Primitive V2

Smart Contract Security Assessment

18.10.2021

ABSTRACT

Dedaub was commissioned to perform an updated audit on a recent version of Primitive V2. This audit was meant to cover architectural changes and to go through some of the calculations in more depth prior to final deployment. This logic was subdivided over two repositories: (i) protocol core and (ii) periphery contracts at commit hashes `2273c3a480bd4210d1df3edfaff8f1c8891c473b` and `0367217aa3677845e03f99bba2dd250e12238525` respectively. Two auditors worked over the codebase over four days. The audit examined:

- The core protocol contracts
- The periphery protocol contracts
- Math libraries written by the Primitive V2 authors, including units

No Critical or High Severity vulnerabilities were found. The audit did not include ABDKMath or standard OpenZeppelin code.

Primitive V2 is the second version of the Primitive Protocol, a peer-to-peer system for exchanging option-like payoffs. Primitive allows users to supply tokens and receive a position with an option payoff of their choice to which they can convert instantly.

The protocol consists of the core (low-level functionality) and the periphery (high-level functionality) contracts. The most interesting aspect of this protocol is the use of "replicating market makers". In a nutshell, an "option-like" payoff is encoded in the value of liquidity tokens, as the curve of the AMM changes according to the parameters of the black-scholes option pricing equation. As the time gets closer to maturity, the AMM behaves more closely to a constant sum market maker, further incentivizing arbitrageurs to tilt the composition of the liquidity pools towards market conditions.

The central functionality of the core contracts lies in the PrimitiveEngine contract. The protocol dictates that for each token pair there has to exist only one PrimitiveEngine. This is ensured by the PrimitiveFactory contract, which is used to deploy new PrimitiveEngine contracts. PrimitiveEngine allows the parametrization of option payoffs by supporting multiple curves per pair of tokens. It also implements other main functions of the protocol such as liquidity management and swaps between the risky and the stable tokens.

Periphery contracts are high-level “helper” contracts that aim to make the interaction with the protocol more optimal and secure. The functionality is clearly divided into several base contracts that together synthesize the main periphery contract, PrimitiveHouse. The interaction with different functions of `PrimitiveEngine` is achieved by implementing corresponding wrapper and callback functions in `PrimitiveHouse and its base contracts`. For example, a user that wishes to allocate liquidity to a curve calls the wrapper method PrimitiveHouse::allocate, which calls PrimitiveEngine::allocate. At the end of its execution, PrimitiveEngine::allocate calls PrimitiveEngine::allocateCallback that handles the transfer of user funds (for which PrimitiveHouse has approval by the user) to PrimitiveEngine.

Security Opinion

The audit’s main target is security threats, i.e., what the community understanding would likely call “hacking“, rather than regular use of the protocol. Functional correctness (i.e., issues in “regular use“) was a secondary consideration, however intensive efforts were made to check the correct application of the mathematical formulae in the reviewed code. Functional correctness relative to low-level calculations (including units, scaling, quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing. Although a number of simulations have been carried out, the crypto-economic effectiveness in a real-world scenario is as yet unknown. Therefore, the financial viability of this protocol in real market conditions cannot be fully established.

In terms of architecture, Dedaub notes that there are several checks that currently do not allow attackers to drain the protocol. However, if, as the code evolves, some checks are relaxed, it may become possible to hack the protocol. As in the previous audit, most of the same conditions keep MT32FINDthe protocol safe:

1) There is only one implementation of Primitive House.
2) Primitive house always:
    a) Checks that callbacks come from valid engines.
    b) Uses the “self permit” pattern.
    c) In its callbacks, always transfers to `msg.sender`, which, combined with (b) makes sure that this is the appropriate engine
3) Primitive engine always calls back `msg.sender`, and keeps accounting for `msg.sender.`
4) All withdrawals have `msg.sender` baked in.
5) Reentrancy guards on all external functions in `PrimitiveEngine`, and most of `PrimitiveHouse.`
6) Token transfers into the system are agnostic (balance is checked to verify)
7) Will only be deployed for straightforward ERC20 tokens (i.e., not aTokens, cTokens, etc.)

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues that affect the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: -User or system funds can be lost when third party systems misbehave. -DoS, under specific conditions. -Part of the functionality becomes unusable due to programming error.
LOW	Examples: -Breaking important system invariants, but without apparent consequences. -Buggy functionality for trusted users where a workaround exists. -Security issues which may manifest when the system evolves.

Issue resolution includes “dismissed”, by the client, or “resolved”, per the auditors.

CRITICAL SEVERITY:

[No critical severity issues]

HIGH SEVERITY:

[No high severity issues]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Second depositor can get `MIN_LIQUIDITY` for free	**OPEN**
	In `PrimitiveEngine::create`, the first user to create a position is generally at a high disadvantage, compared to the rest of the users. Apart from the usual risk of additional impermanent loss, some of their liquidity token is also burned. This means that the second depositor will automatically take a large portion of this burned liquidity. On the other hand, if a similar treatment is applied to `PrimitiveEngine::allocate`, this wouldn't be the case anymore.	
L2	Inaccurate token ratios results in lost funds	**OPEN**
	In `PrimitiveEngine::allocate,` the amount of tokens to deposit into a liquidity pool is decided by the user. If the amount of any of these tokens is inaccurate in respect to the other token in the pair, tokens are lost to the rest of the LP pool holders. A more	

reasonable behavior would be to withdraw the appropriate number of tokens via a simple calculation.		
L3	Recipient in `Withdraw` event can be address 0	**OPEN**
In case the withdrawal recipient provided by the caller of `MarginManager::withdraw` is the 0 address, the actual recipient is the PrimitiveHouse contract itself. However, this mapping is not reflected in the recipient parameter of Withdraw event.		
L4	120s "extra time" could change the option payoff	**OPEN**
`PrimitiveEngine::swap`, allows for an additional 120s of swaps by the market participants after the maturity of a pool. This is meant to allow future extension of this logic into option payoffs. However, at the "extra time" the option parameter τ is always 0 so the AMM's curve allows for no slippage. At this point, if the price of the strike tokens relative to the risky tokens is swayed in the opposite direction, the composition of the LP token is expected to follow suit. Although this is rare, with short-term options, the mechanics at maturity may not truly match that of a traditional option.		

OTHER/ ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Funds in `PrimitiveHouse` can be sweeped by anyone	**INFO**

The `PrimitiveHouse` contract is not expected to store any funds. This is clear, as methods `unwrap`, `sweepToken` and `refundETH` of contract `CashManager`, which `PrimitiveHouse` extends, do not enforce any restriction on the caller or the recipient of the funds. Nevertheless, funds could be left in the `PrimitiveHouse` contract by user mistake, e.g., a user might perform a withdrawal with the `PrimitiveHouse` contract as recipient and not follow it up with an `unwrap` of `sweepToken` call to transfer the funds to themselves. A malicious entity can monitor the contract for leftover funds and capitalize on such mistakes, thus it should be clearly stated in the docs that no funds should be left in the `PrimitiveHouse` contract.

ID	Description	STATUS
A2	Calibration structure can be made more efficient	**INFO**

```
struct Calibration {
    uint128 strike;
    uint64 sigma;
    uint32 maturity;
    uint32 lastTimestamp;
    uint32 creationTimestamp;
}
```

The calibration struct currently occupies 288 bits, however sigma can be reduced to 32 bits (it is bounds checked), which enables the entire struct to fit into one word.

ID	Description	STATUS
A3	`SwapManager::swap` reentrancy guard has been removed	**INFO**

In the latest version of the protocol the reentrancy guard on method swap of `SwapManager` has been removed. This change does not leave any room for attackers, as there is a reentrancy guard on `PrimitiveEngine::swap`, which is called by `SwapManager::swap`, and at the same time all state updates in both functions happen prior to any transfers. Even though this guard is not necessary it can serve as an extra layer of security at very little cost.

A4	Missing engine address check	**INFO**

Methods `allocate` and `remove` of `PrimitiveHouse` could add a check to revert with a `EngineNotDeployedError` in case the engine address is 0, as the error already exists and is used for such a case in `PrimitiveHouse::create`.

A5	No HouseBase constructor checks	**INFO**

The `HouseBase` constructor could implement simple checks, e.g., provided addresses are not equal to 0, to prevent accidental mistakes at contract creation.

A6	Incorrect Multicall Error Handling	**INFO**

The `Multicall` error handling mechanism assumes a fixed ABI for error messages. This would have worked in Solidty 0.7.x for the default `Error(string)` ABI, however Solidity has custom ABIs for 0.8.x that can encode valid error with a shorter `returndata`. The correct way to propagate errors is to re-raise them (e.g., by copying the `returndata` to the revert input data).

DISCLAIMER

ABOUT DEDAUB

Dedaub offers technology and auditing services for smart contract security. The founders, Neville Grech and Yannis Smaragdakis, are top researchers in program analysis. Dedaub's smart contract technology is demonstrated in the contract-library.com service, which decompiles and performs security analyses on the full Ethereum blockchain.