

\$15B DeFi Eclipse

Hanan Beer

(alternative names: Rainbow Eclipse, Quantum Exploit, “I Hate It When My Boss Won’t Let Me Hack \$15B”)

No one ever made a 10 digits USD hack... yet. And I’m not about to either. But it serves right for an 11 digits vulnerability to be quantum exploitable. That is - it is known whether it is exploitable only when attempted to exploit.

This is the strange story about \$15B hanging by a thread. It may be considered the biggest blockchain vulnerability to have ever existed¹, as it requires $\sim \$15K^2$ to extract up to $\sim \$15B$. Others would prefer to call it the “1,000,000x trade” (that is - 100M% return) – after all, **Code Is Law**.

It all started soon after the infamous Wormhole Bridge hack where about $\sim \$300M$ were stolen³. In the era of burned bridges it made sense to me this is some opportunity you cannot wait for.

Quite arbitrarily I landed in NEAR’s Rainbow bridge. Why NEAR? Well I just happened to read how they raised (another) \$150M so I figured they will have relatively non-battle-tested security but also high value.

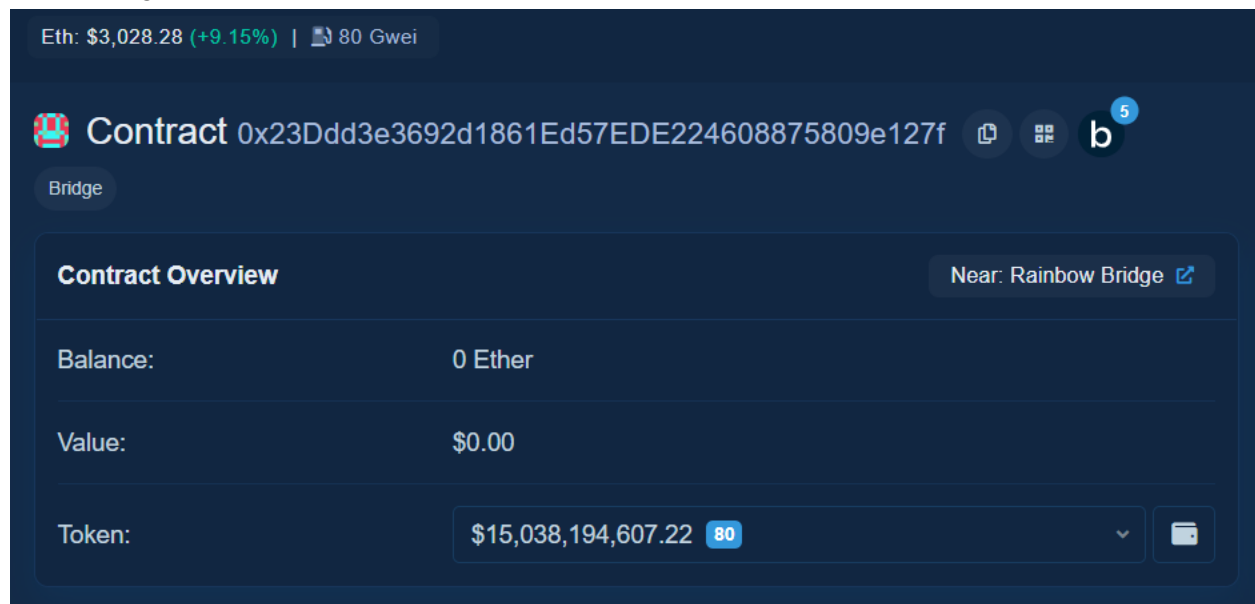


Figure 1.

¹ That was true before saurik came with his [potentially unbounded vulnerability](#) with catastrophic economic impact

² But not really because this only the absolute lower bound

³ Give or take \$50M depending whether you refer to minted value or extracted value

Soon after fiddling with Rainbow bridge's frontend, playing and learning about NEAR & Aurora, I landed in their [Ethereum-side bridge contract](#) – I chose to start with this end of the bridge because Etherscan's block explorer is much more mature and I know my way around it. Let's dive in.

Bridges work by locking assets on one chain and releasing equal-value assets on another chain. BUT – in the process you lose all of blockchain's top-tier security. And this is why we see so many bridges getting hacked.

Bridge contracts increase the attack surface, as we've seen vulnerable bridge contracts, and we even see External Addresses with **billions of dollars**, something which is completely, ridiculously insane in my opinion.

```
1279  function unlockToken(bytes memory proofData, uint64 proofBlockHeight)
1280      public
1281      pausable (PAUSED_UNLOCK)
1282  {
1283      ProofDecoder.ExecutionStatus memory status = _parseAndConsumeProof(proofData, proofBlockHeight);
1284      BurnResult memory result = _decodeBurnResult(status.successValue);
1285      IERC20(result.token).safeTransfer(result.recipient, result.amount);
1286      emit Unlocked(result.amount, result.recipient);
1287  }
```

Figure 2.

However, Rainbow bridge (sort-of) implements NEAR's blockchain proofs on top of their Ethereum-side contracts, such that these contracts verify proofs for blocks on the NEAR chain. Similarly, blocks require Merkle tree hashes and supermajority staked approvals, and anyone can stake some ETH to participate in the block insertion process!

While the blocks themselves are verified & signed by stakeholders on NEAR, the consensus mechanism for Rainbow bridge is somewhat different:

1. Block time is set to 4 hours
2. Once counting starts – this duration can be reset by new & **untrusted** blocks.
3. **The default behavior for untrusted blocks is: to be trusted.**

Here is what it looks like in the code.

unlockTokens calls **_parseAndConsumeProof**:

```
1191     /// Parses the provided proof and consumes it if it's not already used.
1192     /// The consumed event cannot be reused for future calls.
1193     function _parseAndConsumeProof(bytes memory proofData, uint64 proofBlockHeight)
1194     internal
1195     returns (ProofDecoder.ExecutionStatus memory result)
1196     {
1197         require(proofBlockHeight >= minBlockAcceptanceHeight_, "Proof is from the ancient block");
1198         require(prover_.proveOutcome(proofData, proofBlockHeight), "Proof should be valid");
1199
1200         // Unpack the proof and extract the execution outcome.
1201         Borsh.Data memory borshData = Borsh.from(proofData);
1202         ProofDecoder.FullOutcomeProof memory fullOutcomeProof = borshData.decodeFullOutcomeProof();
1203         require(borshData.finished(), "Argument should be exact borsh serialization");
1204
1205         bytes32 receiptId = fullOutcomeProof.outcome_proof.outcome_with_id.outcome.receipt_ids[0];
1206         require(!usedProofs_[receiptId], "The burn event proof cannot be reused");
1207         usedProofs_[receiptId] = true;
1208
1209         require(keccak256(fullOutcomeProof.outcome_proof.outcome_with_id.outcome.executor_id)
1210             == keccak256(nearTokenFactory_),
1211             "Can only unlock tokens from the linked proof producer on Near blockchain");
1212
1213         result = fullOutcomeProof.outcome_proof.outcome_with_id.outcome.status;
1214         require(!result.failed, "Cannot use failed execution outcome for unlocking the tokens");
1215         require(!result.unknown, "Cannot use unknown execution outcome for unlocking the tokens");
1216     }
1217 }
```

Figure 3.

minBlockAcceptanceHeight_ is 0, so this requirement is always satisfied. Immediately after, **prover_.proveOutcome** is called.

```
884     function proveOutcome(bytes memory proofData, uint64 blockHeight)
885     public
886     view
887     override
888     pausable(PAUSED_VERIFY)
889     returns (bool)
890     {
891         Borsh.Data memory borshData = Borsh.from(proofData);
892         ProofDecoder.FullOutcomeProof memory fullOutcomeProof = borshData.decodeFullOutcomeProof();
893         require(borshData.finished(), "NearProver: argument should be exact borsh serialization");
894
895         bytes32 hash =
896             _computeRoot(fullOutcomeProof.outcome_proof.outcome_with_id.hash, fullOutcomeProof.outcome_proof.proof);
897
898         hash = sha256(abi.encodePacked(hash));
899
900         hash = _computeRoot(hash, fullOutcomeProof.outcome_root_proof);
901
902         require(
903             hash == fullOutcomeProof.block_header_lite.inner_lite.outcome_root,
904             "NearProver: outcome merkle proof is not valid"
905         );
906
907         bytes32 expectedBlockMerkleRoot = bridge.blockMerkleRoots(blockHeight);
908
909         require(
910             _computeRoot(fullOutcomeProof.block_header_lite.hash, fullOutcomeProof.block_proof) ==
911             expectedBlockMerkleRoot,
912             "NearProver: block proof is not valid"
913         );
914
915         return true;
916     }
```

Figure 4.

Again, some more magical cryptographic proofs stuff, but the kicker's at line 907:

```
bytes32 expectedBlockMerkleRoot = bridge.blockMerkleRoots(blockHeight);
```

Let me just give you a perspective of what went through my mind at that moment. I like to imagine the flow of a successful execution for **unlockToken**. I skip through the code quickly, assuming and assuming some more, that the function will succeed. (otherwise, what's the point of a function that always fails?)

Following line 907 is a requirement for some expected value. That is, some scary-looking function must return a value equal to the scary-named variable **expectedBlockMerkleRoot**. Obviously that requirement is there for a reason, so in my mind, there should be some input where the flow of execution is *just about right*, and the final decision depends on what is returned from `bridge.blockMerkleRoots(blockHeight)`.

Let's go down this rabbit hole:

```
293  ✓ function blockMerkleRoots(uint64 height) public view override pausable(PAUSED_VERIFY) returns (bytes32 res) {
294      res = blockMerkleRoots_[height];
295  ✓   if (res == 0 && block.timestamp >= lastValidAt && lastValidAt != 0 && height == untrustedHeight) {
296       res = untrustedMerkleRoot;
297   }
298 }
```

Figure 5.

WAT?!

Whenever you see the word “**untrusted**” in a source code - you know someone is going to have a bad time.

Let's analyze this piece of code: there's some mapping called **blockMerkleRoots_**, which as the name suggests, maps the parameter **height** to a block's merkle root.

Now... What the hell is a merkle root?!

We can worry about that later⁴, so let's just continue assuming for now.

So if **height** is not present in the mapping (**res == 0**) AND the **block.timestamp** is greater than **lastValidAt** AND also not zero... AAAND **height** is **untrustedHeight**: then return **untrustedMerkleRoot**. Simple, right?

Believe it or not, but that is probably the simplest piece of code around. So if you got tired of following up to this point, let me give you the kicker: we can control **untrustedMerkleRoot**! That is the root cause of the vulnerability!

Except... we do have to satisfy all these conditions first. Not to mention the horrors of supplying a valid proof. So for the brave reader willing to dig deeper down this rabbit hole, if \$15B aren't motivation enough, let me cheer you up by saying that by the time you finish reading this, you will be one step closer, one step wiser and one step more informed that the next hack could be found by YOU!

⁴ Further details given later in the article with references

Now let's see how we can control **untrustedMerkleRoot**:

```
168     function addLightClientBlock(bytes memory data) public override pausable(PAUSED_ADD_BLOCK) {
169         require(initialized, "Contract is not initialized");
170         require(balanceOf[msg.sender] >= lockEthAmount, "Balance is not enough");
171
172         Borsh.Data memory borsh = Borsh.from(data);
173         NearDecoder.LightClientBlock memory nearBlock = borsh.decodeLightClientBlock();
174         borsh.done();
175
176         unchecked {
177             // Commit the previous block, or make sure that it is OK to replace it.
178 >         if (block.timestamp < lastValidAt) {...
183 >         } else if (lastValidAt != 0) {...
192         }
193
194             // Check that the new block's height is greater than the current one's.
195             require(nearBlock.inner_lite.height > curHeight, "New block must have higher height");
196
197             // Check that the new block is from the same epoch as the current one, or from the next one.
198             bool fromNextEpoch;
199 >             if (nearBlock.inner_lite.epoch_id == epochs[curEpoch].epochId) {...
201 >             } else if (nearBlock.inner_lite.epoch_id == epochs[(curEpoch + 1) % 3].epochId) {...
203 >             } else {...
205         }
206
207             // Check that the new block is signed by more than 2/3 of the validators.
208             Epoch storage thisEpoch = epochs[fromNextEpoch ? (curEpoch + 1) % 3 : curEpoch];
209             // Last block in the epoch might contain extra approvals that light client can ignore.
210             require(nearBlock.approvals_after_next.length >= thisEpoch.numBPs, "Approval list is too short");
211             // The sum of uint128 values cannot overflow.
212             uint256 votedFor = 0;
213 >             for ((uint i, uint cnt) = (0, thisEpoch.numBPs); i != cnt; ++i) {...
224             }
225             require(votedFor > thisEpoch.stakeThreshold, "Too few approvals");
226
227             // If the block is from the next epoch, make sure that next_bps is supplied and has a correct hash.
228 >             if (fromNextEpoch) {...
234             }
235
236             untrustedHeight = nearBlock.inner_lite.height;
237             untrustedTimestamp = nearBlock.inner_lite.timestamp;
238             untrustedHash = nearBlock.hash;
239             untrustedMerkleRoot = nearBlock.inner_lite.block_merkle_root;
240             untrustedNextHash = nearBlock.next_hash;
```

Figure 6.

The method **addLightClientBlock** is used to insert block information from the NEAR blockchain into the Ethereum blockchain. It is routinely called every ~4 hours (**lockDuration**) to update asset transfers from NEAR to Ethereum.

Assuming all the conditions in the collapsed sections succeed, eventually at line 239 **untrustedMerkleRoot** is set to user-controlled input.

Recall when **untrustedMerkleRoot** is returned in the method **blockMerkleRoots**, specifically once `block.timestamp >= lastValidAt`.

This is set a little bit further down in **addLightClientBlock**:

```
236         untrustedHeight = nearBlock.inner_lite.height;
237         untrustedTimestamp = nearBlock.inner_lite.timestamp;
238         untrustedHash = nearBlock.hash;
239         untrustedMerkleRoot = nearBlock.inner_lite.block_merkle_root;
240         untrustedNextHash = nearBlock.next_hash;
241
242         uint256 signatureSet = 0;
243 >         for ((uint i, uint cnt) = (0, thisEpoch.numBPs); i < cnt; i++) { ...
249         }
250         untrustedSignatureSet = signatureSet;
251         untrustedNextEpoch = fromNextEpoch;
252 >         if (fromNextEpoch) { ...
256         }
257         lastSubmitter = msg.sender;
258         lastValidAt = block.timestamp + lockDuration;
259     }
```

Figure 7.

Since **lockDuration** is set to 4 hours, it means **untrustedMerkleRoot** can be returned starting from 4 hours after it was set.

To bypass the final requirement shown in Figure 4 on line 910, an attacker can forge fake proof data, calculate the result of

```
computeRoot(fullOutcomeProof.block_header_lite.hash,
fullOutcomeProof.block_proof)
```

then set **untrustedMerkleRoot** by calling **addLightClientBlock** to the calculated value and wait for 4 hours.

But what about the original block producer calling **addLightClientBlock** periodically? Won't it overwrite the fake **untrustedMerkleRoot**?

```
177         // Commit the previous block, or make sure that it is OK to replace it.
178         if (block.timestamp < lastValidAt) {
179             require( // time_lt
180                 nearBlock.inner_lite.timestamp >= untrustedTimestamp + replaceDuration,
181                 "Can only replace with a sufficiently newer block"
182             );
183         } else if (lastValidAt != 0) { // time_gte
184             curHeight = untrustedHeight;
185             if (untrustedNextEpoch) {
186                 curEpoch = (curEpoch + 1) % 3;
187             }
188             lastValidAt = 0;
189
190             blockHashes_[curHeight] = untrustedHash;
191             blockMerkleRoots_[curHeight] = untrustedMerkleRoot;
192         }
```

Figure 8. **addLightClientBlock** internals

As shown in line 180 on Figure 8, if someone is attempting to insert block data before the locking period is over, they will also be required to introduce a “sufficiently newer block” – by showing its timestamp is greater than the timestamp of the pending block. **replaceDuration** is set to 5 hours and we know the bot is called every 4 hours, hence it is safe to assume that an attacker can craft block data such that blocks attempting to overwrite it will necessarily fail on this requirement. Moreover, **nearBlock.inner_lite.timestamp** is uint64 while **untrustedTimestamp + replaceDuration** is uint256, meaning a sufficiently large **untrustedTimestamp** (e.g. $\text{uint64}(-1)$) will cause newer calls to **addLightClientBlock** to necessarily fail.

Alright, so far we know that we need to:

1. Forge desired proof data
2. Forge block data based on that proof's hash
3. Ensure the block survives the locking period

To understand why **addLightClientBlock** accepts just about any kind of data, we need to understand NEAR. This blockchain uses [ed25519](#) signatures of stakers to verify block data. Each signature is associated with a validator who puts a certain stake to secure the blockchain. It is required that validators with at least $\frac{2}{3}$ of the total stake sign each block. However, EVM does not natively implement ed25519, and in fact it is quite costly. That is why NEAR opted for an optimistic approach - “trust, but verify”. Oh boy...



Cthulhu, Lord of The Underworld. (credit: unknown artist)

You Can't Have A Rainbow During An Eclipse

So far the logic seems solid, at least for steps 1 & 2. For step 3 however... little does our fake block producer know – there are other dangers lurking in the dark forest.

```
91  function challenge(address payable receiver, uint signatureIndex) public override pausable(PAUSED_CHALLENGE) {
92      require(block.timestamp < lastValidAt, "No block can be challenged at this time");
93      require(!checkBlockProducerSignatureInHead(signatureIndex), "Can't challenge valid signature");
94
95      balanceOf[lastSubmitter] = balanceOf[lastSubmitter] - lockEthAmount;
96      receiver.transfer(lockEthAmount / 2);
97      lastValidAt = 0;
98  }
99
100 function checkBlockProducerSignatureInHead(uint signatureIndex) public view override returns (bool) {
101     // Shifting by a number >= 256 returns zero.
102     require((untrustedSignatureSet & (1 << signatureIndex)) != 0, "No such signature");
103     unchecked {
104         Epoch storage untrustedEpoch = epochs[untrustedNextEpoch ? (curEpoch + 1) % 3 : curEpoch];
105         NearDecoder.Signature storage signature = untrustedSignatures[signatureIndex];
106         bytes memory message = abi.encodePacked(
107             uint8(0),
108             untrustedNextHash,
109             Utils.swapBytes8(untrustedHeight + 2),
110             bytes23(0)
111         );
112         (bytes32 arg1, bytes9 arg2) = abi.decode(message, (bytes32, bytes9));
113         return edwards.check(untrustedEpoch.keys[signatureIndex], signature.r, signature.s, arg1, arg2);
114     }
115 }
```

Figure 9.

There exists a **challenge** method, and although never previously called, it is safe to assume that there is some bot, somewhere, lurking, checking, verifying, awaiting to invoke a **challenge**.

But what if there was some way to avoid a **challenge**? The answer is: “it’s complicated”.

Let’s start with **how** a challenge works and what I thought **would** work (but didn’t), then I will explain what **could** work (maybe) and finally what **should** work (probably).



Captain Hindsight with Shoulda, Coulda & Woulda (credit: southpark)

The How

To quickly summarize the **challenge** method – which calls **checkBlockProducerSignatureInHead** – line 113 calls **edwards.check**, where **edwards** is an **Ed25519** library. As mentioned earlier, it is quite costly to execute this method, let alone on all of the 100 signatures, which is why a **signatureIndex** is provided.

For **challenge** to succeed, the check must fail (due to invalid signature). Otherwise, the challenger just wasted a lot of gas. If it succeeds however, the challenger is reimbursed by half of the staker's amount, 2.5 ETH, while the other 2.5 ETH is locked in the contract forever.

Notice how the public keys used for the signature verification, as shown in Figure 9 line 113, come from either the current epoch or the next epoch. Signatures are included in the **addLightClientBlock** call.

```

274 function check(
275     bytes32 k,
276     bytes32 r,
277     bytes32 s,
278     bytes32 m1,
279     bytes9 m2
280 ) public pure returns (bool) {
281     unchecked {
282         uint256 hh;
283         // Step 1: compute SHA-512(R, A, M)
284         {
285 >         uint256[5][16] memory kk = [ ...
398     ];
399     uint256 w0 = (uint256(r) & 0xffffffffffffffff_00000000_00000000_00000000_00000000_ffffffff_ffffffff) |
400         ((uint256(r) & 0xffffffffffffffff_00000000_00000000_00000000_00000000) >> 64) |
401         ((uint256(r) & 0xffffffffffffffff_00000000_00000000) << 64);
402     uint256 w1 = (uint256(k) & 0xffffffffffffffff_00000000_00000000_00000000_00000000_ffffffff_ffffffff) |
403         ((uint256(k) & 0xffffffffffffffff_00000000_00000000_00000000_00000000) >> 64) |
404         ((uint256(k) & 0xffffffffffffffff_00000000_00000000) << 64);
405     uint256 w2 = (uint256(m1) & 0xffffffffffffffff_00000000_00000000_00000000_00000000_ffffffff_ffffffff) |
406         ((uint256(m1) & 0xffffffffffffffff_00000000_00000000_00000000_00000000) >> 64) |
407         ((uint256(m1) & 0xffffffffffffffff_00000000_00000000) << 64);
408     uint256 w3 = (uint256(bytes32(m2)) &
409         0xffffffffffffffff_00000000_00000000_00000000_00000000_00000000_00000000) |
410         ((uint256(bytes32(m2)) & 0xffffffffffffffff_00000000_00000000_00000000_00000000) >> 64) |
411         0x800000_00000000_00000000_00000000_00000000_00000000_00000000);
412     uint256 a = 0x6a09e667_f3bcc908;
413     uint256 b = 0xbb67ae85_84caa73b;
414     uint256 c = 0x3c6ef372_fe94f82b;
415     uint256 d = 0xa54ff53a_5f1d36f1;
416     uint256 e = 0x510e527f_ade682d1;
417     uint256 f = 0x9b05688c_2b3e6c1f;
418     uint256 g = 0x1f83d9ab_fb41bd6b;
419     uint256 h = 0x5be0cd19_137e2179;
420     for (uint256 i = 0; ; i++) {
421         // Round 16 * i
422         {
423             uint256 temp1;
424             uint256 temp2;
425             e &= 0xffffffffffffffff;
426             {
427                 uint256 ss = e | (e << 64);
428                 uint256 s1 = (ss >> 14) ^ (ss >> 18) ^ (ss >> 41);
429                 uint256 ch = (e & (f ^ g)) ^ g;
430                 temp1 = h + s1 + ch;
431             }
432             temp1 += kk[0][i];
433             temp1 += w0 >> 192;
434             a &= 0xffffffffffffffff;
435             {
436                 uint256 ss = a | (a << 64);
437                 uint256 s0 = (ss >> 28) ^ (ss >> 34) ^ (ss >> 39);
438                 uint256 maj = (a & (b | c)) | (b & c);
439                 temp2 = s0 + maj;
440             }
441             h = g;
442             g = f;
443             f = e;
444             e = d + temp1;
445             d = c;
446             c = b;
447             b = a;
448             a = temp1 + temp2;
449         }
450         // Round 16 * i + 1
451         {
452             uint256 temp1;
453             uint256 temp2;
454             e &= 0xffffffffffffffff;
455             {

```

Figure 10. (Only about 10% of the code is shown)

untrustedSignatureSet is a bitmask specifying which signatures were included in the latest call to **addLightClientBlock**, so which signatures can be challenged is actually in our control. *However*, signatures from enough validators are required, such that the sum of their stake is at least $\frac{2}{3}$ of the total stake.

```
212     uint256 votedFor = 0;
213     for ((uint i, uint cnt) = (0, thisEpoch.numBPs); i != cnt; ++i) {
214         bytes32 stakes = thisEpoch.packedStakes[i >> 1];
215         if (nearBlock.approvals_after_next[i].some) {
216             votedFor += uint128(bytes16(stakes));
217         }
218         if (++i == cnt) {
219             break;
220         }
221         if (nearBlock.approvals_after_next[i].some) {
222             votedFor += uint128(uint256(stakes));
223         }
224     }
225     require(votedFor > thisEpoch.stakeThreshold, "Too few approvals");
```

Figure 11. More **addLightClientBlock** internals. PoS requires at least $\frac{2}{3}$ approvals

Moreover, the bridge's [source code is available on github](#) - including the block producer bot & the challenge bot!

[Near2eth-block-relay](#) is the block producer service, and [watchdog](#) is the **challenge** service. (links point to the latest commit as of writing, such that the source shown is what I worked with)

Looking at the code, it is implied that the block producer and the challenger use the same private key `ethMasterSk` (coming from the same config file `.rainbow/config.json`) => meaning their transactions will originate from the same address. While it may be the case, it is still not safe to assume it is the only instance of the challenger. *However* I will elaborate in the epilog why I think it actually may be the case. And in the following sections you will learn why this is important.

The Would

After all the excitement of finding a vulnerability of this magnitude, overworked and underslept me made an off-by-one logical error, where I mistook a +2 for a +1, before realizing that +2 was very intentional and in fact, the difference between an unstoppable⁵ exploit and a risky one.

I thought I could replace the validators' signatures with my own, meaning I could've signed the fake block with my own fake signature(s) and bypass the **edwards.check**.

However...

```
252         if (fromNextEpoch) {
253             Epoch storage nextEpoch = epochs[(curEpoch + 2) % 3];
254             nextEpoch.epochId = nearBlock.inner_lite.next_epoch_id;
255             setBlockProducers(nearBlock.next_bps.blockProducers, nextEpoch);
256         }
```

Figure 12. Near the end of **addLightClientBlock**

This little piece of code within **addLightClientBlock** is the difference between a nice blog post & one of the worst vulnerabilities in DeFi and perhaps outside of crypto.

It's trivial to set **fromNextEpoch** by specifying the next epoch's id:

```
197         // Check that the new block is from the same epoch as the current one, or from the next one.
198         bool fromNextEpoch;
199         if (nearBlock.inner_lite.epoch_id == epochs[curEpoch].epochId) {
200             fromNextEpoch = false;
201         } else if (nearBlock.inner_lite.epoch_id == epochs[(curEpoch + 1) % 3].epochId) {
202             fromNextEpoch = true;
203         } else {
204             revert("Epoch id of the block is not valid");
205         }
```

Figure 13. **addLightClientBlock** requirements

And then the **setBlockProducers** method will be called, which as the name suggests – sets the next block producers, including their public keys for signatures verification:

```
262         function setBlockProducers(NearDecoder.BlockProducer[] memory src, Epoch storage epoch) internal {
263             uint cnt = src.length;
264             require(cnt <= MAX_BLOCK_PRODUCERS, "It is not expected having that many block producers for t
265             epoch.numBPs = cnt;
266             unchecked {
267                 for (uint i = 0; i < cnt; i++) {
268                     epoch.keys[i] = src[i].publicKey.k;
269                 }
270                 uint256 totalStake = 0; // Sum of uint128, can't be too big.
271                 for (uint i = 0; i != cnt; ++i) {
272                     uint128 stake1 = src[i].stake;
273                     totalStake += stake1;
```

Figure 14. If condition in Figure 12 is satisfied

⁵ contract still pause-able, but there is no bot to do this automatically

However, if **fromNextEpoch** is true, then as shown in Figure 12 on line 253, the **next epoch after that** (ie. the next-next) is the one that is assigned the new public keys. Hence the +2. But as shown in Figure 9 on line 104, sadly (to me), public keys from either the current epoch or the next epoch are used to verify signatures.

As you can imagine, I was – VERY DISAPPOINTED.

The Could

Okay so the bot's source code looks pretty solid, so that's off the table. And this is a decentralized blockchain, being uncensorable is kind of its thing, so no tricks here. An external attack is also unlikely, as even with immense resources it may be proven difficult-to-impossible to locate the bot, logically (IP address) or physically (server or geographically).

But... there is a way to "get rid" of the bot. And who knows, maybe in the end you will find out it isn't even running! (HA!)

You may have realized that there is one thing we *can* do to interact with it – present some **challenge**-able blocks! A **challenge** means a transaction, a transaction means... gas used!

Surprisingly, in the bot's source code **there are no indications the gas ran out!** No logs nor alerts. In fact, in this case their bot will attempt to **double** the gas price! 😏

Technically, one **could** do this: you would only have to burn potentially some millions of dollars in ETH in the process (or even as low as hundreds of thousands!), and there are no guarantees. But given the extremely high reward, I can already imagine some VCs drooling about such an... "investment".

Of course there's the issue of having our ETH slashed and given to the bot, further distancing ourselves from the end goal. But looking at **addLightClientBlock**, the transactions aren't private! Suggesting we should be able to detect **challenge** transactions in the mempool... and frontrun them!

Trivially, the flow would be:

1. **addLightClientBlock** with signatures 0~99 (100% stake)
2. Detect **challenge** in the mempool – first signature is invalid, so **signatureIndex** is 0
3. Frontrun with a new **addLightClientBlock** but with signatures 1~99 (<100% stake but still >66% required stake)
4. Now **challenge** is executed, but early-returned due to **signatureIndex** 0 not present in the **untrustedSignatureSet** mask.

Of course, an early return will cost such a negligible amount of ETH, an attacker might need thousands of successful frontruns to drain the gas. Also, **addLightClientBlock** is costly, around 0.05~0.3 ETH. So for about every \$10 of gas drained to the bot, the attacker would be about \$500 short as well. Also, failing once sets you back about 50 attempts - as well as costing you 5 ETH.

You may be good at frontrunning, but nobody's *that* good at frontrunning. And if you say you are, feel free to put your money where your mouth is ;)

BUT WHAT IF THERE'S ANOTHER WAY?

Here's what a well-funded attack *can* definitely do:

1. Copy the last legitimate **addLightClientBlock**, but still untrusted (uncommitted)
2. Issue **addLightClientBlock** 1st time with **challenge**-able block, will override 1.
3. Detect **challenge** in the mempool
4. Frontrun with a contract that does:
 - a. Attacker's own **challenge** to lose & save 2.5 ETH & set **lastValidAt** = 0
 - b. Deposit another 5 ETH
 - c. **addLightClientBlock** 2nd time from step 1, **untrustedTimestamp** will reset
5. Original **challenge** fails (!) - this will cost the bot about 0.03 ETH
6. Repeat until gas runs out (possible because **untrustedTimestamp** was reset)
7. Issue original attack unobstructed

Let's review:

- The cost of two **addLightClientBlock** calls and one successful **challenge** is roughly 0.3-0.6 ETH
- The attacker will lose 2.5 ETH in the process
- The total cost of the above is just about 3 ETH per iteration
- The cost for the original **challenge** bot is about 0.03 ETH per iteration
- Multiply by 33 and you get 100 ETH for the attacker per each 1 ETH drained for the bot
- That's about ~\$300K at the time of writing

And with only about 5 ETH at the moment of writing, this attack becomes practical - requiring only \$1.5M in total (!!!)

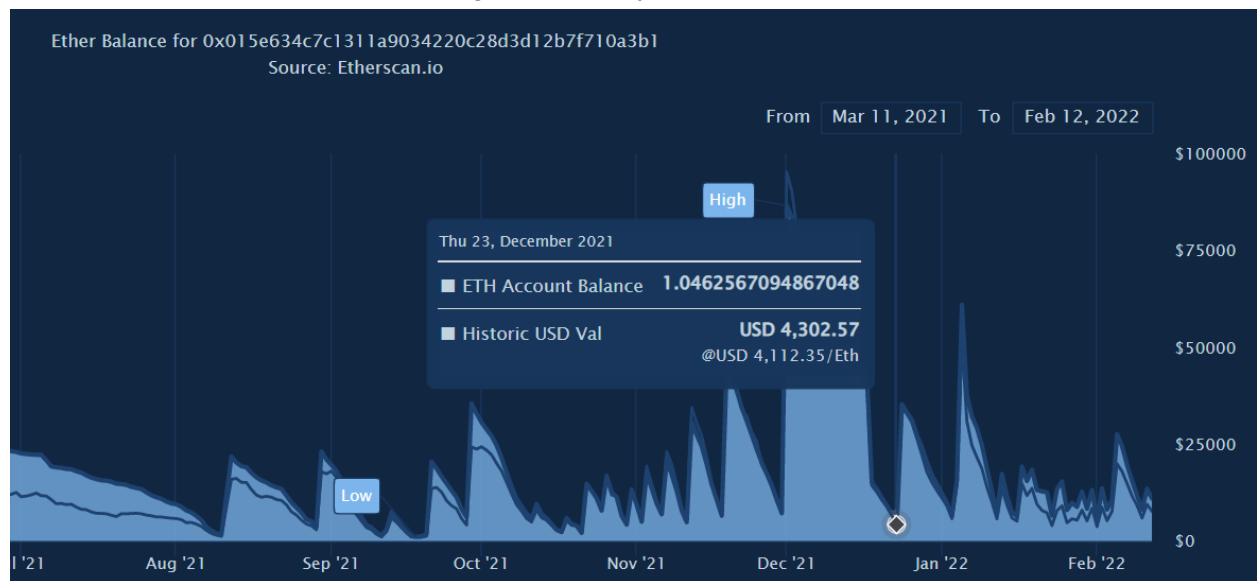
Moreover, executing such an attack during the weekend now becomes an actual possibility, reducing the chance of detection significantly.

The Should

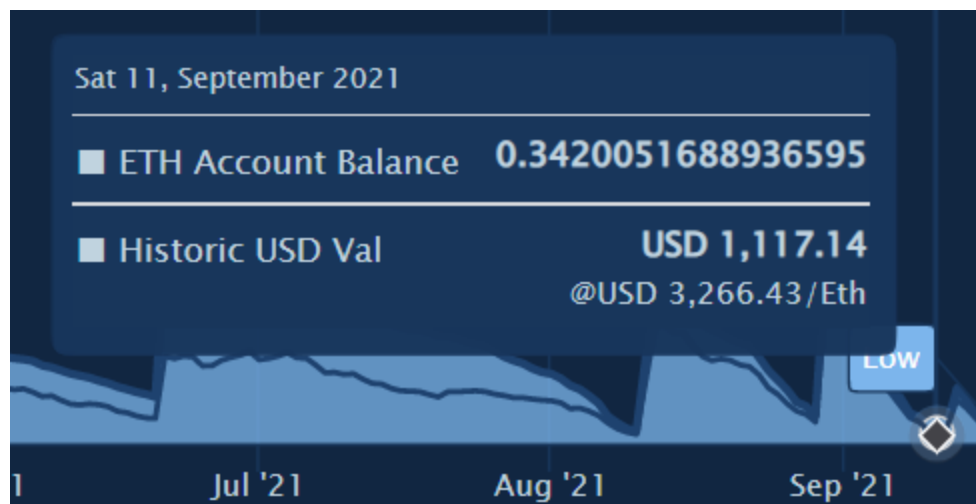
And finally, last but not least, the ultimate solution may be incredibly simple: do nothing. Or more specifically – to just wait.

The bot operators really like to live on the edge. Like *reeaaaallllyyy* on the edge, \$15B-hanging-by-a-thread kind of edge.






On Dec 23, 2021 the bot was looking kinda thirsty:



In fact, at one point it had balance as low as 0.342 ETH! (*although it was before the current version was deployed*)



And what about the refuel bot? Well, today on Feb 13, 2022 as I'm writing these words it doesn't look too good either.

 Address `0xD9cB077700AA4D32d30bDA5e99bb171549b5a382`    

Overview


Balance:


0.961148464896793425 Ether


Value:

\$2,764.22 (@ \$2,875.96/ETH)

Token:

\$9.95 





Still though it may not be enough. The challenge method costs around $\frac{1}{3}$ of the average **addLightClientBlock**, such that there is still a greater chance, roughly ~66%, the bot will run out of gas for new blocks but still be able to challenge. There is a last mile solution, though.

The Last Mile

Consider a situation as described above – low on gas, but just about enough for the **challenge**. Given such a rare opportunity, the event where the bot ran out of gas, implying the refuel bot had probably run out of gas too - is such a rare event and something unlikely to happen twice. Especially given that the bridge will be DoS'ed by itself, quickly triggering alerts in the form of anxious users, followed by what is likely a hefty some of ETH for gas and (preferably) a complete restructure in the way the system is managed, it's safe to say only one such opportunity will be given.

And so we've come this far – and still our chances are bound at no more than 33%?? But this section is called *The Last Mile*, after all...

The sharp reader (potentially a sophisticated attacker who has patiently waited thus far) may have figured out the solution by now: send the bot some ETH!

Haha, yes!

It may not be completely trivial without running some statistics (as I did), but transactions calling **addLightClientBlock** have a median gas usage of ~1.7M units, and 90% deviate from this mean by ~0.2M units, giving us the likely range of 1.5M~1.9M units of gas. And as I mentioned earlier, the challenge has about 1/3 of the cost at almost exactly 0.6M units of gas. (a fixed amount as it is not dependent on the amount of block data, unlike **addLightClientBlock**)

And perhaps the solution is still not apparent given all these numbers - do not worry, as this type of hack isn't something you do in DeFi just every day.

Let's go back to the scenario where the bot ran out of gas, but has just enough for a **challenge**. We know exactly how much ETH it has left. And... we can estimate with satisfying accuracy what the cost of **addLightClientBlock** might be, in gas units. BUT - we need to know the exact price in ETH. So we wait. We know the transaction will be issued almost exactly 4 hours (and 1 or 2 minutes) after the previous call. So minutes before that, we can check the base fee for gas, multiply that by the higher range we estimated at 1.9M gas units and we get the estimated transaction price in ETH! We subtract the current balance from that and send the bot this amount of ETH. If executed properly (and a bit of luck), the next **addLightClientBlock** transaction goes through, leaving the bot completely high & dry! Even if the transaction ended up costing only 1.5M gas, that leaves enough ETH for only about 0.4M gas units (at the same fee). Now isn't that just a mind blowing operation that wouldn't embarrass even

██████████.

Merkle Trees

You may have noticed one key piece is missing - and that is that we need an actual valid merkle tree to pass the requirement at Figure 4 on line 903 (shown here again for convenience).

```
884  function proveOutcome(bytes memory proofData, uint64 blockHeight)
885      public
886      view
887      override
888      pausable(PAUSED_VERIFY)
889      returns (bool)
890  {
891      Borsh.Data memory borshData = Borsh.from(proofData);
892      ProofDecoder.FullOutcomeProof memory fullOutcomeProof = borshData.decodeFullOutcomeProof();
893      require(borshData.finished(), "NearProver: argument should be exact borsh serialization");
894
895      bytes32 hash =
896          _computeRoot(fullOutcomeProof.outcome_proof.outcome_with_id.hash, fullOutcomeProof.outcome_proof.proof);
897
898      hash = sha256(abi.encodePacked(hash));
899
900      hash = _computeRoot(hash, fullOutcomeProof.outcome_root_proof);
901
902      require(
903          hash == fullOutcomeProof.block_header_lite.inner_lite.outcome_root,
904          "NearProver: outcome merkle proof is not valid"
905      );
906
907      bytes32 expectedBlockMerkleRoot = bridge.blockMerkleRoots(blockHeight);
908
909      require(
910          _computeRoot(fullOutcomeProof.block_header_lite.hash, fullOutcomeProof.block_proof) ==
911              expectedBlockMerkleRoot,
912          "NearProver: block proof is not valid"
913      );
914
915      return true;
916  }
```

Figure 4. again.

I already described how to pass the requirement on line 910 by calculating the expected block merkle root and... well, setting **expectedBlockMerkleRoot** to this value.

This is not a tutorial but if you are unfamiliar with the topic (as was I during this research) then I highly encourage you to read⁶ about⁷ it⁸, as [Merkle Trees](#) are absolutely fascinating! and in the following section I show how I quickly hacked a simple tree, a valid one nonetheless, and explain why it's crucial for the exploit and an exercise for the reader.

TODO: describe merkle trees

⁶ High level: https://en.wikipedia.org/wiki/Merkle_tree

⁷ Deeper dive with visuals:

<https://ethereum.stackexchange.com/questions/6415/eli5-how-does-a-merkle-patricia-trie-tree-work>

⁸ Extreme deep dive: <https://medium.com/@chiqing/merkle-patricia-trie-explained-ae3ac6a7e123>

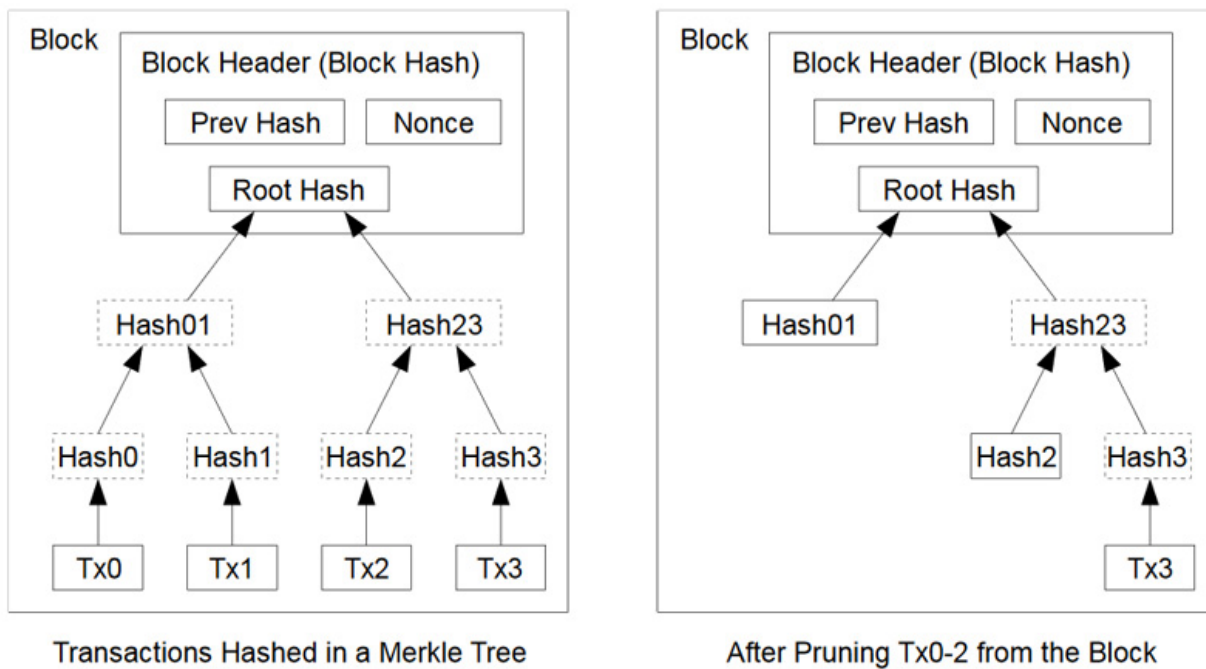


Figure 14

On the left a Merkle tree of height 3 is shown. On the right the Merkle proof for Tx3 is shown. In **addLightClientBlock**, the **untrustedMerkleRootHash** refers to the Root Hash shown in Figure 14.

When calling **unlockToken**, the **proofData** parameter includes the path to the respective transaction as shown on the right. That is, it includes the transaction information (Tx3 in the example shown) and the hashes inside solid squares - Hash2, Hash01 and Root Hash. Because Tx3 is known, Hash3 can be calculated. Because Hash2 is known, Hash23 can be calculated. Because Hash01 is known, the Root Hash can be calculated and finally compared to the **expectedBlockMerkleRoot**.

To quickly set up a Proof of Concept exploit, I hacked *around* the complexity of building entire Merkle trees. What I've done was to take the transaction, let's say Tx3 of our example - hashed it to get Hash3 and simply treated that as the Root Hash! I set that as the **untrustedMerkleRootHash** via **addLightClientBlock** and the requirement in Figure 4 line 910 is satisfied. This way, an actual valid block was presented - except it was never mined, never signed and never included in the blockchain!

The problem with this method is that only a single token can be extracted per each **addLightClientBlock**. The exploit is provided (TODO: insert git link here) and I leave it as an exercise for the reader to implement full Merkle trees and extract multiple tokens in a single exploit.

Epilog

...

Finally, I would like to explain why I think there are no other challengers lurking around. The problem with the current system is that, while there is some incentive to execute a challenge, there is no reason to assume such an opportunity will arise. The mere existence of the **challenge** method is a deterrent against rogue block producers. Assuming the main/official/centralized bot will one day run out of gas is not enough of an incentive to keep another challenger up & running. Of course the sum of 2.5 ETH is decent, but it is more likely than not that such a bot will never cover its operating costs.

This was an attempt to make a decentralized redundancy system, perhaps inspired by PoS or other DeFi opportunities such as liquidations, but without recurring income streams the incentives are essentially non-existent.

...

(TODO: recalc statistics over larger sample size & smaller sample from latest blocks)

TODO: further explain merkle roots! Cool analogy of unmined/out-of-chain block

TODO: explain about the bot more

TODO: explain further about insides of addLightClientBlock

And finish epilog!

Also, prolog?

TODO: show more of addLightClientBlock, refer to recipe?

```
168     function addLightClientBlock(bytes memory data) public override pausable(PAUSED_ADD_BLOCK) {
169         require(initialized, "Contract is not initialized");
170         require(balanceOf[msg.sender] >= lockEthAmount, "Balance is not enough");
171
172         Borsh.Data memory borsh = Borsh.from(data);
173         NearDecoder.LightClientBlock memory nearBlock = borsh.decodeLightClientBlock();
174         borsh.done();
175
176         unchecked {
177             // Commit the previous block, or make sure that it is OK to replace it.
178             if (block.timestamp < lastValidAt) {
179                 require(
180                     nearBlock.inner_lite.timestamp >= untrustedTimestamp + replaceDuration,
181                     "Can only replace with a sufficiently newer block"
182                 );
183             } else if (lastValidAt != 0) {
184                 curHeight = untrustedHeight;
185                 if (untrustedNextEpoch) {
186                     curEpoch = (curEpoch + 1) % 3;
187                 }
188                 lastValidAt = 0;
189
190                 blockHashes_[curHeight] = untrustedHash;
191                 blockMerkleRoots_[curHeight] = untrustedMerkleRoot;
192             }
193         }
194     }
```