前書き

本書では、暗号通貨ビットコインの送金時に内部で使用されるスクリプト言語を解説します。

前回のコミックマーケット C93 で頒布した「ZIP、完全に理解した」では、圧縮形式 ZIP を取り上げ、特に暗号化とその解読方法に焦点を合わせて、解説しました。せっかく本を作るのですから、あまり多くの人には知られていないことを書きたいものですが、有用な技術については多くの書籍やウェブ上の記事が存在します。そこで、「技術の本流からは少し外れた、でも面白い」ことを題材にしようという考えでテーマを選んでいます。

ZIP の暗号化は今でも広く使われてはいますが、すでに脆弱性が見つかっており、セキュリティに詳しい人ならば「使うべきではない」と口を揃えるでしょう。ビットコインのスクリプト言語にそのような脆弱性があるわけではありません。しかし、現在のビットコインの公式実装ではスクリプトに厳しい制約が課せられていて、ほぼ全てのノードは制約を満たすスクリプトしか受理しません。また、Segwit では、スクリプトを用いずに、単に公開鍵と署名を符号化するだけの仕様が追加されました。徐々にスクリプトから離れようとしているという印象を受けます。一方で、最近でもスクリプト言語に新たな命令が追加されましたし、新しい技術であるライトニングネットワークなどで、必要性が高まるかもしれません。

本書のタイトルの「モナーでもわかる」は、筆者がプログラミングを最初に学ぶときに読んだサイト「猫でもわかるプログラミング[*1]」を懐かしんで付けました。「〇〇でもわかる」という題名は良く目にしますが、20 年以上前から存在したサイトですし、どれもこのサイトが元ネタでしょうか？

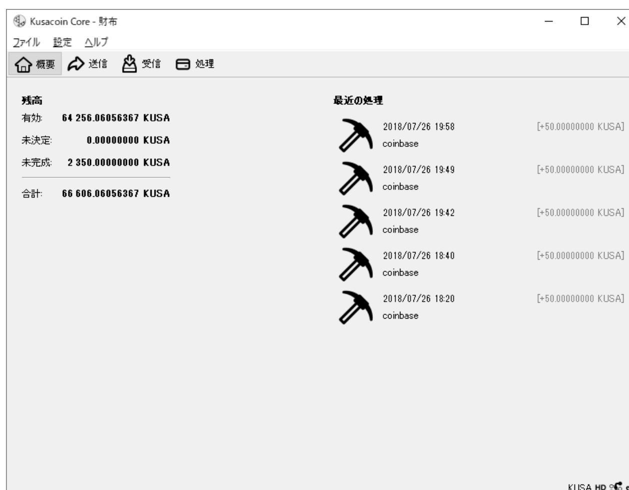
執筆時点の 2018 年 7 月 22 日の bitFlyer におけるビットコインの価格は、1 BTC あたり 82 万 9313 円です。暗号通貨に関する少し前の記事を読むと、「こんなに安かったのか」もしくは「こんなに高かったのか」という驚きがあります。次回のコミックマーケットのときにでも本書を見直すとどういう感想になるでしょう。楽しみですね。

[*1] http://www.kumei.ne.jp/c_lang/

Kusacoin

ビットコインを fork して、削除されていた命令を再実装し、デフォルトの設定で非標準のスクリプトを受け付けるようにした暗号通貨 Kusacoin を作りました。取引をネットワークに送信せず、自分が採掘するブロックにだけ含めるようにするコマンド `poolrawtransaction` を追加しています。色々なスクリプトを試す際に使ってみてください。今のところ（採掘している人が少ないので）CPU でも 1 時間に 1-2 ブロックは採掘できます。

https://kusacoin.github.io/



目次

前書き 1

第 1 章　はじめに 4

スクリプト 4

本書の構成 6

参考文献 6

第 2 章　スクリプトの作成 8

概要 8

命令セット 9

例 17

第 3 章　トランザクションスクリプトの送信 24

準備 25

`scriptPubKey` での送金 25

Pay to Script Hash（P2SH）での送金 29

第 4 章　ブロックチェーンのフォーマット 33

ブロック 33

取引 35

入力、出力、witness 36

第 5 章　ブロックチェーン中のスクリプトの解析 38

`scriptPubKey` の集計 38

P2SH の集計 39

P2WSH の集計 40

Null Data の集計 41

Non Standard の解析 42

（UTXO、Unspent Transaction Output）の合計がそのアドレスの残高となります。ビットコインではある取引で送金されたコインを使用するとき、その一部だけを使用することはできません。1 BTC のうち 0.1 BTC だけを送金したいという場合には、0.1 BTC を相手に送金し、0.9 BTC を自分の所有する別のアドレス（もしくは元のアドレス）に送金するという取引を作成します。ビットコインのクライアントは複数のアドレスを管理しており、その残高の合計がクライアントに表示される残高となります。このようにすることで、参加者全員に全取引が公開される状況下でも、ある程度のプライバシーを確保することができます。ビットコインでは取引の度に新しいアドレスを作ることが推奨されています。

実は、ビットコインのプロトコルは、単に公開鍵や電子署名を書き込むのではなく、より汎用的な作りになっています。送金の宛先は単なる公開鍵ではなく「入力が正しい電子署名であるか検証する」というスクリプト（`scriptPubKey`）であり、送金への電子署名は実際には「電子署名を入力する」というスクリプト（`scriptSig`）です。各ノードは `scriptSig` と `scriptPubKey` をこの順に続けて実行して、結果が真であれば、正しい取引であると判断します。スクリプトになっていることによって、プロトコルの大枠を変更することなく、電子署名の方式を変更することができます。なお、スクリプトへの命令の追加などの変更が自由にできるわけではありません（コラム：ソフトフォークとハードフォークを参照）。実際に何通りかの方法が用いられており、この本の中でも紹介します。なお、電子署名のアルゴリズムがこのスクリプトで一から実装されているわけではなく、このスクリプトの命令セットには電子署名の検証を行う専用の命令が組み込まれています。

Ethereum などの暗号通貨ではブロックチェーン上でプログラムを動かせることをウリにしています。このビットコインの電子署名に用いられるスクリプト（トランザクションスクリプト）も、活用すれば何か面白いことができるかもしれません。

- https://github.com/bitcoin/bitcoin/
- https://sourceforge.net/p/bitcoin/

ビットコインの詳細について理解するには Mastering Bitcoin がオススメです。

- https://bitcoinbook.info/translations-of-mastering-bitcoin/

Bitcoin Wiki には用語の説明が多く書かれています。

- https://en.bitcoin.it/wiki/

ブロックチェーンやネットワーク上の取引の詳細を見るには、chainFlyer を使うと良いでしょう。他の類似のエクスプローラーと異なり、Pay To Script Hash のスクリプトや Witness をデコードして表示してくれます（Pay To Script Hash のスクリプトはマウスカーソルをデータの上に重ねると表示されます）。

- https://chainflyer.bitflyer.jp/

命令	値	説明	例		
OP_2ROT	0x71	6 個の値を 2 個単位で回転	... u v w x y z	→	... w x y z u v
OP_2SWAP	0x72	4 個の値を 2 個単位で交換	... w x y z	→	... y z w x
OP_DROP	0x75	先頭の値を削除	... x	→	...
OP_DUP	0x76	先頭の値を複製	... x	→	... x x
OP_NIP	0x77	2 番目の値を削除	... x y	→	... y
OP_OVER	0x78	2 番目の値を複製	... x y	→	... x y x
OP_ROT	0x7b	3 個の値を回転	... x y z	→	... y z x
OP_SWAP	0x7c	2 個の値を交換	... x y	→	... y x
OP_TUCK	0x7d	先頭の値を 3 番目に複製	... x y	→	... y x y

バイト列操作

OP_CAT（0x7e）はスタックの先頭 2 個のバイト列を pop し、1 番目の値、2 番目の値の順で連結して、push します。

OP_SUBSTR（0x7f）は、スタックの先頭から順に数値 size、数値 begin、バイト列 x を pop し、x の begin バイト目から size バイトを取り出して push します。size と begin が負の場合はエラーとなります。size や begin が大きすぎるときはエラーにはならず、x の末尾までが使用されます。

OP_LEFT（0x80）と OP_RIGHT（0x81）は、スタックの先頭から順に数値 size とバイト列 x を pop し、x のそれぞれ左と右の size バイトを push します。size が x のバイト数より大きくてもエラーとはならず、x がそのまま push されます。

OP_CAT、OP_SUBTR、OP_LEFT、OP_RIGHT は、現在は削除されています。

OP_SIZE はスタック先頭の値のバイト数をスタックに push します。元の先頭の値は削除されず、2 番目の値として残ります。

ビット演算

OP_INVERT（0x83）は、スタック先頭の各バイトをビット反転します。OP_AND（0x84）、OP_OR（0x85）、OP_XOR（0x86）はスタック先頭の 2 個の値を pop し、それぞれビット単位の積、和、xor の結果を push します。2 個の値の長さが異なる場合には、短いほうのバイト列の末尾に 00 を

命令	値	値の個数	結果	削除
OP_LESSTHANOREQUAL	0xa1	2	y <= x	
OP_GREATERTHANOREQUAL	0xa2	2	y >= x	
OP_MIN	0xa3	2	min(y, x)	
OP_MAX	0xa4	2	max(y, x)	
OP_WITHIN	0xa5	3	y <= z && z < x	

OP_NUMEQUALVERIFY（0x9d）は OP_NUMEQUAL OP_VERIFY と等価です。

暗号

OP_RIPEMD160（0xa6）、OP_SHA1（0xa7）、OP_SHA256（0xa8）は、スタック先頭の値を pop し、それぞれ RIPEMD-160、SHA-1、SHA-256 を算出して push します。

OP_HASH160（0xa9）は SHA-256 と PRIMED-160 をこの順に適用した値が push されます。このハッシュ関数は、公開鍵のハッシュ値をトランザクションスクリプトに含める場合などに使われています。

OP_HASH256（0xaa）は SHA-256 を 2 回適用したハッシュ値を push します。このハッシュ関数はブロックのハッシュ値などに使用されています。

OP_CHECKSIG（0xac）はスタックの先頭 2 個の値を pop し、先頭の値を公開鍵、2 番目の値を署名として、このスクリプトが含まれている取引が公開鍵に対応する秘密鍵で署名されたものであるかどうかを push します。署名の検証方法は次項で解説します。OP_CHECKSIGVERIFY（0xad）は OP_CHECKSIG OP_VERIFY と等価です。

OP_CHECKMULTISIG（0xae）は、複数の公開鍵と複数の署名で電子署名を検証します。スタックに積まれている値を下から順に x sig[m-1] ... sig[1] sig[0] m pubkey[n-1] ... pubkey[1] pubkey[0] n として、これらを pop し、m 個の署名 sig が n 個の公開鍵 pubkey のうちの m 個に対して正しい署名になっているかを判定します。全ての署名が正しい署名ならば、スタックに真を push します。そうでない場合は偽が push されます。pubkey のうち対応する署名が存在する m 個の公開鍵は対応する署名と同じ順番に並んでいなければなりません。バグによって（！）、1 個余計にスタックから pop されるので、何らかの値 x をスタックに積んでおく必要があります。ブロックチェーン中の過去の取引が検証に失敗するようになるので、今さらこのバグを修正することはできません。m 以降を scriptPubKey とすることで、n の公開鍵のうち m 個

ノードでスクリプトの検証に失敗するのを防ぐためです。

OP_NOP1（0xb0）、OP_NOP4（0xb3）、OP_NOP5（0xb4）、...、OP_NOP10（0xb9）は現在は NOP 命令であり、何もしません。OP_CHECKLOCKTIMEVERIFY や OP_CHECKSEQUENCEVERIFY のように、将来的に何らかの命令が割り当てられる可能性があるので、NOP 命令としては OP_NOP（0x61）を使うのが良いでしょう。

OP_RESERVED1（0x89）と OP_RESERVED2（0x8a）は予約済みであり、実行するとエラーになります。OP_INVALIDOPCODE（0xff）も同様に無効な命令です。無効な命令を明示的に使用したい場合には OP_INVALIDOPCODE を使うと良いでしょう。

OP_SMALLINTEGER（0xfa）、OP_PUBKEYS（0xfb）、OP_PUBKEYHASH（0xfd）、OP_PUBKEY（0xfe）は、ソースコード中に定数が存在しますが、スクリプトで使われることはなく、出現した場合はエラーになります。内部的にスクリプトの種類を識別する際、それぞれ 16 以下のリテラル、公開鍵の列、公開鍵のハッシュ値、公開鍵を表すために使用されます。

例

スクリプトの例として、Bitcoin Core が output type として認識するものと、連立方程式の問題と解答を入力するものを示します。また、P2SH や Segwit で行われる追加の検証についても説明します。

Pay to Public Key (P2PK)

```
scriptPubKey: <公開鍵> OP_CHECKSIG
scriptSig: <署名>
```

最も基本的なスクリプトです。scriptPubKey と scriptSig を繋げると、<署名> <公開鍵> OP_CHECKSIG となります。あるアドレスへの送金は scriptPubKey が使用され、そのアドレスから送金されたコインを別のアドレスに送金するときには scriptSig が必要なりますが、スクリプトの検証時には scriptSig scriptPubKey と時系列とは逆順になることに注意してください。このスクリプトは OP_CHECKSIG による電子署名の検証を行うだけです。

でスタックの値が 1 個余計に消費されることに対応するためのものです。

Null Data

```
scriptPubKey: OP_RETURN <Data 1> <Data 2> ... <Data n>
scriptSig: -
```

`OP_RETURN` でスクリプトの実行が失敗となるので、この `scriptPubKey` に対して送金されたビットコインを取り出す方法はありません。ブロックチェーンに何らかのデータを書き込むために使用されます。

`OP_RETURN` 以降が実行されることはないので、どのようなスクリプトでも良さそうなものですが、Null Data として認識されるためには、リテラルのみで構成されている必要があります。例えば、`"data"`（`64617461`）を書き込む時には、`6a 64 61 74 61` ではなく、`6a 04 64 61 74 61` とします。`04 64 61 74 61` は 4 バイトの `64 61 74 61` をスタックに積む命令です。

Pay to Script Hash (P2SH)

```
scriptPubKey: OP_HASH160 <Script Hash> OP_EQUAL
scriptSig: <Input 1> <Input 2> ... <Input n> <Script>
```

アドレスの種別が `0x05` である（Base58 で先頭が 3 になる）とき、ビットコインのクライアントはアドレスに含まれるスクリプトのハッシュ値から、この取引を生成します。

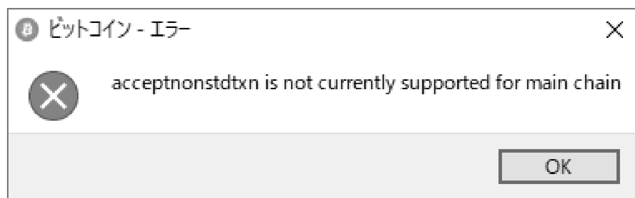
この形式ではスクリプトを出金側の `scriptSig` に含めることができます。`scriptPubKey` は送金側が作成します。例えば、誰かに multisig の `scriptPubKey` に対して送金を依頼するとき、その人の使用するクライアントが multisig 形式の `scriptPubKey` を作成できなければいけません。P2PSH ならばスクリプトがどのような形式でも、送金側はスクリプトのハッシュ値を埋め込むだけなので、クライアントが P2SH にさえ対応していれば送金することができます。

一段階目のチェックは `Script Hash` が `Script` の 160 ビットハッシュ値であれば通ります。その後、`scriptPubKey` がこの形式で、`Script Hash` が 160 ビット（20 バイト）の値であれば、二段階目のチェックが行われます。

命令とスタックの状態は下表のようになります。

命令	スタック
<x>	<x>
<y>	<x> <y>
OP_2DUP	<x> <y> <x> <y>
OP_ADD	<x> <y> <x+y>
0x68ac	<x> <y> <x+y> 0x68ac
OP_NUMEQUALVERIFY	<x> <y>（一致しなければここで終了）
OP_DUP	<x> <y> <y>
OP_ADD	<x> <2y>
OP_SUB	<x-2y>
-0x9abc	<x-2y> -0x9abc
OP_NUMEQUAL	01（一致しなければ空バイト列（0））

3 章では、このスクリプトを実際にビットコインのネットワーク上に送信してみます。



ソースコードを書き換えて、ネットワークに送信したところで、多くのノードに中継を拒否されるとマイナーまで届かなそうです。非標準の scriptPubKey が含まれたブロックが無効とみなされるわけではないので、自分でブロックを採掘すればこのような特殊な scriptPubKey を含めることも可能ですが、今のビットコインの採掘難易度では現実的ではありません。テストネットならば、非標準の scriptPubKey も有効なようです。

Pay to Script Hash（P2SH）での送金

P2SH の scriptPubKey と scriptSig は次の通りです。

```
scriptPubKey: OP_HASH160 <Script Hash> OP_EQUAL
scriptSig: <Input 1> <Input 2> ... <Input n> <Script>
```

scriptPubKey と異なり、二段階目のチェックで実行される Script には、ほとんど制約がありません。一方、scriptSig に相当する、<Script>より前の部分は、スタックに値を積むリテラルのみしか許されません。この制約は、Bitcoin Core の実装による標準形式かどうかのルールではなく、ビットコインのプロトコルのルールです。たとえこの制約を満たさないブロックを自分で採掘しても、ネットワークからはこのブロックが承認されません。今回送信しようとしているスクリプトの scriptSig は 0x1234 0x5678 なので、この制約を満たします。

P2SH を用いて、先の連立方程式の問題に対してビットコインを送金し、連立方程式の答えで取り出してみます。decodescript でスクリプトをで逆アセンブルした結果には p2sh としてアドレスが表示されています。

```
decodescript 6e9302ac689d76939403bc9a809c
{
  "asm": "OP_2DUP OP_ADD 26796 OP_NUMEQUALVERIFY OP_DUP OP_ADD OP_SUB -39612
          OP_NUMEQUAL",
  "type": "nonstandard",
```

```
ation) (code -26)
```

x = 0x1235、y = 0x5677 と書き換えて OP_NUMEQUAL が失敗するようにした場合は、

```
sendrawtransaction 0200...150235120277560e6e9302ac689d76939403bc9a809cfffffff...
16: mandatory-script-verify-flag-failed (Script evaluated without error but fini
shed with a false/empty top stack element) (code -26)
```

となります。書き換える前の取引を送信するとエラーにはならず、取引の ID が出力されて、送信に成功したことがわかります。

```
sendrawtransaction 0200...150234120278560e6e9302ac689d76939403bc9a809cfffffff...
fb4a1feda78f2a5e219ce39d793e180fa62b4707b3b021b5632abd0696ed9cec
```

ブロックに取り入れられるのを待っていましたが、誰かに scriptSig はそのままで出金先のアドレスだけを書き換えた取引をネットワークに流され、その取引がブロックに取り入れられて、私の取引は二重取引として失敗になってしまいました。さようなら、私の 0.001 BTC (´；ω；‘)

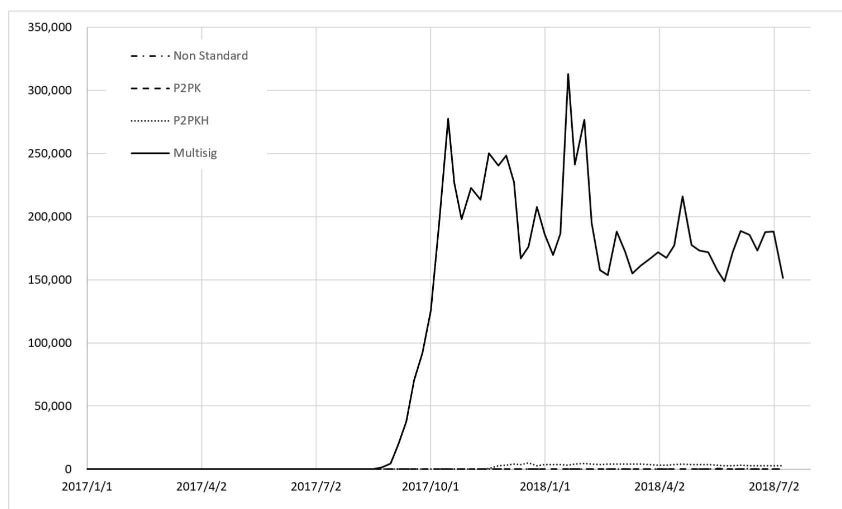
出力のフォーマットは次の通りです。

項目	サイズ	説明
`value`	8 バイト	金額
`nScriptPubKey`	CompactSize	`scriptPubKey` のバイト数
`ScriptPubKey`	`nScriptPubKey`	スクリプト

`value` は satoshi（10^{-8}）単位で指定します。

`vWitness` の要素数は入力の個数と同じです。各要素は値の配列であり、最初に CompactSize で値の個数が書かれ、その後に ComactSize で表された値の長さと、値が繰り返されます。

まれるブロックで個数を数えています。



種別	個数
Non Standard	1,393
P2PK	1
P2PKH	123,220
Multisig	8,728,569
計	8,853,183

ほぼ全てが multisig のスクリプトです。P2WPKH があるため、P2WSH で 1 個の公開鍵に対する支払いをする必要が無いからでしょう。唯一の P2WSH を用いた P2PK の取引は `264887a5a42319a4b25cab843f3bbe8699432fbdab95742157b32a0239230d1a` です。

Null Data の集計

Null Data への出力は 5,947,701 個です。`OP_RETURN` とリテラルのみで構成される Null Data に対してビットコインを送金しても取り出すことができないので、通常は金額は 0 とします。

Null Data に対してビットコインを送金している出力が 56,746 個ありました。大半は 1 satoshi 程度の少額ですが、合計は 3.71161990 BTC にもなりました。最も金額が高いのは、取引 `381917e8d1232aa8eba9893a3b43b7d6e2522654cabf770accb1c63ffbeacb1e` の 0.018454 BTC です。

個ありました。`scriptSig` では署名をスタックに入れる必要は無く、バグで取り出される分の値を 1 個積むだけで出金することができます。

分類するときに $m \geq 1$ という条件を付けていたので、Non Standard としてしまいましたが、Bitcoin Core は multisig と認識するので、Multisig に含めるべきでした。

`c4aaf7fbec7a9a079e670e50f6a672315451c7618814494ab1f89cf3fd97b3bb`

```
20
0478d430274f8c5ec1321338151e9f27f4c676a008bdf8638d07c0b6be9ab35c71a1518063243...
0478d430274f8c5ec1321338151e9f27f4c676a008bdf8638d07c0b6be9ab35c71a1518063243...
 :
0478d430274f8c5ec1321338151e9f27f4c676a008bdf8638d07c0b6be9ab35c71a1518063243...
20
OP_CHECKMULTISIG
```

公開鍵数の多いものでは、20-of-20 の multisig が 1 個ありました。同じ公開鍵が 20 個並んでいるので、電子署名も同じものを 20 個並べれば実行結果が真となります。Bitcoin Core は `n` が 16 以下の multisig しか標準形式としません。これはあくまで実装の条件であって、`OP_CHECKMULTISIG` にはそのような条件は無いので、一度ブロックに含まれれば、正当なブロックとして認められます。

`8e2c7cec5006949e1929f70961da8f85eebfe06a4979d611ec93ab384eaa34ed`

```
1
496620776520636f756c6420666163746f72206c6172676520636f6d706f736974657320696e2...
024752639a0cb3a59261d582dd94b1b8c6b7302070fb0e684cb0d4b25e746b1a50
2
OP_CHECKMULTISIG
```

1-of-2 の multisig ですが、公開鍵のうち 1 個が正しい公開鍵のフォーマットではありません。ビットコインの公開鍵は、非圧縮形式（楕円曲線上の点の x と y を含む）ならば先頭が 04 の 65 バイト、圧縮形式（楕円曲線上の点のうち x だけを含む）ならば先頭が 02 か 03 の 33 バイトです。Multisig では一部の公開鍵の電子署名だけがあれば良いので、このようなことをしても後からビットコインを取り出すことができます。1 個目の公開鍵は、ラップの歌詞の一部のようです。

このような正しくない長さの公開鍵を含み、その他は標準形式の multisig の `scriptPubKey` は

です。

多額のビットコインが入った scriptPubKey

fa735229f650a8a12bcf2f14cca5a8593513f0aabc52f8687ee148c9f9ab6665

```
OP_IFDUP
OP_IF
OP_2SWAP
OP_VERIFY
OP_2OVER
OP_DEPTH
```

アセンブルすると、736372697074、script という文字列になります。この scriptPubKey に対して、182 件、合計 0.60280235 BTC の入金があり、今もそのままです。OP_IF の対応が取れていないので、ビットコインの仕様が変わらない限り取り出すことはできなさそうです。

03acfae47d1e0b7674f1193237099d1553d3d8a93ecc85c18c4bec37544fe386

```
OP_DUP
OP_HASH160
0
OP_EQUALVERIFY
OP_CHECKSIG
```

この scriptPubKey はより強烈で、23 件、2609.36304319 BTC の入金があります。 時価総額は約 20 億円です。とはいえ、こちらも出金することはできなさそうです。たとえ OP_HASH160 の結果が 0 となるような公開鍵を見つけても、直後の OP_EQUALVERIFY が通りません。OP_HASH160 が返すのは 160 ビット（20 バイト）の値ですが、0 がスタックに積むのは 0 バイトの値であり、OP_NUMEQUALVERIFY と異なり OP_EQUALVERIFY はバイト列として一致する必要があるからです。