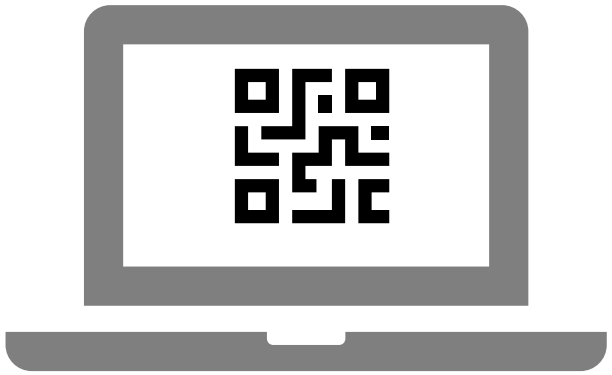
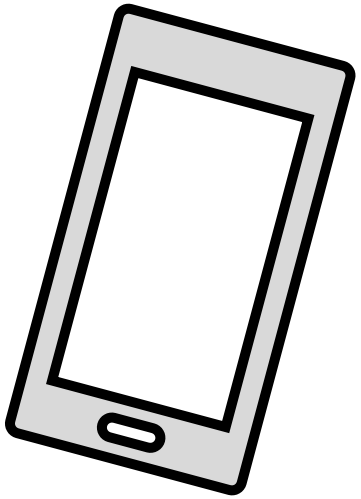
<WA/>
2025





MFA

Beyond Password Authentication

Enrico Masala
Antonio Servetti

https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html
https://auth0.com/docs/secure/multi-factor-authentication

Can you give more than one evidence?

STRONGER AUTHENTICATION

Multi-factor authentication

- Since it is so difficult to protect web applications (XSS, click “hijacking”, etc.), a multi-factor authentication (MFA) is frequently required nowadays
  - Often implemented as a Two-Factor Authentication (2FA), e.g., Gmail, …
- The user is required to present more than one type of evidence to authenticate on a system
- See OWASP:

Factor	Examples
Something you know	Password and PINs, Security questions
Something you have	OTP Tokens, U2F Tokens, Certificates, Smart Cards, Email, SMS and Phone Calls
Something you are	Fingerprints, Facial recognition, Iris scan
Somewhere you are	Source IP Address, Geolocation, Geofencing
Something you do	Behavioral Profiling, Keystroke & Mouse Dynamics, Gait Analysis

https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html

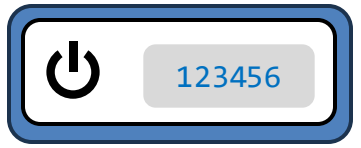
Two-factor authentication

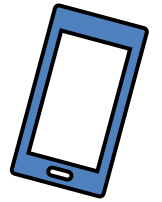
- The two factors must belong to different categories
  - E.g., not: password + PIN
- "*MFA is by far the best defense against the majority of password-related attacks*" [OWASP]
- Typically required at log in, but it may be also appropriate to require a second factor to authorize other sensitive actions such as
  - Changing passwords, contact email, disabling 2FA, granting additional privileges, …
  - Performing sensitive operations (confirm money transfers, risky operations such as permanent deletion, adding new administrators, etc …)

https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html

Example 2FA: Something You Have: Token

- One Time Password Tokens (OTP)
  - Hardware-based and software-based
  - A common secret must be established at creation time
- Token types
  - Hardware tokens
    - Almost impossible to compromise remotely, but handling them may be impractical (distribution costs, risk of losing them), may require special server hw
  - Software OTP Tokens
    - Generally used to generate Time-based One Time Password (TOTP) codes
    - Typically via an App on the user's smartphone (e.g. Google authenticator)
    - Some are standardized (e.g., RFC 6238)





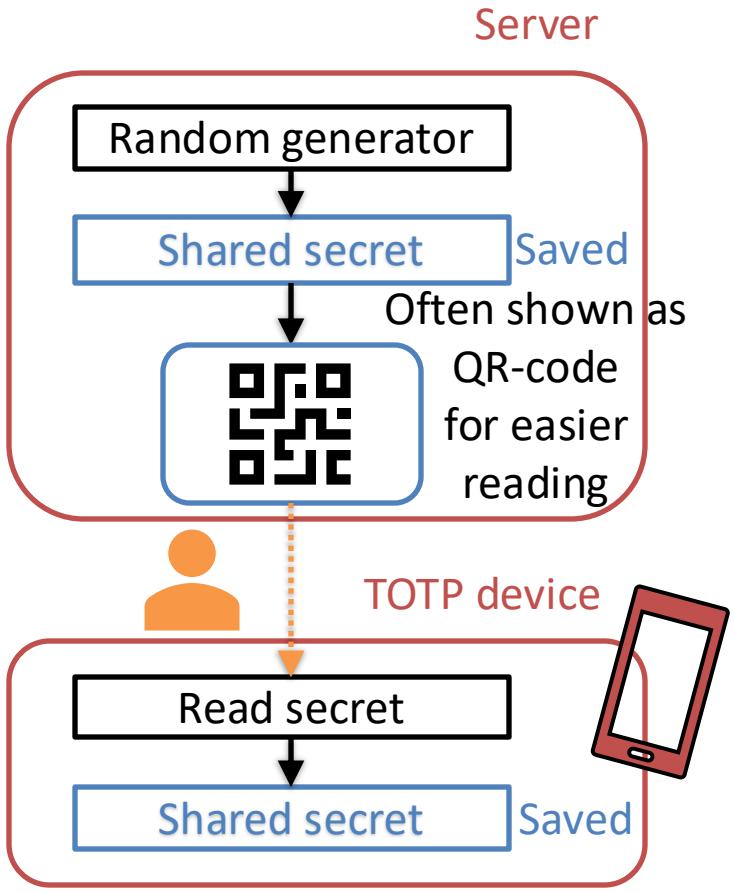
https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html

TOTP Example Usage

- The server requires a TOTP when a sensitive action (incl. login) is required
  - The code changes over time (combines secret and time with some hash function)
- Pros
  - Cheap, no logistic issues, easily changed if user loses one
- Cons
  - Smartphones can be compromised, TOTP app might be on the same device used to authenticate, require the user to have a smartphone
- How to establish a common secret
  - Usually done during the TOTP activation phase: the server chooses and sends the secret to the (web) client, e.g., as a QR-code, which is then memorized in the smartphone and never sent again by the server

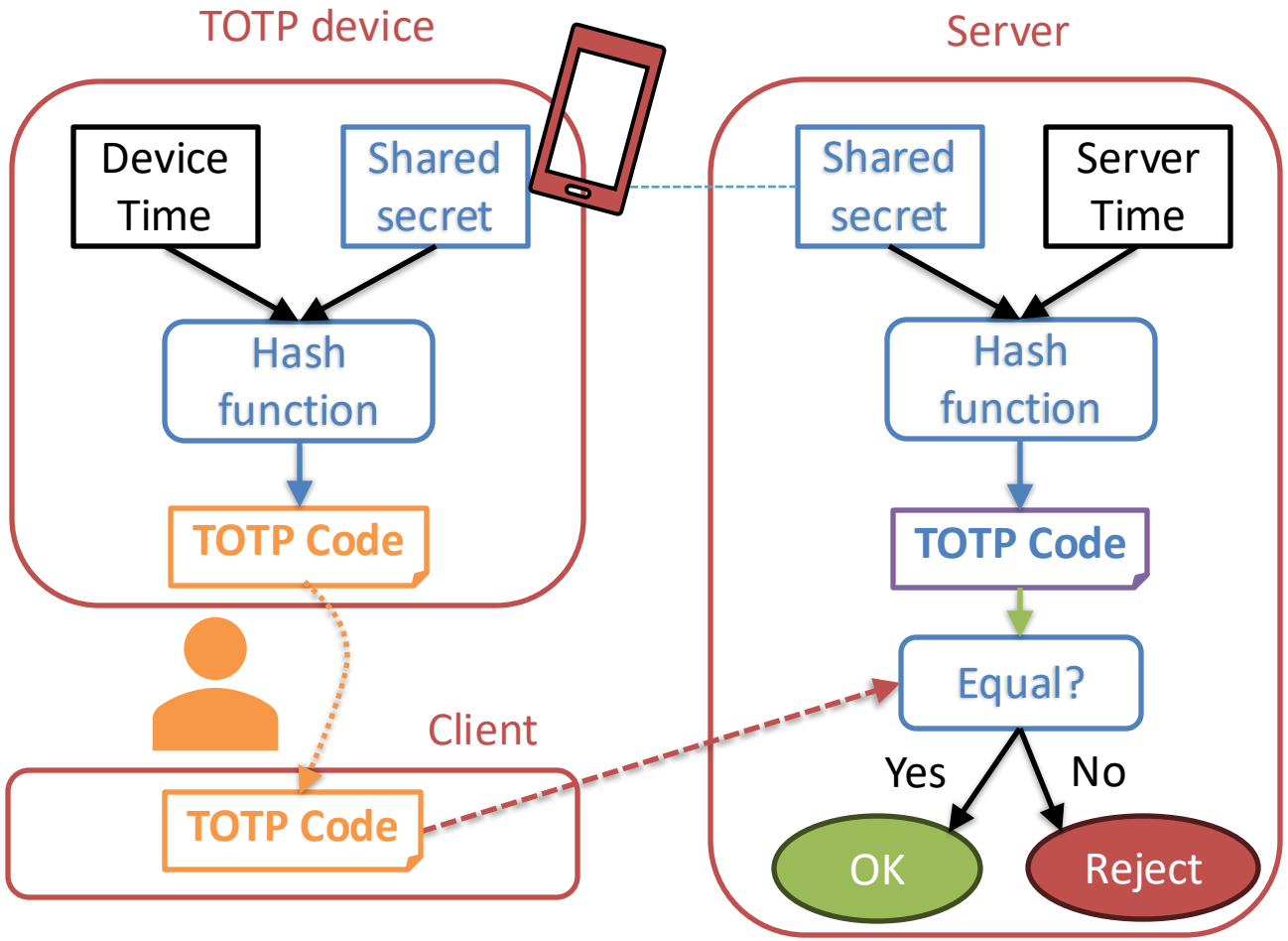
https://cheatsheetseries.owasp.org/cheatsheets/Multifactor_Authentication_Cheat_Sheet.html

TOTP



Initial setup

When OTP is requested



NB: After setup, the secret never leaves the server again, nor it is shown to the user again

https://auth0.com/blog/from-theory-to-practice-adding-two-factor-to-node-dot-js/

Using TOTP tokens in practice

TOTP IN PRACTICE

Secret Key

- Create the secret in the server: a random vector of bytes (e.g., 32 bytes)
- Convert it in a convenient format for reading and storing (nowadays almost always “base32” format)
- Store the corresponding string in the server DB, communicate it to the user
  - Typically through a QR-code: scan it through one of the many app supporting TOTP (Google authenticator, Microsoft authenticator, …), they will store the secret and generate codes when required

```
// Convenience code to create the QR-code
const QRCode = require("qrcode");
const secret = 'LXBSMDTMSP2I5XFXIYRGFVWSFI';  // base32 string to store in DB
QRCode.toDataURL(`otpauth://totp/MyApp?secret=${secret}&issuer=MyApp`, (err, url) => {
  console.log("Paste in browser URL bar:");
  console.log(url);
});
```

Secret Key

- Create the secret in the server: a random vector of bytes (e.g., 32 bytes)
- Convert it in a convenient format for reading and storing (nowadays almost always “base32” format)
- Store the corresponding string in the server DB, communicate it to the user
  - Typically through a QR-code: scan it through one of the many app supporting TOTP (Google authenticator, Microsoft authenticator, …), they will store the secret and generate codes when required

Only for this course: secret generation part not required. Just store the given base32 string. QR-code already given in exam text, same for everybody, just for simplicity and testing speed at exam

Storing Secrets in the Server

- TOTP secret can NOT be hashed, as typically done with passwords
- Secret is to be stored/managed as any other secret information
- See, e.g., OWASP guidelines:
  - https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

- Only for this course, only for simplicity
  - an additional column in the user table is ok, as done with password hash and salt

Client Flow for TOTP after Authentication

1. After authentication, if required, the user is asked to fill out a form in the client with the TOTP code (that will then be validated in the server)

- Just for convenience, store the fact that the user successfully performed the 2FA action in the application memory
  - For instance, in a React State (as done with the session info)

TOTP Form: Use Standard Practice

- A React component with local state (“controlled” form components)

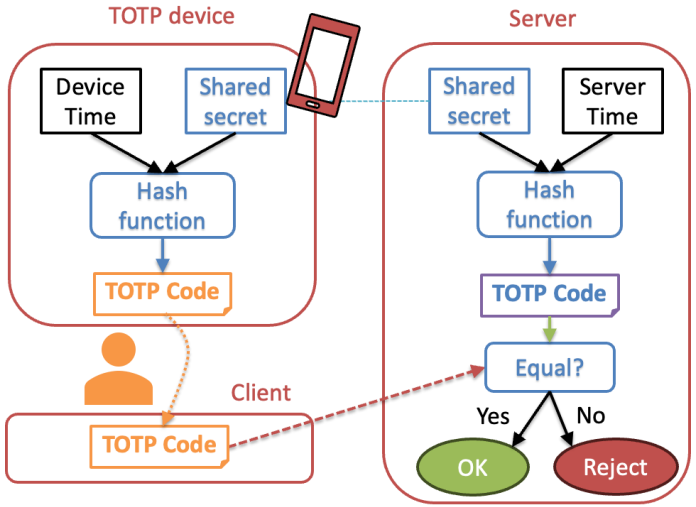
```
const [loggedInTotp, setLoggedInTotp]= useState(false);
<TotpForm setLoggedInTotp={setLoggedInTotp}/>
```

```
function TotpForm(props) => {
    const [totpCode, setTotpCode] = useState('');

    doTotpVerify = (event) => {
        event.preventDefault();
        if (… form valid …) {          // Make POST request to authentication server
            API.totpVerify(totpCode).then(    // API to send object {code: totpCode} to server API /api/login-totp
              () => props.setLoggedInTotp(true) )
            .catch( () => setErrorMessage('Wrong code') );
        } else {
            // show invalid form fields, e.g. empty field
        }
    }
...
```

Server Flow for TOTP after Authentication

2. The code is validated on the server and, if ok, such condition is remembered in the session information for later checking when needed
3. If not, it sends back a generic response saying that the code is not correct
   - Note: in the “TOTP after user/pass login” scenario, it is up to the application logic to decide whether the user needs TOTP and / or the user can proceed without TOTP with less functionalities



Example: 2FA after password authentication

- Create a server endpoint that performs the 2FA and store the result in the session (e.g., `/api/login-totp`)
  - Works only if a session is already established (i.e., after checking user/pass)
- Note: Later, **ALWAYS** check, **on the server**, for each server endpoint that requires 2FA, **every time**, if the user performed 2FA
  - Look for such info into the session on the server side

TOTP in express.js

- Convenient libraries:
  - `passport-totp` https://www.passportjs.org/packages/passport-totp/
  - `thirty-two` https://www.npmjs.com/package/thirty-two

- `npm install passport-totp` (*middleware*)
- `npm install thirty-two` (*base32 utilities to encode/decode secrets*)

TotpStrategy

- The strategy determines how the TOTP value is checked
- The “done” callback must be supplied with the secret stored in the server, in the form of a byte array (base32.decode() convert a string to a byte array)
  - `user` is the currently validated user (in the session)

```
const base32 = require('thirty-two');
const TotpStrategy = require('passport-totp').Strategy; // totp

passport.use(new TotpStrategy(
  function (user, done) {
    // In case .secret does not exist, decode() will return an empty buffer
    return done(null, base32.decode(user.secret), 30);  // 30 = period of key validity
  })
);
```

Caveat

- The user info (including the TOTP secret) is retrieved on the server during authentication and later when session ID is received
- REMOVE THE SECRET from the user object when sending it back to the client
  - Instead, tell the client if user can do TOTP (`canDoTotp`) and if already done (`isTotp`)

```
function clientUserInfo(req) {   // DO NOT return user to the client, it contains the secret!
  const user=req.user;  // contains user.secret, to be removed from return value
  return {id: user.id, username: user.username, name: user.name,
          canDoTotp: user.secret? true: false,  isTotp: req.session.method === 'totp'};
}
...
app.post('/api/sessions', function(req, res, next) {
  passport.authenticate('local', (err, user, info) => {
    if ...
    return res.json(clientUserInfo(req));
...
app.get('/api/sessions/current', (req, res) => {  if(req.isAuthenticated()) {
    res.status(200).json(clientUserInfo(req)); }
```

TOTP Check

- After setting everything up, check the received TOTP
  - add an (authenticated) Express route able to receive the TOTP value
  - pass the `authenticate(<strategy>)` method as the first additional callback
    - `authenticate('totp')` will look for `code` in `req.body` then will check if it is valid or not

```
app.post('/api/login-totp', isLoggedIn,
  passport.authenticate('totp'),   // expect the totp value to be in: body.code
  function(req, res) {
    req.session.method = 'totp';   // store successful result of totp in session
                                   // standard authentication set method as 'plain'
    res.json({otp: 'authorized'}); // returns HTTP status code 200 OK
  }
);
```

Note: the validity check will try codes in a window of N codes in past and future to address clock desynchronization.
Also, for simplicity, this implementation can accept the same TOTP more than once if still in window (not really one-time…)

Storing User Information in React

- The TOTP response will communicate the result to the browser
  - e.g., using the HTTP status code (200 OK or 401 Unauthorized)
- Typically, this status is stored for later usage in the client app
  - Similarly to user info after authentication (e.g., `isLoggedIn` state in JSX)
  - To avoid sending requests that will not succeed and to disable unavailable options (NB: just for users' convenience, it is NOT a security mechanism!!!)
  - In any case the authentication status, including TOTP auth result, can be asked by the app whenever needed (e.g., with `API.getUserInfo()` in a `useEffect` at component mount time)

Protecting Server APIs with TOTP

- Some routes in the server needs to be **protected**
  - i.e., they must provide a response ***only*** for users that can do the operation, ***in addition*** to the check of valid session (i.e., the isLoggedIn middleware)
- Check if a request comes **over a session** (“isLoggedIn”) where TOTP has already been performed successfully
  - **`req.session.method === 'totp' ?`**
  - returns true if previously set upon successful TOTP check
- To be done at the beginning of **every** callback body in each API that needs protection
  - Same code to be inserted in different APIs → Create a custom middleware!

Protecting APIs with a Custom Middleware

- *Create* an Express middleware that includes the check
- Use it at the route level
  - The middleware is especially useful to uniformly handle unauthorized cases

```
function isTotp(req, res, next) {
  if(req.session.method === 'totp')
    return next();
  return res.status(401).json({ error: 'Missing TOTP authentication'});
}
...
app.delete('/api/comments', isLoggedIn, isTotp, ..., (req, res) => {
  ...
});
```

Recommendations

- Authentication/authorization is a complex problem!

- Never invent your own mechanism!
- Use standardized, well tested, mechanisms!

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/