Bramah Systems

Hegic Public Report

PROJECT: hegic/contracts-v1

May 2020

Prepared For:

Molly Wintermute | Hegic

molly.wintermute@protonmail.com

Prepared By:

Team @ Bramah | Bramah Systems, LLC.

info@bramah.systems

Table of Contents

Hegic Protocol Assessment

Executive Summary

Scope of Engagement

Bramah Systems, LLC was engaged in May of 2020 to perform a comprehensive security review of the Hegic protocol smart contract repository. An initial review was conducted over a period of four days by three members of the Bramah Systems, LLC. executive staff. During this period, one individual assessed all Solidity smart contract code (*.sol) as of commit [4fc63d173d54b1a63bb4151737011f7c4e1287e4] that was included within scope, along with JavaScript files (*.js) relevant to testing. JavaScript files were not assessed for their overall security. Following this, a secondary member of Bramah Systems staff also independently performed a full audit, without first having seen the results of the first -- allowing for unbiased bug finding. A third member of the Bramah team then validated each result found.

Bramah Systems completed the assessment using manual, static and dynamic analysis techniques.

Timeline

Audit Commencement: May 22nd, 2020

Report Delivery: May 29th, 2020

Engagement Goals

The primary scope of the engagement was to evaluate and establish the overall security of the Hegic system, with a specific focus on trading actions. In specific, the engagement sought to answer the following questions:

- Is it possible for an attacker to steal or freeze tokens?
- Does the Solidity code match the specification as provided?
- Is there a way to interfere with the balancing mechanisms?
- Are the arithmetic calculations trustworthy?

Protocol Specification

Minimal explicit specification was provided to the Bramah Systems team. While the organization does have a GitBook and multiple diagrams, the specific components of the smart contract should be described to relevant parties (namely, individuals wish to use the protocol)

which would allow them to better understand with which behaviors the dApp is interfacing. Those only interested in the financial mechanics of the protocol would seek the most benefit from this documentation (although we are not attesting towards the legal validity of these instruments).

Overall Assessment

Bramah Systems was engaged to evaluate and identify multiple security concerns in the codebase of the Hegic protocol architecture. During the course of our engagement, Bramah Systems noted numerous instances wherein the protocol deviated from established best practices and procedures of secure software development. **As of the time of publication of this report, Hegic has provided risk acceptance, justification, or a mitigation for all issues we have described below.**

The code has benefited from repeated audits. All publicly listed issues within the Hegic repository have been mitigated.

Throughout the process, We found Hegic to be incredibly responsive and detailed in the mitigations we suggested they implement. Our final issue repository totaled 120 issues, all of which Hegic has provided risk acceptance, justification, or a mitigation for all issues (all of which are addressed below).

In particular, Bramah Systems notes that the code is well commented, particularly in sections where understanding the developer's intent is essential. Additionally, the overall contract organization is consistent throughout (within contracts themselves and their overarching interactions with others).

Disclaimer

As of the date of publication, the information provided in this report reflects the presently held, commercially reasonable understanding of Bramah Systems, LLC.'s knowledge of security patterns as they relate to the Hegic Protocol, with the understanding that distributed ledger technologies ("DLT") remain under frequent and continual development, and resultantly carry with them unknown technical risks and flaws. The scope of the review provided herein is limited solely to items denoted within "Scope of Engagement" and contained within "Directory Structure". The report does NOT cover, review, or opine upon security considerations unique to the Solidity compiler, tools used in the development of the protocol, or distributed ledger technologies themselves, or to any other matters not specifically covered in this report.
The contents of this report must NOT be construed as investment advice or advice of any other kind. This report does NOT have any bearing upon the potential economics of the Hegic protocol or any other relevant product, service or asset of Hegic or otherwise. This report is not and should not be relied upon by Hegic or any reader of this report as any form of financial, tax, legal, regulatory, or other advice.

To the full extent permissible by applicable law, Bramah Systems, LLC. disclaims all warranties, express or implied. The information in this report is provided “as is” without warranty, representation, or guarantee of any kind, including the accuracy of the information provided.

Bramah Systems, LLC. makes no warranties, representations, or guarantees about the Hegic Protocol. Use of this report and/or any of the information provided herein is at the users sole risk, and Bramah Systems, LLC. hereby disclaims, and each user of this report hereby waives, releases, and holds Bramah Systems, LLC. harmless from, any and all liability, damage, expense, or harm (actual, threatened, or claimed) from such use.

Timeliness of Content

All content within this report is presented only as of the date published or indicated, to the commercially reasonable knowledge of Bramah Systems, LLC. as of such date, and may be superseded by subsequent events or for other reasons. The content contained within this report is subject to change without notice. Bramah Systems, LLC. does not guarantee or warrant the accuracy or timeliness of any of the content contained within this report, whether accessed through digital means or otherwise.

Bramah Systems, LLC. is not responsible for setting individual browser cache settings nor can it ensure any parties beyond those individuals directly listed within this report are receiving the most recent content as reasonably understood by Bramah Systems, LLC. as of the date this report is provided to such individuals.

General Recommendations

Best Practices & Solidity Development Guidelines

Usage of "Too Recent" Solidity Version

Contracts within the protocol make usage of a recent Solidity version, **0.6.8**.

It is suggested that while newer versions of Solidity **can** be used, ideally a version existing for a longer period of time with more established testing ***should*** be used. That noted, with current or intended functionality relying on more recently introduced aspects of the compiler, this may be reason to stay on a later version.

Resolution: We received a response from the team denoting that for Hegic protocol's functionality purposes the compiler version should not be below 0.6.x. Given this, Bramah and the team denote that using the most recent version of 0.6.8 is acceptable, as it introduces the most recent version changes of the 0.6.x line.

Usage of Unlocked Solidity Version

Contracts within the protocol make usage of a recent Solidity version, 0.6.8.

However, these versions are not locked to this specific version. As future Solidity versions may handle elements of the language differently than they do at the time of audit, it is recommended that the compiled version be fixed.

Resolution: Where applicable, the team has fixed the compiler version to 0.6.8. This issue has been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Multiplication after Division

Multiple functions within the **HegicOptions.sol** contract utilize multiplication following division. This design pattern is not recommended as it often leads to precision errors.

Fixed point numbers are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from. As a result, integer division must be used. Through performing all multiplication first, we mitigate rounding related concerns.

File: contracts/HegicOptions.sol

Lines: 248-249

File: contracts/HegicOptions.sol

Lines: 241-242

File: contracts/HegicOptions.sol

Lines: 241-243

File: contracts/HegicOptions.sol

Lines: 248-251

File: contracts/HegicOptions.sol

Lines: 248-250

File: contracts/HegicOptions.sol

Lines: 241-244

Resolution: This has been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Integer Overflow in Multiple Functions

A majority of the arithmetic functions within the protocol are at risk of integer overflow. One such function, **sqrt** within **HegicOptions.sol** is immediately vulnerable.

```
function sqrt(uint256 x) private pure returns (uint256 res) {
        res = x;
        uint256 z = (x + 1) / 2;
        while (z < res) (res, z) = (z, (x / z + z) / 2);
}
```

This function, when presented with a **res** value of max integer, will overflow (as it will return 0 divided by 2). This will cause the downstream function **getPeriodFee** to always return 0 when given a period of max integer size. While **period** limitations exist (set at **four weeks**), anyone modifying this protocol to reduce option time limits should be aware of this behavior. The **uint256 amount** performs similarly, in that an **amount** when multiplied by properly by the **period**, **impliedVolRate**, and **strike** (price) of the option, would allow one to return a value incongruent with the anticipated return (potentially leading to huge economic gain if exploited). This could be mitigated by the usage of **SafeMath**, which would protect against similar overflows. Properly written Slither or Echidna rules could detect this behavior. Where **SafeMath** is used, it should be made clear how the function utilized handles the results.

Resolution: Functions not previously guarded through usage of SafeMath have been covered as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Uninitialized Return Values

A majority of the contracts within the protocol have uninitialized return values. These values could lead to potentially unintended behavior. While values in Solidity are inherently set to 0 if not initialized, this could additionally lead to unintended behavior if interacting with the contracts manually.

Resolution: Where applicable, relevant return values were initialized. In order to maintain upgradability and consistency, the Hegic team did not modify any third party library code. These changes were implemented as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Potential Usage of Structs

The fees calculation function **fees** can benefit from the usage of a struct, rather than individually returning multiple values at once.

Resolution: Hegic and Bramah are in mutual agreement that introducing the experimental pragma for a singular struct is overall not worth the risk nor gas cost of invoking ABIEncoderV2.

Low Level Call is not Checked

The **payProfit** function in both put and call option contracts performs an unchecked **send** to an external address. This call does not check for the return value or handle errors. The recommended way of doing ether transfers is via the **transfer** function.

Resolution: Hegic notes that as the payProfit function calls a secondary protocol function, pool.send(address payable, uint amount), this behavior is anticipated.

Reentrancy

A majority of the contracts within the protocol express some form of reentrancy. Usage of **ReentrancyGuard** is suggested to mitigate potential vulnerability from these calls. Relevant calls are expressed within multiple tools, including **Crytic**, which was shared with the team.

Resolution: Hegic provided the following content to the Bramah team, which mitigates concerns regarding reentrancy. Hegic notes that:

> **The issue has been resolved using a check-effects-interactions pattern as it was suggested by Crytic:**
>
> **Reentrancy was found there: HegicERCPool.provide(uint256).**
>
> **token.transferFrom(..) was called before the changes took place in the contract:**

- **_mint(msg.sender,mint) (HegicERCPool.sol#103)**
- **_totalSupply = _totalSupply.add(amount) (@openzeppelin/contracts/token/ERC20/ERC20.sol#232)**

The code has been improved using a check-effects-interactions pattern: https://github.com/hegic/contracts-v1/blob/master/contracts/HegicERCPool.sol#L114-L140

So now all the external calls are made only after the changes took place in the contract.

The other issues found by Crytic.io have been resolved in a similar manner.

Bramah believes this inclusion adequately addresses relevant risk associated with potential reentrancy in the context of this audit.

Uninitialized Local Variables

A majority of the contracts within the protocol have uninitialized local variables. While values in Solidity are inherently set to 0 if not initialized, this could additionally lead to unintended behavior if interacting with the contracts manually. Restricting interactions to the smart contract address of a server acting as a delegator could resolve these concerns (if filters are built at this level).

Resolution: This has been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Variable Shadowing

Multiple variables suffer from variable shadowing in the protocol. Relevant variables impacted should be renamed to remove these concerns. A full list was supplied within the private repository between Bramah and Hegic.

Resolution: This has been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838. All results returned at present via static analysis indicate variable shadowing only exists now within third-party contracts.

Function should be declared External

Multiple functions should be declared **external**. The **external** attribute should be used for functions never called from the contract. Moreover, the **external** visibility modifier functions spends lesser amounts of gas compared to functions with **public** visibility modifiers. Each of

these functions should be assessed for whether or not **external** would benefit them.

File: contracts/HegicOptions.sol

Lines: 124-130

File: contracts/HegicOptions.sol

Lines: 196-198

File: contracts/HegicOptions.sol

Lines: 83-85

File: contracts/HegicOptions.sol

Lines: 179-190

File: contracts/HegicOptions.sol

Lines: 74-77

File: contracts/Migrations.sol

Lines: 16-18

File: contracts/Migrations.sol

Lines: 20-23

File: contracts/HegicPutOptions.sol

Lines: 82-85

File: contracts/HegicPutOptions.sol

Lines: 66-69

File: contracts/HegicPutOptions.sol

Lines: 57-60

File: contracts/HegicERCPool.sol

Lines: 189-192

File: contracts/HegicERCPool.sol

Lines: 156-162

File: contracts/HegicERCPool.sol

Lines: 168-171

File: contracts/HegicERCPool.sol

Lines: 69-71

File: contracts/HegicERCPool.sol

Lines: 147-150

File: contracts/HegicERCPool.sol

Lines: 177-183

File: contracts/HegicERCPool.sol

Lines: 199-206

File: contracts/HegicERCPool.sol

Lines: 52-55

File: contracts/HegicCallOptions.sol

Lines: 44-47

File: contracts/HegicCallOptions.sol

Lines: 53-56

Resolution: This has been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Usage of Block.timestamp

Miners can affect **block.timestamp** for their benefits. Thus, one should not rely on the exact value of **block.timestamp**. As a result of such, **block.timestamp** and **now** should traditionally only be used within inequalities (note: the protocol **does not** follow this strategy).

This risk is relatively minimal - a deviance of more than roughly 12 seconds from NTP will not allow an individual to connect to the Ethereum network. As a result, a time sensitive change, such as change of power at a certain time and date, could prove troublesome.

This noted, no particular test in the testing files provided by Hegic suggests particularly *highly* time sensitive features, and confirmation with the team ensured the general risk behind block timestamps is known.

Block numbers and average block time can be used to estimate time, but this is not future proof as block times may change (such as the changes expected during Casper). Substantial change to the representation of time would lead to deviance from intended ideals, but future solutions are expected to make note of this (due to the sensitive nature of time throughout the general corpus of published smart contracts).

Resolution: Team recognizes and has accepted the risk.

Integration and Third Party Code Risk

Third party integrations weigh a significant risk if untrusted parties are to be involved. While the general security stature of organizations Hegic has integrated with (and resultantly, built protocol integrations for) is quite high, this report (and present security analysis) cannot say for certain these integrations will be without flaw. It is notable that all integrations have seen some form of security scrutiny (be it a bug bounty, security audit, or security focused testing via the development team). That said, the scope of this audit does not cover the security of these integrations beyond the protocol integrations themselves. This audit also does not assess the impact of integrations such as third-party lenders, who we determine to be out of scope. Additional manual or test-net testing is suggested for these individuals.

Notably, minimal testing exists for each integration and verification does not exist for each step of the integration process. These should be introduced to mitigate these concerns.

Resolution: Team recognizes and has accepted the risk. Hegic notes that:

> **Hegic V1 contracts utilize following parties' integrations:**
>
> **Uniswap Protocol V2: https://github.com/Uniswap/uniswap-v2-core**
>
> **ChainLink ETH/USD price feed contract: https://feeds.chain.link/eth-usd**
>
> **Tests have been conducted manually in testnet while integrating Uniswap V2 and ChainLink as well as in automated tests written in .js**

Solidity “Style Guide” not Followed

There are multiple instances wherein the Solidity style guide is not followed. It is recommended that this style guide be followed for general readability purposes. Many linters exist that can assist in enforcing style guide constraints.

Resolution: Solhint is now integrated into the codebase. Existing issues have been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Outdated NPM Module Usage

Throughout the project, NPM modules are utilized in order to import various functionality (notably, **OpenZeppelin** contracts). While this practice enables relatively minimal modifications to be made in order to invoke certain functions securely (such as with **SafeMath**), these libraries must be continuously updated in order to ensure they are used securely.

Virtually every non-blockchain application has these issues because most development teams do not focus on ensuring their components/libraries are up to date. In the case of blockchain codebases, however, knowing all outside components utilized is critical.

It is suggested the following steps are followed (as noted by the OWASP project):

1. Identify all components and the versions you are using, including all dependencies.

   (NPM package lock can help determine these).
2. Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date.
3. Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.
4. Where appropriate, consider adding security wrappers around components to disable unused functionality and/or secure weak or vulnerable aspects of the component.

Resolution: Hegic notes that:

> **Npm modules [were] updated and will be updated again before the deployment of contracts.", resolving concern with this particular item.**

Bramah confirms the NPM update as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

ERC20 Race Condition

Throughout the project, **transfer** and **transferFrom** are utilized heavily. Both of these functions are vulnerable to an attack pattern as follows:

1. Alice allows Bob to transfer N of Alice's tokens ($N > 0$) by calling approve method on Token smart contract passing Bob's address and N as method arguments
2. After some time, Alice decides to change from N to M ($M > 0$) the number of Alice's tokens Bob is allowed to transfer, so she calls approve method again, this time passing Bob's address and M as method arguments
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls transferFrom method to transfer N Alice's tokens somewhere
4. If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain an ability to transfer another M tokens
5. Before Alice noticed that something went wrong, Bob calls transferFrom method again, this time to transfer M Alice's tokens.

Alice's attempt to change Bob's allowance from N to M ($N > 0$ and $M > 0$) made it possible for Bob to transfer $N + M$ of Alice's tokens, while Alice never wanted to allow so many of her tokens to be transferred by Bob. The attack described above is possible because the **approve** method overrides current allowance regardless of whether spender already used it or not, so

there is no way to increase or decrease allowance by certain value atomically, unless the token owner is a smart contract, not an account.

Resolution: There is some level of mitigation that can be performed:

EIP20 refers readers to the MiniMeToken implementation which mitigates concern by adding the line:

```
require((_amount == 0) || (allowed[msg.sender][_spender] == 0));
```

No backward compatible resolution to this problem is presently known, and the solution is still being discussed.

Highly Privileged Owner Account

Much of the power of the smart contract is centralized to the **owner**, an address granted special privileges to make certain modifications to the smart contract operation. Understandably, this poses a fairly unique challenge of ensuring this wallet (regardless of how it is managed) and the associated keys are secured. This centralization of power should be made clear to the users, especially depending on the level of privilege the contract allows to the owner.

Resolution: Hegic supplied the below within the primary README.md file, which is made visible to users viewing the GitHub page. This content could also be made clear to individuals utilizing the dApp.

Added to README.md:

[Added on 28.05.2020] ATTENTION! PLEASE READ THIS! During the first 90 days after the V1.1 contracts deployment (these contracts are not deployed yet) the owner address will be a highly privileged account. It means that the contracts will be under the owner's control. After 90 days from the contractCreationTimestamp time, these priviledges will be lost forever and the contracts owner will only be able to use setLockupPeriod (LockupPeriod value can only be <60 days), setImpliedVolRate, setMaxSpread functions of the contracts.

See below:

```
/**
 * @notice Can be used to update the contract in critical situations
 *         in the first 90 days after deployment
 */
function transferPoolOwnership() external onlyOwner {
   require(now < contractCreationTimestamp + 90 days);
```

```
                pool.transferOwnership(owner());
}
```

Bramah feels this information provided directly to those utilising the protocol adequately addresses the risk posed. Where possible, it should be made continually transparent to users that this is how the protocol will operate for the set time interval.

Specific Recommendations

Unique to the Hegic Protocol

Time Passage does not account for Leap Years and Seconds

Multiple variables are set relying upon the premise of time being roughly equivalent to one day, one week, and so on. However, because not every year equals 365 days and not even every day has 24 hours because of leap seconds, this one day/week/year period is inexact. Due to the fact that leap seconds cannot be predicted, an exact calendar library would require updating by an external oracle.

Note, the direct comparison of these variables within their respective functions poses additional concern, as discussed in "Usage of **block.timestamp**" above (namely, a proper comparison may not be set). It is worth noting that this has downstream implications on calculations utilising this passage of time (such as interest rates and APY calculations).

Resolution: The team recognizes and has accepted this risk. Hegic notes that the following:

> **Contracts rely on the time difference of block.timestamp-X and block.timestamp-Y. The only harm that can be done is that the period of an active option will be -1/+1 second long than the user thought it would last. The minimum period for the options contracts on Hegic is 1 day by design. The 1 second difference could be considered as a not critical one for the end-user.**

Bramah believes given this information the risk has been adequately assessed.

Excess Gas Consumption and Costly Loops

If the state variables **.balance** or **.length** are used several times, holding its value in a local variable is more gas efficient (as the variable does not need to be accessed every loop iteration).

Moreover, as Ethereum miners impose a limit on the total number of gas consumed in a block, if **array.length** is large enough, the function will exceed the block gas limit, and transactions calling it will never be confirmed. As a result, if an external entity is to influence **array.length**, this could pose an issue. Where possible, avoiding loops with a large number of iterations (or an unknown number of iterations) is advised.

Resolution: The Hegic team implemented a fix which stores these costly loops within local variables. Hegic notes that:

> **This particular method is only required for unlocking a number of expired options at once. If a user decides to use this method with a list that is too long, the transaction will be rejected. Nevertheless, she can use her list in parts or unlocking the expired options IDs one by one. In the end of the day, it doesn't influence the workflow and logics of the protocol.**

Incorrect ERC20 Specification

The **ERC20** specification included within the contract repository provides incorrect return values for the generic **ERC20**. This may cause the **ERC20** associated with the contract to work incorrectly with exchanges and other entities that rely on normal **ERC20** definitions.

Resolution: Hegic notes that:

> **Contracts use OZ IERC20 implementation for DAI token. Incorrect implementation was added for the future.**

As of commit hash b1e7ab61d318b83040d0649a6efe576792305838, this specification has been commented out, mitigating these concerns.

No zero address validation

Custom built functions of the contract (such as **send**) do not perform zero-address validation (and thus do not ensure funds are sent to a null address). Additionally, many functions could benefit from validating that they are not accessing a 0 index of a data type (such as arrays).

Resolution: This has been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Unclear documentation surrounding function provide

The **provide** function requires a minimum amount of tokens that should be received by a provider -- however, **provide** enforces this value as the actual maximum tokens that can be received.

Resolution: Greater documentation as to the functionality and components of provide has been provided as of commit hash 45d27b447418b8841770733a826b3c0e0a5c7e45.

Function approve response is ignored

The **approve** function response is ignored and does not produce any actionable events

regardless of what is returned (this includes the lack of an **emit** event).

Resolution: Team has accepted the risk.

Unclear documentation surrounding function setLockupPeriod

The **setLockupPeriod** function requires a **value** of less than 60 days -- however, stating as a value if the provided number does not follow this restriction -- “LockupPeriod is too small”.

Resolution: This has been resolved as of commit hash b1e7ab61d318b83040d0649a6efe576792305838.

Unclear documentation surrounding function unlock

The **unlock** function requires a **value** of less than **lockedAmount** -- however, stating as a value if the provided number does not follow this restriction -- “Pool: Insufficient locked funds”.

Resolution: Greater documentation as to the functionality and components of unlock has been provided as of commit hash 45d27b447418b8841770733a826b3c0e0a5c7e45.

Nothing preventing admin actions as per documentation

The Hegic documentation states the following:

Hegic Protocol V1 contracts admin key holder CAN'T:

call withdraw function (can't withdraw users' funds from the pools contracts)

call lock function (can't lock funds on the liquidity pools contracts)

call unlock function (can't unlock funds on unexercised active contracts)

call transfer function (can't send users' writeETH / writeERC tokens)

call exercise function (can't exercise users' active options contracts)

However, there is no limitation of any sort on the **owner** calling these functions. Resolution of this deviance is solved through limiting all calls to the function to block the owner address.

Resolution: Hegic notes that:

> **Added to README.md:**
>
> **[Added on 28.05.2020] ATTENTION! PLEASE READ THIS! During the first 90 days after the V1.1 contracts deployment (these contracts are not deployed yet) the owner address will be a highly privileged account. It means that the contracts will be under the owner's control. After 90 days from the contractCreationTimestamp time,**

these priviledges will be lost forever and the contracts owner will only be able to use setLockupPeriod (LockupPeriod value can only be <60 days), setImpliedVolRate, setMaxSpread functions of the contracts.

Bramah believes that this inclusion adequately illustrates the risk for usage with the contract.

Previous Audit Findings & Bug Bounties

Prior to this audit, the protocol underwent an audit by an external auditing firm. The results of the previous audit should be taken heavily into consideration and applicable fixes where denoted should be made. As multiple findings were present in the previous audit and this audit, it is suggested that careful precautions are made during deployment and that a bug bounty should be considered, alongside an emergency switch be made.

Resolution: Hegic provides the following: All ten issues that were found during the previous audit were fixed. Commits are below:

hegic/contracts-v1@b3323ca
hegic/contracts-v1@c5f9e57
hegic/contracts-v1@d824e6a
hegic/contracts-v1@7c7491f
hegic/contracts-v1@4f7530a
hegic/contracts-v1@2232c48
hegic/contracts-v1@22b328d
hegic/contracts-v1@34b784f
hegic/contracts-v1@e71b110
hegic/contracts-v1@09150eb

Bramah confirms the existence of the relevant mitigation steps following the initial public audit. We suggest additional research be provided into attacks involving liquidity pool manipulation.

Toolset Warnings

Unique to the Hegic Protocol

Overview

In addition to our manual review, our process involves utilizing concolic analysis and dynamic testing in order to perform additional verification of the presence security vulnerabilities. An additional part of this review phase consists of reviewing any automated unit testing frameworks that exist.
The following sections detail warnings generated by the automated tools and confirmation of false positives where applicable, in addition to findings generated through manual inspection.

Test Coverage

The contract repository presents minimal unit test coverage throughout. This testing provides a variety of unit tests which encompass the various operational stages of the contract. The Hegic protocol (and its relevant components and their respective subcomponents) possesses basic tests validating functionality and ensuring that certain behaviors (those relating to erroneous or overflow-prone input) do not see successful execution.

The Hegic team constructed tests in both JavaScript and native Solidity.

Static Analysis Coverage

The contract repository underwent heavy scrutiny with multiple static analysis agents, including:

- Securify
- MAIAN
- Mythril
- Oyente
- Slither
- SmartCheck

In each case, the team had mitigated relevant concerns raised by each of these tools. In particular, many tools pointed to potential areas of reentrancy, in which multiple state variables are written following external calls. For each of these individual calls, Bramah confirmed the lack of existence of a mitigating factor (we suggest the usage of **ReentrancyGuard**).

Fuzzing Coverage

The contract repository underwent heavy scrutiny with fuzzing utility Echidna, running custom rulesets as well as those within the Crytic.io platform. For each potential area of vulnerability Crytic uncovered, the Hegic team resolved all relevant events.

Directory Structure

At time of initial review, the directory structure of the Hegic contracts (**./contracts**) repository was as follows:

```
├─── LICENSE
├─── README.md
├─── contracts
│    ├─── HegicCallOptions.sol
│    ├─── HegicERCPool.sol
│    ├─── HegicETHPool.sol
│    ├─── HegicOptions.sol
│    ├─── HegicPutOptions.sol
│    ├─── Interfaces.sol
│    └─── Migrations.sol
├─── migrations
│    └─── 1_initial_migration.js
├─── package-lock.json
├─── package.json
├─── test
│    ├─── HegicCallOptions.js
│    ├─── HegicERCPool.js
│    ├─── HegicETHPool.js
│    ├─── HegicPutOptions.js
│    ├─── Import.flat
│    ├─── Prices.js
│    ├─── TestContracts.sol
│    └─── utils
│         └─── utils.js
└─── truffle-config.js
```

4 directories, 21 files

Appendix

Mythril Detection Capabilities

Issue	Description	Mythril Detection Module(s)	References
Unprotected functions	Critical functions such as sends with non-zero value or suicide() calls are callable by anyone, or msg.sender is compared against an address in storage that can be written to. E.g. Parity wallet bugs.	Unchecked_suicide, Ether_send unchecked_retval	
Missing check on CALL return value		unchecked_retval	Handle errors in external calls
Re-entrancy	Contract state should never be relied on if untrusted contracts are called. State changes after external calls should be avoided.	external calls to untrusted contracts	Call external functions lastAvoid state changes after external calls
Multiple sends in a single transaction	External calls can fail accidentally or deliberately. Avoid combining multiple send() calls in a single transaction.		Favor pull over push for external calls

External call to untrusted contract		external calls to untrusted contracts	
Delegatecall or callcode to untrusted contract		delegatecall_forward	
Integer overflow/underflow		integer	Validate arithmetic
Timestamp dependence		Dependence on predictable variables	Miner time manipulation
Payable transaction does not revert in case of failure			
Use of tx.origin		tx_origin	Solidity documentation, Avoid using tx.origin
Type confusion			
Predictable RNG		Dependence on predictable variables	
Transaction order dependence		Transaction order dependence	Front Running
Information exposure			
Complex fallback function (uses more than 2,300 gas)	A too complex fallback function will cause send() and transfer() from other contracts to fail. To implement this we first need to fully implement gas simulation.		

Use require()instead of assert()	Use assert() only to check against states which should be completely unreachable.	Exceptions	Solidity docs
Use of deprecated functions	Use revert()instead of throw(), selfdestruct() instead of suicide(), keccak256() instead of sha3()		
Detect tautologies	Detect comparisons that always evaluate to 'true', see also #54		
Call depth attack	Deprecated		

Oyente Detection Capabilities

Issue	Description
Re-entrancy	Contract state should never be relied on if untrusted contracts are called. State changes after external calls should be avoided.
Timestamp Dependence	The timestamp of the block can be manipulated by the miner, and so should not be used for critical components of the contract. Block numbers and average block time can be used to estimate time, but this is not future proof as block times may change (such as the changes expected during Casper).
Assertion Failure	An assertion is a boolean expression at a specific point in a program which will be true unless there is a bug in the program.

	Assertion failures as such denote critical instances in which assumptions made by the developer no longer hold to be true.
Callstack Depth Attack	Deprecated
Transaction Order Dependence (TOD)	Since a transaction is in the mempool for a short while, one can know what actions will occur, before it is included in a block. This can be troublesome for things like decentralized markets, where a transaction to buy some tokens can be seen, and a market order implemented before the other transaction gets included.
Parity Multisig Bug 2	Unchecked kill/selfdestruct functions, such as those within the Parity Multisig Bug 2 can lead to destruction of the contract, sending funds to the given address provided.

Slither Detection Capabilities

Detector	What it detects	Impact	Confidence
name-reused	Contract's name reused	High	High
rtlo	Right-To-Left-Override control character is used	High	High
shadowing-state	State variables shadowing	High	High
suicidal	Functions allowing anyone to destruct the contract	High	High
uninitialized-state	Uninitialized state variables	High	High
uninitialized-storage	Uninitialized storage	High	High

	variables		
arbitrary-send	Functions that send ether to arbitrary destinations	High	Medium
controlled-delegatecall	Controlled delegatecall destination	High	Medium
reentrancy-eth	Reentrancy vulnerabilities (theft of ethers)	High	Medium
erc20-interface	Incorrect ERC20 interfaces	Medium	High
erc721-interface	Incorrect ERC721 interfaces	Medium	High
incorrect-equality	Dangerous strict equalities	Medium	High
locked-ether	Contracts that lock ether	Medium	High
shadowing-abstract	State variables shadowing from abstract contracts	Medium	High
tautology	Tautology or contradiction	Medium	High
boolean-cst	Misuse of Boolean constant	Medium	Medium
constant-function-asm	Constant functions using assembly code	Medium	Medium
constant-function-state	Constant functions changing the state	Medium	Medium
divide-before-multiply	Imprecise arithmetic operations order	Medium	Medium
reentrancy-no-eth	Reentrancy vulnerabilities (no theft of ethers)	Medium	Medium
tx-origin	Dangerous usage of tx.origin	Medium	Medium

unchecked-lowlevel	Unchecked low-level calls	Medium	Medium
unchecked-send	Unchecked send	Medium	Medium
uninitialized-local	Uninitialized local variables	Medium	Medium
unused-return	Unused return values	Medium	Medium
shadowing-builtin	Built-in symbol shadowing	Low	High
shadowing-local	Local variables shadowing	Low	High
void-cst	Constructor called not implemented	Low	High
calls-loop	Multiple calls in a loop	Low	Medium
reentrancy-benign	Benign reentrancy vulnerabilities	Low	Medium
reentrancy-events	Reentrancy vulnerabilities leading to out-of-order Events	Low	Medium
timestamp	Dangerous usage of block.timestamp	Low	Medium
assembly	Assembly usage	Informational	High
boolean-equal	Comparison to boolean constant	Informational	High
deprecated-standards	Deprecated Solidity Standards	Informational	High
erc20-indexed	Un-indexed ERC20 event parameters	Informational	High
low-level-calls	Low level calls	Informational	High
naming-convention	Conformance to Solidity naming conventions	Informational	High
pragma	If different pragma directives are used	Informational	High
solc-version	Incorrect Solidity version	Informational	High
unused-state	Unused state variables	Informational	High

reentrancy-unlimited-gas	Reentrancy vulnerabilities through send and transfer	Informational	Medium
too-many-digits	Conformance to numeric notation best practices	Informational	Medium
constable-states	State variables that could be declared constant	Optimization	High
external-function	Public function that could be declared as external	Optimization	High