Secure Application Design

Authentication

Summer 2025

Jakob Heher, www.isec.tugraz.at
he/his

GitLab Community Edition

Username or email

jakob.heher@iaik.tugraz.at

Password

•••

☐ Remember me

Forgot your password?

Sign in

IAIK: Teaching related repository management.

By signing in you accept the Terms of Use and acknowledge the Privacy Policy and Cookie Policy.


Registration Phase


Authentication Phase

“Traditional” Password Registration

- User sends username $\mathbf{u}$ and password $\mathbf{p}$ to Server
- Server generates random salt $\boldsymbol{s}_{\mathrm{u}}$ and calculates $\mathbf{d}_{\mathrm{u}}$ as $\mathbf{H}(\mathbf{p}, \boldsymbol{s}_{\mathrm{u}})$
- Server stores $\boldsymbol{s}_{\mathrm{u}}$ and $\mathbf{d}_{\mathrm{u}}$ indexed by $\mathbf{u}$

“Traditional” Password Authentication

- User sends username $\mathbf{u}$ and password $\mathbf{p}$ to Server
- Server retrieves stored salt $\boldsymbol{s}_{\mathbf{u}}$ and digest $\mathbf{d}_{\mathbf{u}}$ based on $\mathbf{u}$
- Server calculates $\mathbf{H}(\mathbf{p}, \boldsymbol{s}_{\mathbf{u}})$ and checks $\mathbf{H}(\mathbf{p}, \boldsymbol{s}_{\mathbf{u}}) == \mathbf{d}_{\mathbf{u}}$

- Problem: password is transmitted to the server!
  - User has to trust server to handle it properly
  - Server has to worry about it being logged, leaked, etc.

Password Authentication – Issues

- Phishing sites might trick users
  - This is essentially a MitM attack (*online* or *offline*)
- Passwords are routinely re-used
  - No, *you* are not the typical user
- Cleartext passwords are sent to the server
  - You need to trust the server to handle them responsibly
  - The server has to worry about accidental logging, in-memory compromise, …
- Compromised credentials are valid forever
  - Password expiration is not a great solution

(Cryptographic)

Authentication Factors

Password-Authenticated Key Exchange (PAKE)

(Asymmetric)

Password Authenticated Key Exchange

- Idea: prove we *know* the password without *showing* the password

- Server doesn’t need to worry about handling the password
- Client doesn’t need to trust the server implementation

- Complications:
  - We don’t want to store any key material on the client!
  - We still want to be able to throttle brute-force attempts

(Asymmetric)

Password Authenticated Key Exchange

- Idea: prove we *know* the password without *showing* the password

- **<u>O</u>**blivious **<u>P</u>**seudo-**<u>R</u>**andom **<u>F</u>**unction
    - Client has input $\mathbf{x}$
    - Server has secret key $\mathbf{k}$
    - … magic happens …
    - Client obtains $\mathbf{f(x, k)}$, but no information about $\mathbf{k}$
    - Server obtains no information about either $\mathbf{x}$ or $\mathbf{f(x, k)}$

See the RFC: https://datatracker.ietf.org/doc/draft-irtf-cfrg-voprf/

(A Quick Recap of)
Elliptic-Curve Operations

- Curve Points (Uppercase) and scalars ($l$owercase)

- Point addition: $A + B = C$
- Scalar multiplication: $s \cdot P = P + P + P + \cdots + P = Q$
  - $s \cdot P$ is similar to r^s in modulo arithmetic
  - Given P and $s \cdot P$, it is hard to find s!
- Multiplicative inverse: $s^{-1} \cdot Q = s^{-1} \cdot s \cdot P = P$
  - Given s, it is easy to find s^{-1}!

See the RFC: https://datatracker.ietf.org/doc/draft-irtf-cfrg-voprf/

(An example of an)

Oblivious Pseudo-Random Function

- Client has secret input p

p

$H(p)$

$r \cdot H(p)$

- Client learns $F(p, k)$, but nothing about k

- Server has secret key k

k

- Server learns nothing about p, $F(p, k)$

(An example of an)

Oblivious Pseudo-Random Function

- Client has secret input p

p

$H(p)$

$r \cdot H(p)$

- Server has secret key k

k

$r \cdot H(p)$

$k \cdot r \cdot H(p)$

- Client learns $F(p, k)$, but nothing about k
- Server learns nothing about p, $F(p, k)$

(An example of an)

Oblivious Pseudo-Random Function

- Client has secret input p

$$p$$

$$H(p)$$

$$r \cdot H(p)$$

$$k \cdot r \cdot H(p)$$

$$r^{-1} \cdot k \cdot r \cdot H(p)$$

- Client learns $F(p,k)$, but nothing about k

- Server has secret key k

$$r \cdot H(p)$$

$$k \cdot r \cdot H(p)$$

- Server learns nothing about p, $F(p,k)$

(An example of an)

Oblivious Pseudo-Random Function

- Client has secret input p

$$p$$

$$H(p)$$

$$r \cdot H(p)$$

$$k \cdot r \cdot H(p)$$

$$F(p, k) \coloneqq k \cdot H(p)$$

- Client learns $F(p, k)$, but nothing about k

- Server has secret key k

$$r \cdot H(p)$$

$$k \cdot r \cdot H(p)$$

- Server learns nothing about p, $F(p, k)$

See also:

705.054 Privacy-Enhancing Technologies

(A conceptual overview of)

The OPAQUE Protocol – Registration

- Idea: prove we *know* the password without *showing* the password

- Client generates asymmetric key pair $(\mathbf{K_{pub}}, \mathbf{K_{priv}})$
- Client sends $\mathbf{K_{pub}}$ to the server
- Server generates a random user-specific OPRF secret key $\mathbf{L_u}$
- Client & Server perform OPRF protocol
  - Client input = password $\mathbf{p}$; Server key = $\mathbf{L_u}$
  - Client learns derived key $\mathbf{f(p, L_u)}$
- Client encrypts $\mathbf{K_{priv}}$ with key $\mathbf{f(p, L_u)}$ and sends it to the server

For all the gory details, read the RFC draft: https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-02.html

(A conceptual overview of)

The OPAQUE Protocol – Authentication

- Idea: prove we *know* the password without *showing* the password

- Server retrieves $\mathbf{L_u}$, $\mathbf{K_{pub}}$, and the encrypted $\mathbf{K_{priv}}$
- Server sends the encrypted $\mathbf{K_{priv}}$ to the user
- Client & Server perform OPRF protocol
    - Client input = password $\mathbf{p}$; Server key = $\mathbf{L_u}$
    - Client learns derived key $\mathbf{f(p, L_u)}$
- Client uses $\mathbf{f(p, L_u)}$ to decrypt $\mathbf{K_{priv}}$ and authenticate

For all the gory details, read the RFC draft: https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-02.html

The Web Context Dilemma

- Who supplies your client code?
  - The server!
- If an attacker controls the server, will they run your cryptography?
  - No, they’ll just send the password input in plain text…
- What attack scenario are you defending against?

(Cryptographic)

Authentication Factors

Time-Based One-Time Password (TOTP)

Multi-Factor Authentication

- Idea: we want a safeguard against password compromise

- Authentication factor categories:
  - Proving **knowledge** of some information — Something you **know**
  - Proving **possession** of some device — Something you **have**
  - Proving **inherence** of some property — Something you **are**

User Settings

Account

User Settings › Account › Two-Factor Authentication

Register Two-Factor Authenticator

Use a one-time password authenticator on your mobile device or computer to enable two-factor authentication (2FA).

We recommend using cloud-based authenticator applications that can restore access if you lose your hardware device. What are some examples?

otpauth://totp/git.teaching.iaik.tugraz.at:git.teaching.iaik.tugraz.at_jakob.heher%40iaik.tugraz.at?secret=4OWR2BWMIZFMG7WYHMIFTNQAKLHV43N6&issuer=git.teaching.iaik.tugraz.at

Can't scan the code?

To add the entry manually, provide the following details to the application on your phone.

Account: git.teaching.iaik.tugraz.at:jakob.heher@iaik.tugraz.at
Key: `4OWR 2BWM IZFM G7WY HMIF TNQA KLHV 43N6`
Time based: Yes

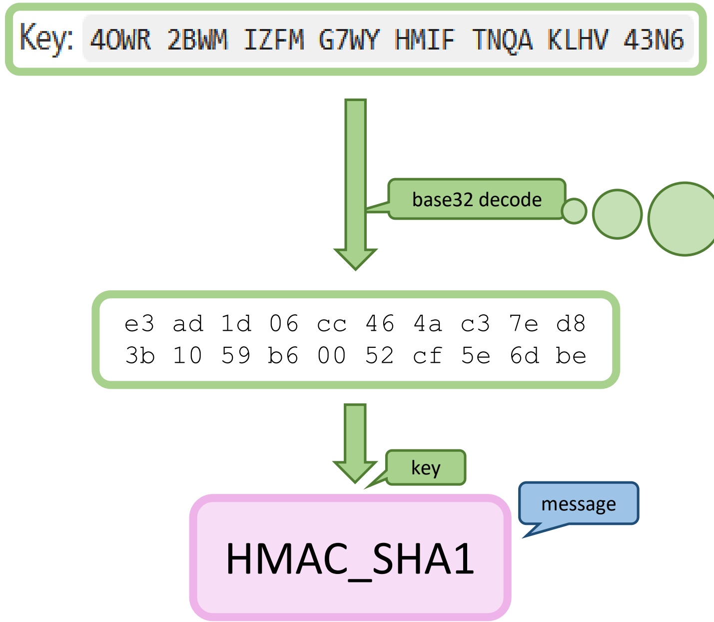
Pin code

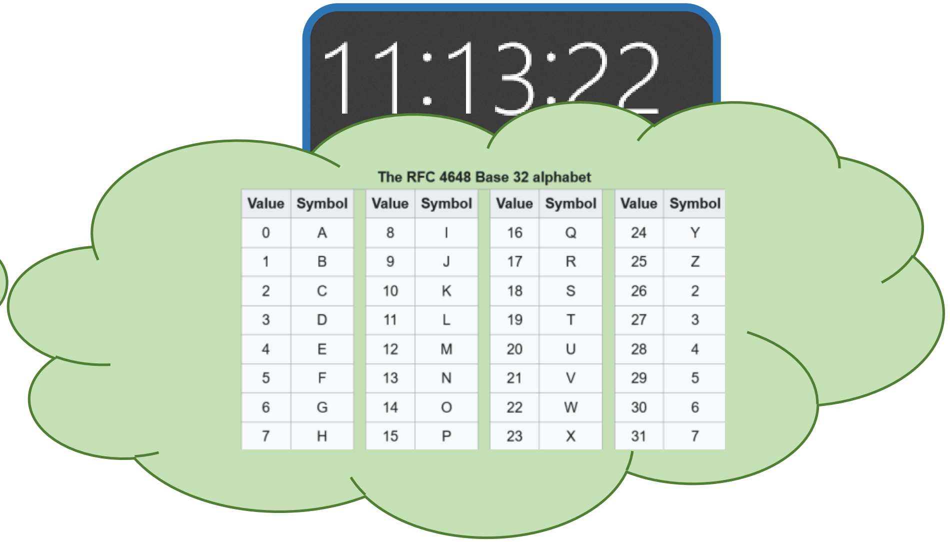
Current password

Your current password is required to register a two-factor authenticator app.

Register with two-factor app

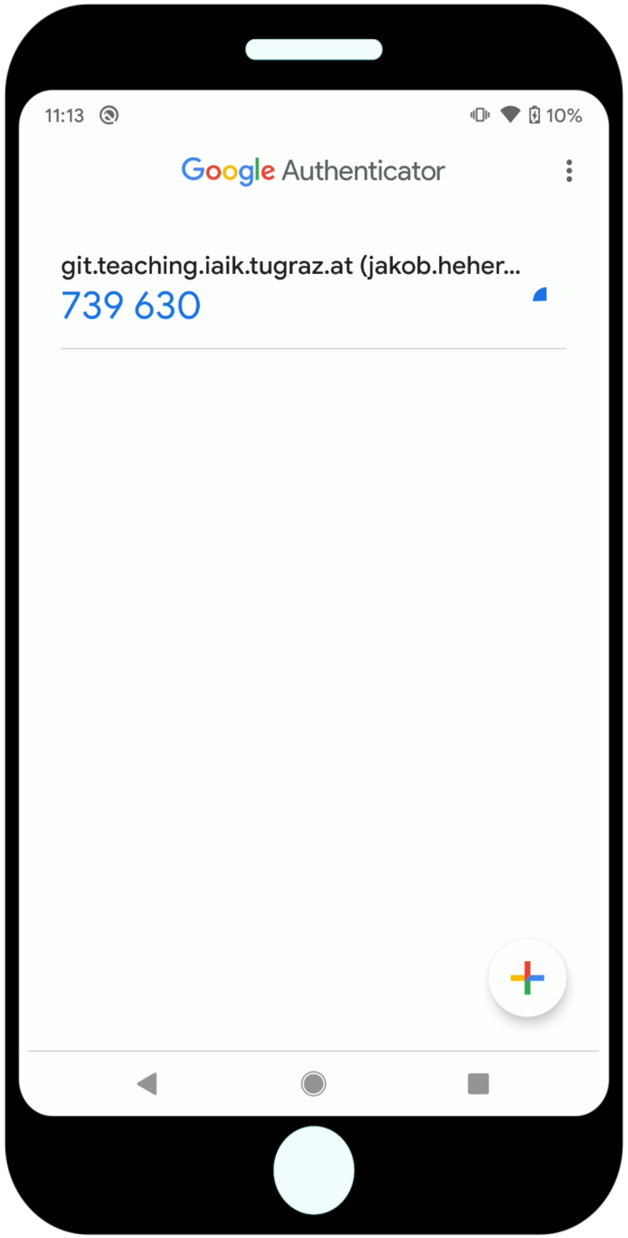
Collapse sidebar

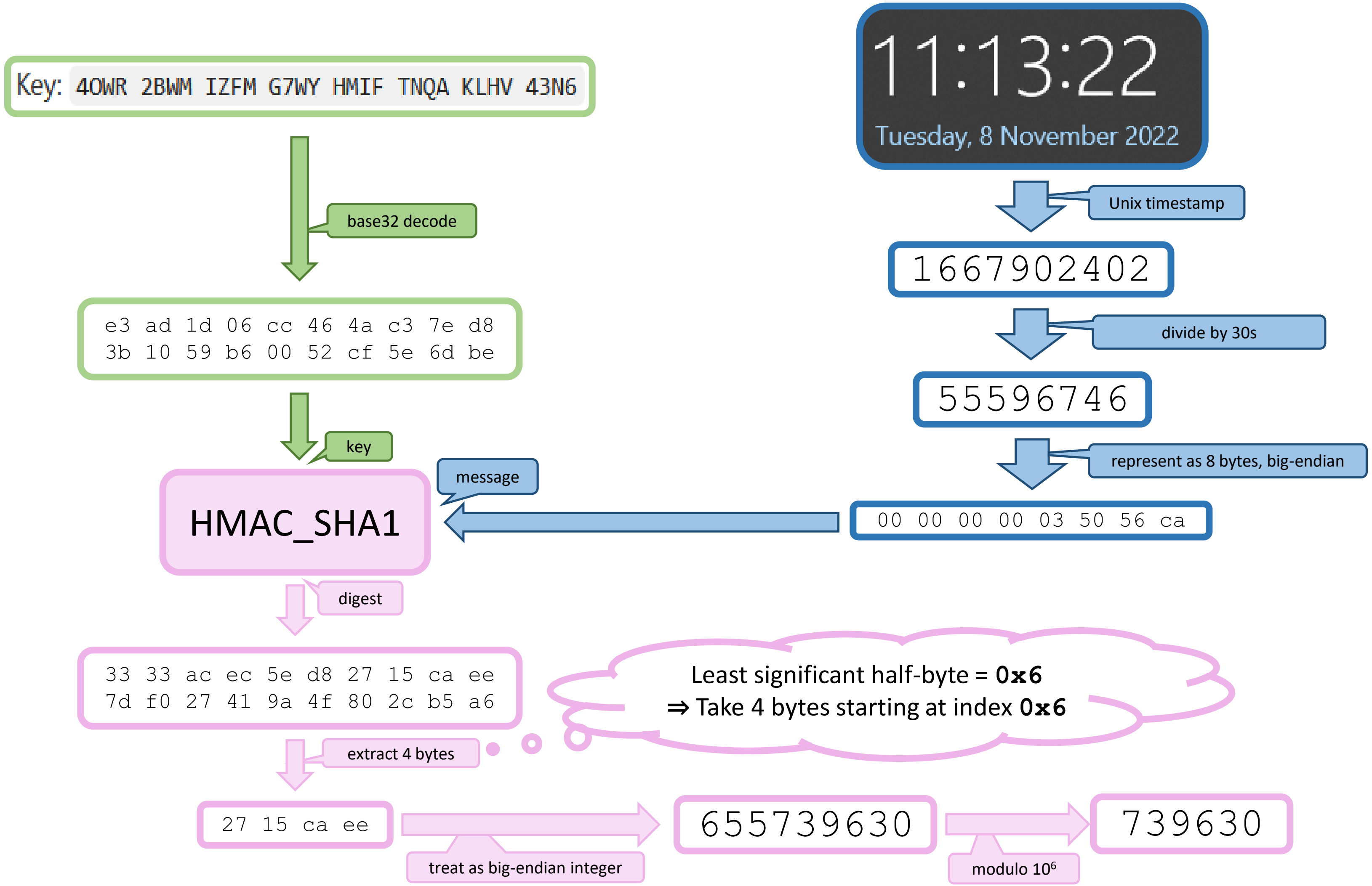


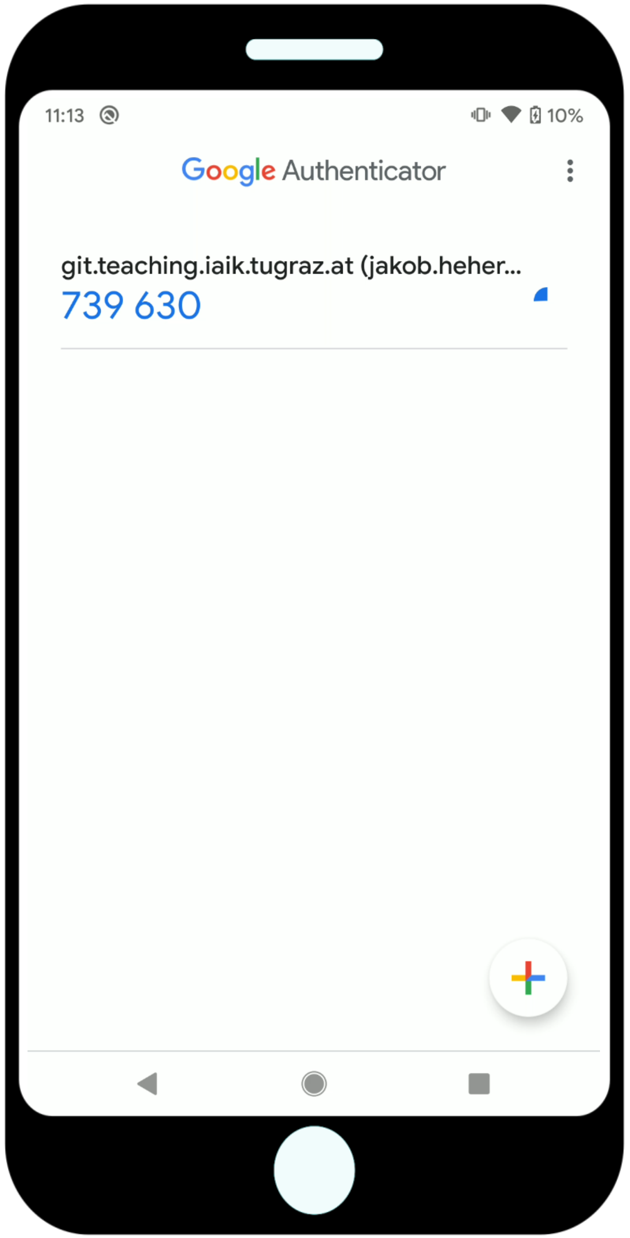


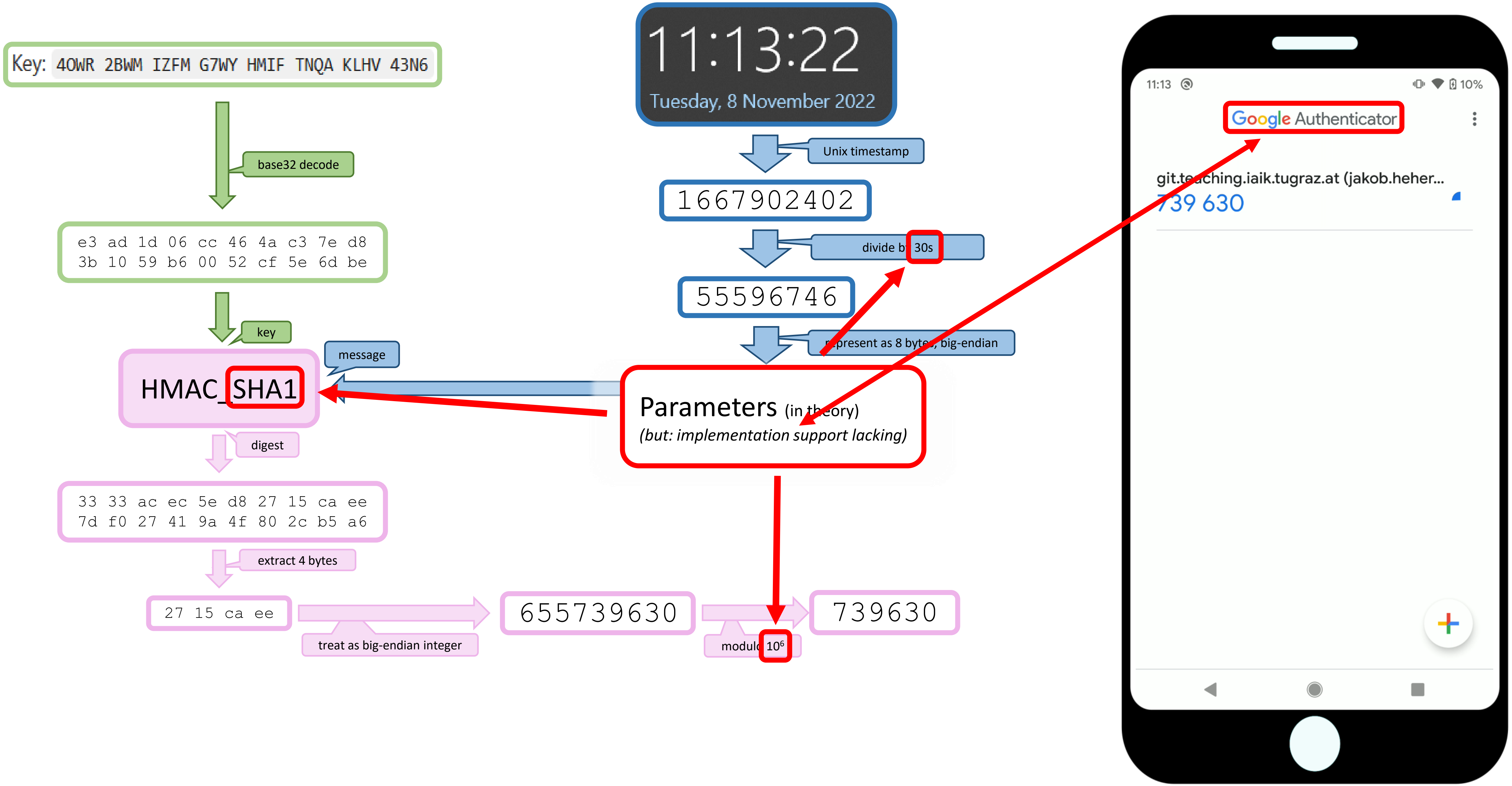
The RFC 4648 Base 32 alphabet

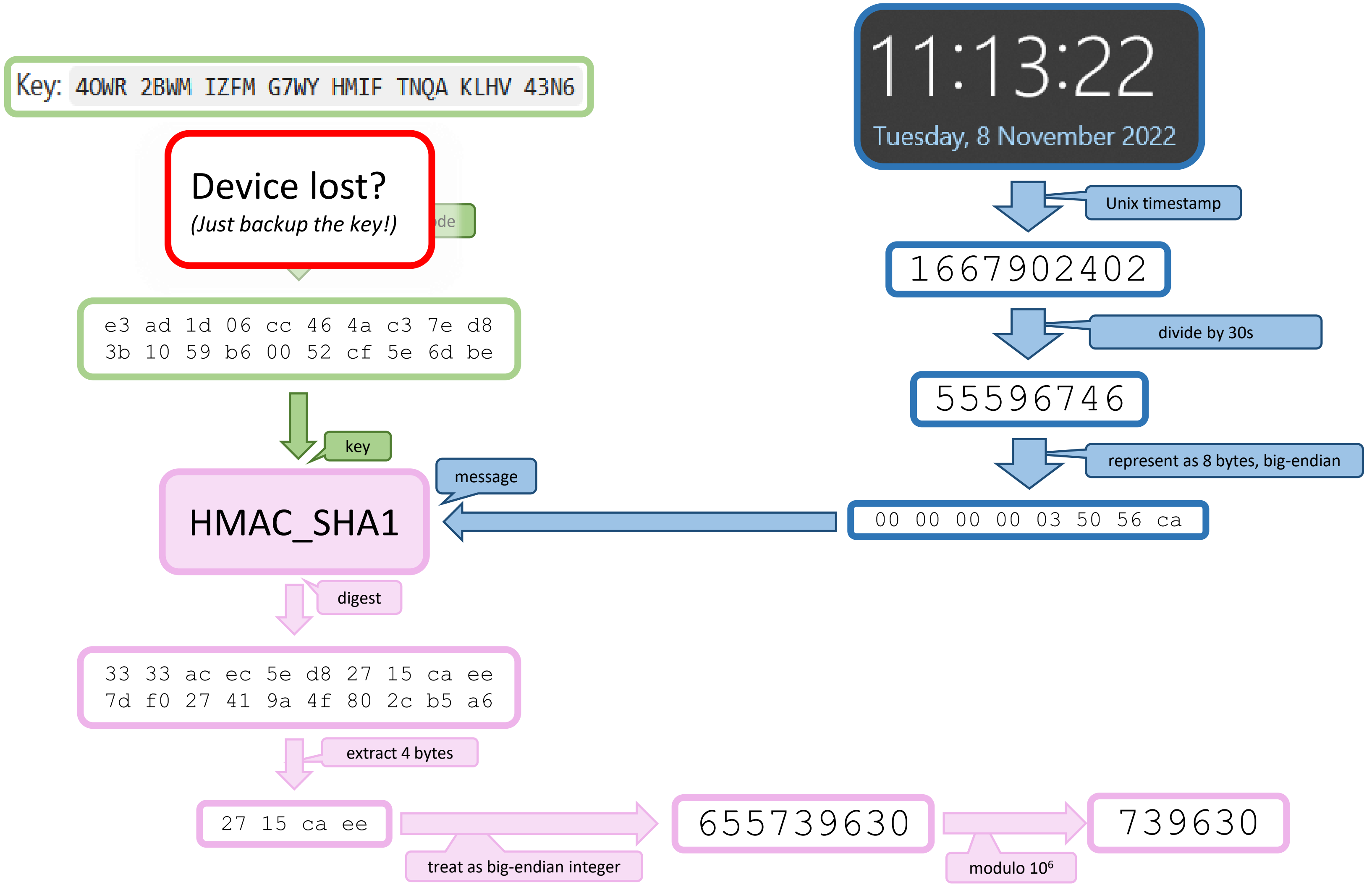
Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
0	A	8	I	16	Q	24	Y
1	B	9	J	17	R	25	Z
2	C	10	K	18	S	26	2
3	D	11	L	19	T	27	3
4	E	12	M	20	U	28	4
5	F	13	N	21	V	29	5
6	G	14	O	22	W	30	6
7	H	15	P	23	X	31	7

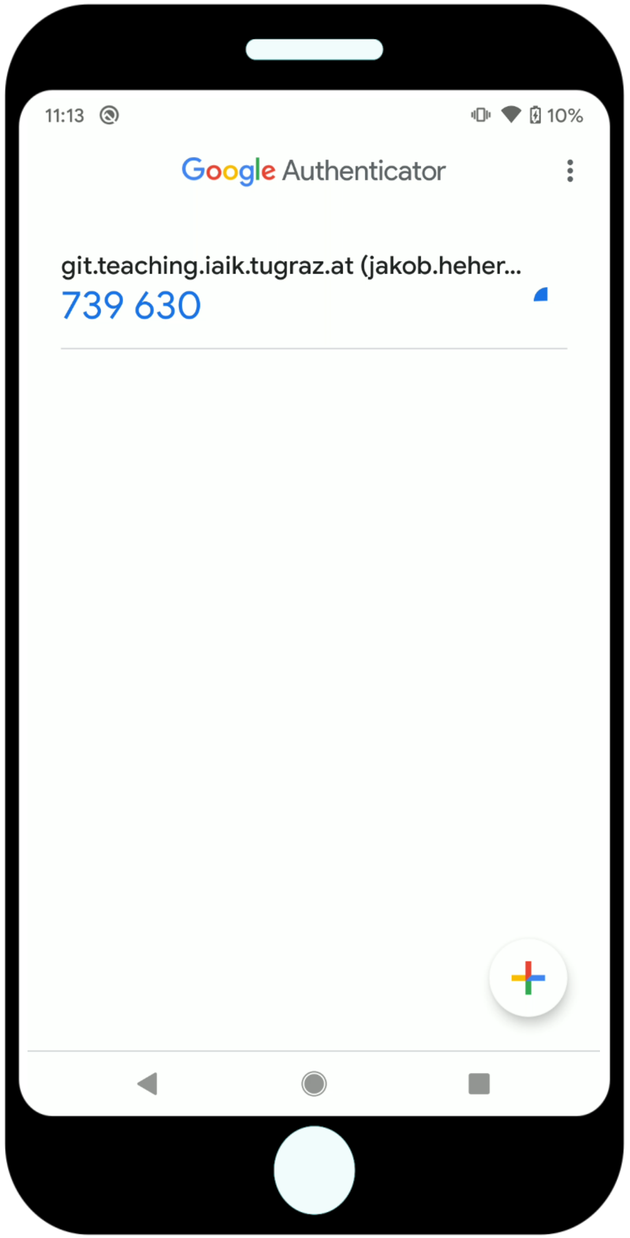


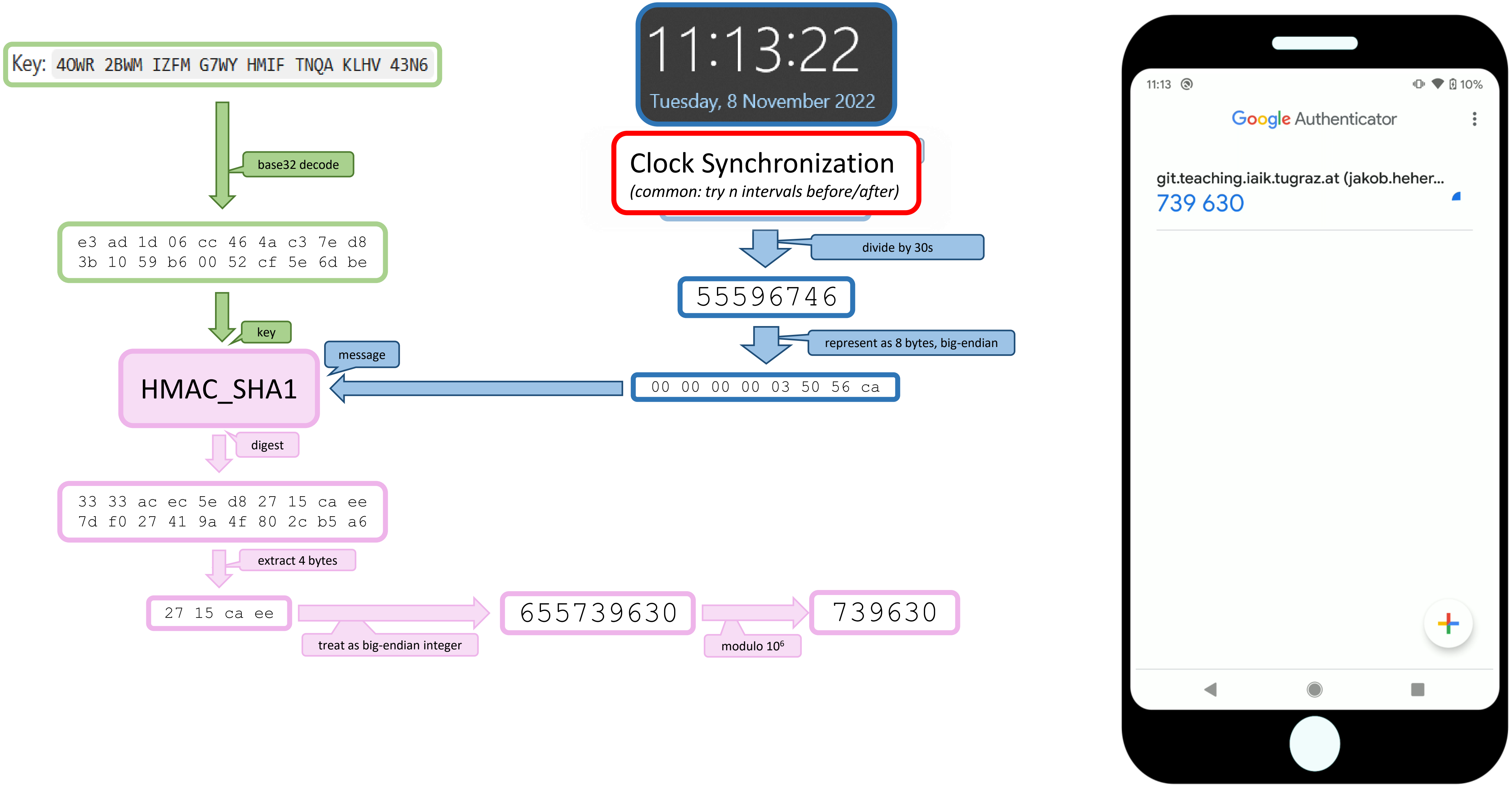

Key: 4OWR 2BWM IZFM G7WY HMIF TNQA KLHV 43N6
base32 decode
e3 ad 1d 06 cc 46 4a c3 7e d8
3b 10 59 b6 00 52 cf 5e 6d be
key
HMAC_SHA1
message
digest
33 33 ac ec 5e d8 27 15 ca ee
7d f0 27 41 9a 4f 80 2c b5 a6
extract 4 bytes
Least significant half-byte = 0x6
⇒ Take 4 bytes starting at index 0x6
27 15 ca ee
treat as big-endian integer
655739630
modulo 10^6
739630
11:13:22
Tuesday, 8 November 2022
Unix timestamp
1667902402
divide by 30s
55596746
represent as 8 bytes, big-endian
00 00 00 00 03 50 56 ca


11:13
10%
Google Authenticator
git.teaching.iaik.tugraz.at (jakob.heher...
739 630

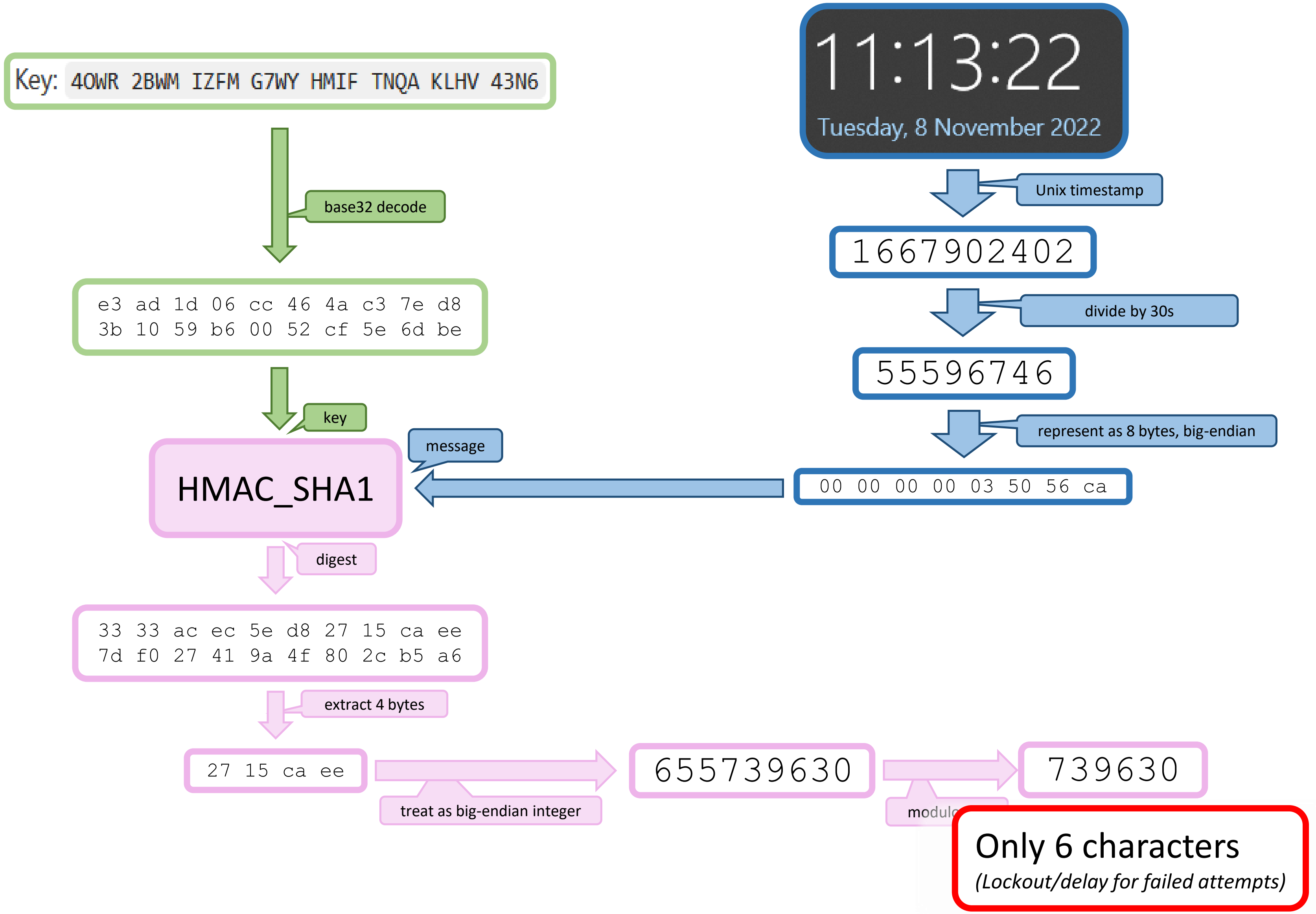

Key: 4OWR 2BWM IZFM G7WY HMIF TNQA KLHV 43N6
base32 decode
e3 ad 1d 06 cc 46 4a c3 7e d8
3b 10 59 b6 00 52 cf 5e 6d be
key
message
HMAC_SHA1
digest
33 33 ac ec 5e d8 27 15 ca ee
7d f0 27 41 9a 4f 80 2c b5 a6
extract 4 bytes
27 15 ca ee
treat as big-endian integer
655739630
modulo 10^6
739630
11:13:22
Tuesday, 8 November 2022
Unix timestamp
1667902402
divide by 30s
55596746
represent as 8 bytes, big-endian
Parameters (in theory)
(but: implementation support lacking)
11:13
10%
Google Authenticator
git.teaching.iaik.tugraz.at (jakob.heher...
739 630

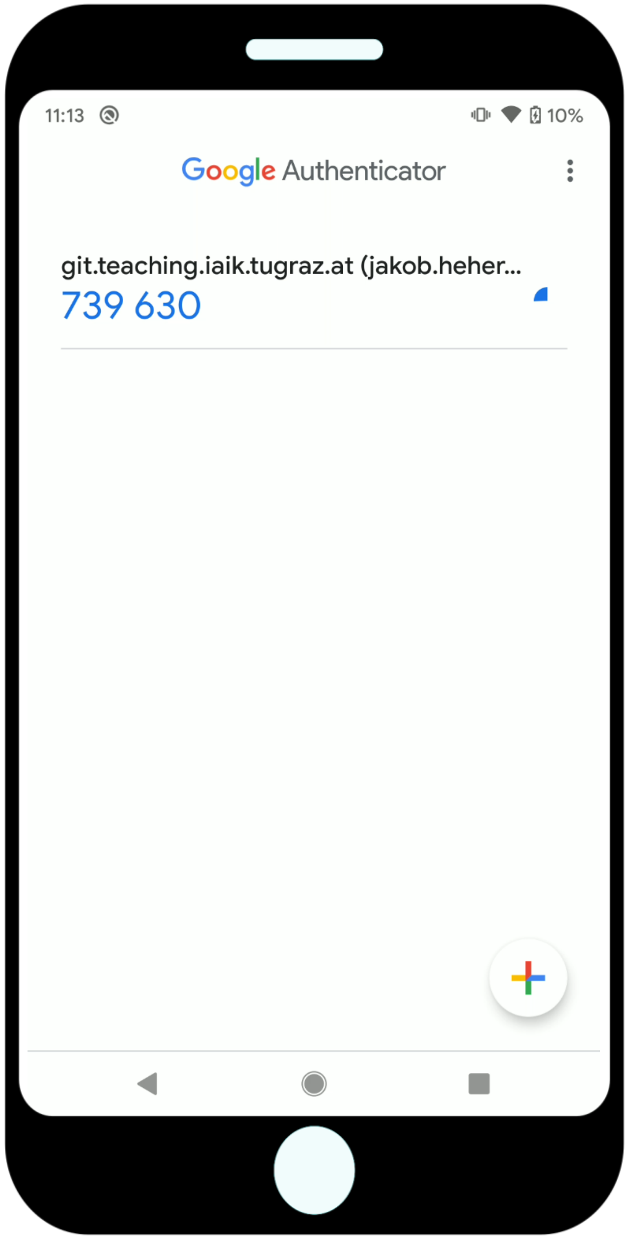

Key: 4OWR 2BWM IZFM G7WY HMIF TNQA KLHV 43N6
Device lost?
(Just backup the key!)
e3 ad 1d 06 cc 46 4a c3 7e d8
3b 10 59 b6 00 52 cf 5e 6d be
key
HMAC_SHA1
message
digest
33 33 ac ec 5e d8 27 15 ca ee
7d f0 27 41 9a 4f 80 2c b5 a6
extract 4 bytes
27 15 ca ee
treat as big-endian integer
655739630
modulo 10⁶
739630
11:13:22
Tuesday, 8 November 2022
Unix timestamp
1667902402
divide by 30s
55596746
represent as 8 bytes, big-endian
00 00 00 00 03 50 56 ca


11:13
10%
Google Authenticator
git.teaching.iaik.tugraz.at (jakob.heher...
739 630



Key: 4OWR 2BWM IZFM G7WY HMIF TNQA KLHV 43N6
base32 decode
e3 ad 1d 06 cc 46 4a c3 7e d8
3b 10 59 b6 00 52 cf 5e 6d be
key
HMAC_SHA1
message
digest
33 33 ac ec 5e d8 27 15 ca ee
7d f0 27 41 9a 4f 80 2c b5 a6
extract 4 bytes
27 15 ca ee
treat as big-endian integer
655739630
modulo 10^6
739630
11:13:22
Tuesday, 8 November 2022
Clock Synchronization
(common: try n intervals before/after)
divide by 30s
55596746
represent as 8 bytes, big-endian
00 00 00 00 03 50 56 ca
11:13
10%
Google Authenticator
git.teaching.iaik.tugraz.at (jakob.heher...
739 630


Key: 4OWR 2BWM IZFM G7WY HMIF TNQA KLHV 43N6
base32 decode
e3 ad 1d 06 cc 46 4a c3 7e d8
3b 10 59 b6 00 52 cf 5e 6d be
key
HMAC_SHA1
message
11:13:22
Tuesday, 8 November 2022
Unix timestamp
1667902402
divide by 30s
55596746
represent as 8 bytes, big-endian
00 00 00 00 03 50 56 ca
digest
33 33 ac ec 5e d8 27 15 ca ee
7d f0 27 41 9a 4f 80 2c b5 a6
extract 4 bytes
27 15 ca ee
treat as big-endian integer
655739630
modul
739630
Only 6 characters
(Lockout/delay for failed attempts)


11:13
10%
Google Authenticator
git.teaching.iaik.tugraz.at (jakob.heher...
739 630

Time-Based One-Time Password (TOTP)

- Shared secret key + current timestamp ⇒ six-digit passcode

✓ Random secret
  - Users cannot reuse passcode between websites

✓ Passcode changes every 30 seconds
  - Phished credentials quickly become stale

Time-Based One-Time Password (TOTP)

- Shared secret key + current timestamp ⇒ six-digit passcode

× Server can still impersonate user
  - Authentication is based on a symmetric, shared secret

× Secure storage is still paramount
  - … and more difficult, since you can’t hash a secret key

× Real-time phishing still works

Time-Based One-Time Password (TOTP)

- Shared secret key + current timestamp ⇒ six-digit passcode

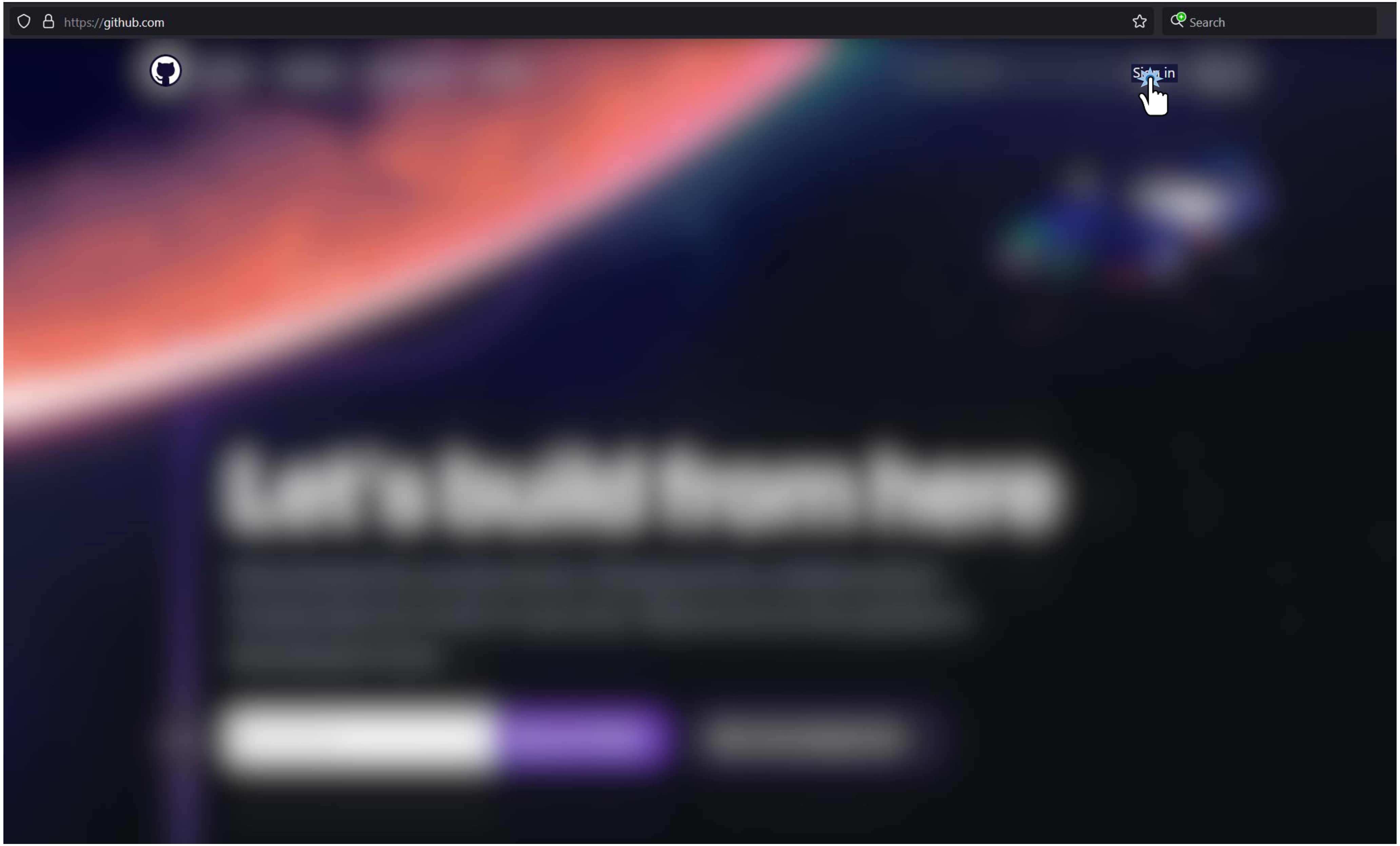
- Authentication factor categories:
  - Proving **knowledge** of some information — Something you **know**
  - Proving **possession** of some device — Something you **have**
  - Proving **inherence** of some property — Something you **are**

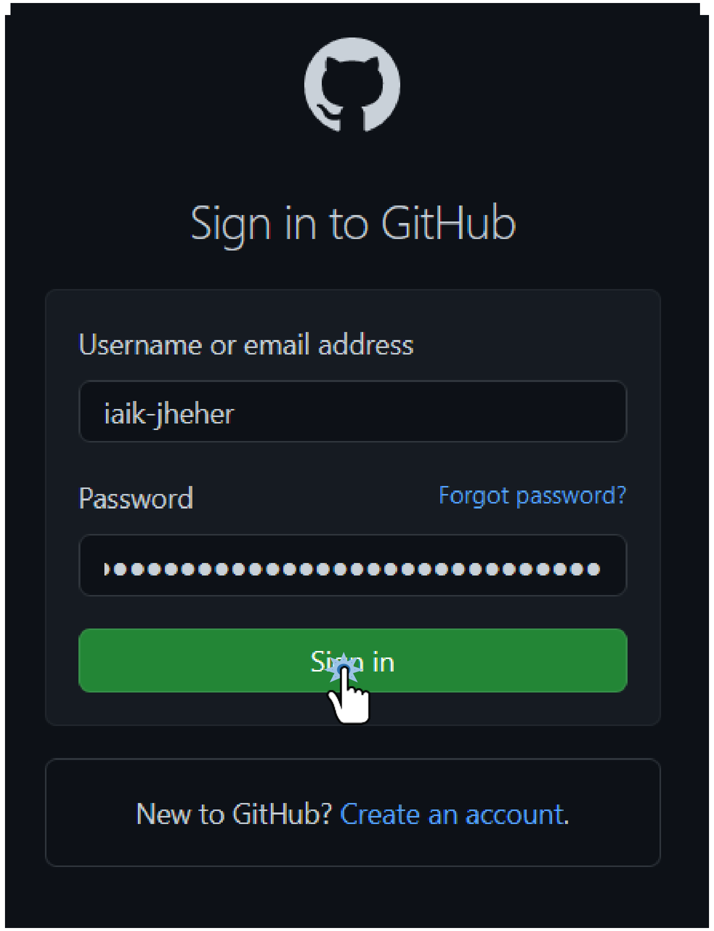
- What category does TOTP fit into?
  - **Possession** of your mobile phone?
  - **Knowledge** of the shared secret?

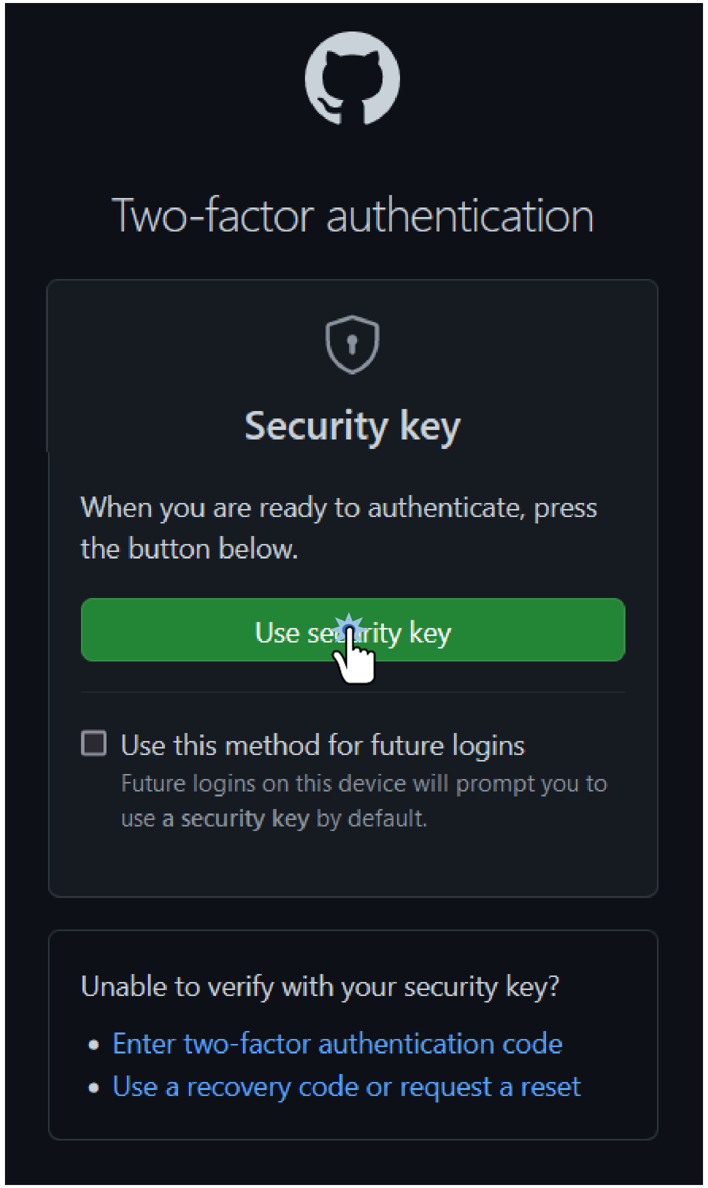
(Cryptographic)

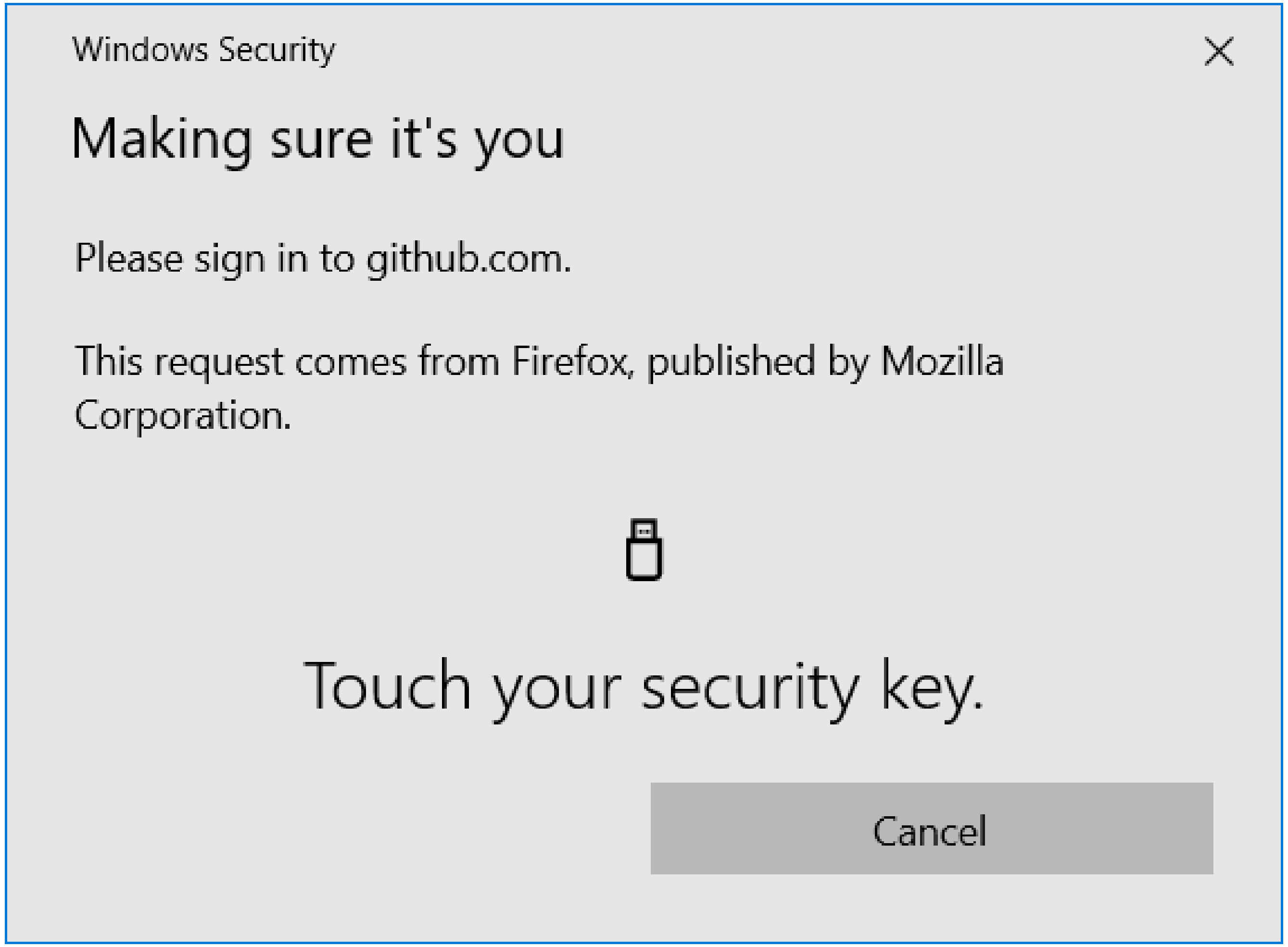
Authentication Factors

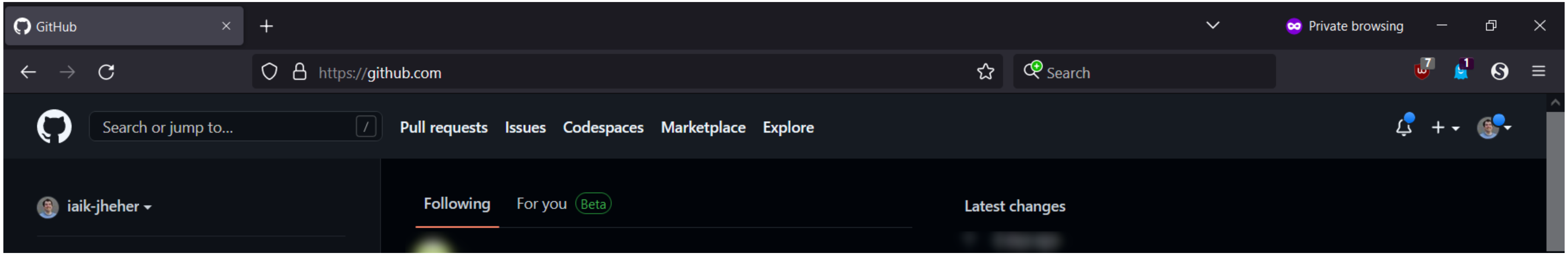
Web Authentication (WebAuthn)


https://github.com
Search

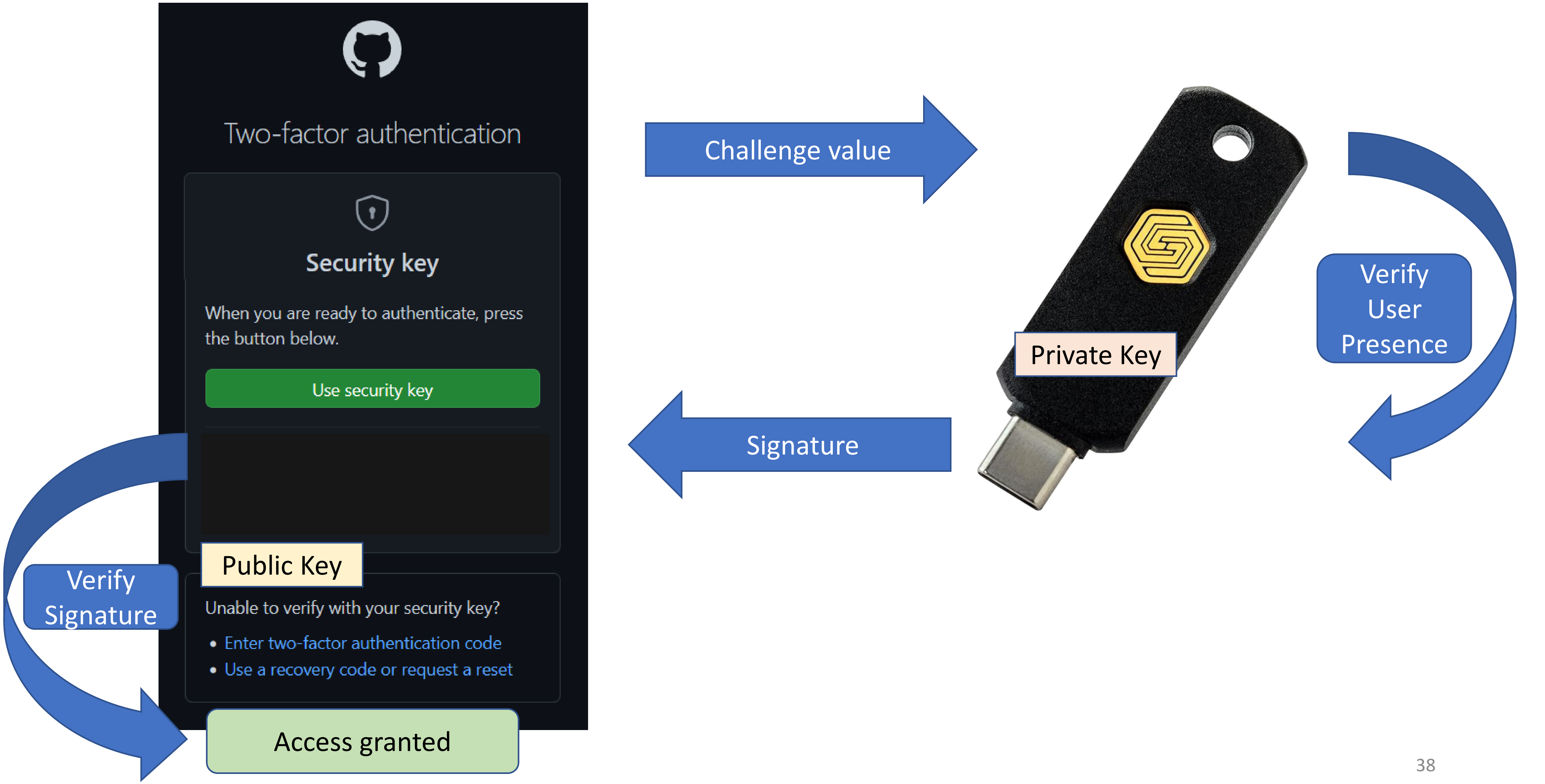

Sign in to GitHub
Username or email address
iaik-jheher
Password
Forgot password?
Sign in
New to GitHub? Create an account.


Two-factor authentication
Security key
When you are ready to authenticate, press the button below.
Use security key
Use this method for future logins
Future logins on this device will prompt you to use a security key by default.
Unable to verify with your security key?
Enter two-factor authentication code
Use a recovery code or request a reset


Windows Security
Making sure it's you
Please sign in to github.com.
This request comes from Firefox, published by Mozilla Corporation.
Touch your security key.
Cancel

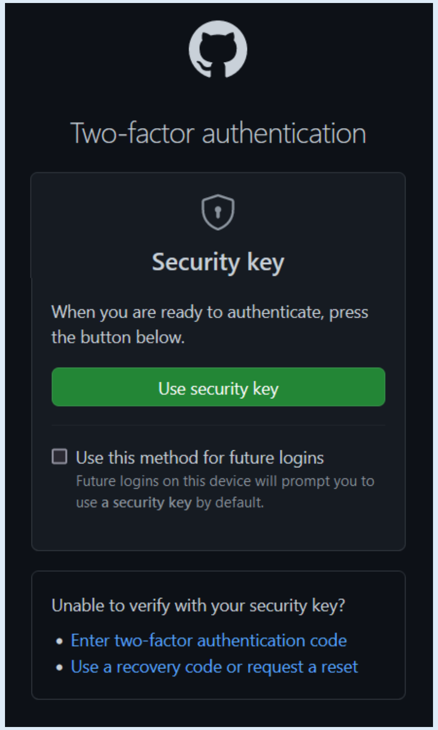


- OK, what just happened?

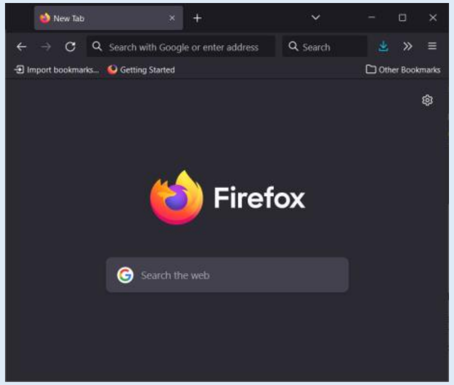


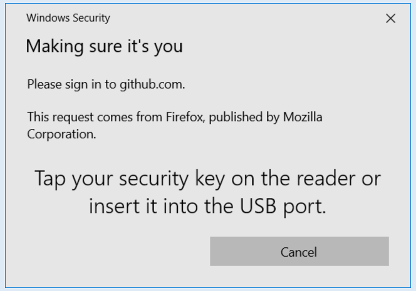
Two-factor authentication
Security key
When you are ready to authenticate, press the button below.
Use security key
Public Key
Unable to verify with your security key?
Enter two-factor authentication code
Use a recovery code or request a reset
Verify Signature
Access granted
Challenge value
Signature
Private Key
Verify User Presence

Relying Party

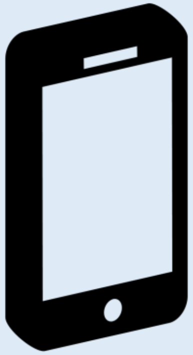


Client





Authenticator



Design Goals

- Public-Key Authentication
  - Keys stored in secure hardware ⇒ true possession factor
- Prevent MitM by Phishing Websites
  - Support HTTPS only & tie authentication to a specific *web origin*
- Prevent Replay Attacks
- Provide (Optional) Device Attestation
  - Provides guarantees about security of key storage & operation of device

WebAuthn – Outline

- RP JavaScript passes information from RP server to client
- Client adds some information & passes it to authenticator
- Authenticator signs the entire data
  - Gets user confirmation/verification first if required
- Signature gets passed back to client → JS → RP server

- Remaining question:
  - What data do we need to sign? Who do we trust to supply it?

Do you want to support WebAuthn on your website? https://webauthn.guide/

Attack Scenario – Phishing/MitM

- Relying Party server is genuine
- JavaScript is not genuine
  - Phishing server supplies fake page
- Client is trustworthy
- Authenticator is trustworthy

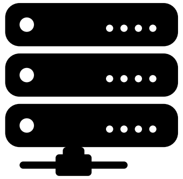
- Page looks genuine, but is *not* at a byte-for-byte identical origin
  - E.g., https://google.com -> https://gooogle.com/

WebAuthn – Registration

- Client requests credential creation for origin $\mathbf{O}$
- Authenticator generates key pair $(\mathbf{K}_{\mathbf{pub}}, \mathbf{K}_{\mathbf{priv}})$
- Authenticator picks credential ID $\mathbf{C}$ and stores $(\mathbf{O}, \mathbf{K}_{\mathbf{priv}})$ indexed by $\mathbf{C}$
- Authenticator sends $(\mathbf{C}, \mathbf{K}_{\mathbf{pub}})$ to client, which forwards it to RP

Want to implement this? See https://w3c.github.io/webauthn/#sctn-rp-operations!

WebAuthn – Authentication



- Get public key based on credential ID
- Verify challenge integrity
- Verify origin
- Verify flags
- Verify signature

Challenge:	0x1873e8ff
Origin:	
Flags:	
Credential:	0x55a473c21b

Challenge:	0x1873e8ff
Origin:	https://github.c
Flags:	
Credential:	0x55a473c21bc49…

Challenge:	0x1873e8ff
Origin:	https://github.com
Flags:	UP: yes, UV: no
Credential:	0x55a473c21bc49…

Signature:	0xafed86a40e57d864…

`UP` and **`UV`**?

- **<u>U</u>**ser **<u>P</u>**resence: a user needs to be physically present to authorize
  - usually: requiring you to push a button on the authenticator
- **<u>U</u>**ser **<u>V</u>**erification: additional user authentication is performed
  - PIN prompt or on-device biometric sensor

- This authentication is done *by the authenticator*!
  - Even a compromised client cannot bypass this requirement

Device Attestation & Certification

- Devices come with a burnt-in *attestation key pair*
  - The manufacturer signs the attestation public key
  - The attestation private key signs the created credential
- This lets us be confident in the credential's origin and storage!

- Built on top of attestation: device certification
  - Delegation of trust in individual device models
  - e.g.: FIDO2 certification levels
    - ID Austria supports WebAuthn, but only with FIDO2 Level 2 certified authenticators

Non-Discoverable Credential Storage

- Bonus: we can “store” infinite credentials

- Credential ID is a opaque byte string that is returned by the server
  - We can use it for storage!

- Authenticator only has a single master device key
  - Generated securely in the device at start-up
  - This key encrypts the private key → credential ID!

Client-Side Discoverable Credentials

- Standard authentication flow:
  - Client sends username
  - Server looks up credential ID(s) & sends them to client
- Idea: we want to get rid of this extra round trip
  - Save user identifier alongside credential on authenticator
  - Find & offer credentials using only target origin
- Problem: storage limits on authenticators!

Web Authentication (WebAuthn)

- Public key cryptography using hardware tokens

✓ No secure server storage necessary

- Public keys are not sensitive information

✓ Phishing impossible

- The browser embeds the current origin into the signed data

Web Authentication (WebAuthn)

- Public key cryptography using hardware tokens

× Users might lose hardware tokens or devices

- Your system is only as secure as the recovery factor…

What if we don't tie each credential to a single device?

(Cryptographic)

Authentication Factors

Synchronized WebAuthn Credentials

Synchronized WebAuthn Credentials

- Public key cryptography with automated key synchronization

✓ No secure server storage necessary
  - Public keys are not sensitive information

✓ Phishing impossible
  - The browser embeds the current origin into the signed data

✓ Credentials survive device failure or loss
  - Synchronized via “sync providers” (Microsoft, Apple, Google)

Synchronized WebAuthn Credentials

- Public key cryptography with automated key synchronization

× Sync providers' implementation is a *huge* point of failure
  - A vulnerability would expose *billions* of single-factor credentials

× Dependency on sync platforms leads to customer lock-in
  - Switching loses every single credential you use to log in, everywhere

× Lack of interoperability reinforces existing cross-sector monopolies
  - Want to use a phone OS, made by Google, to log in?
  - Only if you're using a specific browser that's made by Google!

Synchronized WebAuthn Credentials

- Public key cryptography with automated key synchronization

✓ Definitely more secure than “standard” password usage

? Difficult to compare with password manager usage

× Less secure than hardware token usage

  ? But more usable