vanity hashes

having fun with files and hashes

Michael «hops» Sprecher

what are hashes?

- maps n bytes of data to a fixed size
- one-way functions
- deterministic
- non-/cryptographic hashes
- used for example as integrity checks for files

SHA-256

- **based on the Merkle-Damgard construct**
- **message broken into 64 byte blocks**
- **outputs 256 bits (32 bytes)**

```
$ echo -n LuXeria | sha256sum
```

```
6c8f6c91ba743da60644c635b9db27775515676f
7222240770e3778e0400b750
```

vanity hashes

- «A vanity hash is a hash that contains an interesting substring.» - Royce Williams

done

just kidding

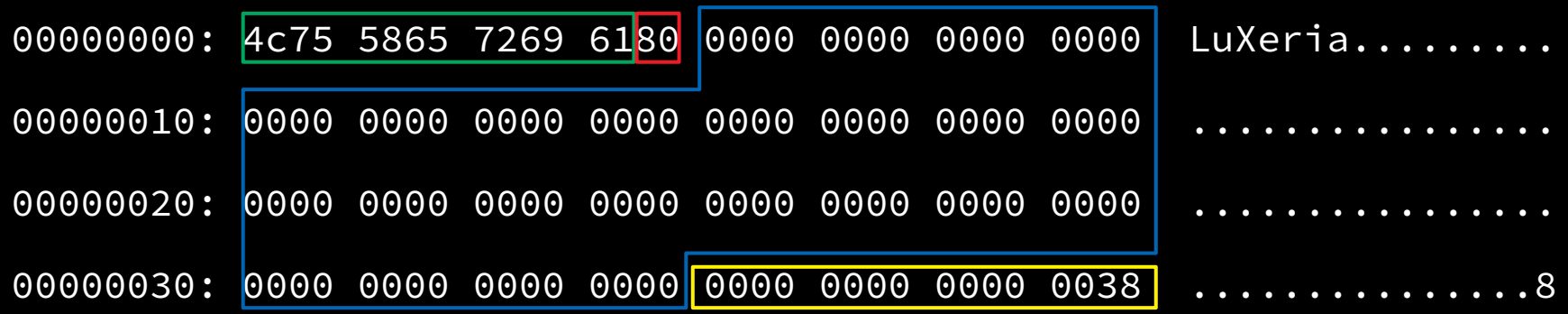
Bitcoin addresses

- 1HZwkjkeaoZfTSaJxDw6aKkxp45agDiEzN
- 1hops41iWUquG4orWvsv4N1KHQwYpYPQU

Tor hidden services

- https://3g2upl4pq6kufc4m.onion
- https://facebookcorewwwi.onion

SHA-256 part2



- message
- «end of message» bit
- padding
- length in bits (64 bit big endian integer)

SHA-256 IV

```
a = 0x6a09e667
b = 0xbb67ae85
c = 0x3c6ef372
d = 0xa54ff53a
e = 0x510e527f
f = 0x9b05688c
g = 0x1f83d9ab
h = 0x5be0cd19
```

SHA-256

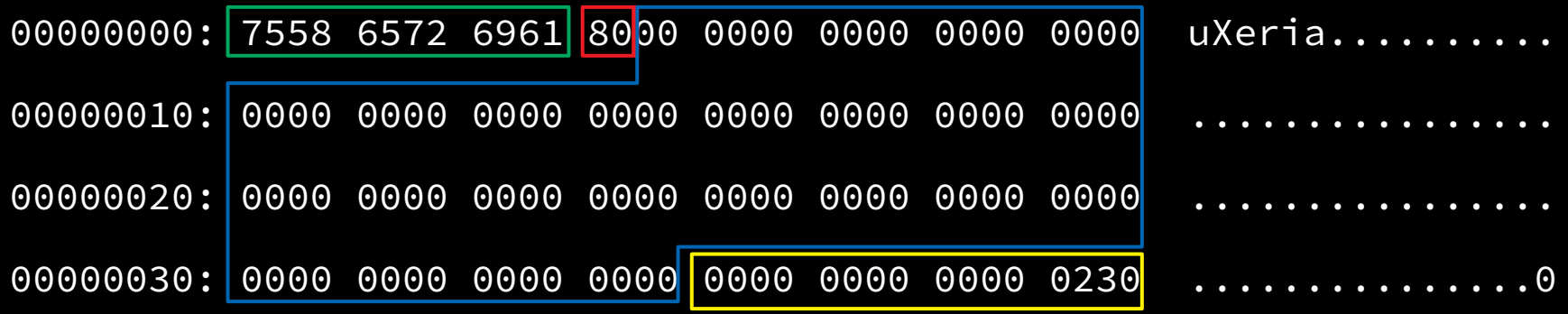
- **input block → crypto magic → hash**
- `a = 0x6a09e667`
- **crypto magic**
- `a = 0x0285862a`
- `0x0285862a + 0x6a09e667 = 0x6c8f6c91`
- `sha256("LuXeria") → 6c8f6c91ba743...`

block one

```
00000000: 4c75 5865 7269 614c 7558 6572 6961 4c75  LuXeriaLuXeriaLu
00000010: 5865 7269 614c 7558 6572 6961 4c75 5865  XeriaLuXeriaLuXe
00000020: 7269 614c 7558 6572 6961 4c75 5865 7269  riaLuXeriaLuXeri
00000030: 614c 7558 6572 6961 4c75 5865 7269 614c  aLuXeriaLuXeriaL
```

- **message part one**

block two



- message part two
- «end of message» bit
- padding
- length in bits (64 bit big endian integer)

Merkle-Damgard

- `a = 0x6a09e667`
- **crypto magic(block one)**
- `a = 0x16ec40b4`
- `0x16ec40b4 + 0x6a09e667 = 0x80f6271b`
- `a = 0x80f6271b`
- **crypto magic(block two)**
- `a = 0x216e8a1c`
- `0x216e8a1c + 0x80f6271b = 0xa264b137`
- `sha256("LuXeria" * 10) → a264b137af3d2...`

why is that important?

- I want to create vanity hashes of files
- I want to do it fast
- That means I don't want to compute n blocks of SHA-256

solution

- use files were we can append data without confusing the decoder (e.g. PNG)
- pad it to align with the block size
- compute SHA-256 hash without the last block
- patch hashcat to compute the last block using the internal state from above
- run it until you found your desired vanity hash

get a PNG file

add padding

- original size: 8292 bytes
- 8292 % 64 = 36
- 64 – 36 = 28
- padding size: 28 bytes
- new size: 8320 bytes

add padding

```
00002040: 015f 1b99 bef0 84fa b80c 8132 9b09 ff07  ._.........2....
00002050: a79e ff49 2351 6791 0000 0000 4945 4e44  ...I#Qg.....IEND
00002060: ae42 6082 2020 2020 2020 2020 2020 2020  .B`.
00002070: 4c75 5865 7269 6120 6973 2063 6f6f 6c21  LuXeria is cool!
```

- **PNG data**
- **our padding**

get internal state

- a = 0x60acebd1
- b = 0x66835eb3
- c = 0x7123f329
- d = 0xbe1cefb6
- e = 0x720c9552
- f = 0x5dbf4bfe
- g = 0xeda7aa75
- h = 0xa5233c14

patch hashcat

choose vanity prefix

```
$ python2 -c "print 'hops'.encode('hex')"
686f7073
```

run hashcat

```
$ ./hashcat -a3 -m 1400 -O --potfile-disable
--keep-guessing 686f707300000000 ?b?b?b?b?b

686f707300000000:$HEX[c6f2f8852f]

686f707300000000:$HEX[ce3b38d985]

686f707300000000:$HEX[a265100702]
```

add any of the above results to the padded image to get your vanity hash

profit

```
$ ./check.py lux_logo-padding.png c6f2f8852f

686f70735bcacaa0ee903dacda06839708f58357dcc5e
5f5158d3b3d18d1bd92

$ python2 -c "print '686f7073'.decode('hex')"

hops
```

image looks still the same

proof

```
$ xxd lux_logo-vanity-hops.png | tail -4
00002050: a79e ff49 2351 6791 0000 0000 4945 4e44  ...I#Qg.....IEND
00002060: ae42 6082 2020 2020 2020 2020 2020 2020  .B`.
00002070: 4c75 5865 7269 6120 6973 2063 6f6f 6c21  LuXeria is cool!
00002080: c6f2 f885 2f                             ..../
$ sha256sum lux_logo-vanity-hops.png
686f70735bcacaa0ee903dacda06839708f58357dcc5e5f5158d3b3d18d1bd92
```

why not LuXeria???

- computation cost grows exponential
- LuXeria would take up to 180 days on my GPUs
- possible solutions:
- add more GPUs! → expensive
- ignore case → meh, doesn't work for LuXeria
- use base64 → yay \o/

choose a bas64 prefix

```
foo = 'LuXeria='.decode('base64')
for i in xrange(256):
  tmp = foo + chr(i)
  if tmp.encode('base64').startswith('LuXeria'):
    print tmp.encode('hex')

$ python2 base64-prefix.py | cut -c-11 | sort -u | sed 's/$/00000/'
2ee5deae26800000
2ee5deae26900000
2ee5deae26a00000
2ee5deae26b00000
```

run hashcat

```
$ ./hashcat -m 1400 -a3 -O --keep-guessing
--potfile-disable b64.hashes ?d?b?b?b?b?b
```

- **~15 minutes later**

```
2ee5deae26800000:$HEX[371b39b8fbae]
```

profit

- `./check-b64.py lux_logo-padding.png 371b39b8fbae`
- `2ee5deae268a8f332d847d1b54762d1c2ae2b0884bb585407241adebcd9171d2`
- `LuXeriaKjzMthH0bVHYtHCrisIhLtYVAckGt682RcdI=`

todo

- turn hacky proof of concept into a framework
- use other hash algorithm
- md5(data) and sha1(data) with the same prefix
- vanity password hashes?
- your idea?

questions?


```
$ sha256sum didit.mp4
644964497421 4aaf7fb51dfb1ce5afb7a6f7ccbc805cc3e3c105006a70a32ab3
$ python2 -c "print '644964497421'.decode('hex')"
dIdIt!
```