Breaking Bridges

An incomplete sampler of bridge hacks

How bridges ~~work~~ break?

Root Causes

- Improper validation
  - Missing validation
  - Delegated validation with wrong assumptions
- Multiple proofs or proof malleability
- Wrong assumptions about underlying networks
- Compromised keys or validators

Side Effects

- Drain locked funds (users) and liquidity (investors)
  - Most common: Nomad, Wormhole
  - transferFrom() approved wallets - AnySwap
- Mint without locking funds
  - Optimism, Polygon, Near
- Arbitrary messages or calls
  - LiFi, Nomad
  - Compromising higher level bridges and projects through composability
  - Governance hijack

EXAMPLES

Li.Fi

Li.Fi

- Bridge aggregator
- Best route chosen off-chain
- User controls route

```
function swap(bytes32 transactionId, SwapData calldata _swapData) internal {
    uint256 fromAmount = _swapData.fromAmount;
    uint256 toAmount = LibAsset.getOwnBalance(_swapData.receivingAssetId);
    address fromAssetId = _swapData.sendingAssetId;
    if (!LibAsset.isNativeAsset(fromAssetId) && LibAsset.getOwnBalance(fromAssetId) < fromAmount) {
        LibAsset.transferFromERC20(_swapData.sendingAssetId, msg.sender, address(this), fromAmount);
    }

    if (!LibAsset.isNativeAsset(fromAssetId)) {
        LibAsset.approveERC20(IERC20(fromAssetId), _swapData.approveTo, fromAmount);
    }

    // solhint-disable-next-line avoid-low-level-calls
    (bool success, bytes memory res) = _swapData.callTo.call{ value: msg.value }(_swapData.callData);
    if (!success) {
        string memory reason = LibUtil.getRevertMsg(res);
        revert(reason);
    }
```

Li.Fi

- Since user can choose which bridge and function to use…
- And user controls both target and call data…
- => Arbitrary function call (similar to classic code injection)
- Cause: no input sanitization

```
function swap(bytes32 transactionId, SwapData calldata _swapData) internal {
    uint256 fromAmount = _swapData.fromAmount;
    uint256 toAmount = LibAsset.getOwnBalance(_swapData.receivingAssetId);
    address fromAssetId = _swapData.sendingAssetId;
    if (!LibAsset.isNativeAsset(fromAssetId) && LibAsset.getOwnBalance(fromAssetId) < fromAmount) {
        LibAsset.transferFromERC20(_swapData.sendingAssetId, msg.sender, address(this), fromAmount);
    }

    if (!LibAsset.isNativeAsset(fromAssetId)) {
        LibAsset.approveERC20(IERC20(fromAssetId), _swapData.approveTo, fromAmount);
    }

    // solhint-disable-next-line avoid-low-level-calls
    (bool success, bytes memory res) = _swapData.callTo.call{ value: msg.value }(_swapData.callData);
    if (!success) {
        string memory reason = LibUtil.getRevertMsg(res);
        revert(reason);
    }
```

Li.Fi

(aka TransferTo.xyz)

$600K

Multichain

Multichain (formerly AnySwap)

- Bridged tokens are wrapped with *anyToken*
- Example:
  - lock USDC (bridge) -> mint anyUSDC
  - burn anyUSDC (un-bridge) -> unlock USDC
- Also implements single-transaction transfers via permit()

Multichain (formerly AnySwap)

- Function expects **token** to be AnySwap wrapped token
- The function tries to **permit()** (i.e. approve) transferring the underlying token
- Since **permit()** reverts on fail this seems safe

```
    function anySwapOutUnderlyingWithPermit(
        address from,
        address token,
        address to,
        uint amount,
        uint deadline,
        uint8 v,
        bytes32 r,
        bytes32 s,
        uint toChainID
    ) external {
        address _underlying = AnyswapV1ERC20(token).underlying();
        IERC20(_underlying).permit(from, address(this), amount, deadline, v, r, s);
        TransferHelper.safeTransferFrom(_underlying, from, token, amount);
        AnyswapV1ERC20(token).depositVault(amount, from);
        _anySwapOut(from, token, to, amount, toChainID);
    }
```

Multichain (formerly AnySwap)

- Notice **token** is user-controlled
- Hence **_underlying** is user-controlled
- Therefore **permit()** is called on controlled address
- How is this vulnerable?

```
function anySwapOutUnderlyingWithPermit(
    address from,
    address token,
    address to,
    uint amount,
    uint deadline,
    uint8 v,
    bytes32 r,
    bytes32 s,
    uint toChainID
) external {
    address _underlying = AnyswapV1ERC20(token).underlying();
    IERC20(_underlying).permit(from, address(this), amount, deadline, v, r, s);
    TransferHelper.safeTransferFrom(_underlying, from, token, amount);
    AnyswapV1ERC20(token).depositVault(amount, from);
    _anySwapOut(from, token, to, amount, toChainID);
}
```

Multichain (formerly AnySwap)

- According to spec *permit()* necessarily reverts
- If *permit()* isn't supported by the token => Undefined Behaviour!
- And transferFrom will still need approval

```
    function anySwapOutUnderlyingWithPermit(
        address from,
        address token,
        address to,
        uint amount,
        uint deadline,
        uint8 v,
        bytes32 r,
        bytes32 s,
        uint toChainID
    ) external {
        address _underlying = AnyswapV1ERC20(token).underlying();
        IERC20(_underlying).permit(from, address(this), amount, deadline, v, r, s);
        TransferHelper.safeTransferFrom(_underlying, from, token, amount);
        AnyswapV1ERC20(token).depositVault(amount, from);
        _anySwapOut(from, token, to, amount, toChainID);
    }
```

Multichain (formerly AnySwap)

- from is also user-controlled
- If from already approved the bridge to use tokens AND permit() is no-op - tokens are essentially up for grabs!

```
    function anySwapOutUnderlyingWithPermit(
        address from,
        address token,
        address to,
        uint amount,
        uint deadline,
        uint8 v,
        bytes32 r,
        bytes32 s,
        uint toChainID
    ) external {
        address _underlying = AnyswapV1ERC20(token).underlying();
        IERC20(_underlying).permit(from, address(this), amount, deadline, v, r, s);
        TransferHelper.safeTransferFrom(_underlying, from, token, amount);
        AnyswapV1ERC20(token).depositVault(amount, from);
        _anySwapOut(from, token, to, amount, toChainID);
    }
```

Multichain (formerly AnySwap)

- Improper delegated verification
  - Delegated authorization to token, but *wrapped* token is user-controlled
- DO NOT ALLOW *transferFrom()* ANYTHING OTHER THAN *msg.sender* (!)
  - permit() is still useful to make single-transaction transfers

```
function depositWithPermit(address target, uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s, address to) external returns (uint) {
    IERC20(underlying).permit(target, address(this), value, deadline, v, r, s);
    IERC20(underlying).safeTransferFrom(target, address(this), value);
    return _deposit(value, to);
}
```

Multichain

(formerly AnySwap)

$3M

Nomad

Nomad

- Optimistic message bridge
- Validators commit merkle roots
- Users supply proofs

Nomad - BEFORE

process() - before

Non-existent messages[_messageHash] is 0x0 and MessageStatus.Proven is 0x1

(Basically 0x0 == 0x1 if hash does not exist)

```
function process(bytes memory _message) public returns (bool _success) {
    bytes29 _m = _message.ref(0);
    // ensure message was meant for this domain
    require(_m.destination() == localDomain, "!destination");
    // ensure message has been proven
    bytes32 _messageHash = _m.keccak();
    require(messages[_messageHash] == MessageStatus.Proven, "!proven");
```

Nomad - AFTER

process() - after

Now it is possible to call acceptableRoot(0x0)

(Remember: non-existent messages[_messageHash])

```
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");
```

Nomad - AFTER

acceptableRoot()

Remember: _root is 0x0 if hash is non-existent!

```
function acceptableRoot(bytes32 _root) public view returns (bool) {
    uint256 _time = confirmAt[_root];
    if (_time == 0) {
        return false;
    }
    return block.timestamp >= _time;
}
```

Nomad - the upgrade

_comittedRoot == 0x0 (!!)

```
    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);
    }
```

Nomad - recap

acceptableRoot()

Remember: _root is 0x0 if hash is non-existent!

```
function acceptableRoot(bytes32 _root) public view returns (bool) {
    uint256 _time = confirmAt[_root];
    if (_time == 0) {
        return false;
    }
    return block.timestamp >= _time;
}
```

Nomad - decentralized robbery

What actually happened?

- process() could be called without prove()
- MEV bots and even non-technical people could copy-paste previous transactions and only replace the receiver address
- Those cause ripple effects, triggering more bots and people grabbing funds

Nomad - summary

- Cause(s):
  - decoupled proof from processing (both prove() and process() are public)
  - upgrade introduced **“Master Password”**
  - unaudited upgrade introduced vulnerabilities

Nomad - BONUS

Remember the call to acceptableRoot()?

Because _commitedRoot was initialized to 0x0 it was possible to bypass this check…

```
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");
```

Nomad - BONUS

But before the upgrade, process() didn't even call acceptableRoot()!

This means that the init alone wasn't exploitable. The code change alone wasn't exploitable. **Only the combination of both enabled exploitation!**

```
function process(bytes memory _message) public returns (bool _success) {
    bytes29 _m = _message.ref(0);
    // ensure message was meant for this domain
    require(_m.destination() == localDomain, "!destination");
    // ensure message has been proven
    bytes32 _messageHash = _m.keccak();
    require(messages[_messageHash] == MessageStatus.Proven, "!proven");
```

Nomad

gang rekt

$190M

Optimism

Optimism

- Optimism is a… well, optimistic rollup
- Modified geth, L2 transaction data written to L1, protected by fault proofs.
- Deviated from standard - Ether was abstracted as ERC20 token

address(this).balance ➡ ERC20(OVM_ETH).balanceOf(this)

(classic way) (the *optimistic* way)

Optimism

- selfdestruct opcode is supposed to send remaining Ether to target address
- Instead, selfdestruct executed something like this:

```
ERC20(OVM_ETH).balances[target] += ERC20(OVM_ETH).balanceOf(this)

address(this).balance = 0
```

- Notice it increases the ERC20 balance, but does not decrease it
- Instead it sets the classic Ether balance (meaningless on Optimism)

Optimism

This allowed unlimited minting of L2 ETH:

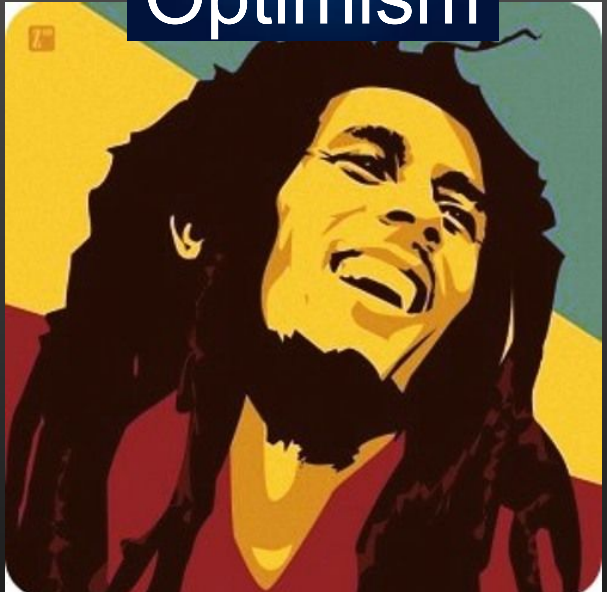
1. send some ERC20 ether to contract
2. selfdestruct contract => send ERC20 ether but also get to keep it!
3. create2 to access same address again.
4. repeat!

Optimism

- Broke a basic assumption by bridges: no double-spend on any chain.
- Funds at risk: all assets on all bridges
  - Main bridge could be drained of ETH
  - Additional minted (unbacked) ETH could buy all tokens on DEXes and drain all bridges.
- Reported by Saurik, $2M bounty paid.

Reported by Saurik

$2M bounty paid



REDEMPTION!

Optimism: Time Travel

- Optimistic rollup sequencers are kept honest by fraud/fault proofs.
- 7 day window to reorg the chain of the sequencer is proven wrong.

Optimism: Time Travel

- Bugs in the proof system may result in arbitrary long term reorgs.
- Bridges assume finality within a short time.
- Optimistic rollups may break that assumption.

Optimism: Time Travel

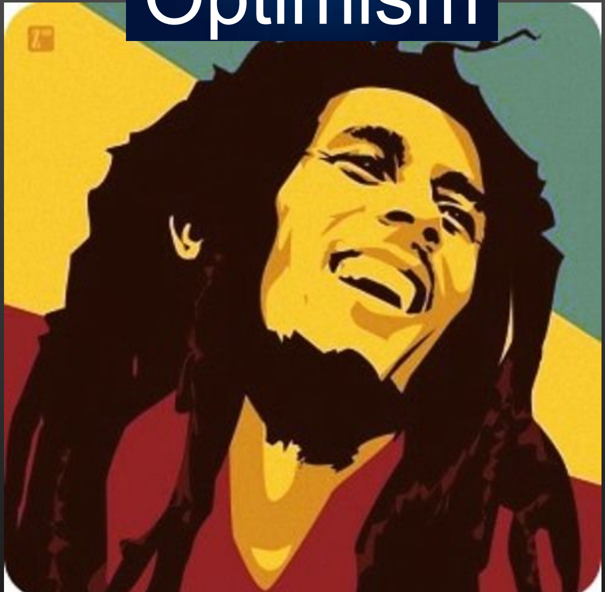
- Multiple bugs in OVM 1 contracts discovered by me :)
- Allowed selectively reverting past transactions up to 7 days back.

Optimism: Time Travel

- Funds at risk: all assets on all bridges, limited by a multiplier of attacker’s liquidity.
  - Double spend: send liquidity into the network, transfer out through bridge, revert on L2. Attacker has the funds on both ends.
  - Retroactively frontrun all DEXes, 7 days arbitrage available only to the attacker. Multiplier on the double-spent amount.
- Optimism deprecated OVM 1 in favor of the more secure OVM 2.

Reported by yours truly

;)



REDEMPTION!

Optimism: Time Travel

- The assumption by bridges remains risky: long term reorgs pose a major risk.

Arbitrum

Arbitrum (pre-prod) - bridges break assumptions too

- Optimistic rollup
- Has a built-in message bridge
- Pre-production bug in ArbOne fixed shortly before official launch.

Arbitrum (pre-prod) - bridges break assumptions too

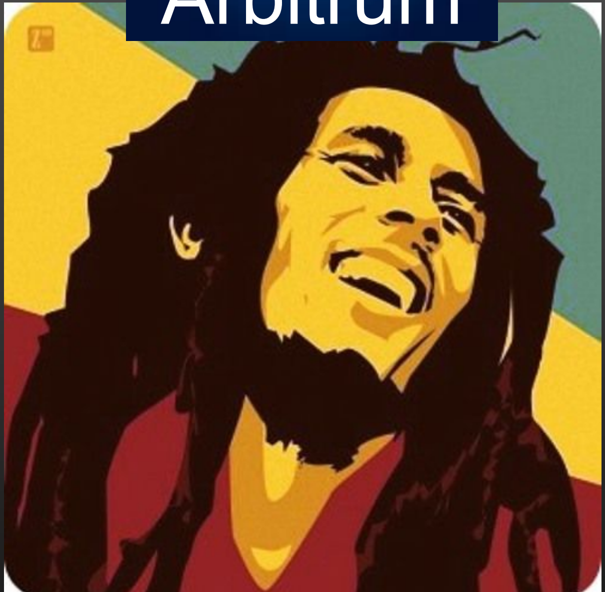
- Message from L1 address would trigger a call from the same address on L2
- Contracts deployed by CREATE on L1 and L2 may have different code
- Attacker may deploy a legit DeFi contract on L2 and later deploy a proxy to the same L1 address
- Proxy can use the bridge to cause L2 contract to transfer all its assets
- Rugpull!

Arbitrum (pre-prod) - bridges break assumptions too

- The bridge broke a basic assumption about smart contracts
- Contract should only perform calls that appear in its code
- Arbitrum discovered and fixed this bug before production. No funds at risk.

Fixed pre-production

Arbitrum



REDEMPTION!

Ronin

Ronin

- 5/9 Multisig
- The attacker managed to get control over five of the nine validator private keys — 4 Sky Mavis validators and 1 Axie DAO
- Sky Mavis keys extracted by spear-phishing emails
- Axie DAO provided the final needed signature - for free
  - At the time it wasn't supposed to have any signing power!

Ronin - summary

- Not all web3 hacks are caused by web3 vulnerabilities
- 5/9 multisig is 2/9 multisig if 4 keys are stored at the same place
- 2/9 multisig is 1/9 multisig if 1 signature is free
- 1/9 multisig is not a multisig


rekt
Ronin
$ 7 0 0 M
certified REKT
rekt

Closing Thoughts

"From rugs to bridges"

- Hacked? Just Rebrand
  - AnySwap => Multichain
  - Wormhole => Portal
  - Li.Fi => TransferTo.xyz
- The market should demand security and accountability
- Upgrade? "It's just a single line change"
  - Remember: audits are 2nd most expensive cost of upgrading
- Keys? "It's okay, I have copies"
  - Keep your friends close and your keys closer

THANK YOU

	Presentor	Collaborator
Twitter:	@yoavw	@high_byte