Introduction

This handbook is a guide to ordinal theory. Ordinal theory concerns itself with satoshis, giving them individual identities and allowing them to be tracked, transferred, and imbued with meaning.

Satoshis, not bitcoin, are the atomic, native currency of the Bitcoin network. One bitcoin can be sub-divided into 100,000,000 satoshis, but no further.

Ordinal theory does not require a sidechain or token aside from Bitcoin, and can be used without any changes to the Bitcoin network. It works right now.

Ordinal theory imbues satoshis with numismatic value, allowing them to be collected and traded as curios.

Individual satoshis can be inscribed with arbitrary content, creating unique Bitcoin-native digital artifacts that can be held in Bitcoin wallets and transferred using Bitcoin transactions. Inscriptions are as durable, immutable, secure, and decentralized as Bitcoin itself.

Other, more unusual use-cases are possible: off-chain colored-coins, public key infrastructure with key rotation, a decentralized replacement for the DNS. For now though, such use-cases are speculative, and exist only in the minds of fringe ordinal theorists.

For more details on ordinal theory, see the overview.

For more details on inscriptions, see inscriptions.

When you're ready to get your hands dirty, a good place to start is with inscriptions, a curious species of digital artifact enabled by ordinal theory.

Links

- GitHub
- BIP
- Discord
- Open Ordinals Institute Website
- Open Ordinals Institute X
- Mainnet Block Explorer
- Signet Block Explorer

Videos

- Ordinal Theory Explained: Satoshi Serial Numbers and NFTs on Bitcoin
- Ordinals Workshop with Rodarmor

Ordinal Theory Overview

Ordinals are a numbering scheme for satoshis that allows tracking and transferring individual sats. These numbers are called ordinal numbers. Satoshis are numbered in the order in which they're mined, and transferred from transaction inputs to transaction outputs first-in-first-out. Both the numbering scheme and the transfer scheme rely on *order*, the numbering scheme on the *order* in which satoshis are mined, and the transfer scheme on the *order* of transaction inputs and outputs. Thus the name, *ordinals*.

Technical details are available in the BIP.

Ordinal theory does not require a separate token, another blockchain, or any changes to Bitcoin. It works right now.

Ordinal numbers have a few different representations:

- *Integer notation*: `2099994106992659` The ordinal number, assigned according to the order in which the satoshi was mined.
- *Decimal notation*: `3891094.16797` The first number is the block height in which the satoshi was mined, the second the offset of the satoshi within the block.
- *Degree notation*: `3°111094′214″16797‴`. We'll get to that in a moment.
- *Percentile notation*: `99.99971949060254%` . The satoshi's position in Bitcoin's supply, expressed as a percentage.
- *Name*: `satoshi` . An encoding of the ordinal number using the characters `a` through `z` .

Arbitrary assets, such as NFTs, security tokens, accounts, or stablecoins can be attached to satoshis using ordinal numbers as stable identifiers.

Ordinals is an open-source project, developed on GitHub. The project consists of a BIP describing the ordinal scheme, an index that communicates with a Bitcoin Core node to track the location of all satoshis, a wallet that allows making ordinal-aware transactions, a block explorer for interactive exploration of the blockchain, functionality for inscribing satoshis with digital artifacts, and this manual.

Rarity

Humans are collectors, and since satoshis can now be tracked and transferred, people will naturally want to collect them. Ordinal theorists can decide for themselves which sats are rare and desirable, but there are some hints…

Bitcoin has periodic events, some frequent, some more uncommon, and these naturally lend themselves to a system of rarity. These periodic events are:

- *Blocks*: A new block is mined approximately every 10 minutes, from now until the end of time.
- *Difficulty adjustments*: Every 2016 blocks, or approximately every two weeks, the Bitcoin network responds to changes in hashrate by adjusting the difficulty target which blocks must meet in order to be accepted.
- *Halvings*: Every 210,000 blocks, or roughly every four years, the amount of new sats created in every block is cut in half.
- *Cycles*: Every six halvings, something magical happens: the halving and the difficulty adjustment coincide. This is called a conjunction, and the time period between conjunctions a cycle. A conjunction occurs roughly every 24 years. The first conjunction should happen sometime in 2032.

This gives us the following rarity levels:

- `common` : Any sat that is not the first sat of its block
- `uncommon` : The first sat of each block
- `rare` : The first sat of each difficulty adjustment period
- `epic` : The first sat of each halving epoch
- `legendary` : The first sat of each cycle
- `mythic` : The first sat of the genesis block

Which brings us to degree notation, which unambiguously represents an ordinal number in a way that makes the rarity of a satoshi easy to see at a glance:

```
A°B′C″D‴
│ │ │ ╰─ Index of sat in the block
│ │ ╰─── Index of block in difficulty adjustment period
│ ╰───── Index of block in halving epoch
╰─────── Cycle, numbered starting from 0
```

Ordinal theorists often use the terms "hour", "minute", "second", and "third" for *A*, *B*, *C*, and *D*, respectively.

Now for some examples. This satoshi is common:

```
1°1′1″1‴
│ │ │ ╰─ Not first sat in block
│ │ ╰─── Not first block in difficulty adjustment period
│ ╰───── Not first block in halving epoch
╰─────── Second cycle
```

This satoshi is uncommon:

```
1°1′1″0‴
│ │ │ └── First sat in block
│ │ └──── Not first block in difficulty adjustment period
│ └────── Not first block in halving epoch
└──────── Second cycle
```

This satoshi is rare:

```
1°1′0″0‴
│ │ │ └── First sat in block
│ │ └──── First block in difficulty adjustment period
│ └────── Not the first block in halving epoch
└──────── Second cycle
```

This satoshi is epic:

```
1°0′1″0‴
│ │ │ └── First sat in block
│ │ └──── Not first block in difficulty adjustment period
│ └────── First block in halving epoch
└──────── Second cycle
```

This satoshi is legendary:

```
1°0′0″0‴
│ │ │ └── First sat in block
│ │ └──── First block in difficulty adjustment period
│ └────── First block in halving epoch
└──────── Second cycle
```

And this satoshi is mythic:

```
0°0′0″0‴
│ │ │ └── First sat in block
│ │ └──── First block in difficulty adjustment period
│ └────── First block in halving epoch
└──────── First cycle
```

If the block offset is zero, it may be omitted. This is the uncommon satoshi from above:

```
1°1′1″
│ │ └── Not first block in difficulty adjustment period
│ └──── Not first block in halving epoch
└────── Second cycle
```

Rare Satoshi Supply

Total Supply

- `common` : 2.1 quadrillion
- `uncommon` : 6,929,999
- `rare` : 3437
- `epic` : 32
- `legendary` : 5
- `mythic` : 1

Current Supply

- `common` : 1.9 quadrillion
- `uncommon` : 808,262
- `rare` : 369
- `epic` : 3
- `legendary` : 0
- `mythic` : 1

At the moment, even uncommon satoshis are quite rare. As of this writing, 745,855 uncommon satoshis have been mined - one per 25.6 bitcoin in circulation.

Names

Each satoshi has a name, consisting of the letters *A* through *Z*, that get shorter the further into the future the satoshi was mined. They could start short and get longer, but then all the good, short names would be trapped in the unspendable genesis block.

As an example, 1905530482684727°'s name is "iaiufjszmoba". The name of the last satoshi to be mined is "a". Every combination of 10 characters or less is out there, or will be out there, someday.

Exotics

Satoshis may be prized for reasons other than their name or rarity. This might be due to a quality of the number itself, like having an integer square or cube root. Or it might be due to a connection to a historical event, such as satoshis from block 477,120, the block in which SegWit activated, or 2099999997689999°, the last satoshi that will ever be mined.

Such satoshis are termed "exotic". Which satoshis are exotic and what makes them so is subjective. Ordinal theorists are encouraged to seek out exotics based on criteria of their own devising.

Inscriptions

Satoshis can be inscribed with arbitrary content, creating Bitcoin-native digital artifacts. Inscribing is done by sending the satoshi to be inscribed in a transaction that reveals the inscription content on-chain. This content is then inextricably linked to that satoshi, turning it into an immutable digital artifact that can be tracked, transferred, hoarded, bought, sold, lost, and rediscovered.

Archaeology

A lively community of archaeologists devoted to cataloging and collecting early NFTs has sprung up. Here's a great summary of historical NFTs by Chainleft.

A commonly accepted cut-off for early NFTs is March 19th, 2018, the date the first ERC-721 contract, SU SQUARES, was deployed on Ethereum.

Whether or not ordinals are of interest to NFT archaeologists is an open question! In one sense, ordinals were created in early 2022, when the Ordinals specification was finalized. In this sense, they are not of historical interest.

In another sense though, ordinals were in fact created by Satoshi Nakamoto in 2009 when he mined the Bitcoin genesis block. In this sense, ordinals, and especially early ordinals, are certainly of historical interest.

Many ordinal theorists favor the latter view. This is not least because the ordinals were independently discovered on at least two separate occasions, long before the era of modern NFTs began.

On August 21st, 2012, Charlie Lee posted a proposal to add proof-of-stake to Bitcoin to the Bitcoin Talk forum. This wasn't an asset scheme, but did use the ordinal algorithm, and was implemented but never deployed.

On October 8th, 2012, jl2012 posted a scheme to the same forum which uses decimal notation and has all the important properties of ordinals. The scheme was discussed but never implemented.

These independent inventions of ordinals indicate in some way that ordinals were discovered, or rediscovered, and not invented. The ordinals are an inevitability of the mathematics of Bitcoin, stemming not from their modern documentation, but from their ancient genesis. They are the culmination of a sequence of events set in motion with the mining of the first block, so many years ago.

Digital Artifacts

Imagine a physical artifact. A rare coin, say, held safe for untold years in the dark, secret clutch of a Viking hoard, now dug from the earth by your grasping hands. It…

…has an owner. You. As long as you keep it safe, nobody can take it from you.

…is complete. It has no missing parts.

…can only be changed by you. If you were a trader, and you made your way to 18th century China, none but you could stamp it with your chop-mark.

…can only be disposed of by you. The sale, trade, or gift is yours to make, to whomever you wish.

What are digital artifacts? Simply put, they are the digital equivalent of physical artifacts.

For a digital thing to be a digital artifact, it must be like that coin of yours:

- Digital artifacts can have owners. A number is not a digital artifact, because nobody can own it.
- Digital artifacts are complete. An NFT that points to off-chain content on IPFS or Arweave is incomplete, and thus not a digital artifact.
- Digital artifacts are permissionless. An NFT which cannot be sold without paying a royalty is not permissionless, and thus not a digital artifact.
- Digital artifacts are uncensorable. Perhaps you can change a database entry on a centralized ledger today, but maybe not tomorrow, and thus one cannot be a digital artifact.
- Digital artifacts are immutable. An NFT with an upgrade key is not a digital artifact.

The definition of a digital artifact is intended to reflect what NFTs *should* be, sometimes are, and what inscriptions *always* are, by their very nature.

Inscriptions

Inscriptions inscribe sats with arbitrary content, creating bitcoin-native digital artifacts, more commonly known as NFTs. Inscriptions do not require a sidechain or separate token.

These inscribed sats can then be transferred using bitcoin transactions, sent to bitcoin addresses, and held in bitcoin UTXOs. These transactions, addresses, and UTXOs are normal bitcoin transactions, addresses, and UTXOS in all respects, with the exception that in order to send individual sats, transactions must control the order and value of inputs and outputs according to ordinal theory.

The inscription content model is that of the web. An inscription consists of a content type, also known as a MIME type, and the content itself, which is a byte string. This allows inscription content to be returned from a web server, and for creating HTML inscriptions that use and remix the content of other inscriptions.

Inscription content is entirely on-chain, stored in taproot script-path spend scripts. Taproot scripts have very few restrictions on their content, and additionally receive the witness discount, making inscription content storage relatively economical.

Since taproot script spends can only be made from existing taproot outputs, inscriptions are made using a two-phase commit/reveal procedure. First, in the commit transaction, a taproot output committing to a script containing the inscription content is created. Second, in the reveal transaction, the output created by the commit transaction is spent, revealing the inscription content on-chain.

Inscription content is serialized using data pushes within unexecuted conditionals, called "envelopes". Envelopes consist of an `OP_FALSE OP_IF … OP_ENDIF` wrapping any number of data pushes. Because envelopes are effectively no-ops, they do not change the semantics of the script in which they are included, and can be combined with any other locking script.

A text inscription containing the string "Hello, world!" is serialized as follows:

```
OP_FALSE
OP_IF
  OP_PUSH "ord"
  OP_PUSH 1
  OP_PUSH "text/plain;charset=utf-8"
  OP_PUSH 0
  OP_PUSH "Hello, world!"
OP_ENDIF
```

First the string `ord` is pushed, to disambiguate inscriptions from other uses of envelopes.

`OP_PUSH 1` indicates that the next push contains the content type, and `OP_PUSH 0` indicates that subsequent data pushes contain the content itself. Multiple data pushes must be used for large inscriptions, as one of taproot's few restrictions is that individual data pushes may not be larger than 520 bytes.

The inscription content is contained within the input of a reveal transaction, and the inscription is made on the first sat of its input. This sat can then be tracked using the familiar rules of ordinal theory, allowing it to be transferred, bought, sold, lost to fees, and recovered.

Content

The data model of inscriptions is that of a HTTP response, allowing inscription content to be served by a web server and viewed in a web browser.

Fields

Inscriptions may include fields before an optional body. Each field consists of two data pushes, a tag and a value.

Currently, there are six defined fields:

- `content_type`, with a tag of `1`, whose value is the MIME type of the body.
- `pointer`, with a tag of `2`, see pointer docs.
- `parent`, with a tag of `3`, see provenance.
- `metadata`, with a tag of `5`, see metadata.
- `metaprotocol`, with a tag of `7`, whose value is the metaprotocol identifier.
- `content_encoding`, with a tag of `9`, whose value is the encoding of the body.
- `delegate`, with a tag of `11`, see delegate.

The beginning of the body and end of fields is indicated with an empty data push.

Unrecognized tags are interpreted differently depending on whether they are even or odd, following the "it's okay to be odd" rule used by the Lightning Network.

Even tags are used for fields which may affect creation, initial assignment, or transfer of an inscription. Thus, inscriptions with unrecognized even fields must be displayed as "unbound", that is, without a location.

Odd tags are used for fields which do not affect creation, initial assignment, or transfer, such as additional metadata, and thus are safe to ignore.

Inscription IDs

The inscriptions are contained within the inputs of a reveal transaction. In order to uniquely identify them they are assigned an ID of the form:

```
521f8eccffa4c41a3a7728dd012ea5a4a02feed81f41159231251ecf1e5c79dai0
```

The part in front of the `i` is the transaction ID (`txid`) of the reveal transaction. The number after the `i` defines the index (starting at 0) of new inscriptions being inscribed in the reveal transaction.

Inscriptions can either be located in different inputs, within the same input or a combination of both. In any case the ordering is clear, since a parser would go through the inputs consecutively and look for all inscription `envelopes` .

Input	Inscription Count	Indices
0	2	i0, i1
1	1	i2
2	3	i3, i4, i5
3	0	
4	1	i6

Inscription Numbers

Inscriptions are assigned inscription numbers starting at zero, first by the order reveal transactions appear in blocks, and the order that reveal envelopes appear in those transactions.

Due to a historical bug in `ord` which cannot be fixed without changing a great many inscription numbers, inscriptions which are revealed and then immediately spent to fees are numbered as if they appear last in the block in which they are revealed.

Inscriptions which are cursed are numbered starting at negative one, counting down. Cursed inscriptions on and after the jubilee at block 824544 are vindicated, and are assigned positive inscription numbers.

Sandboxing

HTML and SVG inscriptions are sandboxed in order to prevent references to off-chain content, thus keeping inscriptions immutable and self-contained.

This is accomplished by loading HTML and SVG inscriptions inside `iframes` with the `sandbox` attribute, as well as serving inscription content with `Content-Security-Policy` headers.

Reinscriptions

Previously inscribed sats can be reinscribed with the `--reinscribe` command if the inscription is present in the wallet. This will only append an inscription to a sat, not change the initial inscription.

Reinscribe with satpoint: `ord wallet inscribe --fee-rate <FEE_RATE> --reinscribe --file <FILE> --satpoint <SATPOINT>`

Reinscribe on a sat (requires sat index): `ord --index-sats wallet inscribe --fee-rate <FEE_RATE> --reinscribe --file <FILE> --sat <SAT>`

Delegate

Inscriptions may nominate a delegate inscription. Requests for the content of an inscription with a delegate will instead return the content and content type of the delegate. This can be used to cheaply create copies of an inscription.

Specification

To create an inscription I with delegate inscription D:

- Create an inscription D. Note that inscription D does not have to exist when making inscription I. It may be inscribed later. Before inscription D is inscribed, requests for the content of inscription I will return a 404.
- Include tag `11`, i.e. `OP_PUSH 11`, in I, with the value of the serialized binary inscription ID of D, serialized as the 32-byte `TXID`, followed by the four-byte little-endian `INDEX`, with trailing zeroes omitted.

NB The bytes of a bitcoin transaction ID are reversed in their text representation, so the serialized transaction ID will be in the opposite order.

Example

An example of an inscription which delegates to `000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1fi0`:

```
OP_FALSE
OP_IF
  OP_PUSH "ord"
  OP_PUSH 11
  OP_PUSH 0x1f1e1d1c1b1a191817161514131211100f0e0d0c0b0a09080706050403020100
OP_ENDIF
```

Note that the value of tag `11` is decimal, not hex.

The delegate field value uses the same encoding as the parent field. See provenance for more examples of inscrpition ID encodings;

Metadata

Inscriptions may include CBOR metadata, stored as data pushes in fields with tag `5`. Since data pushes are limited to 520 bytes, metadata longer than 520 bytes must be split into multiple tag `5` fields, which will then be concatenated before decoding.

Metadata is human readable, and all metadata will be displayed to the user with its inscription. Inscribers are encouraged to consider how metadata will be displayed, and make metadata concise and attractive.

Metadata is rendered to HTML for display as follows:

- `null`, `true`, `false`, numbers, floats, and strings are rendered as plain text.
- Byte strings are rendered as uppercase hexadecimal.
- Arrays are rendered as `<ul>` tags, with every element wrapped in `<li>` tags.
- Maps are rendered as `<dl>` tags, with every key wrapped in `<dt>` tags, and every value wrapped in `<dd>` tags.
- Tags are rendered as the tag , enclosed in a `<sup>` tag, followed by the value.

CBOR is a complex spec with many different data types, and multiple ways of representing the same data. Exotic data types, such as tags, floats, and bignums, and encoding such as indefinite values, may fail to display correctly or at all. Contributions to `ord` to remedy this are welcome.

Example

Since CBOR is not human readable, in these examples it is represented as JSON. Keep in mind that this is *only* for these examples, and JSON metadata will *not* be displayed correctly.

The metadata `{"foo":"bar","baz":[null,true,false,0]}` would be included in an inscription as:

```
OP_FALSE
OP_IF
    ...
    OP_PUSH 0x05 OP_PUSH '{"foo":"bar","baz":[null,true,false,0]}'
    ...
OP_ENDIF
```

And rendered as:

```
<dl>
  ...
  <dt>metadata</dt>
  <dd>
    <dl>
      <dt>foo</dt>
      <dd>bar</dd>
      <dt>baz</dt>
      <dd>
        <ul>
          <li>null</li>
          <li>true</li>
          <li>false</li>
          <li>0</li>
        </ul>
      </dd>
    </dl>
  </dd>
  ...
</dl>
```

Metadata longer than 520 bytes must be split into multiple fields:

```
OP_FALSE
OP_IF
    ...
    OP_PUSH 0x05 OP_PUSH '{"very":"long","metadata":'
    OP_PUSH 0x05 OP_PUSH '"is","finally":"done"}'
    ...
OP_ENDIF
```

Which would then be concatenated into `{"very":"long","metadata":"is","finally":"done"}`.

Pointer

In order to make an inscription on a sat other than the first of its input, a zero-based integer, called the "pointer", can be provided with tag `2`, causing the inscription to be made on the sat at the given position in the outputs. If the pointer is equal to or greater than the number of total sats in the outputs of the inscribe transaction, it is ignored, and the inscription is made as usual. The value of the pointer field is a little endian integer, with trailing zeroes ignored.

An even tag is used, so that old versions of `ord` consider the inscription to be unbound, instead of assigning it, incorrectly, to the first sat.

This can be used to create multiple inscriptions in a single transaction on different sats, when otherwise they would be made on the same sat.

Examples

An inscription with pointer 255:

```
OP_FALSE
OP_IF
  OP_PUSH "ord"
  OP_PUSH 1
  OP_PUSH "text/plain;charset=utf-8"
  OP_PUSH 2
  OP_PUSH 0xff
  OP_PUSH 0
  OP_PUSH "Hello, world!"
OP_ENDIF
```

An inscription with pointer 256:

```
OP_FALSE
OP_IF
  OP_PUSH "ord"
  OP_PUSH 1
  OP_PUSH "text/plain;charset=utf-8"
  OP_PUSH 2
  OP_PUSH 0x0001
  OP_PUSH 0
  OP_PUSH "Hello, world!"
OP_ENDIF
```

An inscription with pointer 256, with trailing zeroes, which are ignored:

```
OP_FALSE
OP_IF
  OP_PUSH "ord"
  OP_PUSH 1
  OP_PUSH "text/plain;charset=utf-8"
  OP_PUSH 2
  OP_PUSH 0x000100
  OP_PUSH 0
  OP_PUSH "Hello, world!"
OP_ENDIF
```

Provenance

The owner of an inscription can create child inscriptions, trustlessly establishing the provenance of those children on-chain as having been created by the owner of the parent inscription. This can be used for collections, with the children of a parent inscription being members of the same collection.

Children can themselves have children, allowing for complex hierarchies. For example, an artist might create an inscription representing themselves, with sub inscriptions representing collections that they create, with the children of those sub inscriptions being items in those collections.

Specification

To create a child inscription C with parent inscription P:

- Create an inscribe transaction T as usual for C.
- Spend the parent P in one of the inputs of T.
- Include tag `3`, i.e. `OP_PUSH 3`, in C, with the value of the serialized binary inscription ID of P, serialized as the 32-byte `TXID`, followed by the four-byte little-endian `INDEX`, with trailing zeroes omitted.

NB The bytes of a bitcoin transaction ID are reversed in their text representation, so the serialized transaction ID will be in the opposite order.

Example

An example of a child inscription of `000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1fi0`:

```
OP_FALSE
OP_IF
  OP_PUSH "ord"
  OP_PUSH 1
  OP_PUSH "text/plain;charset=utf-8"
  OP_PUSH 3
  OP_PUSH 0x1f1e1d1c1b1a191817161514131211100f0e0d0c0b0a09080706050403020100
  OP_PUSH 0
  OP_PUSH "Hello, world!"
OP_ENDIF
```

Note that the value of tag `3` is binary, not hex, and that for the child inscription to be recognized as a child, `000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1fi0` must be spent as one of the inputs of the inscribe transaction.

Example encoding of inscription ID `000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1fi255`:

```
OP_FALSE
OP_IF
  …
  OP_PUSH 3
  OP_PUSH
0x1f1e1d1c1b1a191817161514131211100f0e0d0c0b0a09080706050403020100ff
  …
OP_ENDIF
```

And of inscription ID `000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1fi256`:

```
OP_FALSE
OP_IF
  …
  OP_PUSH 3
  OP_PUSH
0x1f1e1d1c1b1a191817161514131211100f0e0d0c0b0a090807060504030201000001
  …
OP_ENDIF
```

Notes

The tag `3` is used because it is the first available odd tag. Unrecognized odd tags do not make an inscription unbound, so child inscriptions would be recognized and tracked by old versions of `ord`.

A collection can be closed by burning the collection's parent inscription, which guarantees that no more items in the collection can be issued.

Recursion

An important exception to sandboxing is recursion: access to `ord`'s `/content` endpoint is permitted, allowing inscriptions to access the content of other inscriptions by requesting `/content/<INSCRIPTION_ID>`.

This has a number of interesting use-cases:

- Remixing the content of existing inscriptions.
- Publishing snippets of code, images, audio, or stylesheets as shared public resources.
- Generative art collections where an algorithm is inscribed as JavaScript, and instantiated from multiple inscriptions with unique seeds.
- Generative profile picture collections where accessories and attributes are inscribed as individual images, or in a shared texture atlas, and then combined, collage-style, in unique combinations in multiple inscriptions.

The recursive endpoints are:

- `/r/blockhash/<HEIGHT>`: block hash at given block height.
- `/r/blockhash`: latest block hash.
- `/r/blockheight`: latest block height.
- `/r/blocktime`: UNIX time stamp of latest block.
- `/r/children/<INSCRIPTION_ID>`: the first 100 child inscription ids.
- `/r/children/<INSCRIPTION_ID>/<PAGE>`: the set of 100 child inscription ids on `<PAGE>`.
- `/r/metadata/<INSCRIPTION_ID>`: JSON string containing the hex-encoded CBOR metadata.
- `/r/sat/<SAT_NUMBER>`: the first 100 inscription ids on a sat.
- `/r/sat/<SAT_NUMBER>/<PAGE>`: the set of 100 inscription ids on `<PAGE>`.
- `/r/sat/<SAT_NUMBER>/at/<INDEX>`: the inscription id at `<INDEX>` of all inscriptions on a sat. `<INDEX>` may be a negative number to index from the back. `0` being the first and `-1` being the most recent for example.

Note: `<SAT_NUMBER>` only allows the actual number of a sat no other sat notations like degree, percentile or decimal. We may expand to allow those in the future.

Responses from the above recursive endpoints are JSON. For backwards compatibility additional endpoints are supported, some of which return plain-text responses.

- `/blockheight`: latest block height.
- `/blockhash`: latest block hash.

- `/blockhash/<HEIGHT>` : block hash at given block height.
- `/blocktime` : UNIX time stamp of latest block.

Examples

- `/r/blockheight` :

```
777000
```

- `/r/blockhash/0` :

```
"000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f"
```

- `/r/blocktime` :

```
1700770905
```

- `/r/metadata/35b66389b44535861c44b2b18ed602997ee11db9a30d384ae89630c9fc6f011fi3` :

```
"a2657469746c65664d656d6f727966617574686f726e79656c6c6f775f6f72645f626f74"
```

- `/r/sat/1023795949035695` :

```
{
   "ids":[
      "17541f6adf6eb160d52bc6eb0a3546c7c1d2adfe607b1a3cddc72cc0619526adi0"
   ],
   "more":false,
   "page":0
}
```

- `/r/sat/1023795949035695/at/-1` :

```
{
   "id":"17541f6adf6eb160d52bc6eb0a3546c7c1d2adfe607b1a3cddc72cc0619526adi0"
}
```

- `/r/children/60bcf821240064a9c55225c4f01711b0ebbcab39aa3fafeefe4299ab158536fai0/49` :

```
{
   "ids":[

"7cd66b8e3a63dcd2fada917119830286bca0637267709d6df1ca78d98a1b4487i4900",

"7cd66b8e3a63dcd2fada917119830286bca0637267709d6df1ca78d98a1b4487i4901",
      ...

"7cd66b8e3a63dcd2fada917119830286bca0637267709d6df1ca78d98a1b4487i4935",
      "7cd66b8e3a63dcd2fada917119830286bca0637267709d6df1ca78d98a1b4487i4936"
   ],
   "more":false,
   "page":49
}
```

Ordinal Theory FAQ

What is ordinal theory?

Ordinal theory is a protocol for assigning serial numbers to satoshis, the smallest subdivision of a bitcoin, and tracking those satoshis as they are spent by transactions.

These serial numbers are large numbers, like this 804766073970493. Every satoshi, which is $^1/_{100000000}$ of a bitcoin, has an ordinal number.

Does ordinal theory require a side chain, a separate token, or changes to Bitcoin?

Nope! Ordinal theory works right now, without a side chain, and the only token needed is bitcoin itself.

What is ordinal theory good for?

Collecting, trading, and scheming. Ordinal theory assigns identities to individual satoshis, allowing them to be individually tracked and traded, as curios and for numismatic value.

Ordinal theory also enables inscriptions, a protocol for attaching arbitrary content to individual satoshis, turning them into bitcoin-native digital artifacts.

How does ordinal theory work?

Ordinal numbers are assigned to satoshis in the order in which they are mined. The first satoshi in the first block has ordinal number 0, the second has ordinal number 1, and the last satoshi of the first block has ordinal number 4,999,999,999.

Satoshis live in outputs, but transactions destroy outputs and create new ones, so ordinal theory uses an algorithm to determine how satoshis hop from the inputs of a transaction to its outputs.

Fortunately, that algorithm is very simple.

Satoshis transfer in first-in-first-out order. Think of the inputs to a transaction as being a list of satoshis, and the outputs as a list of slots, waiting to receive a satoshi. To assign input satoshis to slots, go through each satoshi in the inputs in order, and assign each to the first available slot in the outputs.

Let's imagine a transaction with three inputs and two outputs. The inputs are on the left of the arrow and the outputs are on the right, all labeled with their values:

```
[2] [1] [3] → [4] [2]
```

Now let's label the same transaction with the ordinal numbers of the satoshis that each input contains, and question marks for each output slot. Ordinal numbers are large, so let's use letters to represent them:

```
[a b] [c] [d e f] → [? ? ? ?] [? ?]
```

To figure out which satoshi goes to which output, go through the input satoshis in order and assign each to a question mark:

```
[a b] [c] [d e f] → [a b c d] [e f]
```

What about fees, you might ask? Good question! Let's imagine the same transaction, this time with a two satoshi fee. Transactions with fees send more satoshis in the inputs than are received by the outputs, so to make our transaction into one that pays fees, we'll remove the second output:

```
[2] [1] [3] → [4]
```

The satoshis *e* and *f* now have nowhere to go in the outputs:

```
[a b] [c] [d e f] → [a b c d]
```

So they go to the miner who mined the block as fees. The BIP has the details, but in short, fees paid by transactions are treated as extra inputs to the coinbase transaction, and are ordered how their corresponding transactions are ordered in the block. The coinbase transaction of the block might look like this:

```
[SUBSIDY] [e f] → [SUBSIDY e f]
```

Where can I find the nitty-gritty details?

The BIP!

Why are sat inscriptions called "digital artifacts" instead of "NFTs"?

An inscription is an NFT, but the term "digital artifact" is used instead, because it's simple, suggestive, and familiar.

The phrase "digital artifact" is highly suggestive, even to someone who has never heard the term before. In comparison, NFT is an acronym, and doesn't provide any indication of what it means if you haven't heard the term before.

Additionally, "NFT" feels like financial terminology, and the both word "fungible" and sense of the word "token" as used in "NFT" is uncommon outside of financial contexts.

How do sat inscriptions compare to…

Ethereum NFTs?

Inscriptions are always immutable.

There is simply no way to for the creator of an inscription, or the owner of an inscription, to modify it after it has been created.

Ethereum NFTs *can* be immutable, but many are not, and can be changed or deleted by the NFT contract owner.

In order to make sure that a particular Ethereum NFT is immutable, the contract code must be audited, which requires detailed knowledge of the EVM and Solidity semantics.

It is very hard for a non-technical user to determine whether or not a given Ethereum NFT is mutable or immutable, and Ethereum NFT platforms make no effort to distinguish whether an NFT is mutable or immutable, and whether the contract source code is available and has been audited.

Inscription content is always on-chain.

There is no way for an inscription to refer to off-chain content. This makes inscriptions more durable, because content cannot be lost, and scarcer, because inscription creators must pay fees proportional to the size of the content.

Some Ethereum NFT content is on-chain, but much is off-chain, and is stored on platforms like IPFS or Arweave, or on traditional, fully centralized web servers. Content on IPFS is not guaranteed to continue to be available, and some NFT content stored on IPFS has already been lost. Platforms like Arweave rely on weak economic assumptions, and will likely fail catastrophically when these economic assumptions are no longer met. Centralized web servers may disappear at any time.

It is very hard for a non-technical user to determine where the content of a given Ethereum NFT is stored.

Inscriptions are much simpler.

Ethereum NFTs depend on the Ethereum network and virtual machine, which are highly complex, constantly changing, and which introduce changes via backwards-incompatible hard forks.

Inscriptions, on the other hand, depend on the Bitcoin blockchain, which is relatively simple and conservative, and which introduces changes via backwards-compatible soft forks.

Inscriptions are more secure.

Inscriptions inherit Bitcoin's transaction model, which allow a user to see exactly which inscriptions are being transferred by a transaction before they sign it. Inscriptions can be offered for sale using partially signed transactions, which don't require allowing a third party, such as an exchange or marketplace, to transfer them on the user's behalf.

By comparison, Ethereum NFTs are plagued with end-user security vulnerabilities. It is commonplace to blind-sign transactions, grant third-party apps unlimited permissions over a user's NFTs, and interact with complex and unpredictable smart contracts. This creates a minefield of hazards for Ethereum NFT users which are simply not a concern for ordinal theorists.

Inscriptions are scarcer.

Inscriptions require bitcoin to mint, transfer, and store. This seems like a downside on the surface, but the raison d'etre of digital artifacts is to be scarce and thus valuable.

Ethereum NFTs, on the other hand, can be minted in virtually unlimited qualities with a single transaction, making them inherently less scarce, and thus, potentially less valuable.

Inscriptions do not pretend to support on-chain royalties.

On-chain royalties are a good idea in theory but not in practice. Royalty payment cannot be enforced on-chain without complex and invasive restrictions. The Ethereum NFT ecosystem is currently grappling with confusion around royalties, and is collectively coming to grips with the reality that on-chain royalties, which were messaged to artists as an advantage of NFTs, are not possible, while platforms race to the bottom and remove royalty support.

Inscriptions avoid this situation entirely by making no false promises of supporting royalties on-chain, thus avoiding the confusion, chaos, and negativity of the Ethereum

NFT situation.

Inscriptions unlock new markets.

Bitcoin's market capitalization and liquidity are greater than Ethereum's by a large margin. Much of this liquidity is not available to Ethereum NFTs, since many Bitcoiners prefer not to interact with the Ethereum ecosystem due to concerns related to simplicity, security, and decentralization.

Such Bitcoiners may be more interested in inscriptions than Ethereum NFTs, unlocking new classes of collector.

Inscriptions have a richer data model.

Inscriptions consist of a content type, also known as a MIME type, and content, which is an arbitrary byte string. This is the same data model used by the web, and allows inscription content to evolve with the web, and come to support any kind of content supported by web browsers, without requiring changes to the underlying protocol.

RGB and Taro assets?

RGB and Taro are both second-layer asset protocols built on Bitcoin. Compared to inscriptions, they are much more complicated, but much more featureful.

Ordinal theory has been designed from the ground up for digital artifacts, whereas the primary use-case of RGB and Taro are fungible tokens, so the user experience for inscriptions is likely to be simpler and more polished than the user experience for RGB and Taro NFTs.

RGB and Taro both store content off-chain, which requires additional infrastructure, and which may be lost. By contrast, inscription content is stored on-chain, and cannot be lost.

Ordinal theory, RGB, and Taro are all very early, so this is speculation, but ordinal theory's focus may give it the edge in terms of features for digital artifacts, including a better content model, and features like globally unique symbols.

Counterparty assets?

Counterparty has its own token, XCP, which is required for some functionality, which makes most bitcoiners regard it as an altcoin, and not an extension or second layer for bitcoin.

Ordinal theory has been designed from the ground up for digital artifacts, whereas Counterparty was primarily designed for financial token issuance.

Inscriptions for…

Artists

Inscriptions are on Bitcoin. Bitcoin is the digital currency with the highest status and greatest chance of long-term survival. If you want to guarantee that your art survives into the future, there is no better way to publish it than as inscriptions.

Cheaper on-chain storage. At $20,000 per BTC and the minimum relay fee of 1 satoshi per vbyte, publishing inscription content costs $50 per 1 million bytes.

Inscriptions are early! Inscriptions are still in development, and have not yet launched on mainnet. This gives you an opportunity to be an early adopter, and explore the medium as it evolves.

Inscriptions are simple. Inscriptions do not require writing or understanding smart contracts.

Inscriptions unlock new liquidity. Inscriptions are more accessible and appealing to bitcoin holders, unlocking an entirely new class of collector.

Inscriptions are designed for digital artifacts. Inscriptions are designed from the ground up to support NFTs, and feature a better data model, and features like globally unique symbols and enhanced provenance.

Inscriptions do not support on-chain royalties. This is negative, but only depending on how you look at it. On-chain royalties have been a boon for creators, but have also created a huge amount of confusion in the Ethereum NFT ecosystem. The ecosystem now grapples with this issue, and is engaged in a race to the bottom, towards a royalties-optional future. Inscriptions have no support for on-chain royalties, because they are technically infeasible. If you choose to create inscriptions, there are many ways you can work around this limitation: withhold a portion of your inscriptions for future sale, to benefit from future appreciation, or perhaps offer perks for users who respect optional royalties.

Collectors

Inscriptions are simple, clear, and have no surprises. They are always immutable and on-chain, with no special due diligence required.

Inscriptions are on Bitcoin. You can verify the location and properties of inscriptions easily with Bitcoin full node that you control.

Bitcoiners

Let me begin this section by saying: the most important thing that the Bitcoin network does is decentralize money. All other use-cases are secondary, including ordinal theory. The developers of ordinal theory understand and acknowledge this, and believe that ordinal theory helps, at least in a small way, Bitcoin's primary mission.

Unlike many other things in the altcoin space, digital artifacts have merit. There are, of course, a great deal of NFTs that are ugly, stupid, and fraudulent. However, there are many that are fantastically creative, and creating and collecting art has been a part of the human story since its inception, and predates even trade and money, which are also ancient technologies.

Bitcoin provides an amazing platform for creating and collecting digital artifacts in a secure, decentralized way, that protects users and artists in the same way that it provides an amazing platform for sending and receiving value, and for all the same reasons.

Ordinals and inscriptions increase demand for Bitcoin block space, which increase Bitcoin's security budget, which is vital for safeguarding Bitcoin's transition to a fee-dependent security model, as the block subsidy is halved into insignificance.

Inscription content is stored on-chain, and thus the demand for block space for use in inscriptions is unlimited. This creates a buyer of last resort for *all* Bitcoin block space. This will help support a robust fee market, which ensures that Bitcoin remains secure.

Inscriptions also counter the narrative that Bitcoin cannot be extended or used for new use-cases. If you follow projects like DLCs, Fedimint, Lightning, Taro, and RGB, you know that this narrative is false, but inscriptions provide a counter argument which is easy to understand, and which targets a popular and proven use case, NFTs, which makes it highly legible.

If inscriptions prove, as the authors hope, to be highly sought after digital artifacts with a rich history, they will serve as a powerful hook for Bitcoin adoption: come for the fun, rich art, stay for the decentralized digital money.

Inscriptions are an extremely benign source of demand for block space. Unlike, for example, stablecoins, which potentially give large stablecoin issuers influence over the future of Bitcoin development, or DeFi, which might centralize mining by introducing opportunities for MEV, digital art and collectables on Bitcoin, are unlikely to produce individual entities with enough power to corrupt Bitcoin. Art is decentralized.

Inscription users and service providers are incentivized to run Bitcoin full nodes, to publish and track inscriptions, and thus throw their economic weight behind the honest chain.

Ordinal theory and inscriptions do not meaningfully affect Bitcoin's fungibility. Bitcoin users can ignore both and be unaffected.

We hope that ordinal theory strengthens and enriches bitcoin, and gives it another dimension of appeal and functionality, enabling it more effectively serve its primary use case as humanity's decentralized store of value.

Contributing to `ord`

Suggested Steps

1. Find an issue you want to work on.
2. Figure out what would be a good first step towards resolving the issue. This could be in the form of code, research, a proposal, or suggesting that it be closed, if it's out of date or not a good idea in the first place.
3. Comment on the issue with an outline of your suggested first step, and asking for feedback. Of course, you can dive in and start writing code or tests immediately, but this avoids potentially wasted effort, if the issue is out of date, not clearly specified, blocked on something else, or otherwise not ready to implement.
4. If the issue requires a code change or bugfix, open a draft PR with tests, and ask for feedback. This makes sure that everyone is on the same page about what needs to be done, or what the first step in solving the issue should be. Also, since tests are required, writing the tests first makes it easy to confirm that the change can be tested easily.
5. Mash the keyboard randomly until the tests pass, and refactor until the code is ready to submit.
6. Mark the PR as ready to review.
7. Revise the PR as needed.
8. And finally, mergies!

Start small

Small changes will allow you to make an impact quickly, and if you take the wrong tack, you won't have wasted much time.

Ideas for small issues:

- Add a new test or test case that increases test coverage
- Add or improve documentation
- Find an issue that needs more research, and do that research and summarize it in a comment
- Find an out-of-date issue and comment that it can be closed
- Find an issue that shouldn't be done, and provide constructive feedback detailing why you think that is the case

Merge early and often

Break up large tasks into multiple smaller steps that individually make progress. If there's a bug, you can open a PR that adds a failing ignored test. This can be merged, and the next step can be to fix the bug and unignore the test. Do research or testing, and report on your results. Break a feature into small sub-features, and implement them one at a time.

Figuring out how to break down a larger PR into smaller PRs where each can be merged is an art form well-worth practicing. The hard part is that each PR must itself be an improvement.

I strive to follow this advice myself, and am always better off when I do.

Small changes are fast to write, review, and merge, which is much more fun than laboring over a single giant PR that takes forever to write, review, and merge. Small changes don't take much time, so if you need to stop working on a small change, you won't have wasted much time as compared to a larger change that represents many hours of work. Getting a PR in quickly improves the project a little bit immediately, instead of having to wait a long time for larger improvement. Small changes are less likely to accumulate merge conflict. As the Athenians said: *The fast commit what they will, the slow merge what they must.*

Get help

If you're stuck for more than 15 minutes, ask for help, like a Rust Discord, Stack Exchange, or in a project issue or discussion.

Practice hypothesis-driven debugging

Formulate a hypothesis as to what is causing the problem. Figure out how to test that hypothesis. Perform that tests. If it works, great, you fixed the issue or now you know how to fix the issue. If not, repeat with a new hypothesis.

Pay attention to error messages

Read all error messages and don't tolerate warnings.

Donate

Ordinals is open-source and community funded. The current lead maintainer of `ord` is raphjaph. Raph's work on `ord` is entirely funded by donations. If you can, please consider donating!

The donation address for Bitcoin is bc1q8kt9pyd6r27k2840l8g5d7zshz3cg9v6rfda0m248lva3ve5072q3sxelt. The donation address for inscriptions is bc1qn3map8m9hmk5jyqdkkwlwvt335g94zvxwd9aql7q3vdkdw9r5eyqvlvec0.

Both addresses are in a 2 of 4 multisig wallet with keys held by raphjaph, erin, rodarmor, and ordinally.

Donations received will go towards funding maintenance and development of `ord`, as well as hosting costs for ordinals.com.

Thank you for donating!

Ordinal Theory Guides

See the table of contents for a list of guides, including a guide to the explorer, a guide for sat hunters, and a guide to inscriptions.

Ordinal Explorer

The `ord` binary includes a block explorer. We host an instance of the block explorer on mainnet at ordinals.com, and on signet at signet.ordinals.com.

Running The Explorer

The server can be run locally with:

```
ord server
```

To specify a port add the `--http-port` flag:

```
ord server --http-port 8080
```

The JSON-API endpoints are enabled by default, to disable them add the `--disable-json-api` flag (see here for more info):

```
ord server --disable-json-api
```

Search

The search box accepts a variety of object representations.

Blocks

Blocks can be searched by hash, for example, the genesis block:

000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

Transactions

Transactions can be searched by hash, for example, the genesis block coinbase transaction:

4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b

Outputs

Transaction outputs can be searched by outpoint, for example, the only output of the genesis block coinbase transaction:

4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b:0

Sats

Sats can be searched by integer, their position within the entire bitcoin supply:

2099994106992659

By decimal, their block and offset within that block:

481824.0

By degree, their cycle, blocks since the last halving, blocks since the last difficulty adjustment, and offset within their block:

1°0'0"0‴

By name, their base 26 representation using the letters "a" through "z":

ahistorical

Or by percentile, the percentage of bitcoin's supply that has been or will have been issued when they are mined:

100%

JSON-API

By default the `ord server` gives access to endpoints that return JSON instead of HTML if you set the HTTP `Accept: application/json` header. The structure of these objects closely follows what is shown in the HTML. These endpoints are:

- `/inscription/<INSCRIPTION_ID>`
- `/inscriptions`
- `/inscriptions/block/<BLOCK_HEIGHT>`
- `/inscriptions/block/<BLOCK_HEIGHT>/<PAGE_INDEX>`
- `/inscriptions/<FROM>`
- `/inscriptions/<FROM>/<N>`
- `/output/<OUTPOINT>`
- `/output/<OUTPOINT>`
- `/sat/<SAT>`

To get a list of the latest 100 inscriptions you would do:

```
curl -s -H "Accept: application/json" 'http://0.0.0.0:80/inscriptions'
```

To see information about a UTXO, which includes inscriptions inside it, do:

```
curl -s -H "Accept: application/json" 'http://0.0.0.0:80/output/
bc4c30829a9564c0d58e6287195622b53ced54a25711d1b86be7cd3a70ef61ed:0'
```

Which returns:

```
{
  "value": 10000,
  "script_pubkey": "OP_PUSHNUM_1 OP_PUSHBYTES_32
156cc4878306157720607cdcb4b32afa4cc6853868458d7258b907112e5a434b",
  "address":
"bc1pz4kvfpurqc2hwgrq0nwtfve2lfxvdpfcdpzc6ujchyr3ztj6gd9sfr6ayf",
  "transaction":
"bc4c30829a9564c0d58e6287195622b53ced54a25711d1b86be7cd3a70ef61ed",
  "sat_ranges": null,
  "inscriptions": [
    "6fb976ab49dcec017f1e201e84395983204ae1a7c2abf7ced0a85d692e442799i0"
  ]
}
```

Ordinal Inscription Guide

Individual sats can be inscribed with arbitrary content, creating Bitcoin-native digital artifacts that can be held in a Bitcoin wallet and transferred using Bitcoin transactions. Inscriptions are as durable, immutable, secure, and decentralized as Bitcoin itself.

Working with inscriptions requires a Bitcoin full node, to give you a view of the current state of the Bitcoin blockchain, and a wallet that can create inscriptions and perform sat control when constructing transactions to send inscriptions to another wallet.

Bitcoin Core provides both a Bitcoin full node and wallet. However, the Bitcoin Core wallet cannot create inscriptions and does not perform sat control.

This requires `ord`, the ordinal utility. `ord` doesn't implement its own wallet, so `ord wallet` subcommands interact with Bitcoin Core wallets.

This guide covers:

1. Installing Bitcoin Core
2. Syncing the Bitcoin blockchain
3. Creating a Bitcoin Core wallet
4. Using `ord wallet receive` to receive sats
5. Creating inscriptions with `ord wallet inscribe`
6. Sending inscriptions with `ord wallet send`
7. Receiving inscriptions with `ord wallet receive`
8. Batch inscribing with `ord wallet inscribe --batch`

Getting Help

If you get stuck, try asking for help on the Ordinals Discord Server, or checking GitHub for relevant issues and discussions.

Installing Bitcoin Core

Bitcoin Core is available from bitcoincore.org on the download page.

Making inscriptions requires Bitcoin Core 24 or newer.

This guide does not cover installing Bitcoin Core in detail. Once Bitcoin Core is installed, you should be able to run `bitcoind -version` successfully from the command line. Do *NOT* use `bitcoin-qt`.

Configuring Bitcoin Core

`ord` requires Bitcoin Core's transaction index and rest interface.

To configure your Bitcoin Core node to maintain a transaction index, add the following to your `bitcoin.conf`:

```
txindex=1
```

Or, run `bitcoind` with `-txindex`:

```
bitcoind -txindex
```

Details on creating or modifying your `bitcoin.conf` file can be found here.

Syncing the Bitcoin Blockchain

To sync the chain, run:

```
bitcoind -txindex
```

…and leave it running until `getblockcount`:

```
bitcoin-cli getblockcount
```

agrees with the block count on a block explorer like the mempool.space block explorer. `ord` interacts with `bitcoind`, so you should leave `bitcoind` running in the background when you're using `ord`.

The blockchain takes about 600GB of disk space. If you have an external drive you want to store blocks on, use the configuration option `blocksdir=<external_drive_path>`. This is much simpler than using the `datadir` option because the cookie file will still be in the default location for `bitcoin-cli` and `ord` to find.

Troubleshooting

Make sure you can access `bitcoind` with `bitcoin-cli -getinfo` and that it is fully synced.

If `bitcoin-cli -getinfo` returns `Could not connect to the server`, `bitcoind` is not running.

Make sure `rpcuser`, `rpcpassword`, or `rpcauth` are *NOT* set in your `bitcoin.conf` file. `ord` requires using cookie authentication. Make sure there is a file `.cookie` in your bitcoin data directory.

If `bitcoin-cli -getinfo` returns `Could not locate RPC credentials`, then you must specify the cookie file location. If you are using a custom data directory (specifying the `datadir` option), then you must specify the cookie location like `bitcoin-cli -rpccookiefile=<your_bitcoin_datadir>/.cookie -getinfo`. When running `ord` you must specify the cookie file location with `--cookie-file=<your_bitcoin_datadir>/.cookie`.

Make sure you do *NOT* have `disablewallet=1` in your `bitcoin.conf` file. If `bitcoin-cli listwallets` returns `Method not found` then the wallet is disabled and you won't be able to use `ord`.

Make sure `txindex=1` is set. Run `bitcoin-cli getindexinfo` and it should return something like

```
{
  "txindex": {
    "synced": true,
    "best_block_height": 776546
  }
}
```

If it only returns `{}`, `txindex` is not set. If it returns `"synced": false`, `bitcoind` is still creating the `txindex`. Wait until `"synced": true` before using `ord`.

If you have `maxuploadtarget` set it can interfere with fetching blocks for `ord` index. Either remove it or set `whitebind=127.0.0.1:8333`.

Installing `ord`

The `ord` utility is written in Rust and can be built from source. Pre-built binaries are available on the releases page.

You can install the latest pre-built binary from the command line with:

```
curl --proto '=https' --tlsv1.2 -fsLS https://ordinals.com/install.sh | bash
-s
```

Once `ord` is installed, you should be able to run:

```
ord --version
```

Which prints out `ord` 's version number.

Creating a Bitcoin Core Wallet

`ord` uses Bitcoin Core to manage private keys, sign transactions, and broadcast transactions to the Bitcoin network.

To create a Bitcoin Core wallet named `ord` for use with `ord` , run:

```
ord wallet create
```

Receiving Sats

Inscriptions are made on individual sats, using normal Bitcoin transactions that pay fees in sats, so your wallet will need some sats.

Get a new address from your `ord` wallet by running:

```
ord wallet receive
```

And send it some funds.

You can see pending transactions with:

```
ord wallet transactions
```

Once the transaction confirms, you should be able to see the transactions outputs with `ord wallet outputs` .

Creating Inscription Content

Sats can be inscribed with any kind of content, but the `ord` wallet only supports content types that can be displayed by the `ord` block explorer.

Additionally, inscriptions are included in transactions, so the larger the content, the higher the fee that the inscription transaction must pay.

Inscription content is included in transaction witnesses, which receive the witness discount. To calculate the approximate fee that an inscribe transaction will pay, divide the content size by four and multiply by the fee rate.

Inscription transactions must be less than 400,000 weight units, or they will not be relayed by Bitcoin Core. One byte of inscription content costs one weight unit. Since an inscription transaction includes not just the inscription content, limit inscription content to less than 400,000 weight units. 390,000 weight units should be safe.

Creating Inscriptions

To create an inscription with the contents of `FILE`, run:

```
ord wallet inscribe --fee-rate FEE_RATE --file FILE
```

Ord will output two transactions IDs, one for the commit transaction, and one for the reveal transaction, and the inscription ID. Inscription IDs are of the form `TXIDiN`, where `TXID` is the transaction ID of the reveal transaction, and `N` is the index of the inscription in the reveal transaction.

The commit transaction commits to a tapscript containing the content of the inscription, and the reveal transaction spends from that tapscript, revealing the content on chain and inscribing it on the first sat of the input that contains the corresponding tapscript.

Wait for the reveal transaction to be mined. You can check the status of the commit and reveal transactions using the mempool.space block explorer.

Once the reveal transaction has been mined, the inscription ID should be printed when you run:

```
ord wallet inscriptions
```

Parent-Child Inscriptions

Parent-child inscriptions enable what is colloquially known as collections, see provenance for more information.

To make an inscription a child of another, the parent inscription has to be inscribed and present in the wallet. To choose a parent run `ord wallet inscriptions` and copy the inscription id (`<PARENT_INSCRIPTION_ID>`).

Now inscribe the child inscription and specify the parent like so:

```
ord wallet inscribe --fee-rate FEE_RATE --parent <PARENT_INSCRIPTION_ID> --
file CHILD_FILE
```

This relationship cannot be added retroactively, the parent has to be present at inception of the child.

Sending Inscriptions

Ask the recipient to generate a new address by running:

```
ord wallet receive
```

Send the inscription by running:

```
ord wallet send --fee-rate <FEE_RATE> <ADDRESS> <INSCRIPTION_ID>
```

See the pending transaction with:

```
ord wallet transactions
```

Once the send transaction confirms, the recipient can confirm receipt by running:

```
ord wallet inscriptions
```

Receiving Inscriptions

Generate a new receive address using:

```
ord wallet receive
```

The sender can transfer the inscription to your address using:

```
ord wallet send ADDRESS INSCRIPTION_ID
```

See the pending transaction with:

```
ord wallet transactions
```

Once the send transaction confirms, you can confirm receipt by running:

```
ord wallet inscriptions
```

Batch Inscribing

Multiple inscriptions can be created inscriptions at the same time using the pointer field. This is especially helpful for collections, or other cases when multiple inscriptions should share the same parent, since the parent can passed into a reveal transaction that creates multiple children.

To create a batch inscription using a batchfile in `batch.yaml`, run the following command:

```
ord wallet inscribe --fee-rate 21 --batch batch.yaml
```

Example `batch.yaml`

```yaml
# example batch file

# inscription modes:
# - `separate-outputs`: inscribe on separate postage-sized outputs
# - `shared-output`: inscribe on a single output separated by postage
# - `same-sat`: inscribe on the same sat
mode: separate-outputs

# parent inscription:
parent: 6ac5cacb768794f4fd7a78bf00f2074891fce68bd65c4ff36e77177237aacacai0

# postage for each inscription:
postage: 12345

# sat to inscribe on, can only be used with `same-sat`:
# sat: 5000000000

# inscriptions to inscribe
inscriptions:
  # path to inscription content
- file: mango.avif
  # inscription to delegate content to (optional)
  delegate:
6ac5cacb768794f4fd7a78bf00f2074891fce68bd65c4ff36e77177237aacacai0
  # destination (optional, if no destination is specified a new wallet change
address will be used)
  destination: bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4
  # inscription metadata (optional)
  metadata:
    title: Delicious Mangos
    description: >
      Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam
semper,
      ligula ornare laoreet tincidunt, odio nisi euismod tortor, vel blandit
      metus est et odio. Nullam venenatis, urna et molestie vestibulum, orci
      mi efficitur risus, eu malesuada diam lorem sed velit. Nam fermentum
      dolor et luctus euismod.
    # inscription metaprotocol (optional)
    metaprotocol: DOPEPROTOCOL-42069

- file: token.json

- file: tulip.png
  destination: bc1pdqrcrxa8vx6gy75mfdfj84puhxffh4fq46h3gkp6jxdd0vjcsdyspfxcv6
  metadata:
    author: Satoshi Nakamoto
```

Sat Hunting

This guide is out of date. Since it was written, the `ord` binary was changed to only build the full satoshi index when the `--index-sats` flag is supplied. Additionally, `ord` now has a built-in wallet that wraps a Bitcoin Core wallet. See `ord wallet --help`.

Ordinal hunting is difficult but rewarding. The feeling of owning a wallet full of UTXOs, redolent with the scent of rare and exotic sats, is beyond compare.

Ordinals are numbers for satoshis. Every satoshi has an ordinal number and every ordinal number has a satoshi.

Preparation

There are a few things you'll need before you start.

1. First, you'll need a synced Bitcoin Core node with a transaction index. To turn on transaction indexing, pass `-txindex` on the command-line:

   ```
   bitcoind -txindex
   ```

   Or put the following in your Bitcoin configuration file:

   ```
   txindex=1
   ```

   Launch it and wait for it to catch up to the chain tip, at which point the following command should print out the current block height:

   ```
   bitcoin-cli getblockcount
   ```

2. Second, you'll need a synced `ord` index.

   - Get a copy of `ord` from the repo.
   - Run `RUST_LOG=info ord index`. It should connect to your bitcoin core node and start indexing.
   - Wait for it to finish indexing.

3. Third, you'll need a wallet with UTXOs that you want to search.

Searching for Rare Ordinals

Searching for Rare Ordinals in a Bitcoin Core Wallet

The `ord wallet` command is just a wrapper around Bitcoin Core's RPC API, so searching for rare ordinals in a Bitcoin Core wallet is Easy. Assuming your wallet is named `foo`:

1. Load your wallet:

   ```
   bitcoin-cli loadwallet foo
   ```

2. Display any rare ordinals wallet `foo`'s UTXOs:

   ```
   ord --wallet foo --index-sats wallet sats
   ```

Searching for Rare Ordinals in a Non-Bitcoin Core Wallet

The `ord wallet` command is just a wrapper around Bitcoin Core's RPC API, so to search for rare ordinals in a non-Bitcoin Core wallet, you'll need to import your wallet's descriptors into Bitcoin Core.

Descriptors describe the ways that wallets generate private keys and public keys.

You should only import descriptors into Bitcoin Core for your wallet's public keys, not its private keys.

If your wallet's public key descriptor is compromised, an attacker will be able to see your wallet's addresses, but your funds will be safe.

If your wallet's private key descriptor is compromised, an attacker can drain your wallet of funds.

1. Get the wallet descriptor from the wallet whose UTXOs you want to search for rare ordinals. It will look something like this:

   ```
   wpkh([bf1dd55e/84'/0'/
   0']xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
   JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/0/*)#csvefu29
   ```

2. Create a watch-only wallet named `foo-watch-only`:

```
bitcoin-cli createwallet foo-watch-only true true
```

Feel free to give it a better name than `foo-watch-only`!

3. Load the `foo-watch-only` wallet:

```
bitcoin-cli loadwallet foo-watch-only
```

4. Import your wallet descriptors into `foo-watch-only`:

```
bitcoin-cli importdescriptors \
  '[{ "desc": "wpkh([bf1dd55e/84h/0h/
0h]xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/0/*)#tpnxnxax", "timestamp":0
}]'
```

If you know the Unix timestamp when your wallet first started receive transactions, you may use it for the value of `"timestamp"` instead of `0`. This will reduce the time it takes for Bitcoin Core to search for your wallet's UTXOs.

5. Check that everything worked:

```
bitcoin-cli getwalletinfo
```

6. Display your wallet's rare ordinals:

```
ord wallet sats
```

Searching for Rare Ordinals in a Wallet that Exports Multi-path Descriptors

Some descriptors describe multiple paths in one descriptor using angle brackets, e.g., `<0;1>`. Multi-path descriptors are not yet supported by Bitcoin Core, so you'll first need to convert them into multiple descriptors, and then import those multiple descriptors into Bitcoin Core.

1. First get the multi-path descriptor from your wallet. It will look something like this:

```
wpkh([bf1dd55e/84h/0h/
0h]xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/<0;1>/*)#fw76ulgt
```

2. Create a descriptor for the receive address path:

   ```
   wpkh([bf1dd55e/84'/0'/
   0']xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
   JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/0/*)
   ```

   And the change address path:

   ```
   wpkh([bf1dd55e/84'/0'/
   0']xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
   JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/1/*)
   ```

3. Get and note the checksum for the receive address descriptor, in this case `tpnxnxax`:

   ```
   bitcoin-cli getdescriptorinfo \
     'wpkh([bf1dd55e/84h/0h/
   0h]xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
   JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/0/*)'
   ```

   ```
   {
     "descriptor": "wpkh([bf1dd55e/84'/0'/
   0']xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
   JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/0/*)#csvefu29",
     "checksum": "tpnxnxax",
     "isrange": true,
     "issolvable": true,
     "hasprivatekeys": false
   }
   ```

   And for the change address descriptor, in this case `64k8wnd7`:

   ```
   bitcoin-cli getdescriptorinfo \
     'wpkh([bf1dd55e/84h/0h/
   0h]xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
   JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/1/*)'
   ```

```
{
  "descriptor": "wpkh([bf1dd55e/84'/0'/
0']xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/1/*)#fyfc5f6a",
  "checksum": "64k8wnd7",
  "isrange": true,
  "issolvable": true,
  "hasprivatekeys": false
}
```

4. Load the wallet you want to import the descriptors into:

```
bitcoin-cli loadwallet foo-watch-only
```

5. Now import the descriptors, with the correct checksums, into Bitcoin Core.

```
bitcoin-cli \
 importdescriptors \
 '[
   {
     "desc": "wpkh([bf1dd55e/84h/0h/
0h]xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/0/*)#tpnxnxax"
     "timestamp":0
   },
   {
     "desc": "wpkh([bf1dd55e/84h/0h/
0h]xpub6CcJtWcvFQaMo39ANFi1MyXkEXM8T8ZhnxMtSjQAdPmVSTHYnc8Hwoc11VpuP8cb8
JUTboZB5A7YYGDonYySij4XTawL6iNZvmZwdnSEEep/1/*)#64k8wnd7",
     "timestamp":0
   }
 ]'
```

If you know the Unix timestamp when your wallet first started receive transactions, you may use it for the value of the `"timestamp"` fields instead of `0`. This will reduce the time it takes for Bitcoin Core to search for your wallet's UTXOs.

6. Check that everything worked:

```
bitcoin-cli getwalletinfo
```

7. Display your wallet's rare ordinals:

```
ord wallet sats
```

Exporting Descriptors

Sparrow Wallet

Navigate to the `Settings` tab, then to `Script Policy`, and press the edit button to display the descriptor.

Transferring Ordinals

The `ord` wallet supports transferring specific satoshis. You can also use `bitcoin-cli` commands `createrawtransaction`, `signrawtransactionwithwallet`, and `sendrawtransaction`, how to do so is complex and outside the scope of this guide.

Teleburning

Teleburn addresses can be used to burn assets on other blockchains, leaving behind in the smoking rubble a sort of forwarding address pointing to an inscription on Bitcoin.

Teleburning an asset means something like, "I'm out. Find me on Bitcoin."

Teleburn addresses are derived from inscription IDs. They have no corresponding private key, so assets sent to a teleburn address are burned. Currently, only Ethereum teleburn addresses are supported. Pull requests adding teleburn addresses for other chains are welcome.

Ethereum

Ethereum teleburn addresses are derived by taking the first 20 bytes of the SHA-256 hash of the inscription ID, serialized as 36 bytes, with the first 32 bytes containing the transaction ID, and the last four bytes containing big-endian inscription index, and interpreting it as an Ethereum address.

Example

The ENS domain name rodarmor.eth, was teleburned to inscription zero.

Running the inscription ID of inscription zero is `6fb976ab49dcec017f1e201e84395983204ae1a7c2abf7ced0a85d692e442799i0`.

Passing `6fb976ab49dcec017f1e201e84395983204ae1a7c2abf7ced0a85d692e442799i0` to the teleburn command:

```
$ ord teleburn
6fb976ab49dcec017f1e201e84395983204ae1a7c2abf7ced0a85d692e442799i0
```

Returns:

```
{
  "ethereum": "0xe43A06530BdF8A4e067581f48Fae3b535559dA9e"
}
```

Indicating that `0xe43A06530BdF8A4e067581f48Fae3b535559dA9e` is the Ethereum teleburn address for inscription zero, which is, indeed, the current owner, on Ethereum, of `rodarmor.eth`.

Collecting

Currently, ord is the only wallet supporting sat-control and sat-selection, which are required to safely store and send rare sats and inscriptions, hereafter ordinals.

The recommended way to send, receive, and store ordinals is with `ord`, but if you are careful, it is possible to safely store, and in some cases send, ordinals with other wallets.

As a general note, receiving ordinals in an unsupported wallet is not dangerous. Ordinals can be sent to any bitcoin address, and are safe as long as the UTXO that contains them is not spent. However, if that wallet is then used to send bitcoin, it may select the UTXO containing the ordinal as an input, and send the inscription or spend it to fees.

A guide to creating an `ord`-compatible wallet with Sparrow Wallet, is available in this handbook.

Please note that if you follow this guide, you should not use the wallet you create to send BTC, unless you perform manual coin-selection to avoid sending ordinals.

Collecting Inscriptions and Ordinals with Sparrow Wallet

Users who cannot or have not yet set up the ord wallet can receive inscriptions and ordinals with alternative bitcoin wallets, as long as they are *very* careful about how they spend from that wallet.

This guide gives some basic steps on how to create a wallet with Sparrow Wallet which is compatible with `ord` and can be later imported into `ord`

⚠️⚠️ Warning!! ⚠️⚠️

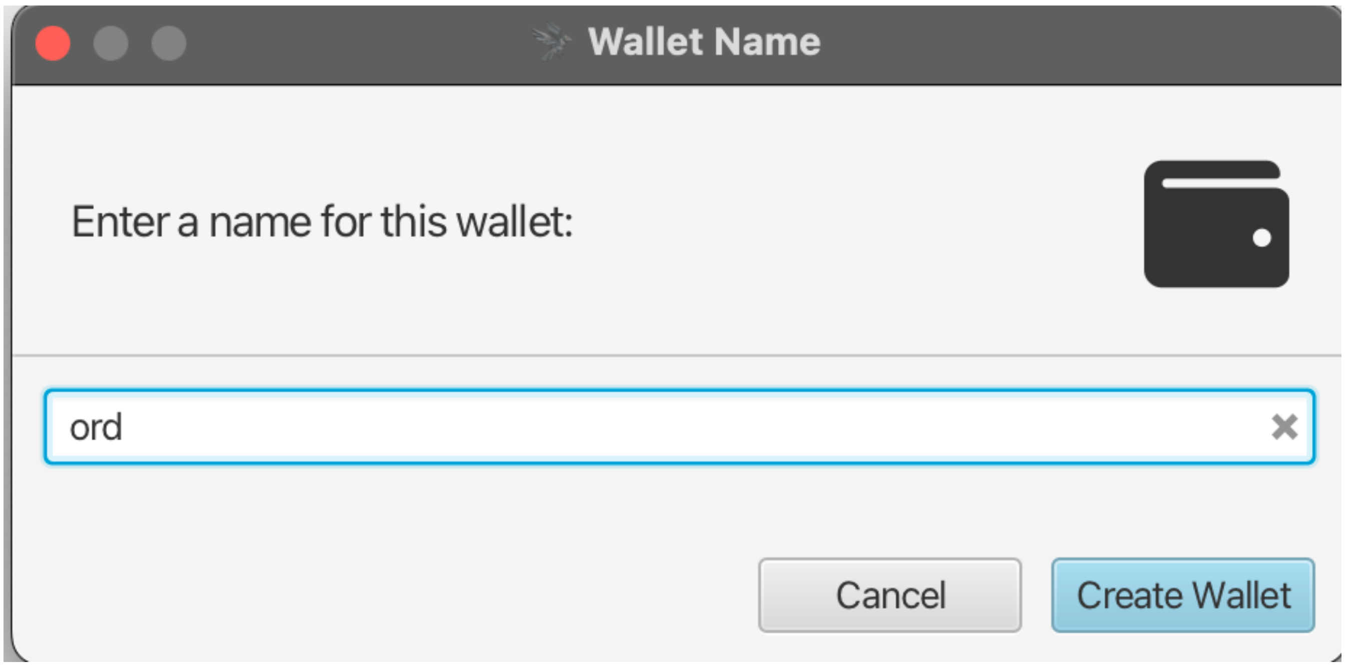
As a general rule if you take this approach, you should use this wallet with the Sparrow software as a receive-only wallet.

Do not spend any satoshis from this wallet unless you are sure you know what you are doing. You could very easily inadvertently lose access to your ordinals and inscriptions if you don't heed this warning.

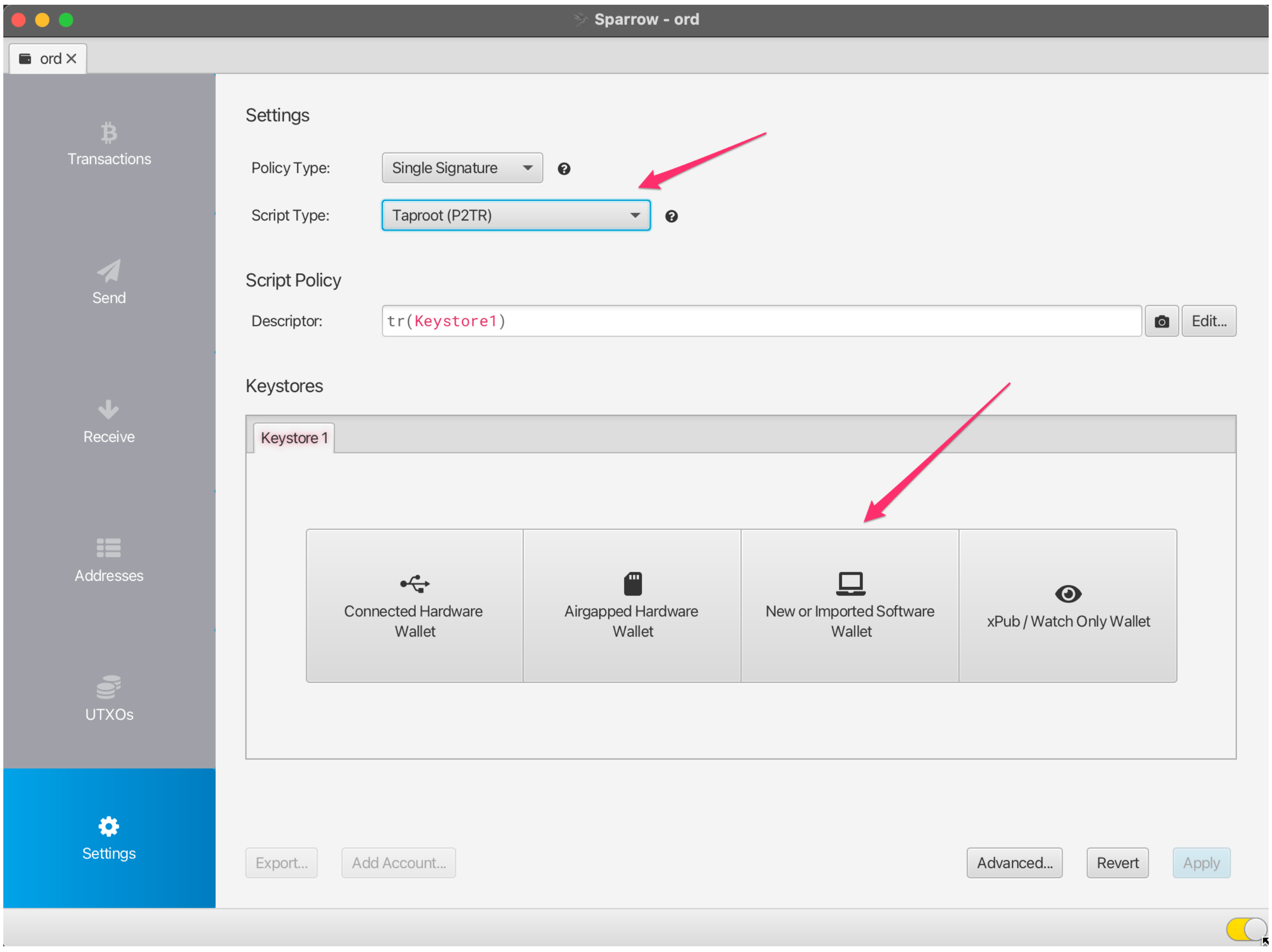
Wallet Setup & Receiving

Download the Sparrow Wallet from the releases page for your particular operating system.

Select `File -> New Wallet` and create a new wallet called `ord`.



Change the `Script Type` to `Taproot (P2TR)` and select the `New or Imported Software Wallet` option.


Settings

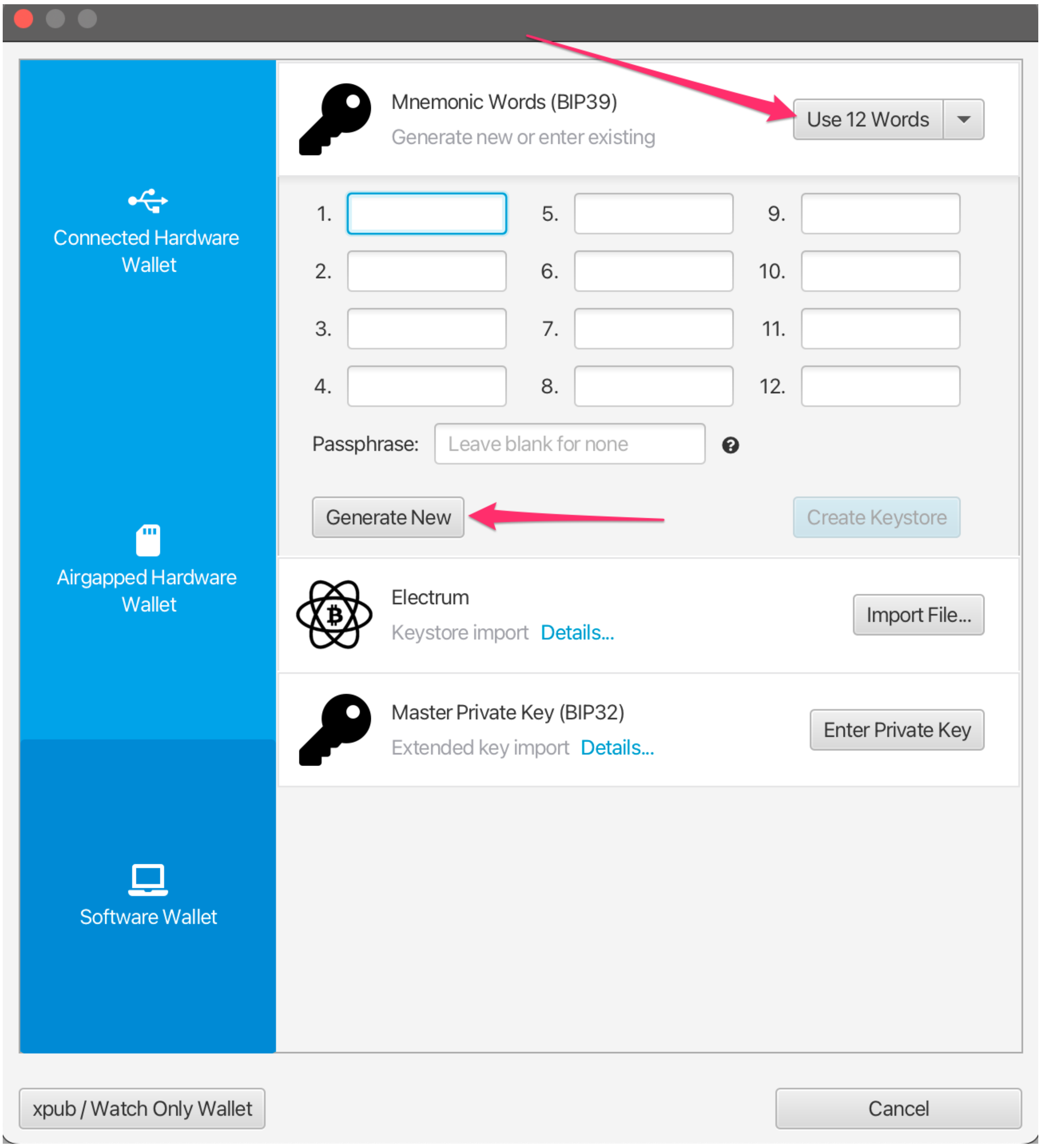
Policy Type: Single Signature

Script Type: Taproot (P2TR)

Script Policy

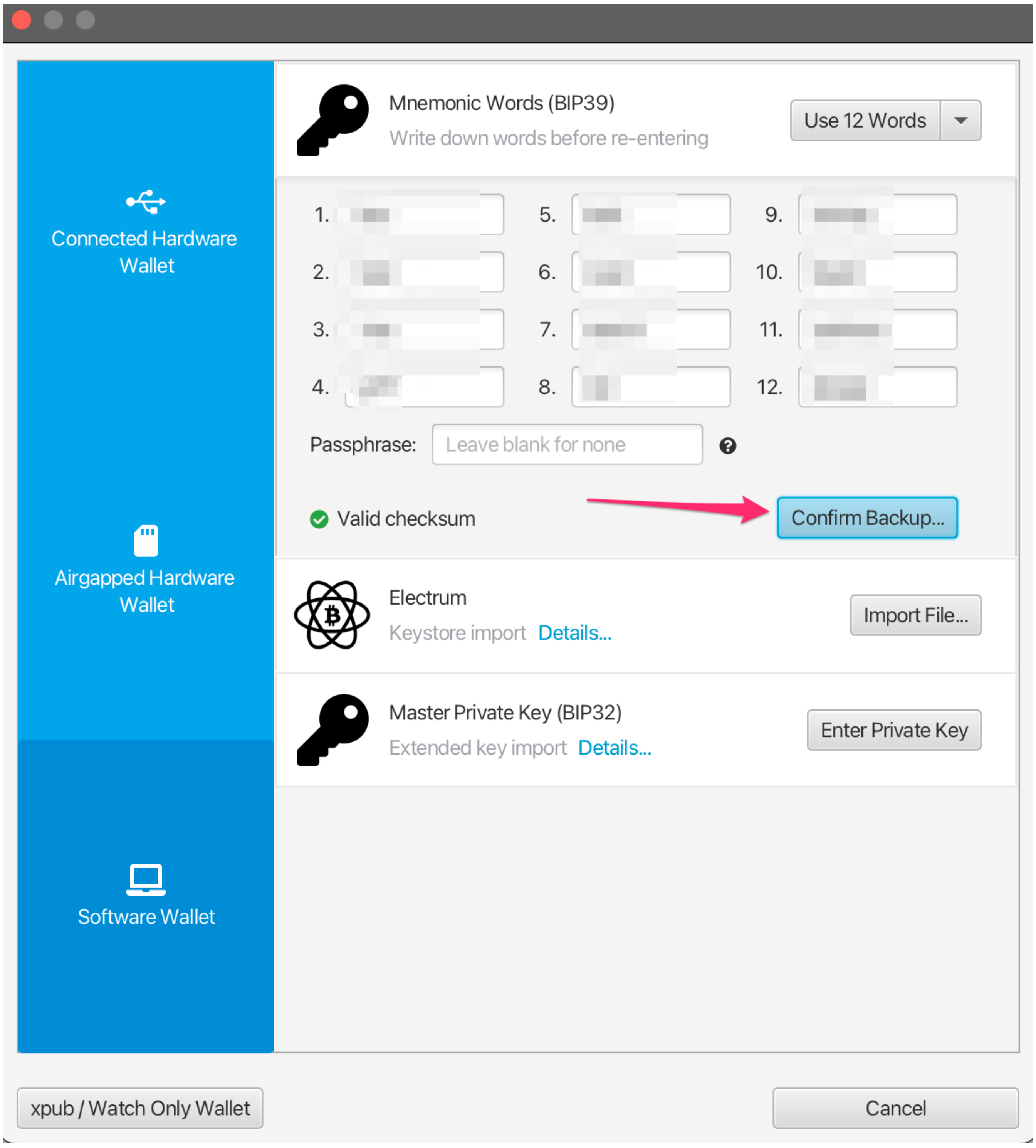
Descriptor: tr(Keystore1)

Select `Use 12 Words` and then click `Generate New`. Leave the passphrase blank.

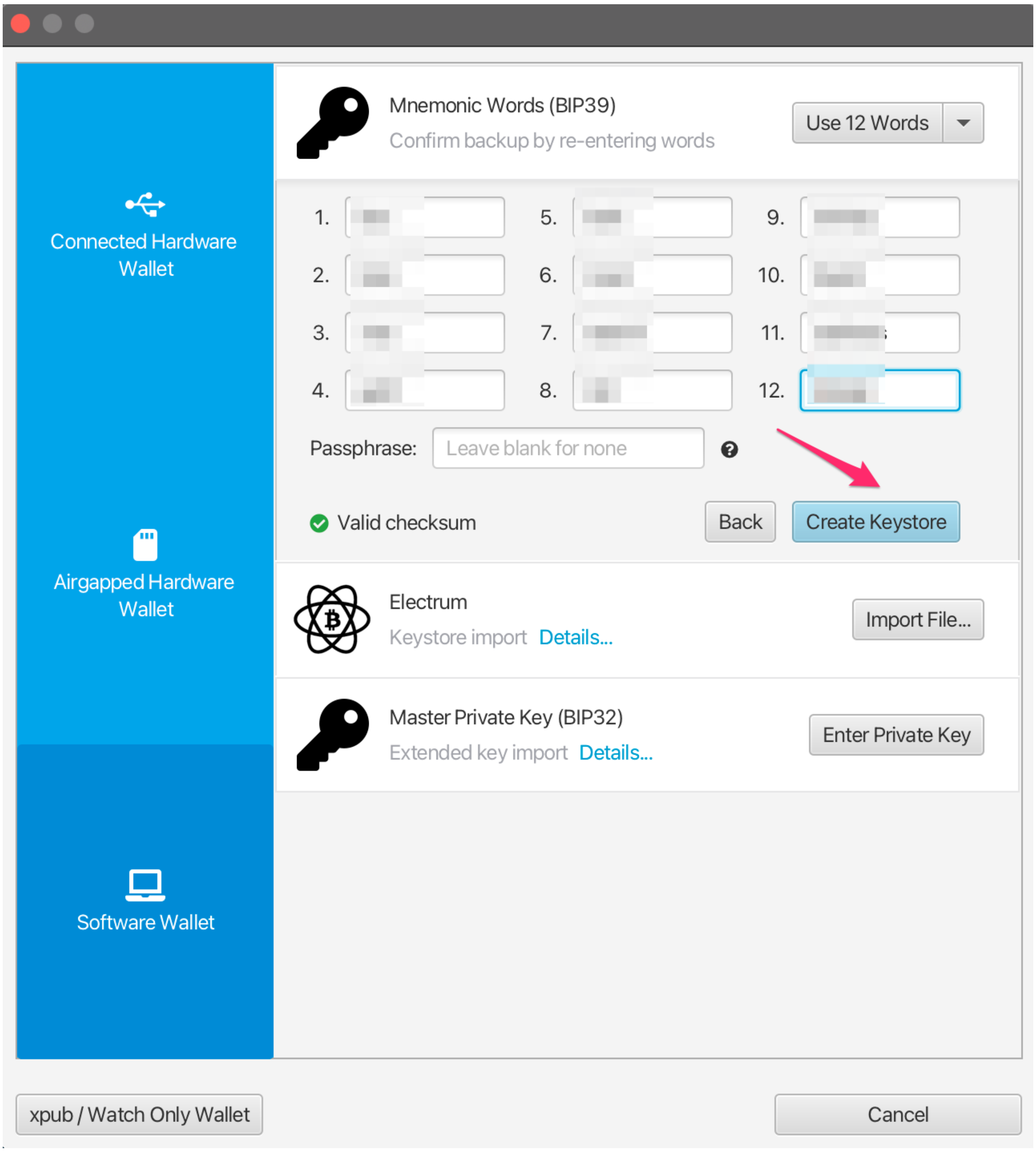


A new 12 word BIP39 seed phrase will be generated for you. Write this down somewhere safe as this is your backup to get access to your wallet. NEVER share or show this seed phrase to anyone else.

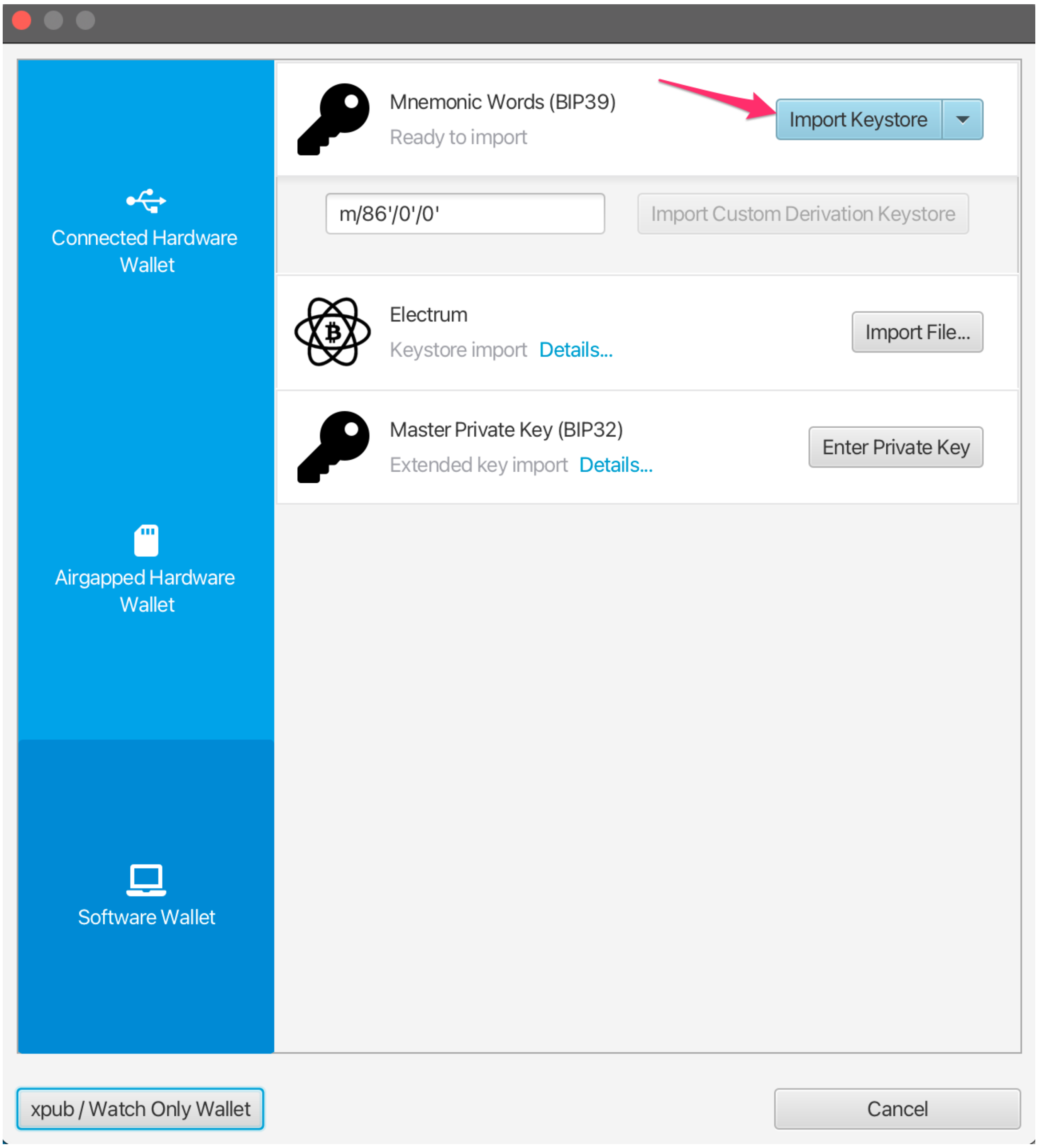
Once you have written down the seed phrase click `Confirm Backup`.



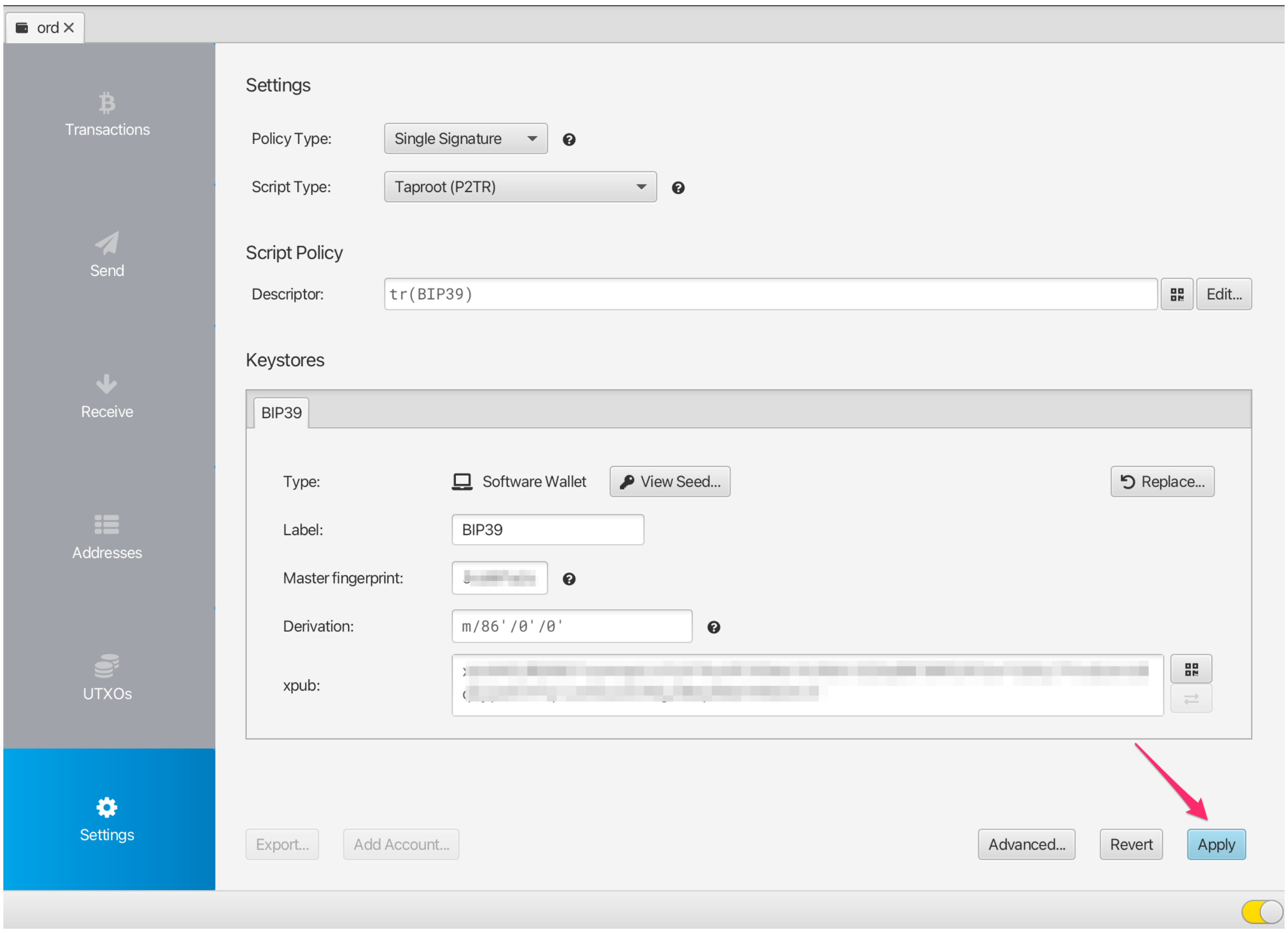
Re-enter the seed phrase which you wrote down, and then click `Create Keystore`.



Click `Import Keystore`.


Connected Hardware Wallet
Airgapped Hardware Wallet
Software Wallet
Mnemonic Words (BIP39)
Ready to import
Import Keystore
m/86'/0'/0'
Import Custom Derivation Keystore
Electrum
Keystore import Details...
Import File...
Master Private Key (BIP32)
Extended key import Details...
Enter Private Key
xpub / Watch Only Wallet
Cancel

Click `Apply`. Add a password for the wallet if you want to.



Settings

Policy Type: Single Signature

Script Type: Taproot (P2TR)

Script Policy

Descriptor: tr(BIP39)

Keystores

BIP39

Type: Software Wallet

Label: BIP39

Master fingerprint:

Derivation: m/86'/0'/0'

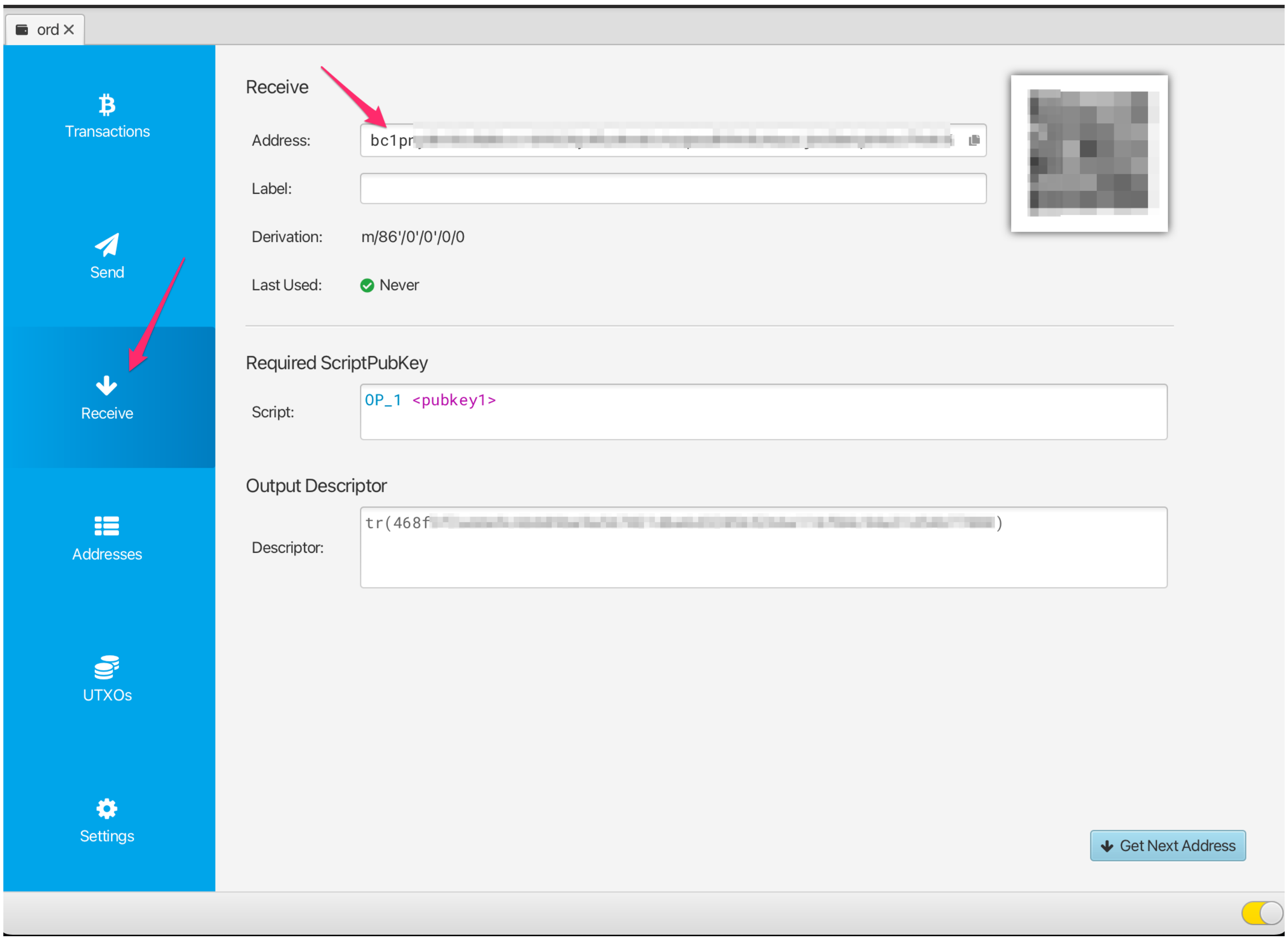
xpub:

You now have a wallet which is compatible with `ord`, and can be imported into `ord` using the BIP39 Seed Phrase. To receive ordinals or inscriptions, click on the `Receive` tab and copy a new address.

Each time you want to receive you should use a brand-new address, and not re-use existing addresses.

Note that bitcoin is different to some other blockchain wallets, in that this wallet can generate an unlimited number of new addresses. You can generate a new address by clicking on the `Get Next Address` button. You can see all of your addresses in the `Addresses` tab of the app.

You can add a label to each address, so you can keep track of what it was used for.



Receive

Address: bc1p[illegible]

Label:

Derivation: m/86'/0'/0'/0/0

Last Used: Never

Required ScriptPubKey

Script: `OP_1 <pubkey1>`

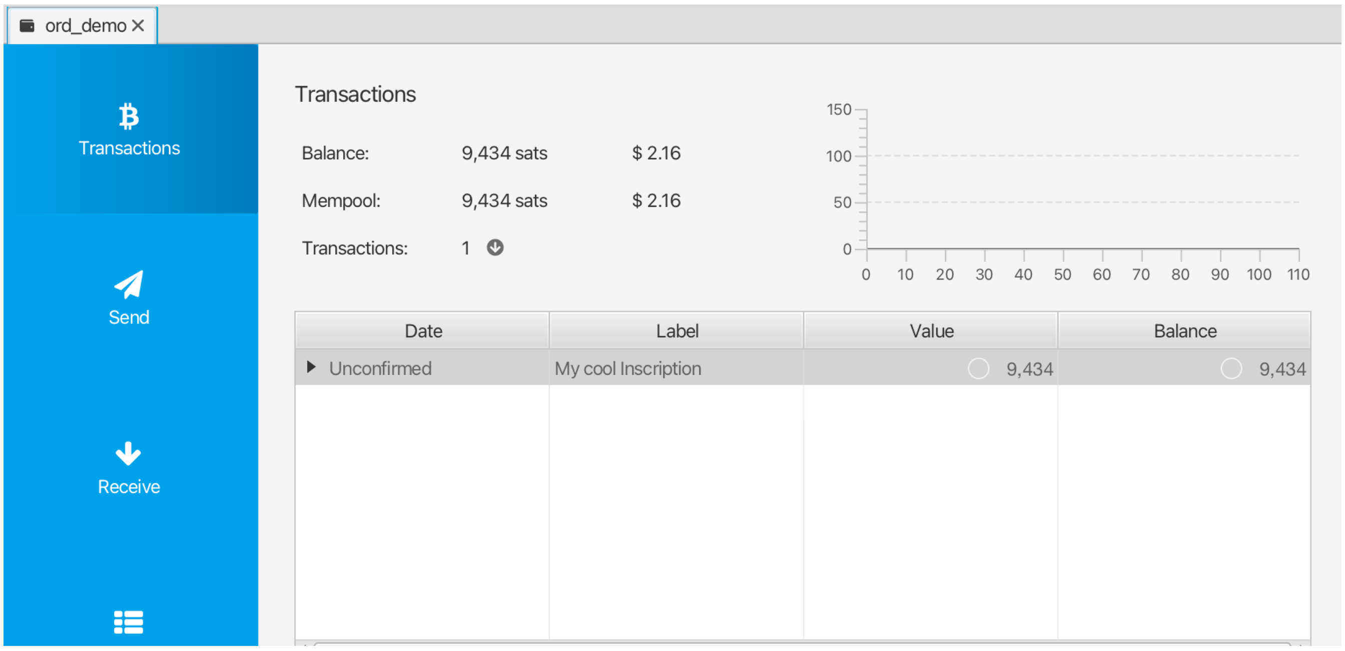
Output Descriptor

Descriptor: `tr(468f[illegible])`

Validating / Viewing Received Inscriptions

Once you have received an inscription you will see a new transaction in the `Transactions` tab of Sparrow, as well as a new UTXO in the `UTXOs` tab.

Initially this transaction may have an "Unconfirmed" status, and you will need to wait for it to be mined into a bitcoin block before it is fully received.

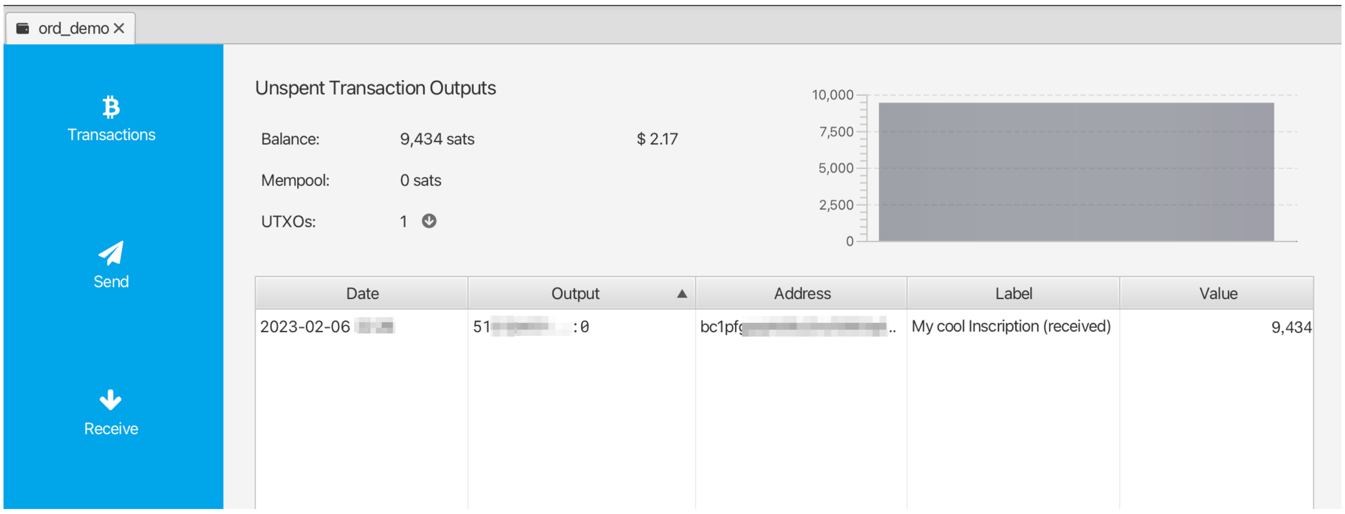


Transactions

Balance:	9,434 sats	$ 2.16
Mempool:	9,434 sats	$ 2.16
Transactions:	1	

Date	Label	Value	Balance
Unconfirmed	My cool Inscription	9,434	9,434

To track the status of your transaction you can right-click on it, select `Copy Transaction ID` and then paste that transaction id into mempool.space.



Unspent Transaction Outputs

Balance:	9,434 sats	$ 2.17
Mempool:	0 sats	
UTXOs:	1	

Date	Output	Address	Label	Value
2023-02-06 [redacted]	51[redacted]:0	bc1pfg[redacted]..	My cool Inscription (received)	9,434

Once the transaction has confirmed, you can validate and view your inscription by heading over to the `UTXOs` tab, finding the UTXO you want to check, right-clicking on the `Output` and selecting `Copy Transaction Output`. This transaction output id can then be pasted into the ordinals.com search.

Freezing UTXO's

As explained above, each of your inscriptions is stored in an Unspent Transaction Output (UTXO). You want to be very careful not to accidentally spend your inscriptions, and one way to make it harder for this to happen is to freeze the UTXO.

To do this, go to the `UTXOs` tab, find the UTXO you want to freeze, right-click on the `Output` and select `Freeze UTXO`.

This UTXO (Inscription) is now un-spendable within the Sparrow Wallet until you unfreeze it.

Importing into `ord` wallet

For details on setting up Bitcoin Core and the `ord` wallet check out the Inscriptions Guide

When setting up `ord`, instead of running `ord wallet create` to create a brand-new wallet, you can import your existing wallet using `ord wallet restore "BIP39 SEED PHRASE"` using the seed phrase you generated with Sparrow Wallet.

There is currently a bug which causes an imported wallet to not be automatically rescanned against the blockchain. To work around this you will need to manually trigger a rescan using the bitcoin core cli: `bitcoin-cli -rpcwallet=ord rescanblockchain 767430`

You can then check your wallet's inscriptions using `ord wallet inscriptions`

Note that if you have previously created a wallet with `ord`, then you will already have a wallet with the default name, and will need to give your imported wallet a different name. You can use the `--wallet` parameter in all `ord` commands to reference a different wallet, eg:

```
ord --wallet ord_from_sparrow wallet restore "BIP39 SEED PHRASE"
```

```
ord --wallet ord_from_sparrow wallet inscriptions
```

```
bitcoin-cli -rpcwallet=ord_from_sparrow rescanblockchain 767430
```

Sending inscriptions with Sparrow Wallet

⚠️⚠️ Warning ⚠️⚠️

While it is highly recommended that you set up a bitcoin core node and run the `ord` software, there are certain limited ways you can send inscriptions out of Sparrow Wallet in a safe way. Please note that this is not recommended, and you should only do this if you fully understand what you are doing.

Using the `ord` software will remove much of the complexity we are describing here, as it is able to automatically and safely handle sending inscriptions in an easy way.

⚠️⚠️ Additional Warning ⚠️⚠️

Don't use your sparrow inscriptions wallet to do general sends of non-inscription bitcoin. You can setup a separate wallet in sparrow if you need to do normal bitcoin transactions, and keep your inscriptions wallet separate.

Bitcoin's UTXO model

Before sending any transaction it's important that you have a good mental model for bitcoin's Unspent Transaction Output (UTXO) system. The way Bitcoin works is fundamentally different to many other blockchains such as Ethereum. In Ethereum generally you have a single address in which you store ETH, and you cannot differentiate between any of the ETH - it is just all a single value of the total amount in that address. Bitcoin works very differently in that we generate a new address in the wallet for each receive, and every time you receive sats to an address in your wallet you are creating a new UTXO. Each UTXO can be seen and managed individually. You can select specific UTXO's which you want to spend, and you can choose not to spend certain UTXO's.

Some Bitcoin wallets do not expose this level of detail, and they just show you a single summed up value of all the bitcoin in your wallet. However, when sending inscriptions it is important that you use a wallet like Sparrow which allows for UTXO control.

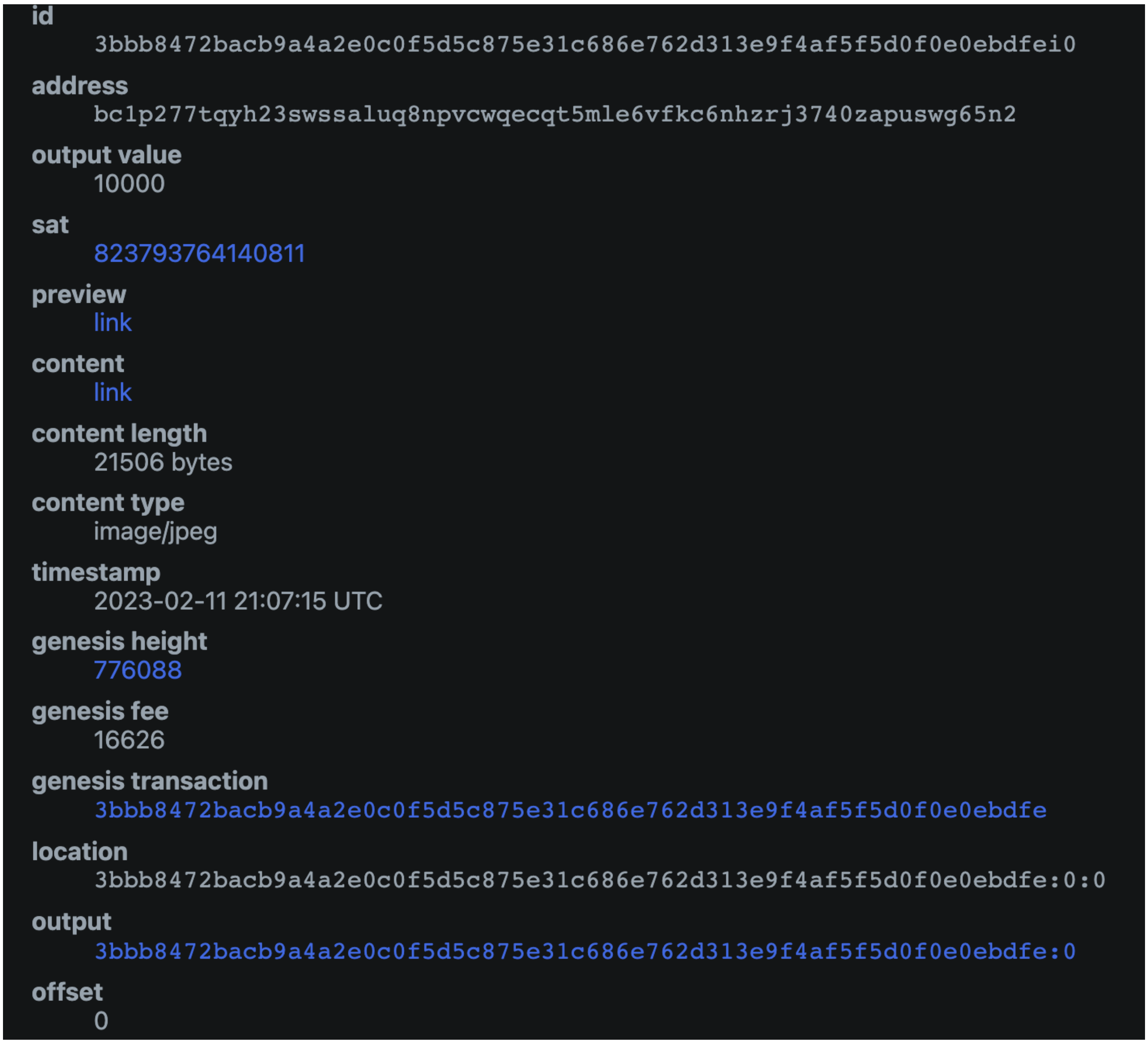
Inspecting your inscription before sending

Like we have previously described inscriptions are inscribed onto sats, and sats are stored within UTXOs. UTXO's are a collection of satoshis with some particular value of the number of satoshis (the output value). Usually (but not always) the inscription will be inscribed on the first satoshi in the UTXO.

When inspecting your inscription before sending the main thing you will want to check is which satoshi in the UTXO your inscription is inscribed on.

To do this, you can follow the Validating / Viewing Received Inscriptions described above to find the inscription page for your inscription on ordinals.com

There you will find some metadata about your inscription which looks like the following:



```
id
     3bbb8472bacb9a4a2e0c0f5d5c875e31c686e762d313e9f4af5f5d0f0e0ebdfei0
address
     bc1p277tqyh23swssaluq8npvcwqecqt5mle6vfkc6nhzrj3740zapuswg65n2
output value
     10000
sat
     823793764140811
preview
     link
content
     link
content length
     21506 bytes
content type
     image/jpeg
timestamp
     2023-02-11 21:07:15 UTC
genesis height
     776088
genesis fee
     16626
genesis transaction
     3bbb8472bacb9a4a2e0c0f5d5c875e31c686e762d313e9f4af5f5d0f0e0ebdfe
location
     3bbb8472bacb9a4a2e0c0f5d5c875e31c686e762d313e9f4af5f5d0f0e0ebdfe:0:0
output
     3bbb8472bacb9a4a2e0c0f5d5c875e31c686e762d313e9f4af5f5d0f0e0ebdfe:0
offset
     0
```

There is a few of important things to check here:

- The `output` identifier matches the identifier of the UTXO you are going to send
- The `offset` of the inscription is `0` (this means that the inscription is located on the first sat in the UTXO)
- the `output_value` has enough sats to cover the transaction fee (postage) for sending the transaction. The exact amount you will need depends on the fee rate you will select for the transaction

If all of the above are true for your inscription, it should be safe for you to send it using the method below.

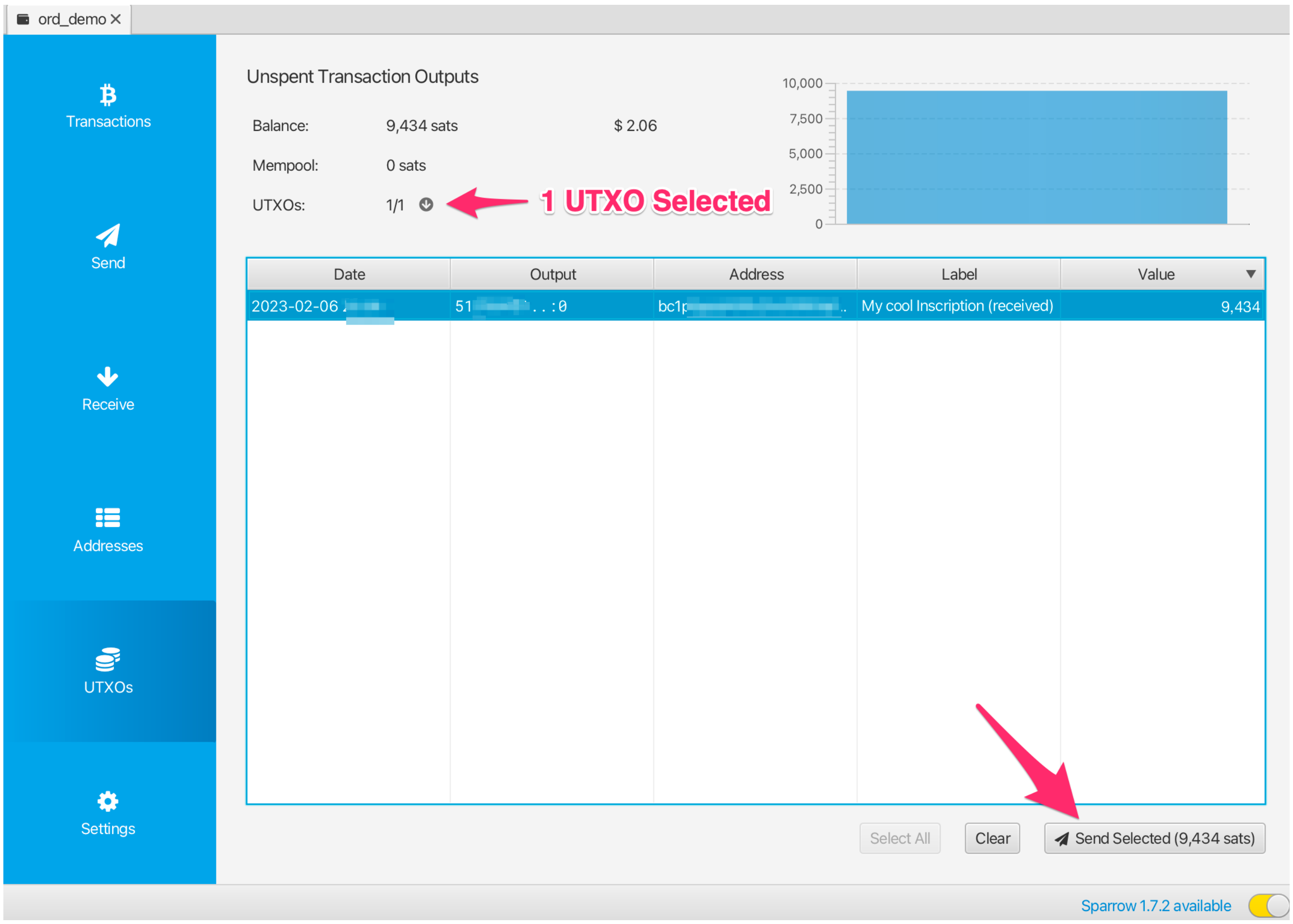
⚠️⚠️ Be very careful sending your inscription particularly if the `offset` value is not `0`. It is not recommended to use this method if that is the case, as doing so you could accidentally send your inscription to a bitcoin miner unless you know what you are doing.

Sending your inscription

To send an inscription navigate to the `UTXOs` tab, and find the UTXO which you previously validated contains your inscription.

If you previously froze the UXTO you will need to right-click on it and unfreeze it.

Select the UTXO you want to send, and ensure that is the *only* UTXO is selected. You should see `UTXOs 1/1` in the interface. Once you are sure this is the case you can hit `Send Selected`.



Unspent Transaction Outputs

Balance: 9,434 sats $ 2.06

Mempool: 0 sats

UTXOs: 1/1

Date	Output	Address	Label	Value
2023-02-06 [redacted]	51[redacted]..:0	bc1p[redacted]..	My cool Inscription (received)	9,434

You will then be presented with the transaction construction interface. There is a few things you need to check here to make sure that this is a safe send:

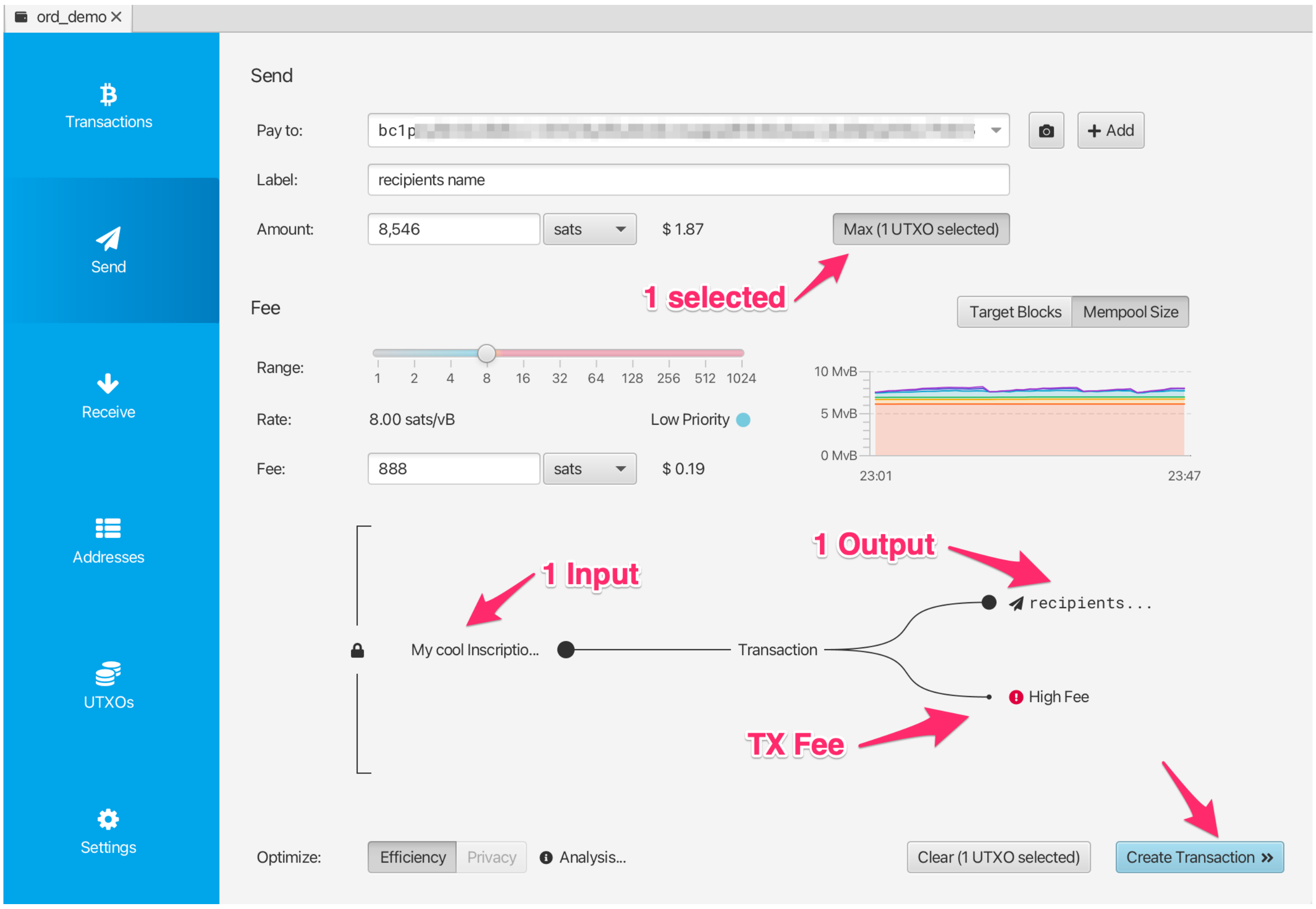
- The transaction should have only 1 input, and this should be the UTXO with the label you want to send
- The transaction should have only 1 output, which is the address/label where you want to send the inscription

If your transaction looks any different, for example you have multiple inputs, or multiple outputs then this may not be a safe transfer of your inscription, and you should abandon sending until you understand more, or can import into the `ord` wallet.

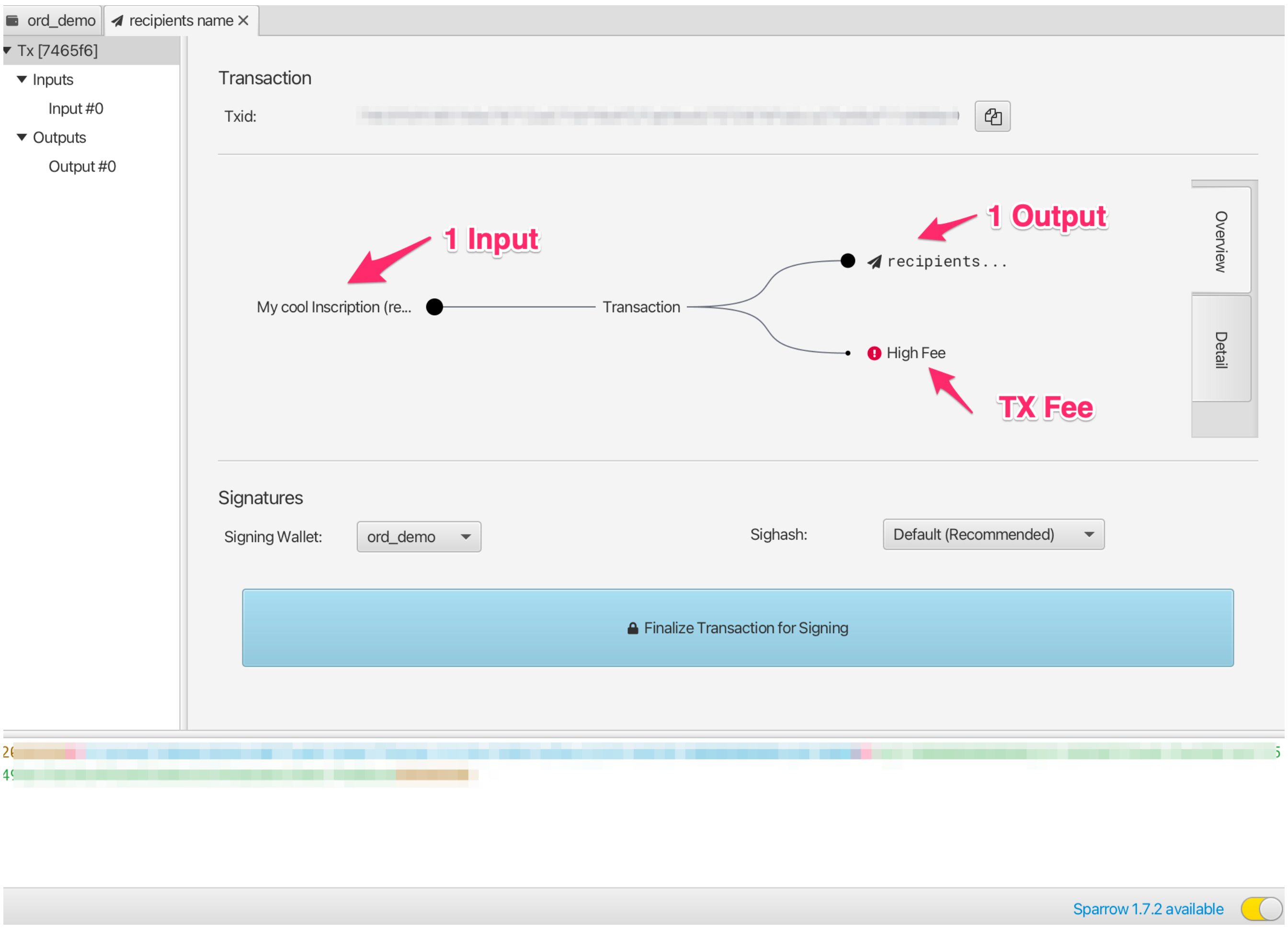
You should set an appropriate transaction fee, Sparrow will usually recommend a reasonable one, but you can also check mempool.space to see what the recommended fee rate is for sending a transaction.

You should add a label for the recipient address, a label like `alice address for inscription #123` would be ideal.

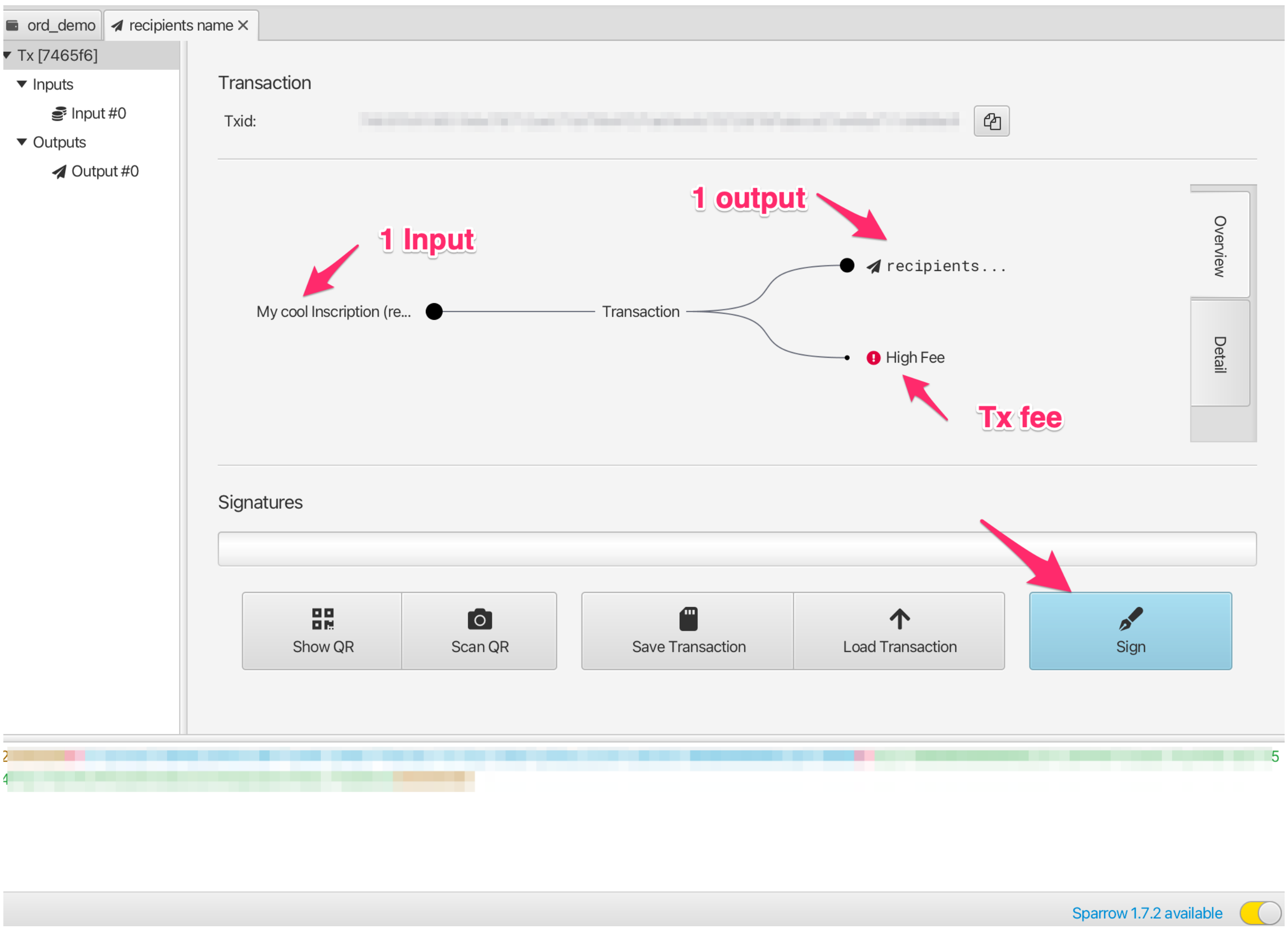
Once you have checked the transaction is a safe transaction using the checks above, and you are confident to send it you can click `Create Transaction`.



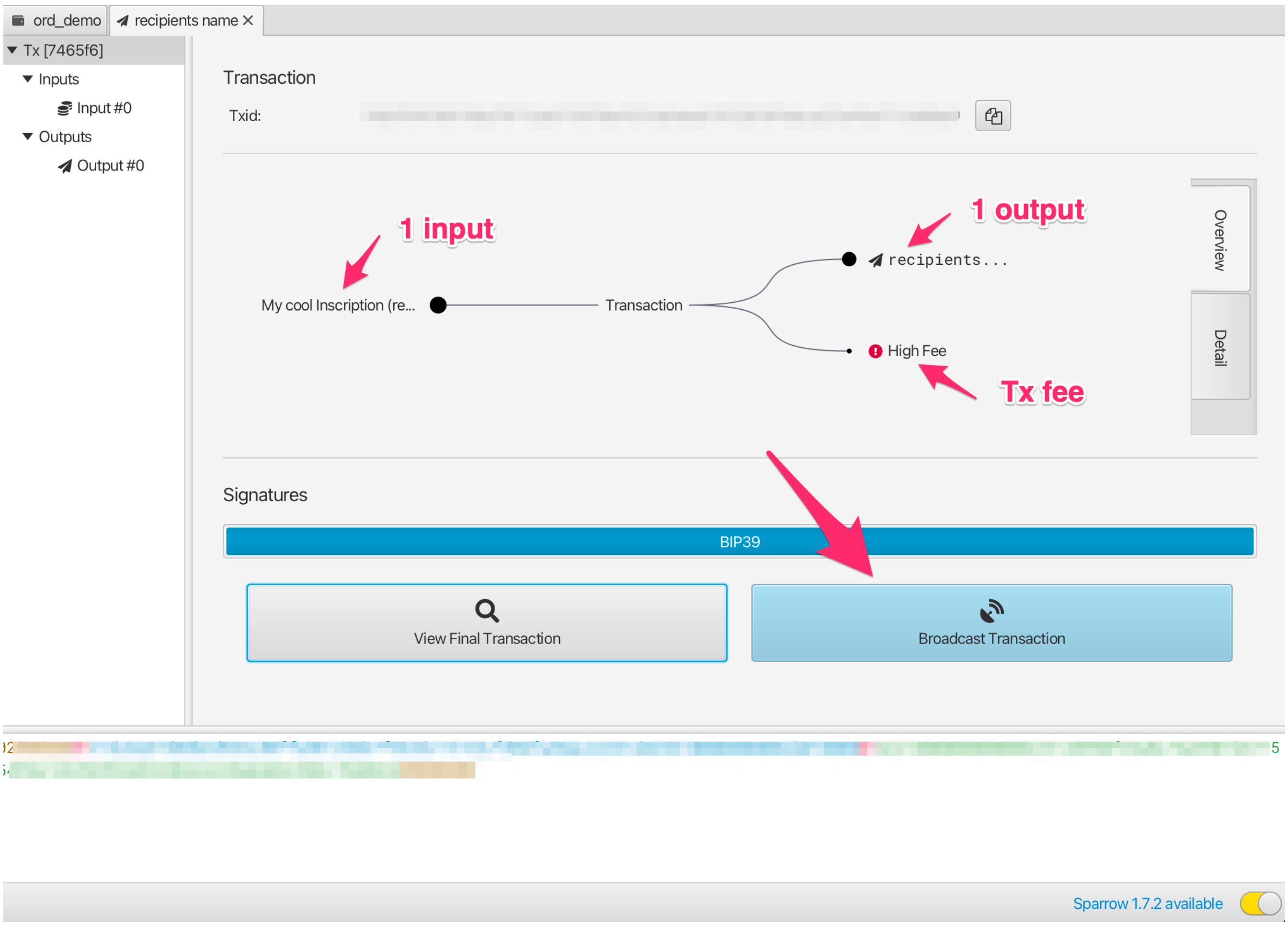
Here again you can double check that your transaction looks safe, and once you are confident you can click `Finalize Transaction for Signing`.



Here you can triple check everything before hitting `Sign`.



And then actually you get very very last chance to check everything before hitting `Broadcast Transaction`. Once you broadcast the transaction it is sent to the bitcoin network, and starts being propagated into the mempool.



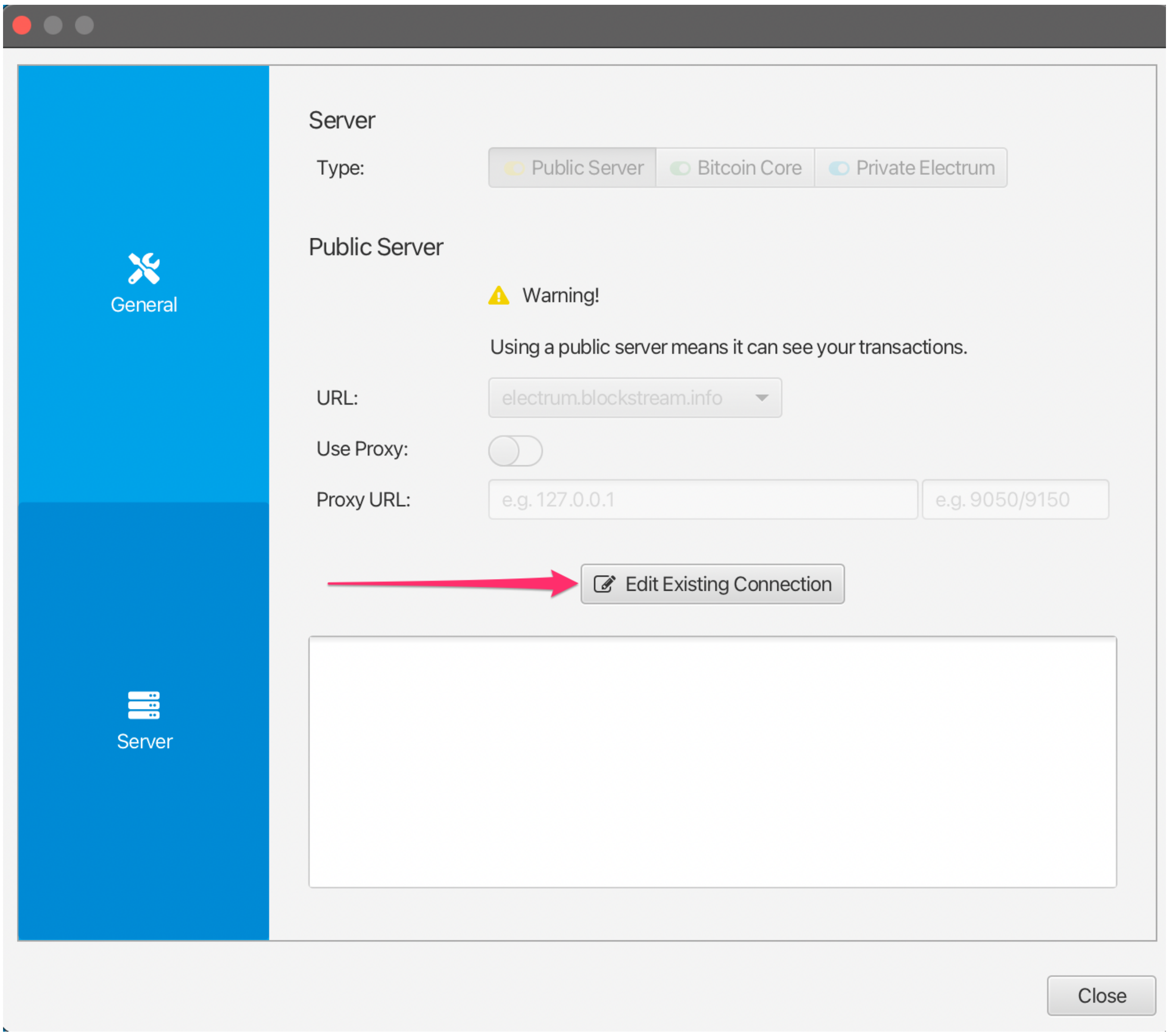
If you want to track the status of your transaction you can copy the `Transaction Id (Txid)` and paste that into mempool.space

Once the transaction has confirmed you can check the inscription page on ordinals.com to validate that it has moved to the new output location and address.

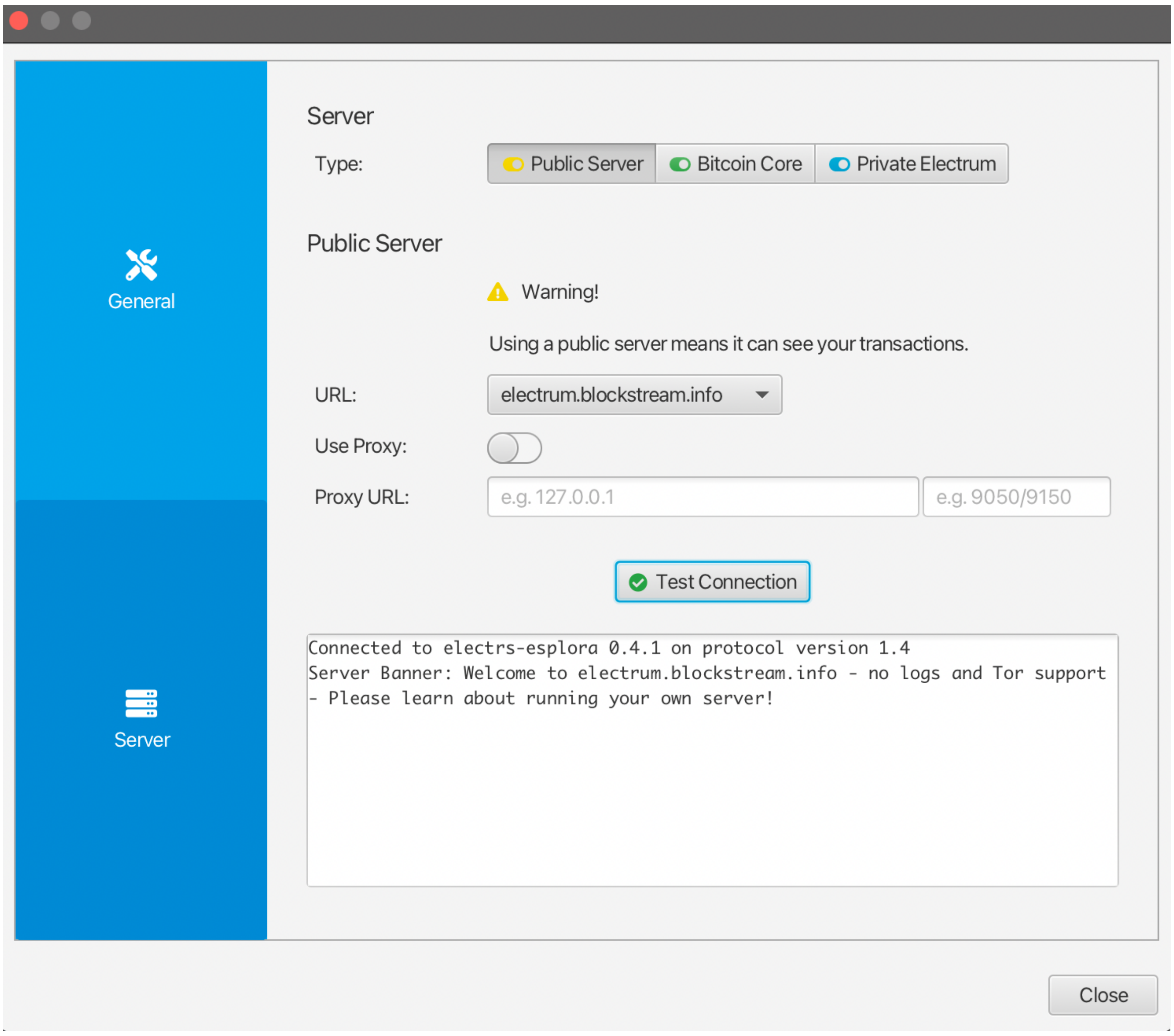
Troubleshooting

Sparrow wallet is not showing a transaction/UTXO, but I can see it on mempool.space!

Make sure that your wallet is connected to a bitcoin node. To validate this, head into the `Preferences`-> `Server` settings, and click `Edit Existing Connection`.



From there you can select a node and click `Test Connection` to validate that Sparrow is able to connect successfully.



Server

Type: Public Server

Public Server

Warning!

Using a public server means it can see your transactions.

URL: electrum.blockstream.info

Use Proxy:

Proxy URL: e.g. 127.0.0.1 e.g. 9050/9150

```
Connected to electrs-esplora 0.4.1 on protocol version 1.4
Server Banner: Welcome to electrum.blockstream.info - no logs and Tor support
- Please learn about running your own server!
```

Testing

Ord can be tested using the following flags to specify the test network. For more information on running Bitcoin Core for testing, see Bitcoin's developer documentation.

Most `ord` commands in inscriptions and explorer can be run with the following network flags:

Network	Flag
Testnet	`--testnet` or `-t`
Signet	`--signet` or `-s`
Regtest	`--regtest` or `-r`

Regtest doesn't require downloading the blockchain or indexing ord.

Example

Run bitcoind in regtest with:

```
bitcoind -regtest -txindex
```

Create a wallet in regtest with:

```
ord -r wallet create
```

Get a regtest receive address with:

```
ord -r wallet receive
```

Mine 101 blocks (to unlock the coinbase) with:

```
bitcoin-cli -regtest generatetoaddress 101 <receive address>
```

Inscribe in regtest with:

```
ord -r wallet inscribe --fee-rate 1 --file <file>
```

Mine the inscription with:

```
bitcoin-cli -regtest generatetoaddress 1 <receive address>
```

View the inscription in the regtest explorer:

```
ord -r server
```

By default, browsers don't support compression over HTTP. To test compressed content over HTTP, use the `--decompress` flag:

```
ord -r server --decompress
```

Testing Recursion

When testing out recursion, inscribe the dependencies first (example with p5.js):

```
ord -r wallet inscribe --fee-rate 1 --file p5.js
```

This should return a `inscription_id` which you can then reference in your recursive inscription.

ATTENTION: These ids will be different when inscribing on mainnet or signet, so be sure to change those in your recursive inscription for each chain.

Then you can inscribe your recursive inscription with:

```
ord -r wallet inscribe --fee-rate 1 --file recursive-inscription.html
```

Finally you will have to mine some blocks and start the server:

```
bitcoin-cli generatetoaddress 6 <receive address>
ord -r server
```

Moderation

`ord` includes a block explorer, which you can run locally with `ord server`.

The block explorer allows viewing inscriptions. Inscriptions are user-generated content, which may be objectionable or unlawful.

It is the responsibility of each individual who runs an ordinal block explorer instance to understand their responsibilities with respect to unlawful content, and decide what moderation policy is appropriate for their instance.

In order to prevent particular inscriptions from being displayed on an `ord` instance, they can be included in a YAML config file, which is loaded with the `--config` option.

To hide inscriptions, first create a config file, with the inscription ID you want to hide:

```
hidden:
- 0000000000000000000000000000000000000000000000000000000000000000i0
```

The suggested name for `ord` config files is `ord.yaml`, but any filename can be used.

Then pass the file to `--config` when starting the server:

```
ord --config ord.yaml server
```

Note that the `--config` option comes after `ord` but before the `server` subcommand.

`ord` must be restarted in to load changes to the config file.

`ordinals.com`

The `ordinals.com` instances use `systemd` to run the `ord server` service, which is called `ord`, with a config file located at `/var/lib/ord/ord.yaml`.

To hide an inscription on `ordinals.com`:

1. SSH into the server
2. Add the inscription ID to `/var/lib/ord/ord.yaml`
3. Restart the service with `systemctl restart ord`
4. Monitor the restart with `journalctl -u ord`

Currently, `ord` is slow to restart, so the site will not come back online immediately.

Reindexing

Sometimes the `ord` database must be reindexed, which means deleting the database and restarting the indexing process with either `ord index update` or `ord server`. Reasons to reindex are:

1. A new major release of ord, which changes the database scheme
2. The database got corrupted somehow

The database `ord` uses is called redb, so we give the index the default file name `index.redb`. By default we store this file in different locations depending on your operating system.

Platform	Value	Example
Linux	`$XDG_DATA_HOME`/ord or `$HOME`/.local/share/ord	/home/alice/.local/share/ord
macOS	`$HOME`/Library/Application Support/ord	/Users/Alice/Library/Application Support/ord
Windows	`{FOLDERID_RoamingAppData}`\ord	C:\Users\Alice\AppData\Roaming\ord

So to delete the database and reindex on MacOS you would have to run the following commands in the terminal:

```
rm ~/Library/Application Support/ord/index.redb
ord index update
```

You can of course also set the location of the data directory yourself with `ord --data-dir <DIR> index update` or give it a specific filename and path with `ord --index <FILENAME> index update`.

Ordinal Bounty Hunting Hints

- The `ord` wallet can send and receive specific satoshis. Additionally, ordinal theory is extremely simple. A clever hacker should be able to write code from scratch to manipulate satoshis using ordinal theory in no time.
- For more information about ordinal theory, check out the FAQ for an overview, the BIP for the technical details, and the ord repo for the `ord` wallet and block explorer.
- Satoshi was the original developer of ordinal theory. However, he knew that others would consider it heretical and dangerous, so he hid his knowledge, and it was lost to the sands of time. This potent theory is only now being rediscovered. You can help by researching rare satoshis.

Good luck and godspeed!

Ordinal Bounty 0

Criteria

Send a sat whose ordinal number ends with a zero to the submission address:

✅: 1857578125803250

❌: 1857578125803251

The sat must be the first sat of the output you send.

Reward

100,000 sats

Submission Address

`1PE7u4wbDP2RqfKN6geD1bG57v9Gj9FXm3`

Status

Claimed by @count_null!

Ordinal Bounty 1

Criteria

The transaction that submits a UTXO containing the oldest sat, i.e., that with the lowest number, amongst all submitted UTXOs will be judged the winner.

The bounty is open for submissions until block 753984—the first block of difficulty adjustment period 374. Submissions included in block 753984 or later will not be considered.

Reward

200,000 sats

Submission Address

`145Z7PFHyVrwiMWwEcUmDgFbmUbQSU9aap`

Status

Claimed by @ordinalsindex!

Ordinal Bounty 2

Criteria

Send an uncommon sat to the submission address:

✅: 347100000000000

❌: 6685000001337

Confirm that the submission address has not received transactions before submitting your entry. Only the first successful submission will be rewarded.

Reward

300,000 sats

Submission Address

`1Hyr94uypwWq5CQffaXHvwUMEyBPp3TUZH`

Status

Claimed by @utxoset!

Ordinal Bounty 3

Criteria

Ordinal bounty 3 has two parts, both of which are based on *ordinal names*. Ordinal names are a modified base-26 encoding of ordinal numbers. To avoid locking short names inside the unspendable genesis block coinbase reward, ordinal names get *shorter* as the ordinal number gets *longer*. The name of sat 0, the first sat to be mined is `nvtdijuwxlp` and the name of sat 2,099,999,997,689,999, the last sat to be mined, is `a`.

The bounty is open for submissions until block 840000—the first block after the fourth halving. Submissions included in block 840000 or later will not be considered.

Both parts use frequency.tsv, a list of words and the number of times they occur in the Google Books Ngram dataset. filtered to only include the names of sats which will have been mined by the end of the submission period, that appear at least 5000 times in the corpus.

`frequency.tsv` is a file of tab-separated values. The first column is the word, and the second is the number of times it appears in the corpus. The entries are sorted from least-frequently occurring to most-frequently occurring.

`frequency.tsv` was compiled using this program.

To search an `ord` wallet for sats with a name in `frequency.tsv`, use the following `ord` command:

```
ord wallet sats --tsv frequency.tsv
```

This command requires the sat index, so `--index-sats` must be passed to ord when first creating the index.

Part 0

Rare sats pair best with rare words.

The transaction that submits the UTXO containing the sat whose name appears with the lowest number of occurrences in `frequency.tsv` shall be the winner of part 0.

Part 1

Popularity is the font of value.

The transaction that submits the UTXO containing the sat whose name appears with the highest number of occurrences in `frequency.tsv` shall be the winner of part 1.

Tie Breaking

In the case of a tie, where two submissions occur with the same frequency, the earlier submission shall be the winner.

Reward

- Part 0: 200,000 sats
- Part 1: 200,000 sats
- Total: 400,000 sats

Submission Address

`17m5rvMpi78zG8RUpCRd6NWWMJtWmu65kg`

Status

Unclaimed!