Chainlink VRF V2 audit

Smart Contract Security Assessment

30.09.2021

ABSTRACT

Dedaub was commissioned to perform a security audit on the 2nd version of the VRF smart contracts, on branch "develop" of https://github.com/smartcontractkit/chainlink at commit hash 7547b562579b4f201994fd85fb50d799ae1d918a. Two senior auditors worked over the codebase over one week. The audit examined the contracts:

- contracts/src/v0.8/dev/VRFConsumerBaseV2.sol
- contracts/src/v0.8/dev/VRF.sol. This contract was mostly inspected as a diff from the v1 (v0.6/VRF.sol) to v2 (v0.8/dev/VRF.sol) and the cryptographic operations were mostly considered trusted since this code has been well-audited and tested before.
- contracts/src/v0.8/dev/VRFCoordinatorV2.sol
- any other functionality highly related with these contracts, but without exhaustive inspection. Specifically, the offchain operations written in vrf_coordinator_v2.go.

SETTING AND CAVEATS

The exhaustively audited code base is of small size, at around 1KLoC, but with extensive interactions and protocol-level complexity. The code and accompanying artifacts (e.g., test suite, documentation) are of excellent quality, developed with high professional standards.

We emphasized protocol-level considerations, in addition to code review. The items below can be considered context notes, possibly to inform future design, or documentation for the benefit of external users.

1) It may not be clear to users what Verified Random Functions truly verify and, consequently, what are the threats. We find statements concerning guarantees *even if a Chainlink oracle ("node") is compromised* to be hard to interpret correctly. E.g., the following in the past Chainlink VRF documentation:
    - *The proof is published and verified on-chain before it can be used by any consuming applications. This process ensures that the results cannot be tampered with nor manipulated by anyone, including oracle operators,*

miners, users and even smart contract developers. [https://docs.chain.link/docs/chainlink-vrf/]

- *The fundamental benefit of using Chainlink VRF is its verifiable randomness. Even if a node is compromised, it cannot manipulate and/or supply biased answers – the on-chain cryptographic proof would fail. The worst case scenario is that the compromised node does not return a response to a request[...] Even in the unlikely scenario that a node is compromised, its resulting randomness cannot be manipulated.* [https://blog.chain.link/chainlink-vrf-on-chain-verifiable-randomness/]

What the above seem to imply is not accurate. If a Chainlink node's VRF secret key is compromised, the VRF becomes a mere deterministic pseudo-random generator. Its randomness is then, at best, on-chain randomness, which has well-known issues. Certainly the collusion of a miner and an oracle operator can open the door to attacks.

2) The current design does not accept a seed from the user of the VRF functionality. This is ideal for the usual case, but it is limiting for highly sophisticated clients. Specifically, the typical client contract only has access to on-chain information. In this case, accepting a seed from the client is pointless: the current approach of the VRF functionality both gets the most obtainable in terms of on-chain randomness (by taking the blockhash of a block not mined at the time of the request) and would destroy any seed provided by the caller in the process. However, a more advanced API can be envisioned (perhaps in addition to the current one) that allows the caller to supply their own randomness in the form of a seed. (Such a seed should not then be hashed with a blockhash, or a miner can bias the outcome.) This approach permits, for instance, the use of two *separate* off-chain randomness sources. If the VRF oracle accepts randomness from the user (in addition to employing its own secret key) the result is a better randomness guarantee: biasing the random outcome would require compromising both randomness sources (i.e., both the secret key of the Chainlink VRF oracle, and the other source of randomness).

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: 01) User or system funds can be lost when third party systems misbehave. 02) DoS, under specific conditions. 03) Part of the functionality becomes unusable due to programming error.
LOW	Examples: 01) Breaking important system invariants, but without apparent consequences. 02) Buggy functionality for trusted users where a workaround exists. 03) 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	Client can cause `fulfillRandomWords` transactions without getting charged.	**Resolved**

[We collect in this issue a couple of different threads, as analyzed in our discussions with developers. These can be considered separate issues, but with related threats. Remedies are also being discussed in our channel with developers.]

Without precise accounting of the subscription balances and outstanding expenses for request, it is possible for a client to make randomness requests and not be able to pay for the delivery of the results. This will cause the `fulfillRandomWords` call (initiated off-chain) to fail with a transaction revert. However, a) this creates work and causes a gas expense for the caller account, allowing DoS attacks; b) the randomness is publicly broadcast, so it can indeed be harvested and used.

Two such scenarios include:

- The client calls `requestRandomWords` but, before the `fulfillRandomWords` response, the client issues a `defundSubscription`.
- Even if `defundSubscription` is accounted for off-chain, an extreme scenario is for the client to act as an attacker, watch the mempool for `fulfillRandomWords` transactions, and try to front-run them with `defundSubscription` to avoid charges.

Many more attacks along these lines can be drafted, and the developers have already supplemented our list with more.

Better accounting of outstanding randomness requests and subscription balances (both on-chain and off-chain) can mitigate the problem.

MEDIUM SEVERITY:

[No medium severity issues]

LOW SEVERITY:

ID	Description	STATUS
L1	Struct `Consumer` can be simplified.	**Resolved**

There is little reason to keep `subId` both in the key and in the value of the `s_consumers` mapping.

```
  struct Consumer {
      uint64 subId;
      uint64 nonce;
  }
  mapping(address => mapping(uint64 => Consumer)) /* consumer */ /* subId */
      private s_consumers;
```

The information could be kept in a boolean, or encoded in the `nonce` field. (E.g., start nonces from 1, to denote an allocated consumer with 0 requests.)

ID	Description	STATUS
L2	Unreachable code in `getRandomnessFromProof`	**Dismissed**

Under the current definition of the Chainlink blockhash store, the following is dead code (condition never true). The call to get the blochhash would have reverted.

```
blockHash = BLOCKHASH_STORE.getBlockhash(rc.blockNum);
if (blockHash == bytes32(0)) {
  revert BlockhashNotInStore(rc.blockNum);
}
```

Admittedly, it is good to code defensively relative to external calls, so the check is not without merit.

ID	Description	STATUS
L3	Extraneous check in `fulfillRandomWords`	**Resolved**

The check

```
if (gasPreCallback < rc.callbackGasLimit) {
      revert InsufficientGasForConsumer(gasPreCallback, rc.callbackGasLimit);
}
```

is unnecessary, given the stronger check that follows inside the call to `callWithExactGas`, with `gasAmount` being `rc.callbackGasLimit`:

```
assembly {
      let g := gas()
      ...
      if iszero(gt(sub(g, div(g, 64)), gasAmount)) {
          revert(0, 0)
      } ...
```

L4	The meaning of `MIN_GAS_LIMIT` is unclear	**Resolved**

Code comments describe `MIN_GAS_LIMIT` as:

```
 // The minimum gas limit that could be requested for a callback.
 // Set to 5k to ensure plenty of room to make the call itself.
 uint256 public constant MIN_GAS_LIMIT = 5_000;
```

and

```
/**
...
  * The minimum amount of gasAmount is MIN_GAS_LIMIT.
```

(With `gasAmount` being the `callbackGasLimit`.)

However, `MIN_GAS_LIMIT` is never compared against the callback gas limit, only against the currently available gas. Our interpretation was that it intends to account for the gas of other VRFCoordinatorV2 contract operations outside the client callback. If so, the limit of 5000 is too low.

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	Variable `s_fallbackWeiPerUnitLink` left out of `Config`	**Dismissed**

It is unclear why variable `s_fallbackWeiPerUnitLink` is not included in the `Config` structure, since it is essentially handled as one of the variables therein. For example, the return statement of `getConfig()`:

```
return (
  config.minimumRequestConfirmations,
  config.fulfillmentFlatFeeLinkPPM,
  config.maxGasLimit,
  config.stalenessSeconds,
  config.gasAfterPaymentCalculation,
  config.minimumSubscriptionBalance,
  s_fallbackWeiPerUnitLink
);
```

Is there some benefit in keeping the size of Config down to one word, given that it seems to be always read/written together with `s_fallbackWeiPerUnitLink` ?

ID	Description	STATUS
A2	Gas optimizations using unchecked wrapper	**Dismissed**

In VRFCoordinatorV2.sol there are a number of safe mathematical operations that could be made more gas efficient if wrapped in unchecked{}
In fulfillRandomWords:

```
s_subscriptions[rc.subId].balance -= payment;
s_withdrawableTokens[s_provingKeys[keyHash]] += payment;
```

In OracleWithdraw:

```
    s_withdrawableTokens[msg.sender] -= amount;
    s_totalBalance -= amount;
```

In defundSubscription:

```
    s_subscriptions[subId].balance -= amount;
    s_totalBalance -= amount
```

In cancelSubscription:

```
    s_totalBalance -= balance
```

However, this recommendation could slightly downgrade readability and clarity.

A3	Function ordering inside contracts	Dismissed

Consider adopting the official style guide for function ordering within a contract.
In order of priority:
external > public > internal > private and view > pure within the same visibility group.
https://docs.soliditylang.org/en/v0.8.7/style-guide.html#order-of-functions

A4	Floating pragma	INFO

Use of a floating pragma: The floating pragma pragma solidity ^0.8.0; is used, allowing contracts to be compiled with any version of the Solidity compiler that is greater or equal to v0.8.0 and lower than v.0.9.0. Although the differences between these versions should be small, for deployment, floating pragmas should ideally be avoided and the pragma be fixed.

A5	Compiler known issues	INFO

Solidity compiler v0.8.0, at the time of writing, has some known bugs (SignedImmutables, ABIDecodeTwoDimensionalArrayMemory, KeccakCaching). We believe that none of them affects the code: no immutable signed integer variables are declared, no multidimensional arrays seem to be used in the audited contracts, and no keccak hashing of constant memory arrays takes place.

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.