```
BIP: 154
Layer: Peer Services
Title: Rate Limiting via peer specified challenges
Author: Karl-Johan Alm <karljohan-alm@garage.co.jp>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0154
Status: Withdrawn
Type: Standards Track
Created: 2017-04-12
License: BSD-2-Clause
```

Abstract

An anti-DoS system which provides additional service for peers which perform proof of work.

Definitions

- **POW** : a proof of work using some arbitrary algorithm, such as SHA256
- **challenge** : a problem in the form of a POW specification and other data
- **solution** : a set of inputs which solve a given challenge
- **free connection slot** : an inbound connection slot that does not require POW
- **POW connection slot** : an inbound connection slot that requires POW
- **SPH** : Special Purpose Hardware, such as an ASIC chip
- **GPH** : General Purpose Hardware, such as a desktop computer
- **Work** : A measurement of optimized average resources (clock cycles, memory, ...) required to perform a single attempt at solving a given POW algorithm on GPH

Motivation

The Bitcoin network has a maximum number of inbound and outbound connections (125). It is trivial and relatively cheap to flood the network with connections via dummy nodes. Such an attack would result in (1) nodes evicting some other nodes in order to facilitate the new connection, and (2) nodes' ability to connect to each other being severely hampered. In this state, the network is vulnerable to e.g. a Sybil attack.

While the network is under pressure as in the above case, nodes could allow incoming connections anyway by requiring that the incoming peer performs some form of proof of work, to prove that they are not simply spamming the network. This would severely ramp up the costs of a Sybil attack, as the attacker would now have to perform proof of work for each node, beyond the free slots.

However, using the "standard" double-SHA256 POW algorithm in use by Bitcoin nodes to generate blocks means attackers can use special-purpose hardware

to greatly accelerate the POW solving process. To counter this, the proof weight would have to be raised, but this would mean standard nodes would need to solve unacceptably costly challenges for simple operation. Therefore, a different proof of work which is arguably less sensitive to special-purpose hardware implementations is introduced. As this is not consensus sensitive, additional POW algorithms may be added in the future.

Specification

A peer that supports Proof of Work Rate Limiting defines two maximums:

- max connections, from which the maximum inbound connections is calculated as `nMaxConnections - (nMaxOutbound + nMaxFeeler)`
- POW connection slots, which define how many of the above inbound connections require a POW challenge

The peer must interpret two new network peer message types, `challenge` and `solution`.

In addition, the network handshake sequence must be altered slightly to facilitate the exchange of challenges and/or solutions:

- when a node connects, it may send a `solution` message prior to the `version`
- if it does, and
  - the solution satisfies the local node, it is given a connection, but if
  - the solution does not satisfy the local node (unknown, wrong, ...), a new `challenge` is sent and the connection is closed
- if it does not, and it is marked as needing to do POW, a `challenge` is sent and the connection is closed

This means nodes will be disconnected after receiving the challenge. It is then up to the individual nodes whether they solve the challenge and reconnect, or discard it and find a different peer (or wait for the peer to have an open free slot).

POW Identifiers

There are two POW identifiers currently. When a new identifier is introduced, it should be added with an increment of 1 to the last identifier in the list. When an identifier is deprecated, its status should be changed to `Deprecated` but it should retain its place in the list indefinitely.

ID	Algorithm Name	Work	Param size	Solution size	Provably Secure
1	sha256	11k cycles	11+ bytes	0, 4 or 8 bytes	Yes
2	cuckoo-cycle	ss 28: 150G cycles / ~48M RAM	6+ bytes	168 bytes	No

sha256 Properties:

Property	Value
Solution probability	`sum((1/2)^i*(1-targetBE[i]))`

Challenge format:

Range	Field Name	Data Type	Description
0	config_length	varint	Length of configuration part; always 9
1..4	target	uint32	Difficulty target, in the form of a compact size (like nBits in blocks).
5	nonce_size	uint8	Size of nonce in bytes; must be 0 (no nonce), 4 (uint32) or 8 (uint64)
6..9	nonce_offset	uint32	Location of nonce value in target
10..	payload_length	varint	Length of the input data
..	payload	byte array	Input data

Solution format:

Range	Field Name	Data Type	Description
0..	nonce	uint32/64, or data	Nonce value that satisfies challenge; for zero-byte nonces, this is v

Note: SHA256 works in two "modes".

1. One is where the task is to insert a nonce into an existing data block so that the hash of the data block matches a given target; this is the conventional block proof of work behavior.
2. The other is where the whole or parts of the data chunk are given as input (a "big nonce"). In this case, the internal nonce size is zero bytes, and the task is simply to check whether the hash of the data matches the target. If it does not, there is no way to find a solution except by getting different input from the generator (a successor algorithm). This mode is used when SHA256 is a predecessor to another algorithm.

Additional notes:

- The initial nonce value (when present) for finding a suitable digest should be randomized, or a challenger may deliberately pick a challenge with "poor" outcomes to fool a node into spending more than predicted time solving.

cuckoo-cycle Properties:

Property	Value
Solution probability	`~1.0` for sizeshift=28, proofsize-min:-max=12:228

Challenge format:

Range	Field Name	Data Type	Description
0	config_length	varint	Length of configuration part; always 5
1	sizeshift	uint8	Size shift; must be equal to 28, but may be variable in the future
2..3	proofsize-min	uint16	Minimum number of edges in cycle; must be even and greater than o
4..5	proofsize-max	uint16	Maximum number of edges in cycle; must be even, greater than or eq
6	payload_length	varint	Length of the input data; must be 76, but may be variable in the fut
7..	payload	byte array	Input data

Solution format:

Range	Field Name	Data Type	Description
0..3	nonce	uint32	Nonce which is appended to challenge payload to form solution graph
4..171	edges	uint32 array	42 values which identify each of the 42 edges in the cycle

Additional notes:

- The initial nonce value used for finding a graph with a suitable solution should be randomized, or a challenger may deliberately pick a challenge with "poor" outcomes to fool a node into spending more than predicted time solving.
- Further information on the recommended challenge parameters can be found here: http://bc-2.jp/cuckoo-profile.pdf

Purpose Identifiers

There is only one Purpose Identifier currently. In the future, more Purpose Identifiers could be added for at-DoS-risk operations, such as bloom filters. When a new identifier is introduced, it should be added with an increment of 1 to the last identifier in the list. When an identifier is deprecated, its status should be changed to `Deprecated` but it should retain its place in the list indefinitely.

ID	Purpose Name	Description	Status
1	connect	Establish peer to peer connection	Active

Challenges

Challenges consist of one or several chained POW identifiers with accompanying parameters, as well as indicators for the purpose of the challenge, and a signature that lets the node verify the challenge authenticity.

After creating a challenge, the node signs it, delivers it to the peer, then discards it. When a node provides a solution to a challenge, the node verifies the signature and adds the challenge hash to a list of solved challenges along with its expiration time. This list is pruned on each insertion, removing any expired challenges.

If nodes needed to keep track of unsolved challenges, an attacker could hypothetically swarm a node, causing a DoS by having it generate so many challenges that it runs out of memory and crashes. By signing and discarding challenges, a node only has to retain challenges that were solved, and which have not yet expired, effectively DoS- protecting the node via the challenges themselves.

The `challenge` message type

A challenge consists of four parts: the POW specification, a purpose identifier, an expiration date, and a signature. The POW specification contains a list of tuples containing a POW identifier and corresponding POW parameters.

- Each POW identifier specifies a POW algorithm (see POW Identifiers)
- The POW parameters define the inputs and requirements of the POW algorithm
- The purpose identifier specifies the purpose of the challenge (see Purpose Identifiers)
- The expiration date is a UNIX timestamp indicating when the challenge expires
- The signed content should contain a signature of the hash `SHA256(SHA256(pow-count || pow-id || pow-params || ... || purpose-id || expiration))`, i.e. the hash of the entire challenge except for the signature length and data.

Field Size	Description	Data type	Description
1 byte	pow-count	uint8	Number of POW algorithms in the range [1..255]
4 bytes	pow-id	uint32	The POW algorithm to solve the problem with
?	pow-params	?	The POW parameters and payload
...	...	...	pow-id and pow-params for algorithms 2 and beyond
4 bytes	purpose-id	uint32	The purpose of the challenge
8 bytes	expiration	int64	Expiration UNIX timestamp
?	sign-len	varint	The length of the signature
?	sign	byte array	The signature data

For POW specifications with a pow-count > 1, the output of the succeeding POW algorithm will be appended to the input of the predecessor for all POW algorithms

except the last one. Normally mid-layer (all but the last) POW algorithms have a zero-length input. Example implementing sha256(cuckoo-cycle):

Range	Field Name	Value	Comment
0	pow-count	2	Two POW algorithms
1..4	pow-id	1	sha256
5	pow-params (config_length)	9	
6..9	pow-params (target)	0x207fffff	Resulting hash must be <= the compact hash 0x
10	pow-params (nonce_size)	0	No nonce
11..14	pow-params (nonce_offset)	0	--
15..18	pow-params (payload_length)	0	0 byte input (turns into 32 byte input from succ
19..22	pow-id	2	cuckoo-cycle
23	pow-params (config_length)	8	
24	pow-params (sizeshift)	28	
25..26	pow-params (proofsize-min)	12	
27..28	pow-params (proofsize-max)	228	
29	pow-params (payload_length)	76	76 byte input
30..105	pow-params	(random data)	A randomized challenge of 76 bytes
106..109	purpose-id	1	Purpose is a peer-to-peer connection
110..117	expiration	1491285696	Expiration is April 4 2017, 15:01:36 (JST)
118	sign-len	71	71 byte signature
119..189	sign	(signature)	Signature of above challenge

(* Compact 0x207fffff = 0x7fffff0000000000000000000000000000000000000000000000000000000000000000.)

The above should be interpreted as SHA256(cuckoo-cycle(random data || nonce)) < 0x7fffff0000000000000000000000000000000000000000000000000000000000000000.

- Run cuckoo-cycle on random data || nonce; increment nonce until solution is found, then
  - Run SHA256 on 32 byte digest from above; if less than 0x7fffff0000000000000000000000000000000000000000000000000000000000000000,
    - Mark solved.
- Otherwise loop back and increase nonce and continue finding solutions

The `solution` message type

A solution consists of two parts: the entire challenge, and solution parameters:

- The challenge must match the given challenge up to and including the signature bytes
- The solution parameters must form a valid solution to each POW step in the challenge

Field Size	Description	Data type	Description
1 byte	pow-count	uint8	Number of POW algorithms in the range [1..255]
4 bytes	pow-id	uint32	The POW algorithm used to solve the problem
?	pow-params	?	The input to the POW solver for the above algorithm
...	...	...	pow-id and pow-params for algorithms 2 and beyond
4 bytes	purpose-id	uint32	The purpose of the challenge
8 bytes	expiration	int64	Expiration UNIX timestamp
?	sign-len	varint	The length of the signature
?	sign	byte array	The signature data
?	solution	?	The solution to the challenge

Note that the solution contains the parameters for the last algorithm only. For each algorithm except the last one, the input is derived from the output of the successor. Example solution:

Range	Name	Value	Description
0	length	4	The input to the innermost POW is 4 bytes in length
1..4	nonce32	0x12345	The nonce used as input is 0x12345

The above example will provide a single nonce for the inner POW. For the SHA256(SHA256(challenge data || nonce32)) case, the solution would claim that SHA256(SHA256(challenge data || 0x00012345)) solves the challenge.

Signing and Verifying Challenges

Below is a suggestion for how to sign a challenge. The implementation generates a new, random key-pair at launch and uses that to sign all challenges until the node is shutdown.

Signing a Challenge

1. (first time) Create a new random key-pair `key` and `pubkey` and keep these around until shutdown
2. (second+ time) Fetch `key` created above
3. Create a double-SHA256 `sighash` of the challenge in serialized form up until and including the expiration bytes
4. Create a signature `sign` of `sighash` using `key`
5. Append `varint(len(sign))` and `sign` to challenge

Verifying a Challenge

1. Fetch `pubkey` and declare failure if not defined (that means we never issued a challenge)

2. Create a double-SHA256 `sighash` of the challenge provided with the solution up until and including the expiration bytes
3. Verify `sighash` is not known, and add it to known hashes along with its expiration date for pruning purposes
4. Set `sign` to the signature included in the challenge
5. Verify the signature `sign` using `pubkey` and `sighash`
6. Check that the solution solves the challenge

Note that a list of known hashes should be kept and pruned of expired challenges on verification. Otherwise nodes may reuse the same solution repeatedly up until its expiration.

Difficulty and Cost

Estimating Challenge Cost

Nodes need to be able to make a judgement call on whether solving a given challenge is worth their efforts. If a challenge is expected to take so much time that it would expire before being solved (on average), it should be immediately discarded. Beyond this, a threshold should be established for nodes based on their "value" to the node, which is inversely proportional to the current number of connections as a function of uptime, with arbitrary modifiers (a whitelisted node or a node added via -addnode has a much higher threshold).

It is hard to obtain an accurate value for `cycles_per_second`, and as such a fixed value of 1700000000=1.7e9 may be used.

Given a threshold `t`, calculate the estimated work required to solve the challenge as follows:

1. Define `p(alg)` as the probability that an attempt at finding a solution given the algorithm `alg` succeeds
2. Define `w(alg)` as the work parameter of the algorithm `alg`.
3. Let `Wc ← 0, Wm ← 1, Wi ← 1`
4. For each proof of work `pow` in the POW specification:
   (a) Let `p ← p(pow)`, `w ← w(pow)`
   (b) Update `Wc ← Wc + w_cycles`, `Wi ← Wi * 1/p`, `Wm ← Wm + w_ram`
5. Let `eta ← (Wc * Wi) / cycles_per_second`
6. If `date() + eta >= expiration`, discard challenge
7. If `eta > t`, discard challenge

Example: `SHA256(cuckoo-cycle(...)) < 0x7fffff0000000000000000000000000000000000000000000000000000000000`

1. `p(cuckoo-cycle) = 1,  p(sha256, 0x7fffff000...) ~= (1/2)^1 = 1/2`
2. `w(cuckoo-cycle) = (1.5e11 cycles, 5e7 ram), w(sha256, 0x7fffff000...) = (11e3 cycles)`
3. `Wc = 0, Wm = 1, Wi = 1`

   (a) `p = p(cuckoo-cycle) = 1, w = w(cuckoo-cycle) = (1.5e11 cycles, 5e7 ram)`
   (b) `Wc = 0 + 1.5e11 = 1.5e11, Wi = 1 * 1 = 1, Wm = 1 + 5e7 = 5e7`
   (c) `p = p(sha256) = 1/2, w = w(sha256) = (11e3 cycles)`
   (d) `Wc = 1.5e11 + 11e3 ~= 1.5e11, Wi = 1 * 2 = 2, Wm = 5e7 + 0 = 5e7`
4. `eta = (1.5e11 * 2) / cycles_per_second` = `7.5e10 / 1.7e9` = 44.1 seconds

TODO: Determine how memory impacts threshold.

To avoid other nodes dropping our challenges due to early expiration, we use a fairly generous expiration based on the pressure value

```
expiration = date() + 600 * (1 + pressure)
```

which means the expiration is 10 minutes for the weakest challenge, and gradually rises to 20 minutes for the hardest one.

Establishing Difficulty Parameters

The difficulty setting for the network should change based on connection slot availability. The amount of pressure on the network in the sense of connection slot availability is proportional to the number of established connections over the number of total available connections. This can be locally approximated by a node to the number of local connections compared to the local connection maximum.

In other words, the network pressure can be approximated by any node as `connections / max` and the difficulty can be based on e.g. `(connections - free) / pow_slots`.

The challenge difficulty parameters can be set based on this, where 0.0 means "low pressure" and 1.0 means "maximum pressure". The `GetPressure` method below gives 0.0 at 67 connections (for a 50 POW slot set up), and hits the 1.0 mark at `(nMaxConnections - nMaxOutbound - nMaxFeeler)`, incrementing by 0.02 for each new connection:

```
int nMaxInbound = nMaxConnections - (nMaxOutbound + nMaxFeeler + nPOWConnectionSlots);
return ((double)GetNodeCount(CONNECTIONS_ALL) - nMaxInbound) / nPOWConnectionSlots;
```

An example of difficulty for a SHA256(Cuckoo-Cycle) specification would be based on a desired probability of a random SHA256 digest matching a given target:

```
prob_target = 1 / (1 + pressure^2 * 15)
```

This would result in probability targets according to the table below, for varying pressures (where the pressure is in the range [0..1]):

pressure	prob_target	solution time sha256(cc)
0.0	1.00	00:45
0.1	0.87	00:51
0.2	0.63	01:11
0.3	0.43	01:45
0.4	0.29	02:32
0.5	0.21	03:32
0.6	0.16	04:46
0.7	0.12	06:13
0.8	0.09	07:54
0.9	0.08	09:48
1.0	0.06	11:55

Cuckoo Cycle

Cuckoo Cycle[1] is a "graph-theoretic proof-of-work system, based on finding small cycles or other structures in large random graphs."

It is memory hard, which greatly increases the complexity and cost of producing dedicated (special purpose) hardware, an ideal property for an anti-DoS system.

The implementation specifics of the algorithm are beyond the scope of this BIP, but the github repository[2] has several reference implementations in various languages.

Compatibility

This proposal is backward compatible. Non-supporting peers will ignore the `challenge` message and be disconnected, as if they hit the peer connection limit as normal.

Reference implementation

https://github.com/kallewoof/bitcoin/pull/2 (https://github.com/kallewoof/bitcoin/tree/pow-connection-slots)

References

- [1] Cuckoo Cycle https://github.com/tromp/cuckoo/blob/master/doc/cuckoo.pdf?raw=true
- [2] Cuckoo Cycle github https://github.com/tromp/cuckoo

Test vectors

Cuckoo-Cycle

Cuckoo Cycle header (76 bytes):

```
00..1f   68a639cb 3deab5b6 23054d60 e7856037 8afa0f31 4f08dec1 6cc4ec4f d9bef1ff
20..3f   468af883 c6c9c3d5 4260087a 046d12a0 7cc3988f 9ff2957a 384de8ed db75b037
40..4b   798d1073 214b7ea6 954f1b3a
```

Example solution nonce: 0 (`00000000`)

Solution edges (16 number of 32-bit unsigned integers, read horizontally from top left):

```
550b1100 0fc89a00 45034401 ddfce701 08da0e02 6ccc5703 06fe8404 1d3f8504
559e3e05 d41a9905 17075206 97cfa006 59e50d07 7bd71f07 13fe2607 14493007
```

SHA256(Cuckoo-Cycle)

SHA256 target: `0x205fffff`

Cuckoo Cycle header (76 bytes, same as above):

```
00..1f   68a639cb 3deab5b6 23054d60 e7856037 8afa0f31 4f08dec1 6cc4ec4f d9bef1ff
20..3f   468af883 c6c9c3d5 4260087a 046d12a0 7cc3988f 9ff2957a 384de8ed db75b037
40..4b   798d1073 214b7ea6 954f1b3a
```

Example solution nonce: 0 (`00000000`)

SHA256 input (cuckoo-cycle nonce + solution):

```
00000000
550b1100 0fc89a00 45034401 ddfce701 08da0e02 6ccc5703 06fe8404 1d3f8504
559e3e05 d41a9905 17075206 97cfa006 59e50d07 7bd71f07 13fe2607 14493007
```

SHA256 hash: `262c8558c7c589b19b3d513abf5fcb15162745473e603f0146889ceff750bcc3`

Must be less than: `5fffff0000000000000000000000000000000000000000000000000000000000`

Serialized challenge example

```
020100000009ffff5f2000000000000002000000051c0c00e4004c68a639cb3deab5b623054d60e7
8560378afa0f314f08dec16cc4ec4fd9bef1ff468af883c6c9c3d54260087a046d12a07cc3988f9f
f2957a384de8eddb75b037798d1073214b7ea6954f1b3a01000000a49d0659000000004730450221
0095fc5fafe2032097c4d12a8901401cda297aad614e16f23ec42d4b78955856c002206ab7ada4ac
8f6fa9d5bd7cd06f9ba89587a28e14cea14e7f8f8d5ab851541791
```

Hex	Description
`0x02`	Two proofs of work
`0x01000000`	Proof of work ID = 1 (SHA256)
`0x09`	Config is 9 bytes

Hex	Description
0xffff5f20	SHA256: Compact target = 0x205fffff
0x00	SHA256: Nonce size is 0 bytes
0x00000000	SHA256: Nonce offset is 0
0x00	Payload is 0 bytes
0x02000000	Proof of work ID = 2 (cuckoo-cycle)
0x05	Config is 5 bytes
0x1c	Size shift is 28
0x0c00	Proof size min is 12
0xe400	Proof size max is 228
0x4c	Payload is 76 bytes
0x68a639cb3deab5b623054d60e7856037	Payload
0x8afa0f314f08dec16cc4ec4fd9bef1ff	
0x468af883c6c9c3d54260087a046d12a0	
0x7cc3988f9ff2957a384de8eddb75b037	
0x798d1073214b7ea6954f1b3a	
0x01000000	Purpose ID = 1 (PURPOSE_CONNECT)
0xa49d065900000000	UNIX timestamp 1493605796
0x47	71 byte signature
0x304502210095fc5fafe2032097c4d12a	Signature data
0x8901401cda297aad614e16f23ec42d4b	
0x78955856c002206ab7ada4ac8f6fa9d5	
0xbd7cd06f9ba89587a28e14cea14e7f8f	
0x8d5ab851541791	

Serialized solution example

```
020100000009ffff5f2000000000000002000000051c0c00e4004c68a639cb3deab5b623054d60e7
8560378afa0f314f08dec16cc4ec4fd9bef1ff468af883c6c9c3d54260087a046d12a07cc3988f9f
f2957a384de8eddb75b037798d1073214b7ea6954f1b3a01000000a49d0659000000004730450221
0095fc5fafe2032097c4d12a8901401cda297aad614e16f23ec42d4b78955856c002206ab7ada4ac
8f6fa9d5bd7cd06f9ba89587a28e14cea14e7f8f8d5ab8515417914400000000550b11000fc89a00
45034401ddfce70108da0e026ccc570306fe84041d3f8504559e3e05d41a99051707520697cfa006
59e50d077bd71f0713fe260714493007
```

Note that the first 187 bytes are identical to the challenge above.

Hex	Description
0x0201..1791	Challenge
0x44	Solution is 68 bytes long
0x00000000	The cuckoo cycle nonce is 0
0x550b11000fc89a0045034401ddfce701	Cycle edges 0..3
0x08da0e026ccc570306fe84041d3f8504	Cycle edges 4..7
0x559e3e05d41a99051707520697cfa006	Cycle edges 8..11

Hex	Description
`0x59e50d077bd71f0713fe260714493007`	Cycle edges 12..15

Cuckoo-Cycle Example 2

Cuckoo Cycle header (76 bytes):

```
00..1f   3c1e3ee5 c799b7e9 92bcccbb 8985979d cb8dd229 b8d0db06 e677d00b b3a43c88
20..3f   ef8596a7 7cbd1dda 23b0a0b8 4bdf6084 d7aa28dd bd5e91b5 11b3578c baf92707
40..4b   c940b051 a0759b3f 80c5fb65
```

Example solution nonce: 4 (`04000000`)

Solution edges (22 number of 32-bit unsigned integers, read horizontally from top left):

```
5a013700 7074ce00 e3dbeb00 e88f7901 06d71d02 984d3d02 091b5002 378a8e02
90a6d202 b3c67003 757cb703 44d9cf03 297f2004 8e76a604 67e44a05 7b077405
634f8405 23e88c05 0d887606 109d3e07 c4bdcd07 3db2d407
```

SHA256(Cuckoo-Cycle)

SHA256 target: `0x2021642c`

Cuckoo Cycle header (76 bytes, same as above):

```
00..1f   3c1e3ee5 c799b7e9 92bcccbb 8985979d cb8dd229 b8d0db06 e677d00b b3a43c88
20..3f   ef8596a7 7cbd1dda 23b0a0b8 4bdf6084 d7aa28dd bd5e91b5 11b3578c baf92707
40..4b   c940b051 a0759b3f 80c5fb65
```

Example solution nonce: 4 (`04000000`)

SHA256 input (cuckoo-cycle nonce + solution):

```
04000000
5a013700 7074ce00 e3dbeb00 e88f7901 06d71d02 984d3d02 091b5002 378a8e02
90a6d202 b3c67003 757cb703 44d9cf03 297f2004 8e76a604 67e44a05 7b077405
634f8405 23e88c05 0d887606 109d3e07 c4bdcd07 3db2d407
```

SHA256 hash: `08210561257e26776135ec1cb92cfe17f46803613c0bdc02043e5545b18556ce`

Must be less than: `21642c0000000000000000000000000000000000000000000000000000000000`

Serialized challenge example

```
0201000000092c64212000000000000002000000051c0c00e4004c3c1e3ee5c799b7e992bcccbb89
85979dcb8dd229b8d0db06e677d00bb3a43c88ef8596a77cbd1dda23b0a0b84bdf6084d7aa28ddbd
5e91b511b3578cbaf92707c940b051a0759b3f80c5fb650100000024aa0659000000004630440220
0edfb5c4812a31d84cbbd4b24e631795435a0d16b57d37ef773735b8a87caa8a0220631d0b78b7f1
d29c9e54a76f3457ff1a2ee19490ff027c528a896f4bf6aff577
```

Hex	Description
0x02	Two proofs of work
0x01000000	Proof of work ID = 1 (SHA256)
0x09	Config is 9 bytes
0x2c642120	SHA256: Compact target = 0x2021642c
0x00	SHA256: Nonce size is 0 bytes
0x00000000	SHA256: Nonce offset is 0
0x00	Payload is 0 bytes
0x02000000	Proof of work ID = 2 (cuckoo-cycle)
0x05	Config is 5 bytes
0x1c	Size shift is 28
0x0c00	Proof size min is 12
0xe400	Proof size max is 228
0x4c	Payload is 76 bytes
0x3c1e3ee5c799b7e992bcccbb8985979d	Payload
0xcb8dd229b8d0db06e677d00bb3a43c88	
0xef8596a77cbd1dda23b0a0b84bdf6084	
0xd7aa28ddbd5e91b511b3578cbaf92707	
0xc940b051a0759b3f80c5fb65	
0x01000000	Purpose ID = 1 (PURPOSE_CONNECT)
0x24aa065900000000	UNIX timestamp 1493608996
0x46	70 byte signature
0x304402200edfb5c4812a31d84cbbd4b2	Signature data
0x4e631795435a0d16b57d37ef773735b8	
0xa87caa8a0220631d0b78b7f1d29c9e54	
0xa76f3457ff1a2ee19490ff027c528a89	
0x6f4bf6aff577	

Serialized solution example


```
0201000000092c642120000000000000002000000051c0c00e4004c3c1e3ee5c799b7e992bcccbb89
85979dcb8dd229b8d0db06e677d00bb3a43c88ef8596a77cbd1dda23b0a0b84bdf6084d7aa28ddbd
5e91b511b3578cbaf92707c940b051a0759b3f80c5fb650100000024aa0659000000004630440220
0edfb5c4812a31d84cbbd4b24e631795435a0d16b57d37ef773735b8a87caa8a0220631d0b78b7f1
d29c9e54a76f3457ff1a2ee19490ff027c528a896f4bf6aff5775c040000005a0137007074ce00e3
dbeb00e88f790106d71d02984d3d02091b5002378a8e0290a6d202b3c67003757cb70344d9cf0329
7f20048e76a60467e44a057b077405634f840523e88c050d887606109d3e07c4bdcd073db2d407
```


Note that the first 186 bytes are identical to the challenge above.

Hex	Description
0x0201..f577	Challenge
0x5c	Solution is 92 bytes long
0x04000000	The cuckoo cycle nonce is 4

Hex	Description
`0x5a0137007074ce00e3dbeb00e88f7901`	Cycle edges 0..3
`0x06d71d02984d3d02091b5002378a8e02`	Cycle edges 4..7
`0x90a6d202b3c67003757cb70344d9cf03`	Cycle edges 8..11
`0x297f20048e76a60467e44a057b077405`	Cycle edges 12..15
`0x634f840523e88c050d887606109d3e07`	Cycle edges 16..19
`0xc4bdcd073db2d407`	Cycle edges 20..21

Copyright