Primitive V2 Protocol audit

Smart Contract Security Assessment

28.09.2021

ABSTRACT

Dedaub was commissioned to perform a security audit on part of an in-development version of Primitive V2. This was subdivided over two repositories: (i) protocol core and (ii) periphery contracts at commit hashes `c959452c7f96edd05f92a70aa34d3e88b88fce92` and `ea6423ad4aebdae98808eb3136b10df5e5d7dd6e` respectively. Two auditors worked over the codebase over one week. The audit therefore examined:

- The core protocol contracts
- The periphery protocol contracts
- The documentation, for crypto-economic concerns

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

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 protocol consists of the core (low-level functionality) and the periphery (high-level functionality) contracts.

The central functionality of the core contracts lies in the PrimitiveEngine contract. The protocol dictates that for each option tokens 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, borrowing 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`. 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") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e., full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. 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. In addition, to our knowledge Primitive is building the first options platform using replicating market makers, and although a number of simulations have been carried out, the crypto-economic effectiveness in a real-world scenario is as yet unknown.

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 increasingly possible to hack the protocol. These are the following conditions and checks we have identified, which keep the protocol safe:

1) There is only one implementation of Primitive House.
2) Primitive house always:
    a) Sets “payer“ to msg.sender when calling PrimitiveEngine.
    b) Checks that callbacks come from valid engines.
    c) Uses the “self permit” pattern.
    d) 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.)

For full disclosure, Dedaub has previously worked with the Primitive team, finding a critical vulnerability in the live version of the V1 protocol, and assisting the white-hat exploitation of it.

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:

ID	Description	STATUS
H1	Incorrect margins mapping indexing in `SwapManager::swap`	**RESOLVED**

Method `SwapManager::swap` performs an internal balance deposit on the `margins` mapping when `params.toMargin` evaluates to true. The margins mapping is a double mapping, going from a `PrimitiveEngine` address to a user address to the user's margin. Instead of indexing the first mapping with `params.engine` and the second with `msg.sender`, indexing is implemented the other way around, leading to invalid `PrimitiveHouse` state.

ID	Description	STATUS
H2	Incorrect margin deposit value in `SwapManager::swap`	**RESOLVED**

There is a second issue with the `margins` mapping update operation in `SwapManager::swap` (the one discussed in issue H1). The deposited amount of tokens is `deltaIn` instead of `deltaOut`, which creates inconsistency between the states of PrimitiveEngine and PrimitiveHouse and in general is not consistent with the protocol's logic. The following snippet addresses both this issue and issue H1:

```
if (params.toMargin) {
    margins[params.engine][msg.sender].deposit(
        params.riskyForStable ? params.deltaOut : 0,
        params.riskyForStable ? 0 : params.deltaOut
    );
}
```

[*After our report*, the Primitive Finance team identified that the `deltaOut` amount was deposited in the wrong margin, i.e., `deltaOut` risky in stable margin and the other way around. Consequently, the above example has the ternary operator result expressions inverted in its final form.]

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Secret Flash-Loan Functionality	**DISMISSED**
`PrimitiveEngine::swap` can be actually used to get flash loans from the Primitive reserves. However, this functionality is not documented and may have been implemented by mistake. One can get flash loans by implementing a contract with the `swapCallback` function. When this gets called by the engine, the output ERC20 tokens have already been transferred to the engine contract, and all that is required for the rest of the transaction to succeed is to transfer the input tokens back.		
L2	Incorrect Multicall Error Handling	**OPEN**
The Multicall error handling mechanism assumes a fixed ABI for error messages. This would have worked in Solidity 0.7.x for the default `Error(string)` ABI. However, Solidity has custom ABIs for 0.8.x that can encode valid errors 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).		

L3	Mixing Reserve Balance Mechanisms	**DISMISSED**
The balances of the two reserve tokens in the engine are sometimes tracked by incrementing/decrementing internal counters and sometimes by checking `balanceOf()`. This not only causes the system to read more storage locations, and thus consume more gas, but it also automatically disqualifies tokens that have dynamic balances such as aTokens.		
L4	Fixed Swap Fee Might Not Compensate Theta Decay For All Asset Pairs	**SPEC CHANGED**
Options, manifesting themselves as asset pairs of different types will encode different proportions of intrinsic and extrinsic value. Although the swap fee is meant to compensate for theta decay, it seems strange that this cannot be set per curve or per token pair. We note however that other important parameters such as sigma are customizable.		

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	`Transfers::safeTransfer` always returns true	**RESOLVED**
`Transfers::safeTransfer` return value is always true (as noted in a comment), thus can be removed as an optimization.		
A2	No zero liquidity check in `PrimitiveEngine::remove`	**RESOLVED**
`PrimitiveEngine::remove` does not revert in case of 0 provided liquidity, which leads to unnecessary computation and gas fee for the user. `PrimitiveHouse::remove` implements an early check for such a scenario.		
A3	Redundant Bookkeeping and Transfers	**DISMISSED**
The architecture as it currently stands, and the relationship between `PrimitiveHouse` and `PrimitiveEngine` causes multiple token transfers to intermediate contracts, and multiple layers of bookkeeping, with some redundancy. This causes the application to consume more gas. **DISMISSED:** The specific architecture is highly desired by the protocol developers. Nevertheless, a few transfer operations have been optimized.		
A4	No engine-risky-stable sanity check in `PrimitiveHouse` `create` and `allocate` methods	**RESOLVED**
In `PrimitiveHouse::create` and `PrimitiveHouse::allocate` the user has to provide the `PrimitiveEngine` address and the addresses of the risky and stable tokens, while there is no early check that ensures the pair of risky and stable tokens provided corresponds to the engine address. This check is implemented in the respective callback functions, maintaining the security of the protocol. However, the		

execution of the contract will only revert at such a late point (i.e., in the callback) even if a user provides a wrong engine, risky and stable tokens triplet by mistake, leading to unnecessary gas consumption, which could have been avoided with an early check.

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, as well as a public bug bounty program.

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.