Syscoin Bridge: How a Proof Parsing Flaw Minted 5 Billion SYS From Nothing

How a malformed proof minted 5 billion SYS from nothing, and why this was a parsing bug, not a stolen key.

On 7 June 2026, an attacker made the Syscoin bridge mint about **5 billion SYS** (roughly **568% of the whole circulating supply** at that moment) without burning a single real coin to back it. No private key was stolen. No signature was forged. The attacker simply handed the bridge a piece of data that looked like a valid proof, and the bridge believed it.

This post explains how a burn and mint bridge works, where the code that reads the proof lives, why a bug in that reader is enough to print money, and how the whole thing played out on chain. One thing I want to say up front: Syscoin and Halborn confirmed the kind of bug (a proof parsing flaw), but they never published the exact byte that broke. So I will show you the real, open source parsing code where the flaw lived, and I will clearly mark which parts are confirmed facts and which parts are my own reasoning.

Summary

- **Protocol:** Syscoin bridge (UTXO chain to NEVM), the `SyscoinRelay` and `SyscoinVaultManager` contracts. Every line of code I quote is pinned to the `syscoin/sysethereum-contracts` repo at commit `5405c13`.
- **Bug type:** A logic error in the SPV proof parsing. The bridge accepted a malformed (broken) proof as if it were valid.
- **Impact:** About 5,000,000,000 SYS minted out of nothing, worth around **$8.5M** at the 7 June closing price of $0.00171187. The new tokens were about **85% of the supply after the attack**.
- **Date:** 7 June 2026.
- **Status:** The bridge was paused within hours. A first postmortem went out the same day. A whitehat recovery address was posted on 9 June, and the funds were returned by 11 June. The proof validation path was fixed and reviewed.

Background: what a burn and mint bridge does

Syscoin has two sides:

- The **UTXO chain**, the Bitcoin style chain where SYS lives as unspent transaction outputs (UTXOs). It is the same model Bitcoin uses.
- The **NEVM** (Network Enhanced Virtual Machine), an EVM compatible smart contract chain, like the one you know from Ethereum.

A bridge lets you move SYS between these two sides. The rule that keeps it honest is simple, and it is the main idea behind this whole story:

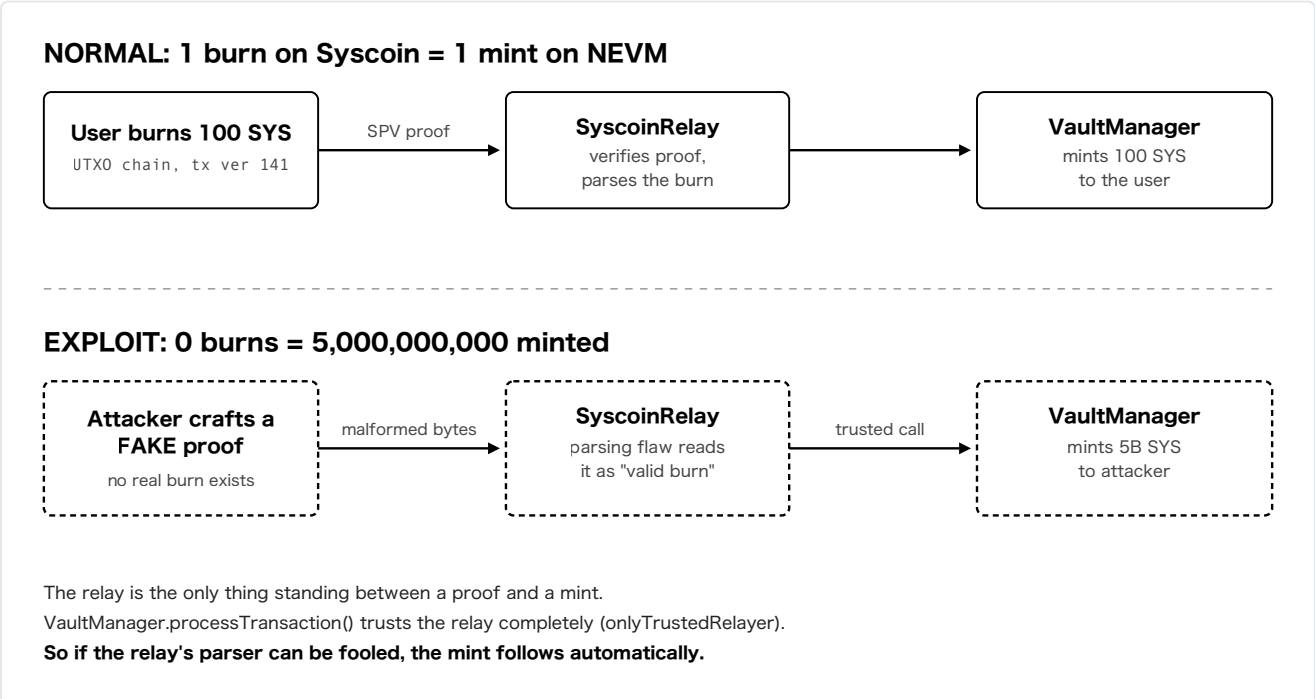
Every coin minted on one side must be backed by a coin **destroyed (burned)** on the other side. One burn in, one mint out. Never more.

When you move SYS from the UTXO chain to the NEVM chain, you make a special **burn transaction** on the UTXO side (Syscoin marks it with transaction version `141`). That burn says "I am destroying 100 SYS here, please release 100 SYS to my NEVM address."

But the NEVM contract cannot see the UTXO chain directly. So how does it know the burn really happened? It uses an **SPV proof**.

SPV stands for Simplified Payment Verification. Instead of giving the contract the whole blockchain, you give it a small proof: the raw burn transaction, the block header it lives in, and a short list of **Merkle siblings**. With those pieces, the contract can rebuild the Merkle root and check that this exact transaction really sits inside a real, mined block. If the math matches, the burn is real.

So the bridge's trust model is: **prove the burn, then mint.** The proof is the only gatekeeper. Hold that thought.



Top: how it should work, where one burn makes one mint. Bottom: the exploit, where a fake proof with no real burn behind it still made a mint, because the relay is the only check and its parser was fooled.

The vulnerable code

The gatekeeper is `SyscoinRelay.relayTx()`. Anyone can call it with a proof. Here is the real entry point (`SyscoinRelay.sol`, lines 277-312):

```
function relayTx(
    uint64 _blockNumber,
    bytes memory _txBytes,        // the raw burn transaction (attacker supplied)
    uint _txIndex,
    uint[] memory _txSiblings,    // Merkle siblings (attacker supplied)
    bytes memory _syscoinBlockHeader
) external override returns (uint) {
    uint txHash = verifySPVProofs(          // (1) is this tx really in a block?
        _blockNumber, _syscoinBlockHeader, _txBytes, _txIndex, _txSiblings
    );
    require(txHash != 0, "SPV proof verification failed");

    (
        uint errorCode,
        uint value,                         // (2) how much was burned?
        address destinationAddress,         //     who gets the minted coins?
        uint64 assetGuid
    ) = parseBurnTx(_txBytes);              // (3) read those fields out of raw bytes
    if (errorCode != 0) {
        emit RelayTransaction(bytes32(txHash), errorCode);
        return errorCode;
    }

    syscoinVaultManager.processTransaction( // (4) mint to the destination
        txHash, value, destinationAddress, assetGuid
    );
    return value;
}
```

Read the four numbered steps as a chain of trust:

1. `verifySPVProofs()` checks that the block header is real and that the transaction sits under the block's Merkle root.
2. Then `parseBurnTx()` walks the raw transaction bytes and pulls out the amount, the destination address, and the asset.
3. `processTransaction()` on the vault then mints that amount to that address.

Now look at what the minting side does (`SyscoinVaultManager.sol`, lines 154-166):

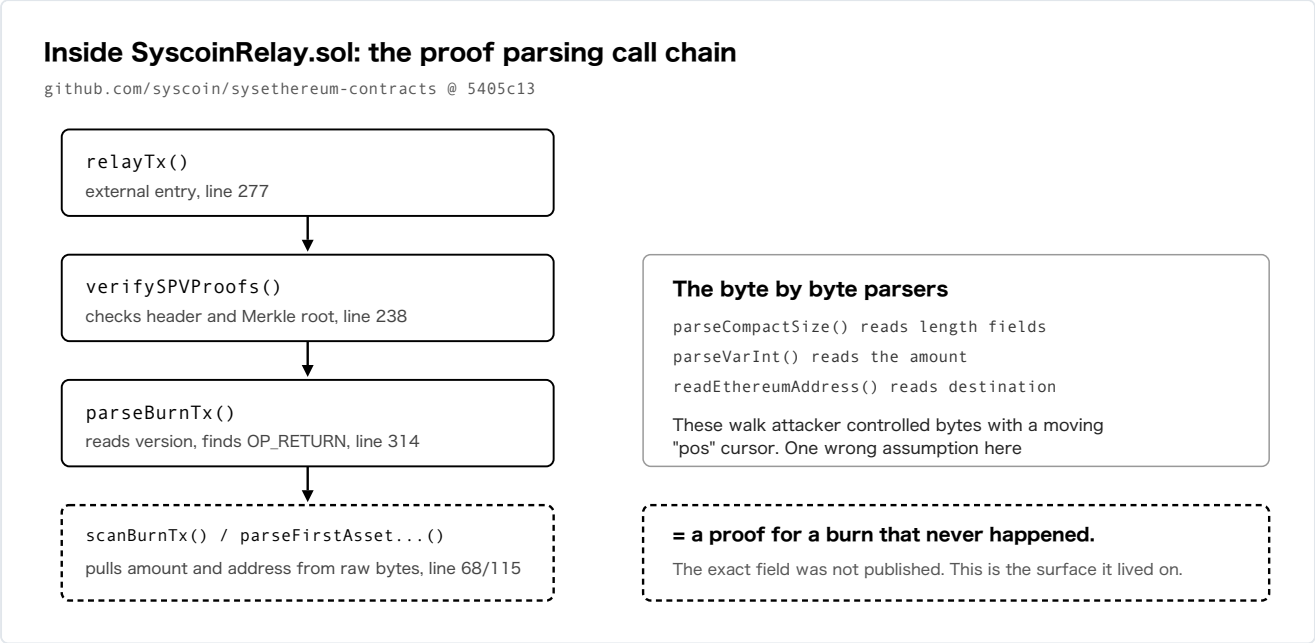
```
function processTransaction(
    uint txHash,
    uint value,
    address destination,
    uint64 assetGuid
) external override onlyTrustedRelayer nonReentrant whenNotPaused {
    require(_insert(txHash), "TX already processed");

    if (assetGuid == SYSAssetGuid) {
        require(value > 0, "Value must be positive");
        uint mintedAmount = scaleFromSatoshi(value, 18);
        _withdrawSYS(mintedAmount, payable(destination));   // the mint happens here
    }
    ...
}
```

Notice the `onlyTrustedRelayer` modifier. The vault does **no checking of its own** about whether the burn was real. It trusts the relay's word completely. That is by design, because the relay is the part that is supposed to prove the burn. But it also means the whole security of the mint rests on one question: **can the relay's proof reader be fooled?**

The reading happens in `parseBurnTx()`, which calls a small stack of byte walking helpers (`SyscoinRelay.sol`, lines 314-340):

```
function parseBurnTx(bytes memory txBytes)
    public pure
    returns (uint errorCode, uint output_value, address destinationAddress, uint64
assetGuid)
{
    uint32 version;
    uint pos = 0;
    uint opIndex = 0;
    version = bytesToUint32Flipped(txBytes, pos);
    if (version != SYSCOIN_TX_VERSION_ALLOCATION_BURN_TO_NEVM) {   // must be ver 141
        return (ERR_PARSE_TX_SYS, 0, address(0), 0);
    }
    (opIndex, pos) = getOpReturnPos(txBytes, 4);                   // find the OP_RETURN
    (output_value, destinationAddress, assetGuid) = scanBurnTx(    // read amount +
address
        txBytes, opIndex, pos
    );
    return (0, output_value, destinationAddress, assetGuid);
}
```

Every value here (the amount, the destination, the asset id) is read straight out of `txBytes`, which is data the **caller supplied**. A cursor named `pos` moves through the byte array. `parseCompactSize()` reads a length, `parseVarInt()` reads a number, `readEthereumAddress()` reads 20 bytes for the address. This is hand parsing of a raw Bitcoin style transaction, written in Solidity by hand.



The flaw lived somewhere on this surface: the chain of byte walking parsers that turn attacker controlled bytes into a "burn amount" and a "destination address". The exact field was never published. This is the code path it sat on.

Root Cause

Here is the **invariant** (the rule that must always hold):

A proof should be accepted only if the bytes inside it stand for a real burn transaction, and that transaction was really mined into a real block on the chain.

For that proof to count as truly valid, **two separate things** must be true.

First, the transaction must really be inside a real block on the chain. The **Merkle proof and block header check** confirm this.

Second, when the transaction bytes are parsed, the meaning that comes out must be the same meaning every node on the network agrees on. This is the **parsing check**.

These are **two different security guarantees**, and a bridge needs **both** to be safe.

Syscoin's relay got the first one right. `verifySPVProofs()` rebuilds the block hash from a precompile and rebuilds the Merkle root, so you really cannot point it at a transaction that was not in a block. As Syscoin and Halborn confirmed, the flaw was in the **proof validation parsing logic of the bridge relay**. In other words, the problem was in the **second guarantee (the parsing check)**.

rekt.news put it this way:

"The attacker didn't forge a valid proof. They forged something the parsing code would read as valid proof."

In plain words, the attacker built a blob of bytes. When the relay's hand written parser read those bytes, it thought "Burn Amount = 5,000,000,000 SYS, Destination = Attacker Address." But in reality the burn amount was 0, and no such burn ever happened. The symptom was 5 billion SYS showing up on NEVM. The cause was that the parser's idea of "what these bytes mean" did not match reality, and nothing after it checked again.

This is the classic shape of a bridge bug. It is the same family as the **Nomad** hack (2022), where a bad setup made the bridge treat any message as already proven, and the **BNB Chain** bridge hack (2022), where the attacker built a forged IAVL Merkle proof and the bridge accepted it as valid. Different chains, different code, same root cause: the proof verifier trusted something it should have rebuilt and checked itself.

Let me also be clear about what Syscoin did **not** get wrong. The relay already defends against the well known **64 byte transaction Merkle attack**. Inside `verifyTx()` there is logic like this:

```
// inside verifyTx()
if (tx.length == 64) {
    revert();
}
```

So a transaction whose length is exactly 64 bytes gets rejected (line 224). That means the flaw was not the 64 byte attack. It was somewhere on the transaction body parsing surface shown above. Because the exact field was never disclosed, I am not going to guess at a specific byte or line and present it as fact. Doing that would mean inventing a "which line had the bug" answer that the public record does not support.

Exploit walkthrough

Here is how it played out on chain. Every hash below comes from the public postmortem and you can check it yourself.

1. The mint. The attacker sent their crafted proof to the bridge. The relay accepted it, called `processTransaction`, and about 5 billion SYS was minted on the UTXO side to an address the attacker controlled.

- Mint tx: `a5b422abbbd89c8e316d1990f696e030d610cb527001ff97524f5317e87fa184`
- Attacker address: `sys1qgaelv690g7wwp2xchfdh0enf5uewzq5sm9wvcw`

2. The transfer. The new minted coins were moved out of the minting address.

- Transfer tx: `ba6798fac98eaf95f18e4622a6d46b5d8547f75d3912ed3665ee2e12537d5ff4`

3. The split. The attacker split the haul into many outputs. This is a common move to make freezing and tracing harder.

- Split tx: `31e12b0dcd9aeffa12e596e0b16d75ce161667104c7e511bfafe67195117113c`
- The two largest tainted balances ended up at:
  - `sys1q2k482wnachkgky4lw60973p4vcf7xlh9kzpv33` (about 4 billion SYS)
  - `sys1qx6jjkq89sdaxftfgre3m0nv7vjfd4jeakg5t38` (about 1 billion SYS)

There was **no flash loan and no setup money** here. That is what makes a proof validation bug so dangerous compared to a price manipulation exploit: the attacker did not need to borrow anything or move a market. The whole "setup" happened off chain, which means building the malformed proof bytes. Once the proof was accepted, the profit was the mint itself.

About a proof of concept: I am not going to publish one. The reason is that the real weapon here is the exact method for building the malformed proof. Responsible disclosure means not shipping a copy paste exploit, even for a bridge whose details are now all public. What is safe to show is the call shape. It is a single `relayTx(...)` call that carries:

- a block header
- a transaction body
- the Merkle siblings

The parser reads the transaction body and turns it into an amount and a destination address that the attacker chose. That is the basic structure of the exploit. But the exact bytes that fool the parser and slip past validation are the dangerous part, so they stay unpublished.

Impact

- **Funds at risk / minted:** about 5,000,000,000 SYS, around **$8.5M** at mint time.
- **Supply shock:** that mint inflated the circulating supply by about **568%**, and made up about **85% of the supply after the attack**. SYS dropped around **20%** on the news.
- **Who was affected:** every SYS holder, through dilution and the price drop, plus any exchange or pool that might have taken in the tainted coins.
- **Preconditions:** the attacker only needed to call the public `relayTx` entry point with a crafted proof. No special access, no key, no money.
- **Still exploitable now:** no. The bridge was paused within hours, and the validation path was fixed before the bridge reopened.

The damage stopped at "minted but mostly contained" instead of "minted and cashed out" because the response was fast. Pausing the bridge and working with exchanges kept most of the tainted SYS from reaching open markets before it was returned.

The Fix

Syscoin has not published a line by line diff, so I will not make one up. From the public statements, the fix had two parts:

1. **Pause first.** `processTransaction` runs under the `whenNotPaused` modifier, so pausing the bridge instantly stopped any more mints while the team worked. This is the bridge version of pulling the emergency brake.
2. **Repair the validation path.** The team fixed and reviewed the relay's proof validation logic (the parsing surface described above), so a malformed proof can no longer be read as a valid burn.

For this kind of bug, a correct fix is strict, complete parsing: the verifier must reject any transaction whose bytes do not parse into exactly one clear meaning, check the bounds on every length and offset, and rebuild every value itself instead of trusting the layout the caller implied. The idea, in before and after form, looks like this:

```
- // accept the proof if the Merkle root matches, then read fields loosely
- (value, dest, guid) = scanBurnTx(txBytes, opIndex, pos);
- processTransaction(txHash, value, dest, guid);

+ // accept ONLY if the bytes parse into exactly one canonical burn,
+ // with strict bounds on every field and no leftover or unclear bytes
+ require(isCanonicalBurnTx(txBytes), "not canonical tx");
+ (value, dest, guid) = scanBurnTxStrict(txBytes, opIndex, pos);
+ processTransaction(txHash, value, dest, guid);
```

That fixes the root cause (unclear parsing) instead of the symptom (one specific bad input). Patching only the exact byte the attacker used would be a symptom fix. The right move is to make the parser reject everything that is not clearly a real burn.

Takeaways

If you build, audit, or use bridges, these are the patterns worth remembering:

- **A proof verifier is only as strong as its parser.** Checking the Merkle root again is not enough if the step that reads what the transaction means can be fooled. Both the "is it in a block" check and the "what does it say" check have to be solid. Bridges keep dying on the second one: Nomad, BNB Chain, and now Syscoin.
- **Parsing raw bytes by hand in Solidity is a very risky surface.** Walking through transaction bytes with a moving cursor, reading fields of unknown length, trusting offsets, this is exactly where edge cases hide. Treat every such parser as input from an

attacker, and check every boundary very carefully. It is better to use strict parsers that reject anything unclear.

- **One unchecked trust boundary undoes every other control.** `SyscoinVaultManager` was careful in other ways (a reentrancy guard, a pause switch, per asset checks), but it trusted the relay completely. When the relay was wrong, all that careful code minted 5 billion coins on command. A weak link upstream makes strong code downstream pointless.
- **Audit the boring relay path too, not just the contracts.** The postmortem record shows the wallet, governance, and other parts had third party audits, but the bridge relay's proof path did not. The least exciting code held the most dangerous power.
- **Pause switches and working with exchanges are real security.** They did not stop the mint, but they are why this is a "funds returned" story instead of a "funds gone for good" story. Incident response is part of security, not something to add later.

Disclosure Timeline

Date	Event
7 June 2026	Exploit run; about 5B SYS minted. Bridge paused within hours.
7 June 2026	First postmortem published the same evening, with transaction hashes and tainted addresses.
9 June 2026	Public whitehat recovery address posted; a bounty offered instead of legal threats.
11 June 2026	Funds confirmed returned by the attacker.
Following days	The proof validation path fixed and reviewed before the bridge reopened.

Sources

- rekt.news, Syscoin Rekt (main writeup, transaction hashes, addresses)
- Halborn, Explained: The Syscoin Bridge Hack (June 2026)
- BeInCrypto, Syscoin pauses bridge after 5 billion SYS exploit
- CryptoPotato, SYS drops 20% after 5B unauthorized tokens minted
- Syscoin bridge contracts source (`sysethereum-contracts`)
- Syscoin Bridge Dapp repository
- Syscoin documentation
- Syscoin postmortem (X/Twitter) and recovery confirmation

- Background on the same bug family: Nomad Bridge hack (2022), BNB Chain bridge hack (2022)

Written by Khant Wai Yan Aung