On TOTP Standards

A rant and ramble on just some of the oddities of TOTP standards

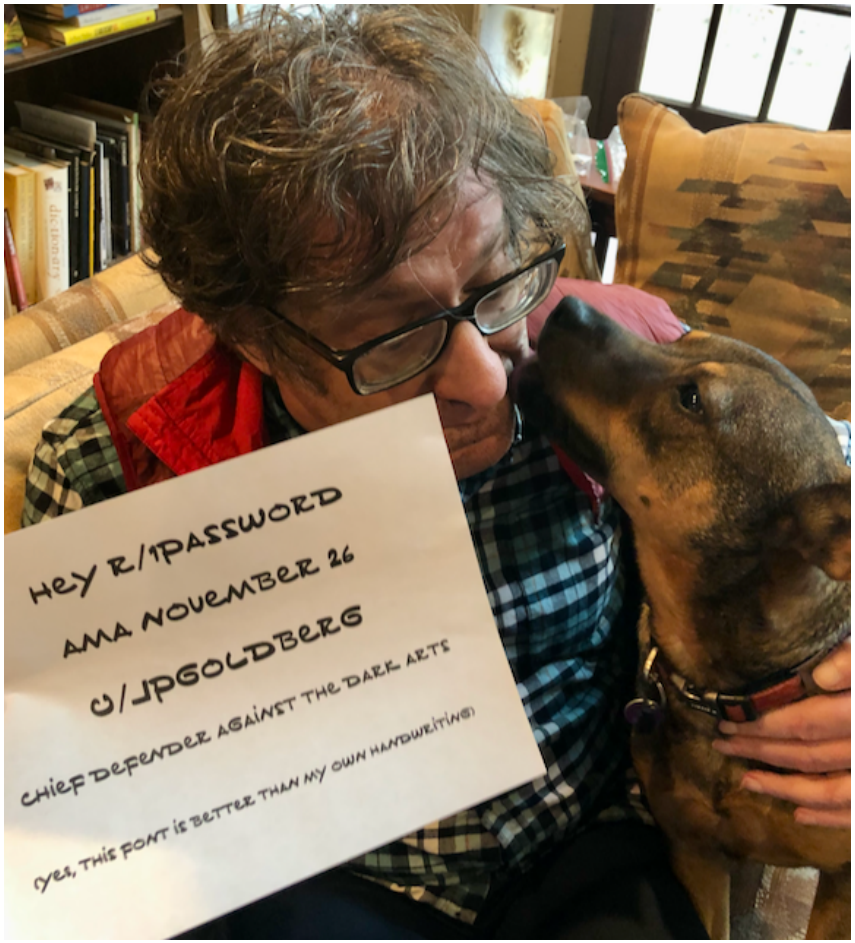
Jeffrey Goldberg
jeffrey@goldmark.org

AgileBits, Inc

Passwords19, 26 November 2019
Last Modified: June 10, 2024

AMA

r/1Password



Where	r/1Password
Who	Me, Pilar Garcia & 9 others
When	Today (26 November, 2019) 20:00 Central Europe Time. 14:00 Eastern Time.

Time-based One Time Passwords

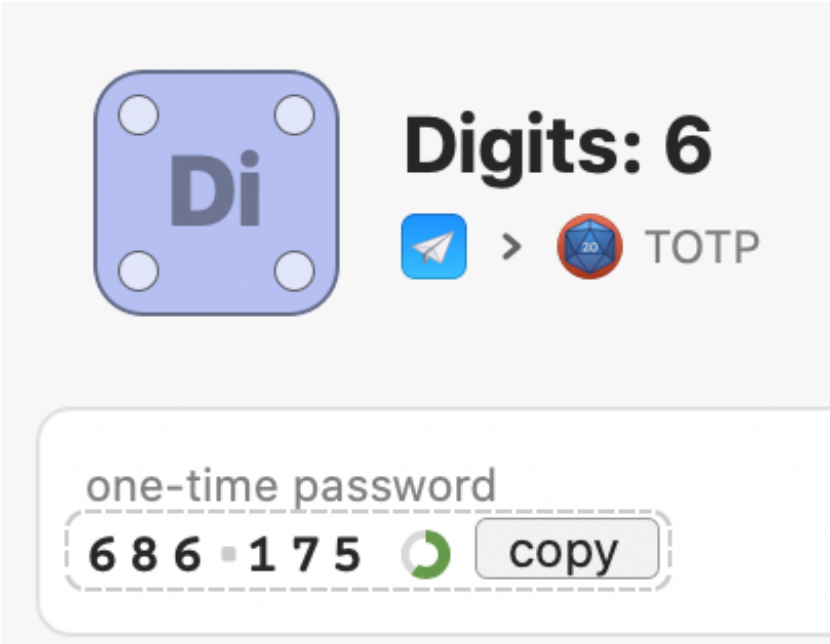
- Initial setup and transmission of long term secret typically by QR code.
- Authenticator apps generate one time password from current time and long term secret

TOTP Enrollment

1. Server generates a shared long term secret and constructs a totp-auth URI, like
   `otpauth://totp/alice@google.com?secret=JBSWY3DPEHPK3PXP`
2. Server presents that to user as a QR code
3. User scans it in to authenticator app

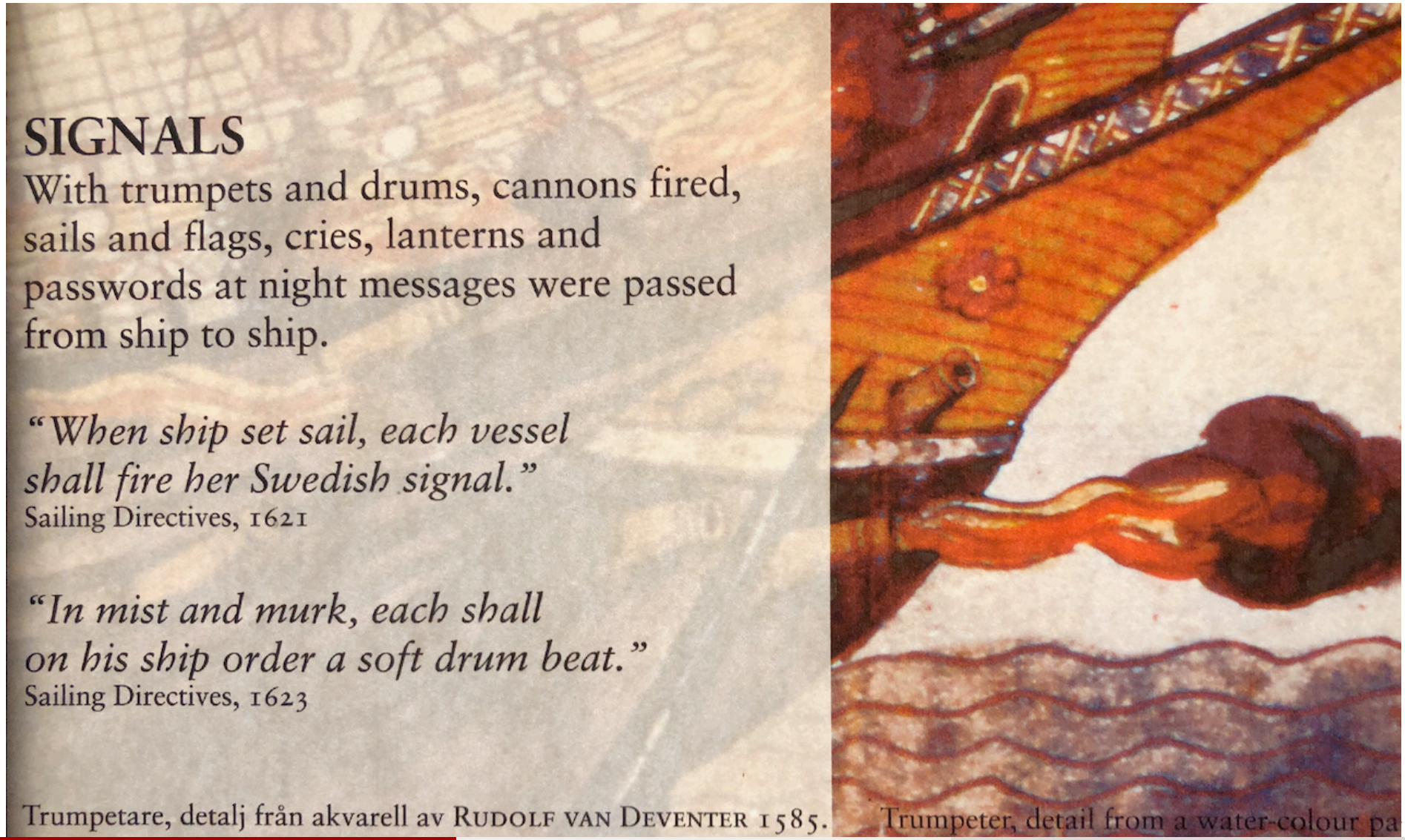
TOTP Use

The one time password is computed by the authenticator app based on the secret and number of seconds since the start of the epoch.



Why we want OTPs

Traditional passwords can by replayed



A good idea

Journalist What do you think of Western Civilization?

Gandhi I think it would be a good idea.

A good idea

Journalist What do you think of Western Civilization?

Gandhi I think it would be a good idea.

A good idea, too

Nobody What do you think of TOTP standards?

Me I think it would be a good idea.

A good idea, too

Nobody What do you think of TOTP standards?

Me I think it would be a good idea.

Standards matter

- Because they allow us to make things that work together
- Because they help us avoid security problems that arise from ambiguity

Dangerous ambiguity

- “Make sure that the kids come to the table and are ready to eat.”
- “Here’s a sharp knife. Get the tomatoes ready to eat.”
- ”Here’s a sharp knife. Get the chicken ready to eat.”

Dangerous ambiguity

- “Make sure that the kids come to the table and are ready to eat.”
- “Here’s a sharp knife. Get the tomatoes ready to eat.”
- ”Here’s a sharp knife. Get the chicken ready to eat.”

Dangerous ambiguity

- “Make sure that the kids come to the table and are ready to eat.”
- “Here’s a sharp knife. Get the tomatoes ready to eat.”
- ”Here’s a sharp knife. Get the chicken ready to eat.”

A good request

Example (Some HTTP request headers)

GET / HTTP/1.1
Host: www.example.com
Accept-Language: tlh, en-cockney, i-cherokee
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT

A bad request

Example (Malformed HTTP header)

```
GET / HTTP/1.1
Host: www.example.com
Accept-Language : tlh, en-cockney, i-cherokee
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

Surely no harm can come from accommodating non-conforming clients which might send such malformed headers, right?

A bad request

Example (Malformed HTTP header)

```
GET / HTTP/1.1
Host: www.example.com
Accept-Language : tlh, en-cockney, i-cherokee
If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT
```

Surely no harm can come from accommodating non-conforming clients which might send such malformed headers, right?

HTTP headers

No whitespace is allowed between the header field-name and colon. In the past, differences in the handling of such whitespace have led to security vulnerabilities in request routing and response handling. A server MUST reject any received request message that contains whitespace between a header field-name and colon [RFC7230 (2014) §3.2.4]

The problem with standards ...

is that we have so many of them

HOTP RFC	RFC 4226 “HOTP: An HMAC-Based One-Time Password Algorithm” (2005)
TOTP RFC	RFC 6238 ”TOTP: Time-Based One-Time Password Algorithm” (2011)
otp-auth	Google “Key Uri Format” (2011)
base32(ops)	RFC 3548 (obsolete) “The Base16, Base32, and Base64 Data Encodings“ (2003)
base32	RFC 4648 “The Base16, Base32, and Base64 Data Encodings” (2006)

HOTP RFC

Where the math is

- “HOTP: An HMAC-Based One-Time Password Algorithm” (2005)
- Only knows about SHA1
- Defines DT algorithm, which takes the HMAC output and produces a 31 bit result
- Defines algorithm for converting that 31-bit result to an *d*-digit code
- Doesn’t specify how long term secret is initially shared between client and server
- Has errata

TOTP RFC

Where the time is

- “RFC 6238 ”TOTP: Time-Based One-Time Password Algorithm” (2011)
- Says to use HOTP but with a time instead of a counter
- HMAC can use SHA-256
- Has lots of stuff about time intervals that we can ignore here
- Doesn’t specify how long term secret is initially shared between client and server

otp-auth scheme

Google makes TOTP usable

- Published with Google Authenticator source code
- Uri like
  `otpauth://totp/alice@google.com?secret=JBSWY3DPEHPK3PXP`
- Has parameters for setting hash function, numbers of digits, and more
- References obsolete Base32 RFC
- Has made it really easy for servers and users to actual set up and use TOTP

Base32(obs)

All your base are being obsolete

- RFC 3548 (obsolete) “The Base16, Base32, and Base64 Data Encodings“ (2003)
- Talks about Base16 and Base64, but we don’t care about those
- Base32 alphabet is all uppercase
- Bytes only
- One-to-one mapping

Base32

Encoding secrets

- RFC 4648 “The Base16, Base32, and Base64 Data Encodings” (2006)
- Talks about Base16 and Base64, but we don’t care about those
- Base32 alphabet is all uppercase
- Bytes only
- One-to-one mapping, unless you don’t want to
- Has warning about strict parsing

Digits allowed by otp-auth

otp-auth allows the setup to specify how many digits should be generated

Except from otp-auth on digits

The **digits** parameter may have the values 6 or 8, and determines how long of a one-time passcode to display to the user. The default is 6.

Currently, on Android and Blackberry the digits parameter is ignored by the Google Authenticator implementation.

Google Authenticator on iOS does respect the parameter.

Digits allowed by otp-auth

otp-auth allows the setup to specify how many digits should be generated

Except from otp-auth on digits

The **digits** parameter may have the values 6 or 8, and determines how long of a one-time passcode to display to the user. The default is 6.

Currently, on Android and Blackberry the digits parameter is ignored by the Google Authenticator implementation.

Google Authenticator on iOS does respect the parameter.

Digits allowed by otp-auth

otp-auth allows the setup to specify how many digits should be generated

Except from otp-auth on digits

The **digits** parameter may have the values 6 or 8, and determines how long of a one-time passcode to display to the user. The default is 6.

Currently, on Android and Blackberry the digits parameter is ignored by the Google Authenticator implementation.

Google Authenticator on iOS does respect the parameter.

HOTP RFC

The minimum

"The HOTP value must be at least a 6-digit value." [§4]

HOTP RFC

Other values

Lemma (Lemma 1 from HOTP RFC)

Let $N \geq m \geq 1$ be integers, and let $(q, r) = \mathrm{IntDiv}(N, m)$. For $z \in \mathbb{Z}_N$ let:

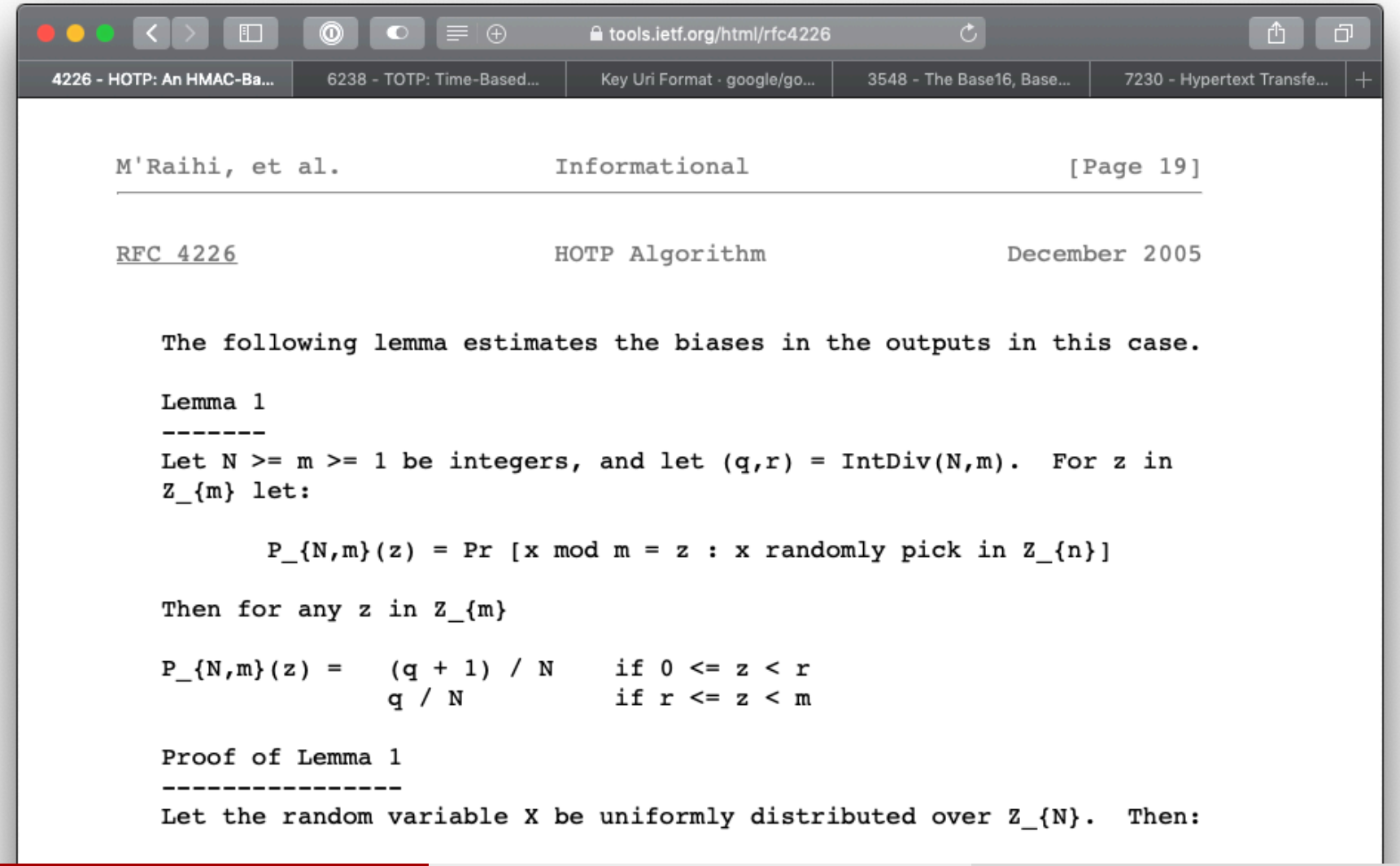
$$P_{N,m}(z) = \Pr[x \equiv z \bmod m : z \xleftarrow{\$} \mathbb{Z}_N]$$

Then for any $z \in Z_m$

$$P_{N,m}(z) = \begin{cases} (q+1)/N & \text{if } 0 \leq z < r \\ q/N & \text{if } r \leq z < m \end{cases} \tag{1}$$

HOTP RFC

Pictures or it didn't happen



Worked example

Example: $d = 9$

- With $N = 2^{31}$, $d = 9$, and $m = 10^d$
- $(q, r) = \mathrm{IntDiv}(N, m) = (2, 147483648)$

The very small value of q is our problem.

$$P_{N,2^9}(z) = \begin{cases} 3/2^{31} & \text{if } 0 \leq z < 147483648 \\ 2/2^{31} & \text{if } 147483648 \leq z < 10^9 \end{cases}$$

The min-entropy, H_∞, is then going to be $-\lg(3/2^{31}) \approx 29.415$

10x strength

Does going from d to $d + 1$ digits increase the strength of the code by 10 times?

- From 5 digits to 6 increases strength 9.99767 times (close enough to 10x)
- From 6 to 7 increases strength 9.99070 times (close enough to 10x)
- From 7 to 8 increases strength 9.77273 times (close enough to 10x)
- From 8 digits to 9 increases strength 7.33333 times
- Going from 9 digits to 10 increases strength 3.00000 times
- From 10 to 11 increases strength 1 time (not at all)

10x strength

Does going from d to $d + 1$ digits increase the strength of the code by 10 times?

- From 5 digits to 6 increases strength 9.99767 times (close enough to 10x)
- From 6 to 7 increases strength 9.99070 times (close enough to 10x)
- From 7 to 8 increases strength 9.77273 times (close enough to 10x)
- From 8 digits to 9 increases strength 7.33333 times
- Going from 9 digits to 10 increases strength 3.00000 times
- From 10 to 11 increases strength 1 time (not at all)

10x strength

Does going from d to $d + 1$ digits increase the strength of the code by 10 times?

- From 5 digits to 6 increases strength 9.99767 times (close enough to 10x)
- From 6 to 7 increases strength 9.99070 times (close enough to 10x)
- From 7 to 8 increases strength 9.77273 times (close enough to 10x)
- From 8 digits to 9 increases strength 7.33333 times
- Going from 9 digits to 10 increases strength 3.00000 times
- From 10 to 11 increases strength 1 time (not at all)

10x strength

Does going from d to $d + 1$ digits increase the strength of the code by 10 times?

- From 5 digits to 6 increases strength 9.99767 times (close enough to 10x)
- From 6 to 7 increases strength 9.99070 times (close enough to 10x)
- From 7 to 8 increases strength 9.77273 times (close enough to 10x)
- From 8 digits to 9 increases strength 7.33333 times
- Going from 9 digits to 10 increases strength 3.00000 times
- From 10 to 11 increases strength 1 time (not at all)

10x strength

Does going from d to $d + 1$ digits increase the strength of the code by 10 times?

- From 5 digits to 6 increases strength 9.99767 times (close enough to 10x)
- From 6 to 7 increases strength 9.99070 times (close enough to 10x)
- From 7 to 8 increases strength 9.77273 times (close enough to 10x)
- From 8 digits to 9 increases strength 7.33333 times
- Going from 9 digits to 10 increases strength 3.00000 times
- From 10 to 11 increases strength 1 time (not at all)

10x strength

Does going from d to $d + 1$ digits increase the strength of the code by 10 times?

- From 5 digits to 6 increases strength 9.99767 times (close enough to 10x)
- From 6 to 7 increases strength 9.99070 times (close enough to 10x)
- From 7 to 8 increases strength 9.77273 times (close enough to 10x)
- From 8 digits to 9 increases strength 7.33333 times
- Going from 9 digits to 10 increases strength 3.00000 times
- From 10 to 11 increases strength 1 time (not at all)

10x strength

Does going from d to $d + 1$ digits increase the strength of the code by 10 times?

- From 5 digits to 6 increases strength 9.99767 times (close enough to 10x)
- From 6 to 7 increases strength 9.99070 times (close enough to 10x)
- From 7 to 8 increases strength 9.77273 times (close enough to 10x)
- From 8 digits to 9 increases strength 7.33333 times
- Going from 9 digits to 10 increases strength 3.00000 times
- From 10 to 11 increases strength 1 time (not at all)

Strength versus user expectations

d	H_∞	H if uniform	$H - H_\infty$	x-fold increase from $d-1$
d	H_∞	$-\lg 10^d$	$H - H_\infty$	$2^{H_\infty(d)-H_\infty(d-1)}$
6	19.931	19.932	0.00035	9.99767
7	23.252	23.253	0.00169	9.99070
8	26.541	26.575	0.03486	9.77273
9	29.415	29.897	0.48232	7.33333
10	31	33.219	2.21928	3.00000
$d > 10$	31	$-\lg 10^d$		1

Table: d: number of digits;
H_∞: Min-entropy;
H if uniform: Entropy of d digit number chosen uniformly;
$H - H_\infty$: Difference between uniform and min-entropy.

Maximum digits

8 Makes perfect sense

9 Somewhat theatrical

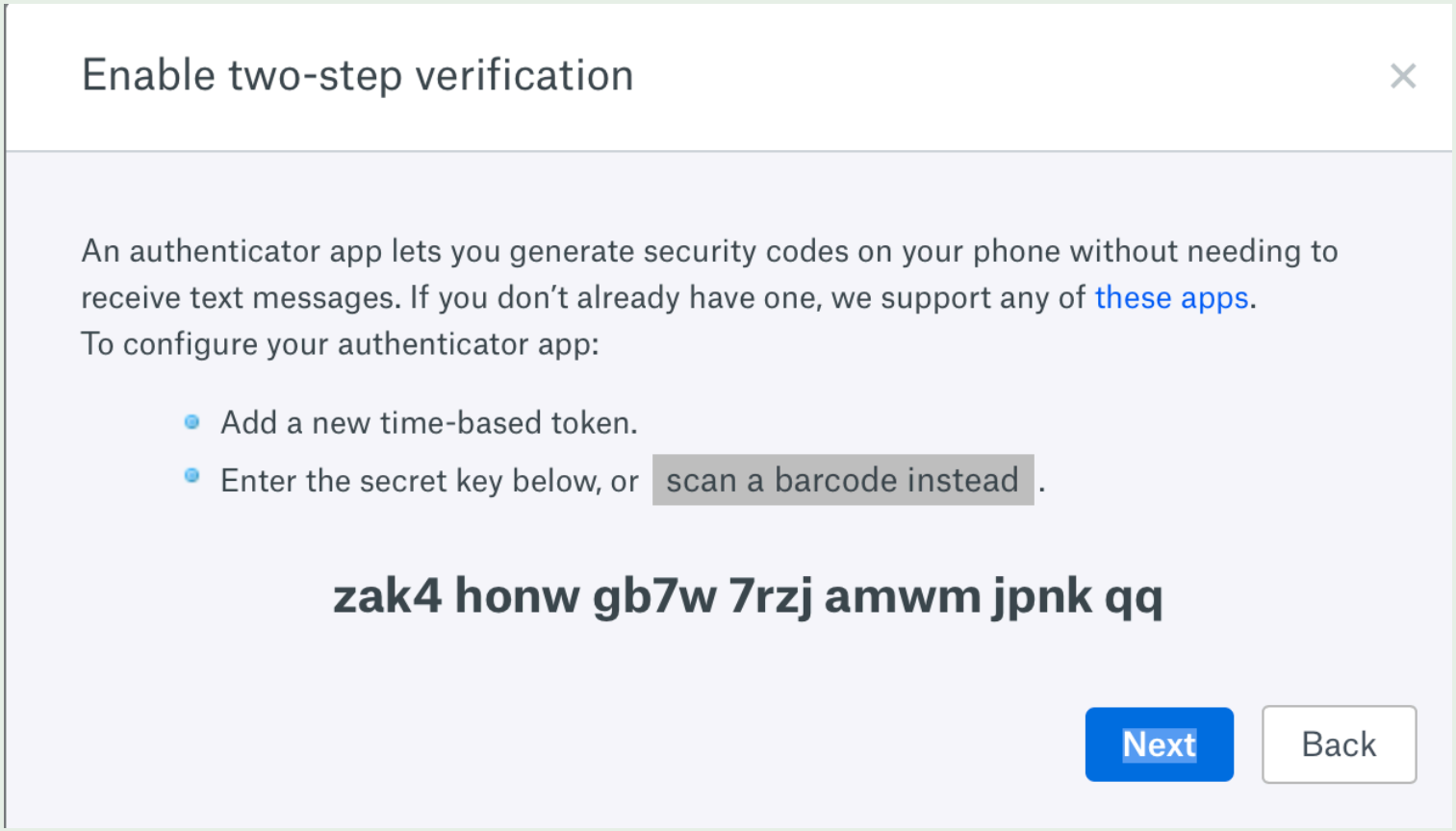
10 Very theatrical

11 absurd by any standard

Manual entry

- otp-auth has the long term secret encoded using base32
- Sometimes users enter only the secret manually

Example (A site offering manual entry of secret)



Spaces & Lowercase

- It makes sense to strip out whitespace, making it easier for people to correct things.
- It makes perfect sense to read case insensitively
- No where is this in any thing resembling a standard

Pasting the wrong thing

Suppose a user pages the wrong thing, and so enters a string like `https://example.com/2fa-setup?r=foo` Should a reader

1. Report an error and not construct a TOTP secret?
2. Truncate at first non-base32 character and so use "`HTTPS`" as the base32 encoding?
3. Strip out all non-base32 characters and so use "`HTTPSEXAMPLECOM2FASETUPRFOO`"?

Most apps do (3)

Pasting the wrong thing

Suppose a user pages the wrong thing, and so enters a string like `https://example.com/2fa-setup?r=foo` Should a reader

1. Report an error and not construct a TOTP secret?
2. Truncate at first non-base32 character and so use "`HTTPS`" as the base32 encoding?
3. Strip out all non-base32 characters and so use "`HTTPSEXAMPLECOM2FASETUPRFOO`"?

Most apps do (3)

Non-spaces

Consider a secret like `ABCDE<script>nastiness()FGHIKL` Should a reader

1. Report an error and not construct a TOTP secret?
2. Truncate at non-base32 character and so use "`ABCDE`" as the base32 encoding?
3. Strip out all non-base32 characters and so use "`ABCDESCRIPTNASTINESSFGHIKL`"?

My opinion is that (1) is the best choice.

Non-spaces

Consider a secret like `ABCDE<script>nastiness()FGHIKL` Should a reader

1. Report an error and not construct a TOTP secret?
2. Truncate at non-base32 character and so use "`ABCDE`" as the base32 encoding?
3. Strip out all non-base32 characters and so use "`ABCDESCRIPTNASTINESSFGHIKL`"?

My opinion is that (1) is the best choice.

And I am not alone

If non-alphabet characters are ignored, instead of causing rejection of the entire encoding (as recommended),a covert channel that can bemused to ”leak” information is made possible. The ignored characters could also be used for other nefarious purposes, such as to avoid a string equality comparison or to trigger implementation bugs. The implications of ignoring non-alphabet characters should be understood in applications that do not follow the recommended practice. [RFC 4648 (2006)]

SHA-1

The once and future hash

Excerpt from otp-auth

OPTIONAL: The algorithm may have the values:

- SHA1 (Default)
- SHA256
- SHA512

Currently, the algorithm parameter is ignored by the Google Authenticator implementations.

As a consequence, servers only go with the default, SHA1

SHA-1

The once and future hash

Excerpt from otp-auth

OPTIONAL: The algorithm may have the values:

- SHA1 (Default)
- SHA256
- SHA512

Currently, the algorithm parameter is ignored by the Google Authenticator implementations.

As a consequence, servers only go with the default, SHA1

SHA-1

The once and future hash

Excerpt from otp-auth

OPTIONAL: The algorithm may have the values:

- SHA1 (Default)
- SHA256
- SHA512

Currently, the algorithm parameter is ignored by the Google Authenticator implementations.

As a consequence, servers only go with the default, SHA1

Bytes & Trailing bits

Encoding `0x10101010`

0 1 2 3 4 5 6 7 8 9

1 0 1 0 1 0 1 0		
1 0 1 0 1	0 1 0	0 0
V	I	
Significant		Tr

Non-canonical

Non-canonical encoding `0x10101010`

0 1 2 3 4 5 6 7	8 9
1 0 1 0 1 0 1 0	
1 0 1 0 1	0 1 0	0 1
V	J	
1 0 1 0 1	0 1 0	1 0
V	K	
1 0 1 0 1	0 1 0	1 1
V	L	

Invalid length

Invalid length `0x10101010`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	1	0	1	0	1	0							
1	0	1	0	1	0	1	0	0	0	0	1	0	1	0
V					I					K				

When length of base32 string is $\{1, 3, 6\} \pmod 8$

Use the source, Luke!

Google Authenticator Android

From `util/Base32String.java`

```
/**
 * The implementation is slightly different than in
 * RFC 4648. During encoding, padding is not added,
 * and during decoding the last incomplete chunk is
 * not taken into account. The result is that multiple
 * strings decode to the same byte array, for example,
 * string of sixteen 7 ("7...7") and seventeen 7s both
 * decode to the same byte array.
 *
 * TODO: Revisit this encoding and whether this
 * ambiguity needs fixing.
 */
```

Source is a well of info

More from GA source

Base32String.java lines 80–85

```
// We'll ignore leftover bits for now.
//
// if (next != outLength || bitsLeft >= SHIFT) {
//  throw new DecodingException("Bits left: " + bitsLeft);
// }
return result;
```

Long term secret length

What does the HOTP RFC Say?

> *The algorithm MUST use a strong shared secret. The length of the shared secret MUST be at least 128 bits. This document RECOMMENDs a shared secret length of 160 bits. [RFC 4226 (2005), §4]*

128 bits requires a 26 character base32 string

Long term secret length

What does the HOTP RFC Say?

> *The algorithm MUST use a strong shared secret. The length of the shared secret MUST be at least 128 bits. This document RECOMMENDs a shared secret length of 160 bits. [RFC 4226 (2005), §4]*

128 bits requires a 26 character base32 string

What should apps do?

Many authenticator apps accept secrets as short as a single byte. Should they

- Continue as they are?
- Reject setups with short secrets?
- Warn users about short secrets?

yjotp

Because things aren't weird enough

If you get bored with the `otpauth` scheme, you can use `yjotp` as the scheme in the URI. At least those are things created by Yahoo! Japan.

Who am I?

- Jeffrey Goldberg
- Email: `jeffrey@goldmark.org`
- Mastodon: `jpgolderg@ioc.exchange`
- Keybase: `jpgoldberg`
- Twitter/X: `jpgoldberg`
- These slides:
  `https://jeffrey.goldmark.org/uploads/totp-talk.pdf`