Script

Bitcoin uses a scripting system for transactions. Forth-like, **Script** is simple, stack-based, and processed from left to right. It is intentionally not Turing-complete, with no loops.

A script is essentially a list of instructions recorded with each transaction that describe how the next person wanting to spend the Bitcoins being transferred can gain access to them. The script for a typical Bitcoin transfer to destination Bitcoin address D simply encumbers future spending of the bitcoins with two things: the spender must provide

1. a public key that, when hashed, yields destination address D embedded in the script, and
2. a signature to prove ownership of the private key corresponding to the public key just provided.

Scripting provides the flexibility to change the parameters of what's needed to spend transferred Bitcoins. For example, the scripting system could be used to require two private keys, or a combination of several keys, or even no keys at all.

A transaction is valid if nothing in the combined script triggers failure and the top stack item is True (non-zero) when the script exits. The party that originally *sent* the Bitcoins now being spent dictates the script operations that will occur *last* in order to release them for use in another transaction. The party wanting to spend them must provide the input(s) to the previously recorded script that results in the combined script completing execution with a true value on the top of the stack.

This document is for information purposes only. De facto, Bitcoin script is defined by the code run by the network to check the validity of blocks.

The stacks hold byte vectors. When used as numbers, byte vectors are interpreted as little-endian variable-length integers with the most significant bit determining the sign of the integer. Thus 0x81 represents -1. 0x80 is another representation of zero (so called negative 0). Positive 0 is represented by a null-length vector. Byte vectors are interpreted as Booleans where False is represented by any representation of zero and True is represented by any representation of non-zero.

Leading zeros in an integer and negative zero are allowed in blocks but get rejected by the stricter requirements which standard full nodes put on transactions before retransmitting them. Byte vectors on the stack are not allowed to be more than 520 bytes long. Opcodes which take integers and bools off the stack require that they be no more than 4 bytes long, but addition and subtraction can overflow and result in a 5 byte integer being put on the stack.

Contents

Opcodes

This is a list of all Script words, also known as opcodes, commands, or functions.

There are some words which existed in very early versions of Bitcoin but were removed out of concern that the client might have a bug in their implementation. This fear was motivated by a bug found in OP_LSHIFT that could crash any Bitcoin node if exploited and by other bugs that allowed anyone to spend anyone's bitcoins. The removed opcodes are sometimes said to be "disabled", but this is something of a misnomer because there is *absolutely no way* for anyone using Bitcoin to use these opcodes (they simply *do not exist anymore* in the protocol), and there are also no solid plans to ever re-enable all of these opcodes. They are listed here for historical interest only.

New opcodes can be added by means of a carefully designed and executed softfork using OP_NOP1-OP_NOP10.

False is zero or negative zero (using any number of bytes) or an empty array, and True is anything else.

Constants

When talking about scripts, these value-pushing words are usually omitted.

Word	Opcode	Hex	Input	Output	Description
OP_0, OP_FALSE	0	0x00	Nothing.	(empty value)	An empty array of bytes is pushed onto the stack. (This is not a no-op: an item is added to the stack.)
N/A	1-75	0x01-0x4b	(special)	data	The next *opcode* bytes is data to be pushed onto the stack
OP_PUSHDATA1	76	0x4c	(special)	data	The next byte contains the number of bytes to be pushed onto the stack.
OP_PUSHDATA2	77	0x4d	(special)	data	The next two bytes contain the number of bytes to be pushed onto the stack in little endian order.
OP_PUSHDATA4	78	0x4e	(special)	data	The next four bytes contain the number of bytes to be pushed onto the stack in little endian order.
OP_1NEGATE	79	0x4f	Nothing.	-1	The number -1 is pushed onto the stack.
OP_1, OP_TRUE	81	0x51	Nothing.	1	The number 1 is pushed onto the stack.
OP_2-OP_16	82-96	0x52-0x60	Nothing.	2-16	The number in the word name (2-16) is pushed onto the stack.

Flow control

Word	Opcode	Hex	Input	Output	Description
OP_NOP	97	0x61	Nothing	Nothing	Does nothing.
OP_IF	99	0x63	<expression> if [statements] [else [statements]]* endif		If the top stack value is not False, the statements are executed. The top stack value is removed.
OP_NOTIF	100	0x64	<expression> notif [statements] [else [statements]]* endif		If the top stack value is False, the statements are executed. The top stack value is removed.
OP_ELSE	103	0x67	<expression> if [statements] [else [statements]]* endif		If the preceding OP_IF or OP_NOTIF or OP_ELSE was not executed then these statements are and if the preceding OP_IF or OP_NOTIF or OP_ELSE was executed then these statements are not.
OP_ENDIF	104	0x68	<expression> if [statements] [else [statements]]* endif		Ends an if/else block. All blocks must end, or the transaction is **invalid**. An OP_ENDIF without OP_IF earlier is also **invalid**.
OP_VERIFY	105	0x69	True / false	Nothing / *fail*	**Marks transaction as invalid** if top stack value is not true. The top stack value is removed.
OP_RETURN	106	0x6a	Nothing	*fail*	**Marks transaction as invalid**. A standard way of attaching extra data to transactions is to add a zero-value output with a scriptPubKey consisting of OP_RETURN followed by exactly one pushdata op. Such outputs are provably unspendable, reducing their cost to the network. Currently it is usually considered non-standard (though valid) for a transaction to have more than one OP_RETURN output or an OP_RETURN output with more than one pushdata op.

Stack

Word	Opcode	Hex	Input	Output	Description
OP_TOALTSTACK	107	0x6b	x1	(alt)x1	Puts the input onto the top of the alt stack. Removes it from the main stack.
OP_FROMALTSTACK	108	0x6c	(alt)x1	x1	Puts the input onto the top of the main stack. Removes it from the alt stack.
OP_IFDUP	115	0x73	x	x / x x	If the top stack value is not 0, duplicate it.
OP_DEPTH	116	0x74	Nothing	<Stack size>	Puts the number of stack items onto the stack.
OP_DROP	117	0x75	x	Nothing	Removes the top stack item.
OP_DUP	118	0x76	x	x x	Duplicates the top stack item.
OP_NIP	119	0x77	x1 x2	x2	Removes the second-to-top stack item.
OP_OVER	120	0x78	x1 x2	x1 x2 x1	Copies the second-to-top stack item to the top.
OP_PICK	121	0x79	xn ... x2 x1 x0 <n>	xn ... x2 x1 x0 xn	The item *n* back in the stack is copied to the top.
OP_ROLL	122	0x7a	xn ... x2 x1 x0 <n>	... x2 x1 x0 xn	The item *n* back in the stack is moved to the top.
OP_ROT	123	0x7b	x1 x2 x3	x2 x3 x1	The top three items on the stack are rotated to the left.
OP_SWAP	124	0x7c	x1 x2	x2 x1	The top two items on the stack are swapped.
OP_TUCK	125	0x7d	x1 x2	x2 x1 x2	The item at the top of the stack is copied and inserted before the second-to-top item.
OP_2DROP	109	0x6d	x1 x2	Nothing	Removes the top two stack items.
OP_2DUP	110	0x6e	x1 x2	x1 x2 x1 x2	Duplicates the top two stack items.
OP_3DUP	111	0x6f	x1 x2 x3	x1 x2 x3 x1 x2 x3	Duplicates the top three stack items.
OP_2OVER	112	0x70	x1 x2 x3 x4	x1 x2 x3 x4 x1 x2	Copies the pair of items two spaces back in the stack to the front.
OP_2ROT	113	0x71	x1 x2 x3 x4 x5 x6	x3 x4 x5 x6 x1 x2	The fifth and sixth items back are moved to the top of the stack.
OP_2SWAP	114	0x72	x1 x2 x3 x4	x3 x4 x1 x2	Swaps the top two pairs of items.

Splice

If any opcode marked as disabled is present in a script, it must abort and fail.

Word	Opcode	Hex	Input	Output	Description
OP_CAT	126	0x7e	x1 x2	out	Concatenates two strings. *disabled.*
OP_SUBSTR	127	0x7f	in begin size	out	Returns a section of a string. *disabled.*
OP_LEFT	128	0x80	in size	out	Keeps only characters left of the specified point in a string. *disabled.*
OP_RIGHT	129	0x81	in size	out	Keeps only characters right of the specified point in a string. *disabled.*
OP_SIZE	130	0x82	in	in size	Pushes the string length of the top element of the stack (without popping it).

Bitwise logic

If any opcode marked as disabled is present in a script, it must abort and fail.

Word	Opcode	Hex	Input	Output	Description
OP_INVERT	131	0x83	in	out	Flips all of the bits in the input. *disabled.*
OP_AND	132	0x84	x1 x2	out	Boolean *and* between each bit in the inputs. *disabled.*
OP_OR	133	0x85	x1 x2	out	Boolean *or* between each bit in the inputs. *disabled.*
OP_XOR	134	0x86	x1 x2	out	Boolean *exclusive or* between each bit in the inputs. *disabled.*
OP_EQUAL	135	0x87	x1 x2	True / false	Returns 1 if the inputs are exactly equal, 0 otherwise.
OP_EQUALVERIFY	136	0x88	x1 x2	Nothing / *fail*	Same as OP_EQUAL, but runs OP_VERIFY afterward.

Arithmetic

Note: Arithmetic inputs are limited to signed 32-bit integers, but may overflow their output.

If any input value for any of these commands is longer than 4 bytes, the script must abort and fail. If any opcode marked as *disabled* is present in a script - it must also abort and fail.

Word	Opcode	Hex	Input	Output	Description
OP_1ADD	139	0x8b	in	out	1 is added to the input.
OP_1SUB	140	0x8c	in	out	1 is subtracted from the input.
OP_2MUL	141	0x8d	in	out	The input is multiplied by 2. *disabled.*
OP_2DIV	142	0x8e	in	out	The input is divided by 2. *disabled.*
OP_NEGATE	143	0x8f	in	out	The sign of the input is flipped.
OP_ABS	144	0x90	in	out	The input is made positive.
OP_NOT	145	0x91	in	out	If the input is 0 or 1, it is flipped. Otherwise the output will be 0.
OP_0NOTEQUAL	146	0x92	in	out	Returns 0 if the input is 0. 1 otherwise.
OP_ADD	147	0x93	a b	out	a is added to b.
OP_SUB	148	0x94	a b	out	b is subtracted from a.
OP_MUL	149	0x95	a b	out	a is multiplied by b. *disabled.*
OP_DIV	150	0x96	a b	out	a is divided by b. *disabled.*
OP_MOD	151	0x97	a b	out	Returns the remainder after dividing a by b. *disabled.*
OP_LSHIFT	152	0x98	a b	out	Shifts a left b bits, preserving sign. *disabled.*
OP_RSHIFT	153	0x99	a b	out	Shifts a right b bits, preserving sign. *disabled.*
OP_BOOLAND	154	0x9a	a b	out	If both a and b are not "" (null string), the output is 1. Otherwise 0.
OP_BOOLOR	155	0x9b	a b	out	If a or b is not "" (null string), the output is 1. Otherwise 0.
OP_NUMEQUAL	156	0x9c	a b	out	Returns 1 if the numbers are equal, 0 otherwise.
OP_NUMEQUALVERIFY	157	0x9d	a b	Nothing / *fail*	Same as OP_NUMEQUAL, but runs OP_VERIFY afterward.
OP_NUMNOTEQUAL	158	0x9e	a b	out	Returns 1 if the numbers are not equal, 0 otherwise.
OP_LESSTHAN	159	0x9f	a b	out	Returns 1 if a is less than b, 0 otherwise.
OP_GREATERTHAN	160	0xa0	a b	out	Returns 1 if a is greater than b, 0 otherwise.
OP_LESSTHANOREQUAL	161	0xa1	a b	out	Returns 1 if a is less than or equal to b, 0 otherwise.
OP_GREATERTHANOREQUAL	162	0xa2	a b	out	Returns 1 if a is greater than or equal to b, 0 otherwise.
OP_MIN	163	0xa3	a b	out	Returns the smaller of a and b.
OP_MAX	164	0xa4	a b	out	Returns the larger of a and b.
OP_WITHIN	165	0xa5	x min max	out	Returns 1 if x is within the specified range (left-inclusive), 0 otherwise.

Crypto

Word	Opcode	Hex	Input	Output	Description
OP_RIPEMD160	166	0xa6	in	hash	The input is hashed using RIPEMD-160.
OP_SHA1	167	0xa7	in	hash	The input is hashed using SHA-1.
OP_SHA256	168	0xa8	in	hash	The input is hashed using SHA-256.
OP_HASH160	169	0xa9	in	hash	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_HASH256	170	0xaa	in	hash	The input is hashed two times with SHA-256.
OP_CODESEPARATOR	171	0xab	Nothing	Nothing	All of the signature checking words will only match signatures to the data after the most recently-executed OP_CODESEPARATOR.
OP_CHECKSIG	172	0xac	sig pubkey	True / false	The entire transaction's outputs, inputs, and script (from the most recently-executed OP_CODESEPARATOR to the end) are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise.
OP_CHECKSIGVERIFY	173	0xad	sig pubkey	Nothing / *fail*	Same as OP_CHECKSIG, but OP_VERIFY is executed afterward.
OP_CHECKMULTISIG	174	0xae	x sig1 sig2 ... <number of signatures> pub1 pub2 <number of public keys>	True / False	Compares the first signature against each public key until it finds an ECDSA match. Starting with the subsequent public key, it compares the second signature against each remaining public key until it finds an ECDSA match. The process is repeated until all signatures have been checked or not enough public keys remain to produce a successful result. All signatures need to match a public key. Because public keys are not checked again if they fail any signature comparison, signatures must be placed in the scriptSig using the same order as their corresponding public keys were placed in the scriptPubKey or redeemScript. If all signatures are valid, 1 is returned, 0 otherwise. Due to a bug, one extra unused value is removed from the stack.
OP_CHECKMULTISIGVERIFY	175	0xaf	x sig1 sig2 ... <number of signatures> pub1 pub2 ... <number of public keys>	Nothing / *fail*	Same as OP_CHECKMULTISIG, but OP_VERIFY is executed afterward.

Locktime

Word	Opcode	Hex	Input	Output	Description
OP_CHECKLOCKTIMEVERIFY (previously OP_NOP2)	177	0xb1	x	x / *fail*	**Marks transaction as invalid** if the top stack item is greater than the transaction's nLockTime field, otherwise script evaluation continues as though an OP_NOP was executed. Transaction is also invalid if 1. the stack is empty; or 2. the top stack item is negative; or 3. the top stack item is greater than or equal to 500000000 while the transaction's nLockTime field is less than 500000000, or vice versa; or 4. the input's nSequence field is equal to 0xffffffff. The precise semantics are described in BIP 0065 (https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki).
OP_CHECKSEQUENCEVERIFY (previously OP_NOP3)	178	0xb2	x	x / *fail*	**Marks transaction as invalid** if the relative lock time of the input (enforced by BIP 0068 (https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki) with nSequence) is not equal to or longer than the value of the top stack item. The precise semantics are described in BIP 0112 (https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki).

Pseudo-words

These words are used internally for assisting with transaction matching. They are invalid if used in actual scripts.

Word	Opcode	Hex	Description
OP_PUBKEYHASH	253	0xfd	Represents a public key hashed with OP_HASH160.
OP_PUBKEY	254	0xfe	Represents a public key compatible with OP_CHECKSIG.
OP_INVALIDOPCODE	255	0xff	Matches any opcode that is not yet assigned.

Reserved words

Any opcode not assigned is also reserved. Using an unassigned opcode makes the transaction **invalid**.

Word	Opcode	Hex	When used...
OP_RESERVED	80	0x50	**Transaction is invalid** unless occuring in an unexecuted OP_IF branch
OP_VER	98	0x62	**Transaction is invalid** unless occuring in an unexecuted OP_IF branch
OP_VERIF	101	0x65	**Transaction is invalid even when occuring in an unexecuted OP_IF branch**
OP_VERNOTIF	102	0x66	**Transaction is invalid even when occuring in an unexecuted OP_IF branch**
OP_RESERVED1	137	0x89	**Transaction is invalid** unless occuring in an unexecuted OP_IF branch
OP_RESERVED2	138	0x8a	**Transaction is invalid** unless occuring in an unexecuted OP_IF branch
OP_NOP1, OP_NOP4-OP_NOP10	176, 179-185	0xb0, 0xb3-0xb9	The word is ignored. Does not mark transaction as invalid.

Script examples

The following is a list of interesting scripts. When notating scripts, data to be pushed to the stack is generally enclosed in angle brackets and data push commands are omitted. Non-bracketed words are opcodes. These examples include the “OP_” prefix, but it is permissible to omit it. Thus “<pubkey1> <pubkey2> OP_2 OP_CHECKMULTISIG” may be abbreviated to “<pubkey1> <pubkey2> 2 CHECKMULTISIG”. Note that there is a small number of standard script forms that are relayed from node to node; non-standard scripts are accepted if they are in a block, but nodes will not relay them.

Standard Transaction to Bitcoin address (pay-to-pubkey-hash)

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
scriptSig: <sig> <pubKey>
```

To demonstrate how scripts look on the wire, here is a raw scriptPubKey:

```
  76       A9             14
OP_DUP OP_HASH160    Bytes to push

89 AB CD EF AB BA AB BA AB BA AB BA AB BA AB BA AB BA AB BA   88         AC
                      Data to push                     OP_EQUALVERIFY OP_CHECKSIG
```

Note: scriptSig is in the input of the spending transaction and scriptPubKey is in the output of the previously unspent i.e. "available" transaction.

Here is how each word is processed:

Stack	Script	Description
Empty.	<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Constants are added to the stack.
<sig> <pubKey> <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is duplicated.
<sig> <pubKey> <pubHashA>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is hashed.
<sig> <pubKey> <pubHashA> <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	Constant added.
<sig> <pubKey>	OP_CHECKSIG	Equality is checked between the top two stack items.
true	Empty.	Signature is checked for top two stack items.

Obsolete pay-to-pubkey transaction

OP_CHECKSIG is used directly without first hashing the public key. This was used by early versions of Bitcoin where people paid directly to IP addresses, before Bitcoin addresses were introduced. scriptPubKeys of this transaction form are still recognized as payments to user by Bitcoin Core. The disadvantage of this transaction form is that the whole public key needs to be known in advance, implying longer payment addresses, and that it provides less protection in the event of a break in the ECDSA signature algorithm.

```
scriptPubKey: <pubKey> OP_CHECKSIG
scriptSig: <sig>
```

Checking process:

Stack	Script	Description
Empty.	<sig> <pubKey> OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_CHECKSIG	Constants are added to the stack.
true	Empty.	Signature is checked for top two stack items.

Provably Unspendable/Prunable Outputs

The standard way to mark a transaction as provably unspendable is with a scriptPubKey of the following form:

```
scriptPubKey: OP_RETURN {zero or more ops}
```

OP_RETURN immediately marks the script as invalid, guaranteeing that no scriptSig exists that could possibly spend that output. Thus the output can be immediately pruned from the UTXO set even if it has not been spent. eb31ca1a4cbd97c2770983164d7560d2d03276ae1aee26f12d7c2c6424252f29 (http://blockexplorer.com/tx/eb31ca1a4cbd97c2770983164d7560d2d03276ae1aee26f12d7c2c6424252f29) is an example: it has a single output of zero value, thus giving the full 0.125BTC fee to the miner who mined the transaction without adding an entry to the UTXO set. You can also use OP_RETURN to add data to a transaction without the data ever appearing in the UTXO set, as seen in 1a2e22a717d626fc5db363582007c46924ae6b28319f07cb1b907776bd8293fc; P2Pool does this with the share chain hash txout in the coinbase of blocks it creates.

Anyone-Can-Spend Outputs

Conversely a transaction can be made spendable by anyone at all:

```
scriptPubKey: (empty)
scriptSig: OP_TRUE
```

With some software changes such transactions can be used as a way to donate funds to miners in addition to transaction fees: any miner who mines such a transaction can also include an additional one after it sending the funds to an address they control. This mechanism may be used in the future for fidelity bonds to sacrifice funds in a provable way.

Anyone-Can-Spend outputs are currently considered non-standard, and are not relayed on the P2P network.

Freezing funds until a time in the future

Using OP_CHECKLOCKTIMEVERIFY it is possible to make funds provably unspendable until a certain point in the future.

```
scriptPubKey: <expiry time> OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
scriptSig: <sig> <pubKey>
```

Stack	Script	Description
Empty.	<sig> <pubKey> <expiry time> OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey> <expiry time>	OP_CHECKLOCKTIMEVERIFY OP_DROP OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Constants are added to the stack.
<sig> <pubKey> <expiry time>	OP_DROP OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is checked against the current time or block height.
<sig> <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is removed.
<sig> <pubKey> <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is duplicated.
<sig> <pubKey> <pubHashA>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is hashed.
<sig> <pubKey> <pubHashA> <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	Constant added.
<sig> <pubKey>	OP_CHECKSIG	Equality is checked between the top two stack items.
true	Empty.	Signature is checked for top two stack items.

Transaction puzzle

Transaction a4bfa8ab6435ae5f25dae9d89e4eb67dfa94283ca751f393c1ddc5a837bbc31b is an interesting puzzle.

```
scriptPubKey: OP_HASH256 6fe28c0ab6f1b372c1a6a246ae63f74f931e8365e15a089c68d6190000000000 OP_EQUAL
scriptSig:
```

To spend the transaction you need to come up with some data such that hashing the data twice results in the given hash.

Stack	Script	Description
Empty.	<data> OP_HASH256 <given_hash> OP_EQUAL	
<data>	OP_HASH256 <given_hash> OP_EQUAL	scriptSig added to the stack.
<data_hash>	<given_hash> OP_EQUAL	The data is hashed.
<data_hash> <given_hash>	OP_EQUAL	The given hash is pushed to the stack.
true	Empty.	The hashes are compared, leaving true on the stack.

This transaction was successfully spent by 09f691b2263260e71f363d1db51ff3100d285956a40cc0e4f8c8c2c4a80559b1. The required data happened to be the Genesis block, and the given hash in the script was the genesis block header hashed twice with SHA-256. Note that while transactions like this are fun, they are not secure, because they do not contain any signatures and thus any transaction attempting to spend them can be replaced with a different transaction sending the funds somewhere else.

Incentivized finding of hash collisions

In 2013 Peter Todd created scripts that result in true if a hash collision is found. Bitcoin addresses resulting from these scripts can have money sent to them. If someone finds a hash collision they can spend the bitcoins on that address, so this setup acts as an incentive for somebody to do so.

For example the SHA1 script:

```
scriptPubKey: OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA1 OP_SWAP OP_SHA1 OP_EQUAL
scriptSig: <preimage1> <preimage2>
```

See the bitcointalk thread [1] and reddit thread[2] for more details.

In February 2017 the SHA1 bounty worth 2.48 bitcoins was claimed.

See Also

- Transactions
- Contracts

External Links

- Bitcoin IDE (https://github.com/siminchen/bitcoinIDE): Bitcoin Script for dummies
- Bitcoin Debug Script Execution (https://webbtc.com/script) - web tool which executes a script opcode-by-opcode
- Script Playground (http://www.crmarsh.com/script-playground/) — convert Script to JavaScript

(cf. "Online Bitcoin Script simulator or debugger? (http://bitcoin.stackexchange.com/q/42576/4334)")

References

1. bitcointalk forum thread on the hash collision bounties (https://bitcointalk.org/index.php?topic=293382.0)
2. https://www.reddit.com/r/Bitcoin/comments/1mavh9/trustless_bitcoin_bounty_for_sha1_sha256_etc/

Bitcoin Core documentation

User documentation	Alert system • Bitcoin Core compatible devices • Data directory • Fallback Nodes • How to import private keys in Bitcoin Core 0.7+ • Installing Bitcoin Core • Running Bitcoin • Transaction fees • Vocabulary
Developer documentation	Accounts explained • API calls list • API reference (JSON-RPC) • Block chain download • Dump format • getblocktemplate • List of address prefixes • Protocol documentation • Script • Technical background of version 1 Bitcoin addresses • Testnet • Transaction Malleability • Wallet import format
History & theory	Common Vulnerabilities and Exposures • DOS/STONED incident • Economic majority • Full node • Original Bitcoin client • Value overflow incident