeamReward)`: Updates `_teamOverrideRewardRecipientForToken[tokenId]`.

Recommendation: Consider add these events.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Test Suite Results

Test suite data was obtained with `forge test`.

```
Ran 3 tests for test/LpLockerv2.t.sol:LpLockerv2Test
[PASS] test_constructor() (gas: 27727)
[PASS] test_ownerOnlyFunctions() (gas: 232579)
[PASS] test_withdrawAndReceive() (gas: 78996)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 6.56s (3.61s CPU time)

Ran 5 tests for test/LpMetaLocker.t.sol:LpMetaLockerTest
[PASS] test_constructor() (gas: 27576)
[PASS] test_onERC721Received() (gas: 205389)
[PASS] test_ownerOnlyFunctions() (gas: 120227)
[PASS] test_setUserFeeRecipients() (gas: 954436)
```

```
[PASS] test_withdrawAndReceive() (gas: 79236)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 38.02s (46.43s CPU time)

Ran 10 tests for test/Clanker.t.sol:ClankerTest
[PASS] test_claimRewards() (gas: 8672137)
[PASS] test_deployToken() (gas: 1056267132)
[PASS] test_deployTokenClankerPoolOnly() (gas: 9820117)
[PASS] test_deployTokenDegenPoolOnly() (gas: 9827153)
[PASS] test_deployTokenHigherPoolOnly() (gas: 9815634)
[PASS] test_generateSalt() (gas: 99461)
[PASS] test_ownerOnlyFunctions() (gas: 58716)
[PASS] test_specificNewSaltBehavior() (gas: 7425340)
[PASS] test_whoCanClaimRewardsOnCurrentBlondeLocker() (gas: 273912)
[PASS] test_whoCanClaimRewardsOnNewTokenWithNewContract() (gas: 8676999)
Suite result: ok. 10 passed; 0 failed; 0 skipped; finished in 39.49s (123.69s CPU time)

Ran 3 test suites in 39.52s (84.07s CPU time): 18 tests passed, 0 failed, 0 skipped (18 total tests)
```

Code Coverage

Unfortunately, `forge coverage` is running in to `Stack too deep` errors, which is a known issue with the legacy compilation pipeline with no fix.

Changelog

- 2025-01-13 - Initial report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp's mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp's team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp's collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Disclaimer

Quantstamp