VLS Security Audit

This report summarizes the audit of the Validating Lightning Signer project in January 2022.

The audit included reviewing:

- validating-lightning-signer/vls-core at commit `c38561c`

- validating-lightning-signer/vls-server at commit `c38561c`

- validating-lightning-signer/vlsd at commit `c38561c`

- docs/policy-controls.md at commit `c38561c`

- docs/oracle.md at commit `c38561c`

- docs/heartbeat.md at commit `c38561c`

The 1st part of the audit reviews the completeness of the policy checks, ensuring this is a correct representation of the channel state. The checks should cover all the phases of the channel lifetime: funding, signing, payments, revocation, HTLC forwards, mutual close and back to the wallet sweeps.

The 2nd part of the audit reviews the core crates of the VLS reference implementation (`vls-core`, `vls-server`, `vlsd`), the grpc API and some core mechanisms as the allowlist and invoice flow.

The 3rd part of the audit reviews the security bounds of the VLS oracles: the UTXO set and the fee-estimation, and the impact of oracle compromise.

The 4rd part highlights additional security enhancements on top of VLS: the heartbeat protocol and the distributed watchtower setup.

The information in this report is produced with the best effort and state of knowledge of its author.

General Projects Overview

Security Goals

The main security argument of the Validating Lightning project is to isolate all the Lightning signing operations from a traditional computing host to a secure environment (e.g HSM) with a restricted attack & risk surface. As of today, Lightning nodes are "hot wallet", of which the compromise of the underlying host leads to a complete loss of the funds. Traditional computing host (e.g Amazon S3) offers a wide attack surface: remote code exploitation affecting the kernel network stack, supply chain attack targeting the platform drivers firmware or even social engineering the cloud provider operators [0] [1].

Additionally, a set of policy controls are implemented by the Validating Lightning signer, not only to replicate a BOLT-compliant state machine, as present in all Lightning implementations, but also to enforce additional series of checks. Those checks can sanitize more dimensions of the flows like velocity, destinations or capacity. They're also declined in function of the use-case considered: payment node, merchant, routing hop.

Those two lines of defenses, compartmentalization of the critical operations and additional security checks, should guarantee that even in case of compromise of the node, there is no jeopardy of the funds safety.

Finally, the signing operations could leverage modern multi-signature schemes (e.g MuSig2 [2], FROST [3], TAPS [4]) dividing the physical location of the funds over multiple, diverse HSMs. This would constitute a step further in removing a single-point of failure to compromise.

There is no confidentiality model of the information stored by the Validating Lightning signer (e.g channels) to encompass multi-user deployment in the future.

Spec

The set of *policy controls* are documented in the canonical document:

https://gitlab.com/lightning-signer/validating-lightning-signer/-/blob/main/docs/policy-controls.md

There are 63 rules described, classified in 3 main categories:

- *Mandatory Policy Controls*

- *Optional Policy Controls*

- *Use-case Specific Controls*

The *Mandatory Policy Controls* replicate the checks corresponding to a Lightning state machine (e.g the HTLC in-flight value or the number of HTLC outputs). The *Optional Policy*

Controls enable to control funds flows beyond the usual Lightning checks (e.g the amount transferred to peers must be under a certain amount at all time). The *Use-case Specific Controls* encompass more specific flows such as *Merchant* or *Routing Hub*. The policy checks are also modulated in function of the channel type considered [5].

All the checks listed in the reference document are not yet implemented, with the one missing tracked in `vls-core/src/policy/simple_validator.rs` (L336+).

Codebase

The project includes 36k LoC of Rust, summing up 1300 commits. The initial commit is dated 15 January 2020.

The following list of crates is under development, of which the level of maturity varies amply:
- `bitcoind-client`: A bitcoind RPC client implementation.

- `bolt-derive`: A BOLT message parsing library.

- `embedded`: A set of experimental memory-management utilities for embedded devices.

- `vls-core`: A library implementing Lightning signer, which externalizes and secures cryptographic operations.

- `vls-server`: A library implementing a Lightning signer server.

- `lightning-storage-server`: A library implementing a Lightning Storage Service [6].

- `vlsd`: Validating Lightning server daemon and client.

- `vls-frontend`: A library implementing VLS frontend functions.

- `vls-persist`: A library implementing a VLS key-value store.

- `vls-protocol`: A library implementing VLS internal protocol.

- `vls-protocol-client`: A library implementing a VLS protocol client.

- `vls-protocol-signer`: A library implementing a VLS protocol signer.

- `vls-proxy`: A drop-in replacement or Core's Lightning's hsmd.

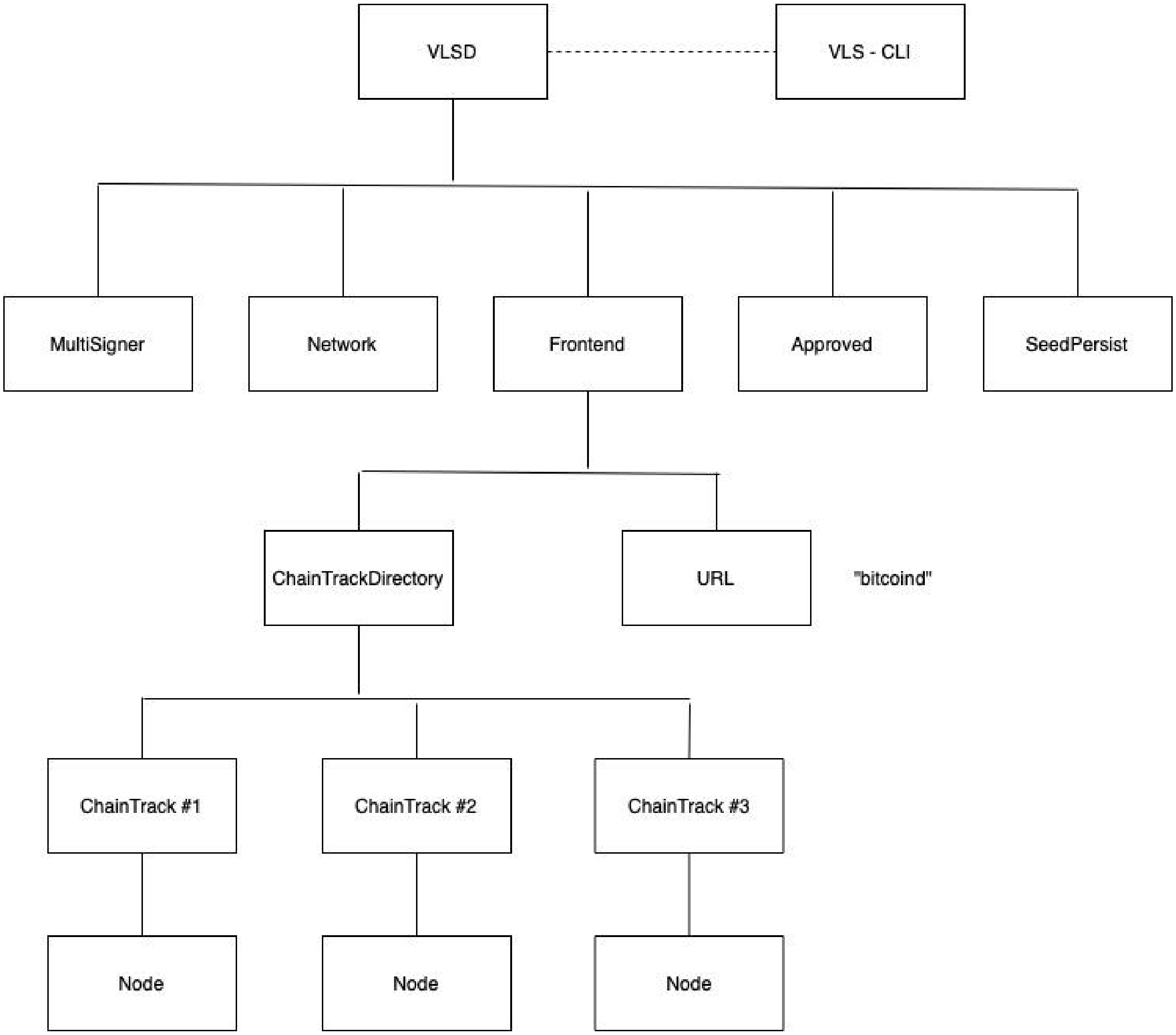
- `vls-signer-stm32`: An experimental VLS signer on 32-bit ARM Cortex processor.

- `wasm`: An experimental signer running in WASM.

For the scope of this report, only the `vls-core`, `vls-server` and the `vlsd` crates have been the object of an in-depth and width audit.

Architecture

Assuming the Validating Lightning signer is deployed in the daemon-client configuration, the architecture is the following:



Part 1: Policy Controls

This section reviews the set of policy controls enforced by the Validating Lightning signer core module.

The set of policy controls should prevent:

- loss or freeze of channel funds/wallet UTXOs.

- significant siphoning of transaction fees.

- leverage of channel transactions as building blocks for more sophisticated attacks.

For the former, e.g channel transactions could be inflated in size with a significant fee, as such being leveraged as mempool backlog for *flood&loot*-style attacks [7].

Channel Opening

The checks are enforced in `SimpleValidator::validate_ready_channel()` (`vls-core/src/policy/simple_validator.rs`):

- the holder selected delay is enforced to be in the 144 - 2016 block range (`L379`).

- the counterparty selected delay is enforced to be in the 144 - 2016 block range (`L386`).

- the committer shutdown script is verified to be spendable by the wallet or in the allowlist (`L392`).

Enforcing the minimal value of 144 on the counterparty commitment and second-stage HTLC transactions is in range with the value selected by standard Lightning implementations and should allow sufficient delay of reaction to confirm justice transactions in case of revoked states broadcast by the counterparty.

Enforcing the maximal value of 2016 on the holder commitment and second-stage HTLC transactions prevents a timevalue Dos where the holder funds could be locked for a long period of time in case of an unilateral closure.

More channel policy parameters (`dust_limit_satoshis`, `max_htlc_value_in_flight_msat`, etc) as described in BOLT2 are enforced during commitment transactions signing operations (`SimpleValidator::validate_commitment_tx()`).

We underscore the lack of ChannelSetup::commitment_type sanitization, leading to downgrade attacks towards legacy, unsafe channel type.

Channel Funding

The checks are enforced in `SimpleValidator::validate_onchain_tx()` (`vls-core/src/policy/simple_validator.rs`):

- the signed transactions are verified to be nversion=2 (`L436`).

- the non-funding channel outputs are verified to be spendable by the wallet (`L462`) or in the allowlist (`L472`).

- the funding output value should match the channel capacity previously registered (`L507`).

- the scriptpubkey should match the funding scriptpubkey previously registered (`L523`).

- the state machine commitment number should be equal to one (L533).

- the channel should be outbound (L540).

- the BOLT2's push_msat value should be null (L548).

- if there is an unknown output on the funding transaction, a policy error is returned (L574).

- if the feerate range of the transaction is also verified to be lower than the maximum allowed by the policy SimplePolicy::max_feerate_per_kw (L271).

We observe there is a CRITICAL LOSS OF FUNDS VULNERABILITY by accepting non-Segwit inputs (P2PKH) in Node::unchecked_sign_onchain_tx(). An initial commitment transaction signature could be made invalid as with pre-Segwit input a signer could produce a new ECDSA signature breaking the txid, and as such the signature digest [11].

We observe the default policy value is equivalent to 100 sat/vb (L1786). While this value sounds reasonable for time-sensitive transactions (e.g HTLC-timeout), this high value is a fee siphoning issue for non-time-sensitive transactions like the funding one. We think two upper bound feerate values should be used, one for the class of time-sensitive transactions, one for the time-insensitive transaction.

We think a detailed threat model could be defined for risks about non-standard transactions. In the present case, while the transaction is checked to be nversion=2, there are other standard checks missing such as the dust threshold or the standardness of the output scriptpubkeys (P2SH, P2PKH, P2WSH, P2WPKH, P2TR).

There are 2 policy checks concerning the channel funding flow described in the reference documentation, of which the implementation is pending:

- policy-onchain-wallet-path-predictable: the change output derivation path for a wallet address must be reasonable.

- policy-velocity-funding: the amount transferred to peers must be under a certain amount per unit time.

We observe the channel funding flow could be protected by an additional policy check:

- policy-velocity-fee: the fee paid must be under a certain amount per unit time.

This check would prevent the following fee-siphoning attack, where a compromised node requests validation and signature of funding transactions with change output paying back to wallet address, in loop. The funding transactions fees are captured with the collusion of a miner. We think the current policy-velocity-funding rule does not protect against this attack vector as the funding transaction channel outputs can be kept at the minimum of 1 sat.

Commitment transaction

The checks sanitizing the holder commitment transaction are enforced in `Channel::validate_holder_commitment_tx()` (`vls-core/src/channel.rs`):
- the channel value upper bound is enforced (`L1796`).

- the holder balance value is verified to be above the dust threshold for `P2WSH` (`L1628`).

- the counterparty balance value is verified to be above the dust threshold for `P2WSH` (`L1638`).

- the number of HTLCs is above the maximum number of HTLCs (`L1650`).

- the HTLC CLTV value is above the min delay of 144 (`L193`) and under the max delay of 2016 (`L203`) both from the chain tip.

- if the HTLC amount value is under the dust limit, the trim is verified for offered (`L1674`) and received (`L1703`).

- the sum of the HTLC amounts is above the max HTLC value requested by the policy (`L1714`).

- the commitment transaction fee is checked to be above the `SimplePolicy::min_feerate_per_kw` (`L229`) and under the `SimplePolicy::max_feerate_per_kw` (`L238`).

- if this is the initial commitment transaction, the number of pending HTLCs must be 0 (`L1741`).

- if this is the initial commitment transaction, the counterparty value must not be above the `push_msat` channel parameter (`L1758`).

- the CSV delay enforced on the commitment transaction is equal to the CSV delay picked up at channel opening (`L768`).

- if the state number is re-signed the commitment transaction must be the same (`L781`).

- the commitment transaction must not have been revoked (`L794`).

- the commitment transaction cannot be signed if the channel is closed (`L810`).

- the recomposed transaction must be equal to the original transaction given (`L1867`).

Beyond there is a serie of policy rules implicitly checked:

- `policy-commitment-version`, in LDK's `chan_utils::make_transaction()`

- `policy-commitment-locktime`, in LDK's `chan_utils::make_transaction()`

- `policy-commitment-sequence`, in LDK's `chan_utils::internal_rebuild_transaction()`

- policy-commitment-input-single, in LDK's chan_utils::internal_rebuild_inputs()

- policy-commitment-revocation-pubkey, in LDK's chan_utils::internal_build_outputs()

- policy-commitment-broadcaster-pubkey, in LDK's chan_utils::internal_build_outputs()

- policy-commitment-htlc-revocation-pubkey, in LDK's chan_utils::internal_build_outputs()

- policy-commitment-htlc-counterparty-htlc-pubkey, in LDK's chan_utils::internal_build_outputs()

- policy-commitment-htlc-holder-htlc-pubkey, in LDK's chan_utils::internal_build_outputs()

We observe the default value of SimplePolicy::max_htlcs is above the maximum number of pending HTLCs on a commitment transaction (L966). This is a security issue as it's a marginal increase in the value that can be captured by a fee siphoning attack.

We think there is a CRITICAL LOSS OF FUNDS VULNERABILITY by processing the option_anchors_outputs channel type. The channel type is not sanitized against safer channel type at channel opening. Therefore, a compromised node can register an anchor_output channel and then extract the fee value by adding new siphoning outputs on the SIGHASH_SINGLE second-stage HTLC transactions [10].

We think there is a CRITICAL LOSS OF FUNDS VULNERABILITY as in case of HTLC trimmed under the dust, there is no enforcement of the global value accumulated as fee on the transaction. We think this opens the door to a lightweight miner fee siphoning attack where 159390 satoshis could be open.

The checks sanitizing the counterparty commitment transaction are enforced in Channel::sign_counterparty_commitment_tx() (vls-core/src/channel.rs).

It should be noted numerous checks are common due to the shared utility validate_commitment_tx(), we note the different ones:

- the CSV delay enforced on the commitment transaction is equal to the CSV delay picked up at channel opening (L662).

- the commitment number must be equal to the current or next state (L677).

- if this is a retry at current number, the commitment point (L689) and state must be the same as previous (L717).

There are 12 policy checks concerning the commitment signing flow described in the reference documentation, of which the implementation is pending:

- policy-commitment-spends-active-utxo

- policy-commitment-htlc-routing-balance

- `policy-commitment-htlc-received-spends-active-utxo`

- `policy-commitment-htlc-cltv-range`

- `policy-commitment-htlc-offered-hash-matches`

- `policy-commitment-anchors-not-when-off`

- `policy-commitment-anchor-to-holder`

- `policy-commitment-anchor-to-counterparty`

- `policy-commitment-anchor-amount`

- `policy-commitment-anchor-static-remotekey`

- `policy-commitment-anchor-match-fundingkey`

We think the lack of implementation of those checks lead to CRITICAL LOSS FUNDS such as receiving a forward HTLC on a non-confirmed channel.

We observe the HTLC forward routing flow for routing hubs should be protected by an additional check:

- `policy-cltv-delta-reasonable`: the CLTV expiry on the received HTLC and the CLTV expiry on the offered HTLC should enable a sufficient claim delay.

We think the lack of this `policy-cltv-delta-reasonable` lead to a CRITICAL LOSS OF FUNDS VULNERABILITY against routing hubs, where the commitment transaction offered HTLC `cltv_expiry` on the downstream channel is inferior to the commitment transaction received HTLC `cltv_expiry` on the upstream channel, making it impossible to avoid a HTLC double-spend by colluding channel counterparties.

Payments

The sanitization of the inbound/outbound payments/forward HTLC is enforced in `NodeState::validate_payments()` (`vls-core/src/node.rs`). It relies on an internal map `NodeState::Payments()`. This map tracks the incoming and outgoing HTLCs on their respective channels for all classes of payments (routed or non-routed). If the payment is non-routed, there must be an invoice to authorize the spend. If the payment is routed, there must be an incoming HTLC to balance outgoing payments.

The payment balance is checked with `validate_payment_balance()` and if it's unbalanced a policy error is returned (`L279`). This helper enforces a bound on the maximum of value spent as routing fees along a Lightning payment path `SimplePolicy::max_routing_fee_msat`, with a value of `10_000` msat.

If the balance enforcement policy is opted-in, the balance should always increase following the rule `policy-routing-deltas-only-htlc` (`L287`).

We think the whole payment flows as pre-authorized by invoices should be detailed in a consistent security model.

There are 5 policy checks concerning the payment flow described in the reference documentation, of which the implementation is pending:

- `policy-commitment-payment-settled-preimage`
- `policy-commitment-payment-allowlisted`
- `policy-commitment-payment-velocity`
- `policy-commitment-payment-approved`
- `policy-commitment-payment-invoiced`

We understand the implementation would allow advanced controls of the outgoing payments, the implementation of the checks are critical for funds security.

TODO.

Revoking Previous Commitment Transaction

The checks sanitizing the counterparty commitment transaction revocation are enforced in `SimpleValidator::validate_counterparty_revocation()` (`vls-core/src/policy/simple_validator.rs`):

- the revoked commitment number is equal to the current or the next one (`L826`).
- the supplied revoked commitment point is not equal to the previously announced commitment point (`L854`).

HTLC Transactions

The checks sanitizing second-stage HTLC transactions are enforced in `SimpleValidator::validate_htlc_tx()` (`vls-core/src/policy/simple_validator.rs`):

- the CLTV expiry on the offered HTLC must be non-zero (`L1021`).
- the feerate must be above the `SimplePolicy::min_feerate_per_kw` (`L1026`) and under the `SimplePolicy::max_feerate_per_kw` (`L1036`).

Beyond there is a serie of policy rules implicitly checked:

- `policy-htlc-version` in LDK’s `build_htlc_transaction()`
- `policy-htlc-locktime` in LDK’s `build_htlc_transaction()`
- `policy-htlc-sequence` in LDK’s `build_htlc_input()`
- `policy-htlc-to-self-delay` in LDK’s `build_htlc_output()`

- `policy-htlc-revocation-pubkey` in LDK's `build_htlc_output()`

- `policy-htlc-delayed-pubkey` in LDK's `build_htlc_output()`

We think there is a FUND FREEZING VULNERABILITY in the fact the `cltv_expiry` at HTLC transaction signature isn't checked to be in range. Following the semantic of the Script interpreter, the HTLC transaction is still valid w.r.t CLTV opcode but shouldn't confirm in the chain until the chain tip height is equal to the `cltv_expiry` value committed by the nLocktime.

If the node is a simple payer, the fund can be locked for a long-time, leading to HTLC amount being frozen. If the node is a routing hub, the fund can be locked so far, there is no time to timeout the outbound HTLC before the inbound HTLC is timeout, leading to a double-spend risk (i.e preimage forward, timeout backward).

Mutual Closing Transaction

The checks sanitizing a mutual close transaction enforced in `SimpleValidator::validate_mutual_close_tx()` (`vls-core/src/policy/simple_validator.rs`):

- the number of outputs is verified to be non-superior to 2 (`L1086`).

- the holder commitment information must be present (`L1098`).

- the counterparty commitment information must be present (`L1101`).

- the holder script must be present if there is a holder value (`L1268`).

- the counterparty script must be present if there is a counterparty value (`L1277`).

- the holder script must match the holder shutdown script (`L1289`).

- the transaction must not have pending HTLCs (`L1298`).

- the feerate is checked to be in range (`L1318`).

- the funder or the counterparty output value is checked to be equal to the commitment info (`L1326` / L1340 or L1357 / `L1369`).

- the holder output script must be in the wallet or allowlist (`L1384`).

We observe the nSequence and nLocktime fields should be sanitized to avoid a timevalue DoS, where the mutual closing transaction cannot be finalized before the holder balances output on the latest commitment transaction, therefore nullifying the interest of a mutual close.

Sweep transaction

The sweep transaction scopes 3 phases:

- the claim of a holder balance on a commitment transaction/second-stage HTLC transaction.

- the punishment of revoked outputs and the claim of counterparty HTLC outputs.

There is a common helper ensuring a serie of policy rules for those 3 phases,, `SimpleValidator::validate_sweep()`:

- the signed transaction is verified to be nversion=2 (`L301`).

- the destination addresses are verified to be spendable by the wallet or in allowlist (`L314`).

We observe there is a code comment documenting the lack of a critical fee-sanitization. This check would implicitly verify the correspondence between the spent input collected in the transaction and the output address amount.

Additional policy rules could be implemented sanitizing the transaction standardness such as the dust threshold on outputs and the type of the scriptpubkey.

The checks sanitizing the claim of a holder balance are enforced in `SimpleValidator::validate_delayed_sweep()` (`vls-core/src/policy/simple_validator.rs`):

- the transaction nLocktime must not be superior to the chain tip height (`L1419`).

- the transaction nSequence must be equal to the CSV delay selected by the counterparty (`L1430`).

The checks sanitizing the punishment of revoked outputs are enforced in `SimpleValidator::validate_justice_sweep()` (`vls-core/src/policy/simple_validator.rs`):

- the transaction nLocktime must not be superior to the chain tip height (`L1555`).

- the transaction nSequence must be compatible with the channel type requirements (`L1566`).

We observe the second check should be updated in consequence to the *anchor output* channel type, where a justice transaction input spending a revoked HTLC output should be equal to one.

The checks sanitizing the claim of counterparty HTLC outputs are enforced in `SimpleValidator::validate_counterparty_htlc_sweep()` (`vls-core/src/policy/simple_validator.rs`):

- on received HTLCs, the HTLC `cltv_expiry` must be in the bound of an unsigned integer (`L1472`).

- on received HTLCs, the transaction locktime must be superior to the HTLC `cltv_expiry` (`L1481`).

- on offered HTLCs, the transaction locktime must be superior to the chain tip height (`L1499`).

- the `witnessScript` must parse as received or offered (`L1510`).

- the transaction nSequence must be compatible with the channel type requirements (`L1525`).

Optional Policy Controls

The Validating Lightning Signer includes a series of checks in addition to the classic one of a Lightning implementation state machine.

Those checks enable fine-grained control of the funds velocity or in function of the Lightning node type.

There are 3 optional policy checks described in the reference documentation, of which the implementation is pending:

- `policy-velocity-funding`

- `policy-velocity-transferred`

- `policy-merchant-no-sends`

We understand the implementation would allow advanced control of the funds flows, however the implementation does not appear critical for funds security.

Part 2: Validating Lightning Server Daemon

This section reviews the Validating Lightning Signer server daemon and few of its core components. Indeed, one of the deployment configurations of VLS is a server daemon (`vlsd`) responding to a CLI client (`vls-cli`) through a gRPC protocol (remotesigner.proto). The server daemon answers asynchronously to all the calls driving the transaction signing.

As described in **architecture** (fig 1), the daemon contains the nodes and their corresponding channels and a persistence module.

The design of the system implies that a security failure in the gRPC server cannot result in a compromise of the system, unless the failure includes the ability to execute arbitrary code or write arbitrary memory.

The `vls-core` crate should be the only safety-critical crate common to the whole VLS architecture.

The `Persist` implementation and the `ChainTracker` implementation are beyond the scope of this audit.

The following sections covers in details:

- the gRPC API interface.

- some safety-critical core components: invoices registering, addresses allowlist and weight computation.

The gRPC API interface

The gRPC API interface contains a list of calls realizing server initialization, channels transactions and signing, invoice and keysend approvals, data objects listing. In the following subsection, an overview of gRPC calls soundness against DoS and other logical bugs is pursued. The `vls-cli` is assumed to be compromised as an adversary and free to trigger any call.

ping()

This call implements a classic networking ping, where the data payload is sent back as a reply.

We observe the message is logged without prior checks on the payload size. If the internal printing buffer is full, it could lead to a crash of the daemon.

init()

This calls implements passthrough of the node configuration parameters, the chain parameters and the BIP32 seed.

We observe there is no current implementation of generating a local HSM secret if the network is production-ready, even if the documentation indicates it should be the practice.

We observe there is no prevention of reetrance, where the Signer could be re-initialized during the operation to destroy some sensitive data (even if there is persistence).

get_node_param()

This call implements yield of implementation-specific key material such as CLN xpub, a BOLT-12 pubkey and node secret.

new_channel()

This call implements the registration of a new channel, starting as a stub until it is marked as ready.

We observe there is protection against re-entrance where if the channel has always been marked as ready, an error is returned.

We observe there is no memory-DoS prevention, where the internal Signer memory map `Node::channels` is overflowed with new channels.

get_channel_basepoints()

This call implements yield of channel basepoints as they're stored in the `MultiSigner`.

We observe the probe is realized on the channel nonce as announced at declaration. A compromised node might use this call to probe the existence of a channel stored by the Signer by iterating randomly on the 8 or 32-bytes space.

`ready_channel()`

This call marks a channel as ready for usage once the channel parameters information from the counterparty have been received.

We observe there is no check on the byte-length of neither the holder shutdown script nor the counterparty shutdown script.

We observe there is protection against re-entrance, where the channel setup is tried to be registered multiple times.

sign_mutual_close_tx()

This call signs a mutual closing transaction spending a funding channel output.

sign_mutual_close_tx_phase2()

This call signs a mutual closing transaction spending a funding channel output. The transaction is rebuilt from the supplied arguments.

check_future_secret()

This calls checks if the counterparty knows a secret that we haven't generated since being restored from backup.

We observe that if the secret is relied on it could be leveraged for some downgrade attack, where the Signer is misled to release a secret or sign a transaction.

get_per_commitment_point()

This call gets the per-commitment point for a specific commitment number.

sign_onchain_tx()

This call signs an on-chain funding or sweep transaction, including validating the transaction.

We observe the transaction raw byte size is not checked before deserialization, this could lead to a memory-DoS.

We observe the number of transaction inputs is not checked, if the witnesses are computationally expensive to generate, it could lead to a CPU-DoS.

sign_counterparty_commitment_tx()

This call signs a counterparty commitment transaction, including validation.

We observe the transaction size is not checked before deserialization, this could lead to a memory-DoS.

validate_holder_commitment_tx()

This call validates the holder commitment transaction.

We observe the transaction raw byte size is not checked before deserialization, this could lead to a memory-DoS.

validate_counterparty_revocation()

This call validates the counterparty revocation secret.

sign_holder_htlc_tx()

This call validates second-stage HTLC transactions.

We observe the transaction raw byte size is not checked before deserialization, this could lead to a memory-DoS.

sign_delayed_sweep()

This call validates a delayed sweep transaction.

We observe the transaction raw byte size is not checked before deserialization, this could lead to a memory-DoS.

We observe the number of transaction inputs is not checked, if the witnesses are computationally expensive to generate, it could lead to a CPU-DoS.

sign_counterparty_htlc_tx()

This call validates the counterparty second-stage HTLC transaction.

We observe the transaction raw byte size is not checked before deserialization, this could lead to a memory-DoS.

sign_counterparty_htlc_sweep()

This call validates the counterparty HTLC sweep transaction.

We observe the transaction raw byte size is not checked before deserialization, this could lead to a memory-DoS.

sign_justice_sweep()

This call signs a justice sweep transaction.

We observe the backend code of some Lightning implementations allows aggregation of revoked output claims, therefore the sanitization check of a single transaction output could lead to issues.

sign_channel_announcement()

This call signs a BOLT 7 channel announcement.

We observe the channel announcement message fields are not sanitized. E.g the features field size could be inflated to provoke a CPU-DoS with the signing operation.

sign_node_announcement()

This call signs a BOLT 7 node announcement.

We observe the node announcement message fields are not sanitized. E.g the number of node addresses could be inflated to provoke a CPU-DoS with the signing operation.

sign_channel_update()

The call signs a BOLT 7 channel update.

We observe the channel update fields are not sanitized. E,g the number of channel flags could be inflated to provoke a CPU-DoS with the signing operation.

ecdh()

This call performs an ECDH operation between the node key and a public key.

The number of ecdh operation could be rate-limited to avoid any kind of cryptographic oracle leaking the node secret.

sign_invoice()

This call signs a BOLT 11 invoice.

We observe the BOLT 11 invoice fields are not sanitized. E.g the invoice `r`-field could be inflated to provoke a CPU-DoS with the signing operation.

sign_bolt12()

This call signs a BOLT 12 invoice.

We observe the BOLT 12 invoice fields are not sanitized. E.g the invoice `r`-field could be inflated to provoke a CPU-DoS with the signing operation.

sign_message()

This call signs a Lightning native message.

We observe the Lightning native message size is not sanitized.

derive_secret()

This call derives a pseudorandom secret from the provided secret key.

sign_counterparty_commitment_tx_phase2()

This call signs a counterparty commitment transaction, including validation.

We observe the number of offered/received HTLCs is not verified before deserialization, this could lead to a memory-DoS.

validate_holder_commitment_tx_phase2()

This call validates a holder commitment transaction, rebuilding the transaction from scratch.

We observe the number of offered/received HTLCs is not verified before deserialization, this could lead to a memory-DoS.

sign_holder_commitment_tx_phase2()

This call signs a holder commitment transaction, including signing commitment transaction..

preapprove_invoice()

This call approves a BOLT 11 invoice for future outgoing payments.

We observe the BOLT 11 invoice fields are not sanitized. E.g the invoice `r`-field could be inflated to provoke a CPU-DoS with the signing operation.

preapprove_keysend()

This call approves a keysend payment, without the necessity of presenting an invoice.

list_nodes()

This call lists the nodes currently registered toward the signer.

We observe this call could trigger a DoS conditional on signer state, if there is a sizable state of nodes pre-registered, it could lead to a memory-DoS.

list_channels()

This call lists the channels currently registered toward the signer.

We observe this call could trigger a DoS conditional on signer state, if there is a sizable state of channels pre-registered, it could lead to a memory-DoS.

list_allowlist()

This call lists the allowlist currently registered toward the signer.

We observe this call could trigger a DoS conditional in signer state, if there is a sizable state of channels pre-registered, it could lead to a memory-DoS.

add_allowlist()

This call registers a new allowlist.

We observe there is no sanitization of the allowlist size, this could lead to a memory-DoS.

remove_allowlist()

This call removes an allowlist.

Core components

This section reviews some core components of the Validating Lightning Signer daemon:

- the allowlist mechanism
- the invoice registration
- the transaction weight computation

Allowlist mechanism

On-chain or off-chain payments should be only sent to an authorized destination to preserve funds security. The destination should be authorized, registered and persisted before the payment occurs.

There is currently a list of on-chain addresses and Lightning node pubkeys maintained by the node along its lifetime, `allowlist`, a set of `Allowable` (in `vls-core/src/node.rs`).

The allowlist can be modified with the following list of functions:

- `new_extended()`, `new_from_persistence()`, `new()`

- `add_allowlist()`, `set_allowlist()`

- `remove_allowlist()`

We observe there is no implemented authorization mechanism for the addition of new allowables. Presently, the modifications are made without verifying if there is a credential or a signature attached to the allowable, or requesting a manual approval. We note there is an approval process for the invoice, however an independent authorization mechanism at the allowlist-level could offer better fine-grained control of funds flows (e.g allow X% per day for this node id).

We think further sanity checks could be implemented on the allowable, such as ensuring the script is spendable (e.g no op_return) or the payee node id is connected to the signer node channel topology (with reasonable number of hops to avoid some routing fee inflation).

Invoice registration

The invoice registration flow starts with `preapprove_invoice()` (`vlsd/src/server/driver.rs`) and follows with `handle_proposed_invoice()` (`vls-protocol-signer/src/approver.rs`).

We observe there is a check against hash duplication (`L45`), if the invoice is already existent in the node state.

This should avoid unsafe interference with a pending inbound/outbound payment, such as changing the `is_fulfilled` flag in-flight.

If the invoice destination in the allowlist, if there is a match the invoice is further checked. Otherwise it is rejected. We observe this logic is currently under implementation.

The invoice is checked for approval with `approve_invoice()`. Assuming the `Approve` implementation is the `VelocityApprover`, the invoice amount msat should be in the temporal bound requested by the velocity control policy. In case of failure, there is a fallback to manual approval, enabling override of the velocity control policy.

We observe there could be duplicated checks as the invoice is also sanitized by `Node::add_invoice()` (`vls-core/src/node.rs`).

The invoice hash is verified for hash duplication (`L1782`) and the invoice amount msat should be in the temporal bound requested by the velocity control policy (`L1800`). This `NodeState::VelocityControl` is a global velocity control that is applied on all funds flows.

A payment state is generated from the invoice (`Node::payment_state_from_invoice()`) and registered in the node state (`L1809`).

We observe there is no sanitization of the invoice fields such as the `expiry` or `min_final_cltv_expiry_delta`.

The BOLT 11 specification recommends some defaults. Abnormal value in those fields could be leveraged as building blocks for more sophisticated attacks.

The keysend flow is outside the audit scope.

The invoiced payments flows are described in Part 1 - Payments.

Transaction weight computation

A significant security risk for a Validating Lightning Signer is a miner fee siphoning attack. A compromised node requests a transaction signature with an inflated feerate. Once the signature has been gained, the compromised node relays the transaction out-of-band to a colluding miner, and extracts the high-fee. As the compromised node constitutes the signer transaction-relay capabilities, there is no access to an honest miner. One mitigation against this issue is the presence of strict fee-estimation oracles, as described in Part 3.

However, even assuming strict fee-estimation oracles, the transaction weight could be artificially inflated by the compromised node to bump the feerate in its favor. There are two dimensions of weight inflation, adding inputs or outputs on malleable transactions (e.g HTLC outputs in a commitment transaction, dummy inputs in a justice transaction) and logic errors miscomputing the real weight of the transaction.

We think sanity checks of the transaction weight could be implemented before each call to `SimpleValidator::validate_fee`, and a policy rule or anomaly detection added in case of abnormal weight or transaction characteristics (e.g the size of the spent witness inputs).

Part 3: Oracles security bounds

This section reviews the oracles security model as documented in `docs/oracle.md`. There is an experimental implementation of a UTXO oracle (`utxoo`), however it is not integrated with the current `vls-core` module.

The Validating Lightning Signer is designed to run on secure environments with no processing capabilities (memory, CPU, I/O ressources), to receive blocks and usual transaction-relay traffic, validate the chainstate (headers, transactions, signatures and accumulated proof-of-work) and constitute a UTXO set.

Those informations are required to validate few signing operations such as the maturity of a time-sensitive commitment transaction, the existence of channels for routing or the efficiency of transactions feerate (in Lightning, some transactions must be confirmed before a height deadline to secure funds, and as such the feerate be compelling in face of current blockspace demand, however a too-high feerate could be a waste, or even an incentive to trigger some miner-based attacks).

Chainstate oracle

The review of the correctness of the chain events tracking is beyond the scope of this audit.

The current UTXO oracle design approach advocated by VLS is the usage of SPV + proof-of-non-spends. The transaction confirmation is proved through the known Merkle proof mechanism as described in the Bitcoin position paper. The proof-of-non-spends are BIP157 (non-standard) filters announcing the TXOs spent in a block.

We assume the mode of operations is the following:

- the Signer receives and validates the block headers from the Oracle set.

- the Signer receives the SPV proof for transaction confirmation.

- the Signer receives and validates the compact headers filters.

- the Oracle set broadcasts per-block signed attestations of the spent TXOs.

- the Signer front-end receives the attestation and generates TXOs proofs for the UTXO of Signer interest.

- the Signer receives the TXO proofs to detect a channel on-chain event.

The following events are tracked:

- *channel activ*e: funding TXO is on-chain and unspent.

- *funding impossible*: a funding input is double-spent.

- *closed by us*: our commitment is on-chain.

- *closed by counterparty*: counterparty commitment is on-chain.

- *HTLC success/failure is on-chain*: a 2nd level HTLC transaction is on-chain.

- *2nd level HTLC swept*: sweep is on-chain.

- *breached by counterparty*: counterparty revoked commitment is on-chain.

- *breach remedied*: remedy transaction is on-chain.

The following security goals are identified:

- `G1`: channel balance or UTXO wallet disequilibrium, i.e the signer releases a secret or a signature enabling the adversary to siphon the holder's balance or spend a UTXO without compensation.

- `G2`: channel state management DoS vector, i.e lack of proper internal memory management provokes a DoS halting the signer operations.

- `G3`: heartbeat anomalies missing, i.e the heartbeat module does not signal back to the consumer application of the node operator (described in Part 4 - Heartbeat protocol).

In the following context, a prover designates an entity submitting API calls requests to the signer (e.g a compromised `vls-cli`).

The *channel active* event is scoped under `G1`, a HTLC forward without an existent inbound channel to receive the incoming HTLC would provoke a loss of funds, as the routing hop doesn't receive a compensation. There is another case, if a payment is received and the preimage is released off-chain (`update_fulfill_htlc`), if the preimage has an out-of-band value (e.g to redeem a gift card), there is a loss as a payee.

The *funding impossibl*e event is scoped under `G2`, a prover could submit new funding transactions to announce channels without broadcasting them, until the internal `Node::channels` is under a memory overflow.

The *closed by us* event is scoped under `G1` and `G3`, releasing a revocation secret of our corresponding commitment transaction on-chain, this revocation could be consumed by our counterparty to punish the commitment transaction, confiscating the funds. If the commitment transaction is time-sensitive, a heartbeat event should be periodically sent to the node operator, in case of missed confirmation, funds could be jeopardized.

The *closed by counterparty* event is scoped under `G2` and `G3`, if there is no HTLC output and no holder balance output, the channel can be safely forgotten once there is enough confirmation, otherwise a prover could submit channels until the internal `Node::channels` is under a memory overflow. If the commitment counterparty has offered HTLC outputs, a heartbeat event should be periodically sent to the node operator, as HTLC-success transactions should be broadcast.

The *HTLC success/failure is on-chain* event is scoped under `G1`, if a HTLC-success is confirmed on a holder or counterparty commitment transaction on the outbound link and a HTLC-timeout is confirmed on a holder or counterparty commitment transaction on the inbound link, a routing hop would be at loss.

The *2nd level HTLC swept* event is scoped under `G2`, the channel can be safely forgotten once all HTLCs have been swept, otherwise a prover could submit channels until the internal `Node::channels` is under a memory overflow.

The *breached by counterparty* event is scoped under G3, the confirmation of a justice transaction is time-sensitive, a heartbeat event should be periodically sent to the node operator. In case of missed confirmation, funds could be jeopardized.

The *breach remedied* event is scoped under `G2`, the channel can be safely forgotten once all revoked outputs have been punished, otherwise a prover could submit channels until the internal `Node::channels` is under a memory overflow.

We can account for few missing events:

- a *burying event* in function of the channel opening parameters, a commitment transaction should not be signed. until a number of block confirmations has elapsed, otherwise the channel could be unsafe against swallow reogs (1 or 2 blocks).

- a *block tick* event, if there is a pending time-sensitive confirmation, a rebroadcast could be scheduled every block as the timelock deadline is approaching.

- a *cooperative closure* event, for accurate memory management of the internal maps, once a cooperative closure appears on-chain, the channel can be forgotten.

- a *local revoked on-chain commitment*, this assumes there has been a leakage of the signer key material, an event could still be triggered in an architecture with distributed monitor replicas.

Semi-trusted third-party UTXO oracles are public servers run by established entities, of which the resource is beyond the administrative scope of the Signer operator. In-house trusted UTXO oracles are private servers run by the Signer operator in its own administrative scope, though running in a different computing space than the *secure environment*.

We outline a few security enhancements that could be adopted by the Signer.

Authentication of the original UTXO set. The UTXO set authentication can be static, i.e hardcoded in the codebase of the Validating Lightning Signer, similar to how it's done for the assumeutxo feature in Bitcoin Core [8]. Or the UTXO set authentication can be dynamic, i.e it should have been countersigned by a public key on a subclass of the allowlist.

Anti-DoS headers syncing strategy. The Signer SPV module should validate headers on the longest valid headers-chain. An approach could be to adopt hardcoded checkpoints ensuring the chain is on the historical valid one. A more sophisticated approach could be to reproduce the novel anti-memory-Dos headers syncing strategy adopted by Bitcoin Core, where the headers are downloaded first, and a commitment to them is stored, then once it has been verified enough work has been accumulated, the headers are download a second time [9].

Block production anomalies detection. A low-rate of block production could be mimicked by the headers prover to prevent the Signer to observe the confirmation of a transaction with a time-sensitive reaction (e.g *breached by counterparty*) and issue the corresponding heartbeat events. A high-rate of block production could be also mimicked by the headers prover to provoke wrongdoing in the heartbeat events issuance (e.g escalation to force-close "sane" channels if a reaction policy enforced by watchtowers is connected to the anomalie protocol). Therefore, we think additional and continuous checks on the normal rate of block issuance should be enforced, beyond the one devised by the *blockgap* research: https://gitlab.com/lightning-signer/blockgap/.

Authentication of the UTXO oracles. The connection to the UTXO oracles themselves should be authenticated, whatever the trust relationship, as the communication channels beyond the "secure environment" could be compromised. A good authentication framework could be to reuse the Noise framework as implemented by BOLT8, with the xk pattern. The initiator must know the public key of the responder, the initiator could be the Signer seeded with the oracle public keys..

Fee-estimation oracle

The oracle design documents, without further detail, that a fee-estimation oracle could be helpful to the signing operations. An "adequate" feerate for Lightning transaction is requested, where adequate can be defined as scoring in the top MB of network mempools.

A feerate under the "adequate" could mean the transaction is stuck during the mempool backlogs. An inflated feerate is even more concerning as it could lead to 2 security breaches.

A *flood-and-loot* escalation scenario [7]. Commitment transactions are inflated to max HTLC output size at max feerate allowed by the policy. E.g if the current feerate is 5 sat/vb, assuming a 4MB blocksize, assuming a max commitment transaction size of 34kb and assuming a max policy feerate of 10sat/vb, if the node has 2000 channels open, HTLC timelocks of a length inferior to 20 blocks could be swallowed by malicious mempool congestion provoked by the *own* node transactions.

A *miner fee siphoning* attack. Commitment transactions are inflated to the max policy feerate allowed by the Signer then captured directly to a colluding miner to siphon the value. Indeed, if an adversary compromises the main Lightning node, there is no more transaction-relay broadcast to the honest majority of miners, and an adversary colluding with any set of low-hashrate miners is plausible.

Beyond that, there is one more risk with a hardcoded bound like the current one of 100sat/vb as in case of current network mempool congestion overreaching the hardcoded bound, a time-sensitive transaction might not confirm within the expected delay.

Part 4: Future security enhancements

This section reviews future security enhancements that could be included in the Validating Lightning Signer architecture in the future.

The first security enhancement devised is the heartbeat protocol, a periodic signal generated by the Signer to indicate either normal operation of the system, or inform about sensitive events for which reaction should be taken.

The second security enhancement is the replication and coupling of Signer with a monitor replica, an instance of a highly-available distributed on-chain channel monitor. A full compromise or failure of a Signer site (e.g fire) enables recovery of the channel funds as long as there is 1 honest and functional monitor replica.

Heartbeat protocol

This subsection reviews the work-in-progress security model detailed in `docs/heartbeat.md`. A heartbeat protocol goal is to alert the node operator about a deadline. Failure to react before the deadline leads to a fund loss. E.g confirming a justice transaction on a revoked output before expiration of its CSV delay.

There are 3 designs for a heartbeat protocol:

- direct signal by node's communication channels

- direct signal by out-of-band communication channels

- dead's man switch

The first design suffers from censoring by the compromised node, therefore the direct signal could never be delivered to the node operator.

The second design is requesting of the secure environment its own wireless network stack, a hardware attack surface increase. This design enables the transmission of anomalies to the node operator (e.g high-rate of channel registration, max velocity systematically reached, etc). At reception, the node operator could connect to the node to detect any intrusion or go to the Signer physical location. Those anomalies could be also connected to a monitor replica force-closing the channels in case of suspicion of compromise of the main Lightning node.

The third design is functioning with a client-side module, where the lack of a regular signal reception should alert the node operator. For this design to work, the time-sensitive outputs should be registered on the client-side module once they're committed in the channels, that way when there is a signal breakup there is already awareness of the reactions to be taken. This "dead's man switch" does not enable the addition of anomalies conveyed to the node operator.

Distributed Watchtower / Monitor replica

A monitor replica is an instance of a highly-available distributed on-chain channel monitor. An on-chain channel monitor is a component of a Lightning node responsible for detecting the confirmation of channel transactions, or the spends of specific channel outputs, broadcast reaction transactions (e.g a justice one) and fee-bump/rebroadcast accordingly.

Each HSM of this monitor replica could host a Validating Lightning Signer with a duplicata of the keys. The channel funds secured by this distributed on-chain channel monitor should be robust against the following events:

- compromise or failure of the main Lightning node

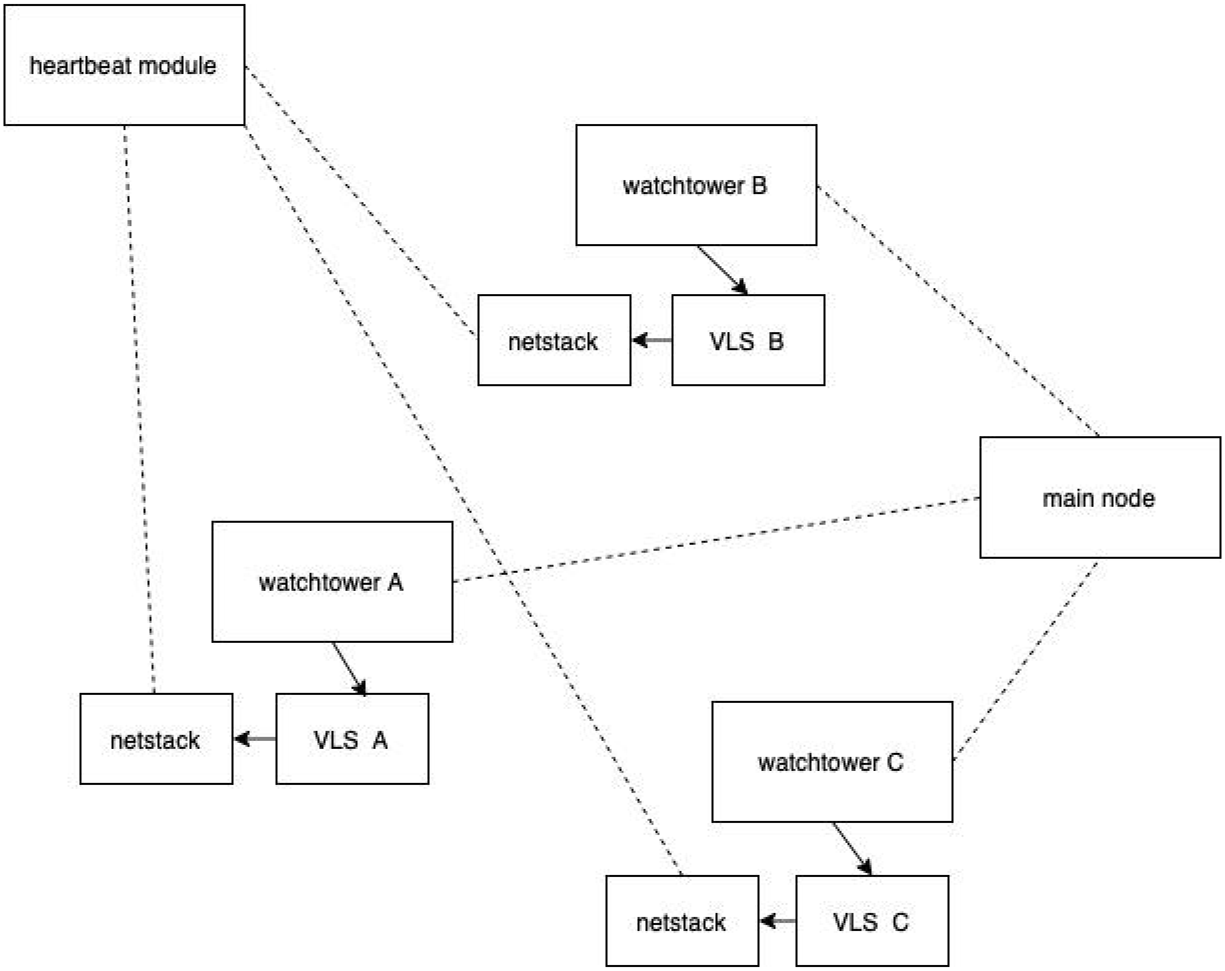
- compromise or failure of all the monitor replica instances minus one

Each distributed on-chain channel monitor should come with its own validation full-node, hardening the Lightning infrastructure against eclipse and time-dilation attacks.

This highly-available distributed on-chain channel monitor comes with a downside of a two-phases commit protocol played between the off-chain coordinator and the monitor replica to avoid accidental issues such as a revocation secret revealed at the same time as a time-sensitive commitment transaction. There is also an additional security risk, as with each key set duplicated in a physical location, there is one more opportunity for a target to compromise.

This concern moves away with the deployment of Eltoo as a channel update mechanism, as the update transaction key can be duplicated to the monitor replica with a single location owning the key to the settlement transaction.

Fig 2: “VLS distributed watchtower deployment”



Sources

[0]: "Securing Lightning Nodes”

[1]: "Blind Signing Considered Harmful”

[2]: "MuSig2: Simple Two-Round Schnorr Multi-Signatures”

[3]: "FROST: Flexible Round-Optimized Schnorr Threshold Signatures”

[4]: "Threshold Signatures with Private Accountability”

[5]: "Lightning Channel Types - BOLT2”

[6]: "Versioned Storage Service - Design Draft”

[7]: "Flood & Loot: A Systemic Attack On The Lightning Network”

[8]: “assumeutxo-docs”

[9]: "p2p: Implement anti-DoS headers sync”

[10]: "SIGHASH_SINGLE + update_fee Considered Harmful”

[11]: "Trust-free unconfirmed transaction dependency chain”

DISCLAIMER: THIS AUDIT REPORT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THIS AUDIT REPORT OR THE USE OR OTHER DEALINGS IN THIS AUDIT REPORT.