# Authentication & Session Management

JOHANNES KEPLER
UNIVERSITÄT LINZ

**Michael Sonntag**
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

# Session management/ Session hijacking/Access control

- Stealing accounts from other persons
  - Account-ID, username, password, authentication token, session-cookie, session-ID…
- Building authentication and session management is hard
  - But most web applications do it on their own (again)
  - Flaws are therefore quite common!
- Biggest problem: the attacker is then not restricted any more
  - He can do what he should be able to do (“impersonation”), as he poses as a legitimate and authorized user!
- Typically high-level accounts are targeted
  - If not, “privilege escalation” is attempted

# Examples

- When logging out, the session is not correctly invalidated
  - Or timeouts are far too long (e.g. 1 hour)
    - User doesn’t log out from a public computer → closes browser
    - <1 hour later another person opens the browser → still logged in!
- Passwords of users are not or only weakly hashed
  - They are still very often stored in the database in cleartext
- “Forgot my password” → send it to the E-Mail address in plain text (or send a link to reset it…)
  - Anyone can initiate this
  - E-Mails may (commonly not!) be easy to read for third parties
    - Mail, as well as access to server, is often unencrypted!
- E.g. a large ISP in Upper Austria → if you forgot your E-Mail password, they will send it to you by E-Mail (to another address after verifying who you are) in cleartext
  - ISP: “The customer service cannot see them, they see only *”

# Examples

- Public session ID
  - http://example.com/page;jsessionid=**2P0OC2JDPXM0OQNDLPSKHCJUN2JV**?param=
  - Send this link to someone else → they “own” your session!
- Predictable numbers in session-IDs or cookies
  - Login and retrieve your session ID
  - Wait a short time
  - Try the following session IDs
  - Or: Session-ID = User-ID/Serial number/Database-Row-ID…
- Login check is commented out
  - Probably done for testing, but made it into the release version
- Default passwords identical on all devices
  - Home routers from ISPs → Everyone has same admin password

# Detection

- Manual testing:
  - When are session IDs assigned and when are they changed?
    - Should be renewed on: login, reauthentication, logout
  - How long is their timeout? Is it enforced (=verified) by the server?
  - What happens on wrong/missing IDs?
  - Cookies should set path as specific as possible (but see __Host-!)
    - Domain also, or even better no domain to restrict to this single host
- Automatic testing:
  - Searching for IDs in URLs, error messages, logs
  - Lockout after too many attempts
  - Check for generated session IDs
    - Include a “server secret” → attackers cannot generate valid IDs
- Ensure that authentication is in a single library/module/…
  - **One** implementation of checking **only**
  - and make sure, that this is actually called!
- Take care to avoid XSS → often used to steal session IDs!

# Session tokens

- Session tokens are used to recognized users
  - HTTP is stateless!
  - If we get this token, we can pass off as another person!
- Basic classes of attacks on session tokens
  - Prediction: we get a specific token now and can deduce, what token the next person(s) will receive
    - So we wait a bit, and then use this number!
  - Capture/replay: we get access to the token in use by someone, e.g. through XSS
    - See above!
  - Fixation: we obtain a token and then make sure that the victim will use exactly this token for logging in
    - We need to get the victim to actually use it → therefor “renew on login”

# General measures

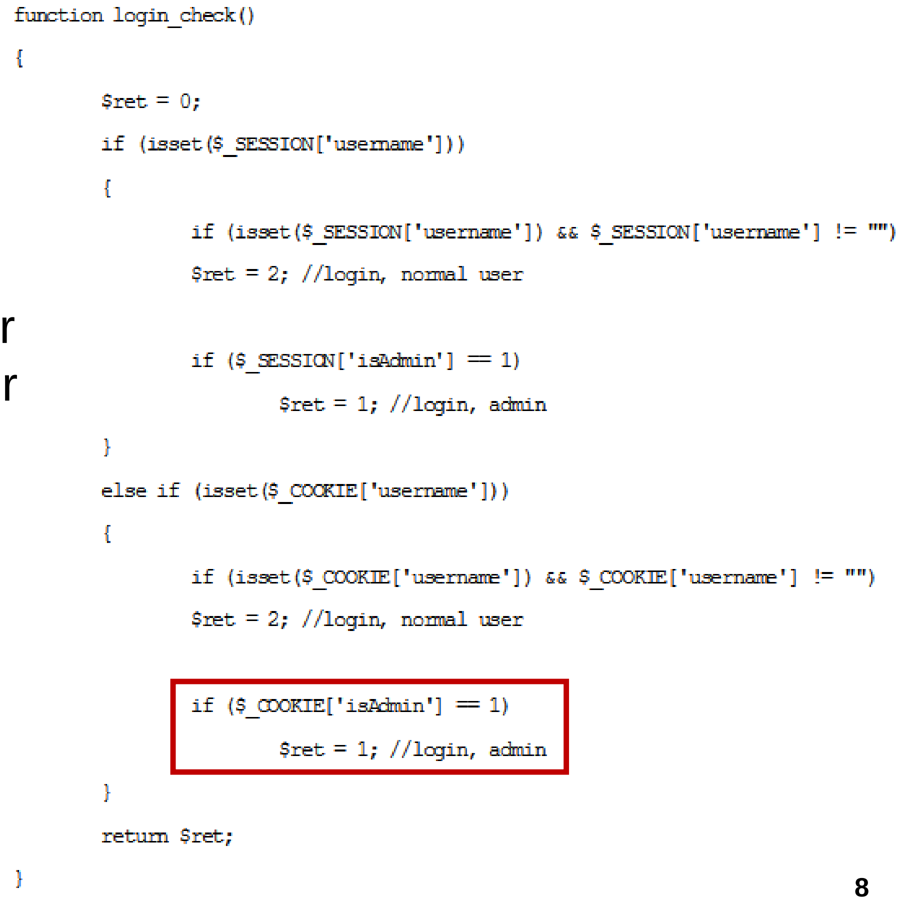
- Session tokens should be really random and long numbers
  - Good random number generator
- Should be cryptographically secure
  - I.e. include a secret, e.g. HASH(<predictable number> | <secret>)
    - Allows detection of “fake” cookies
- Could be tied to an IP address
  - Replay/fixation becomes much more difficult
  - Potential problem for very mobile users (e.g. trains): the device often receives a new IP address in one session
- Take care where to write them to
  - URL is a bad idea!
- Cookies: use security flags (Secure, HTTPOnly, SameSite)
- Enforce session timeout on server
  - Destroy stored session after timeout/check when retrieving it

# Authentication based on user input

- Authentication decisions must be based on the server, and may never be determined by the client
- Example: Western Digital MyCloud Login Bypass

This is the server code:
It relies on data sent from the client in the cookie to determine whether a user is:

- an admin (=unrestricted access), or
- normal user (=restricted access), or
- not yet logged in (=no access)



```
function login_check()
{
        $ret = 0;
        if (isset($_SESSION['username']))
        {
                if (isset($_SESSION['username']) && $_SESSION['username'] != "")
                $ret = 2; //login, normal user

                if ($_SESSION['isAdmin'] == 1)
                        $ret = 1; //login, admin
        }
        else if (isset($_COOKIE['username']))
        {
                if (isset($_COOKIE['username']) && $_COOKIE['username'] != "")
                $ret = 2; //login, normal user

                if ($_COOKIE['isAdmin'] == 1)
                        $ret = 1; //login, admin
        }
        return $ret;
}
```

https://blog.exploitee.rs/2017/hacking_wd_mycloud/

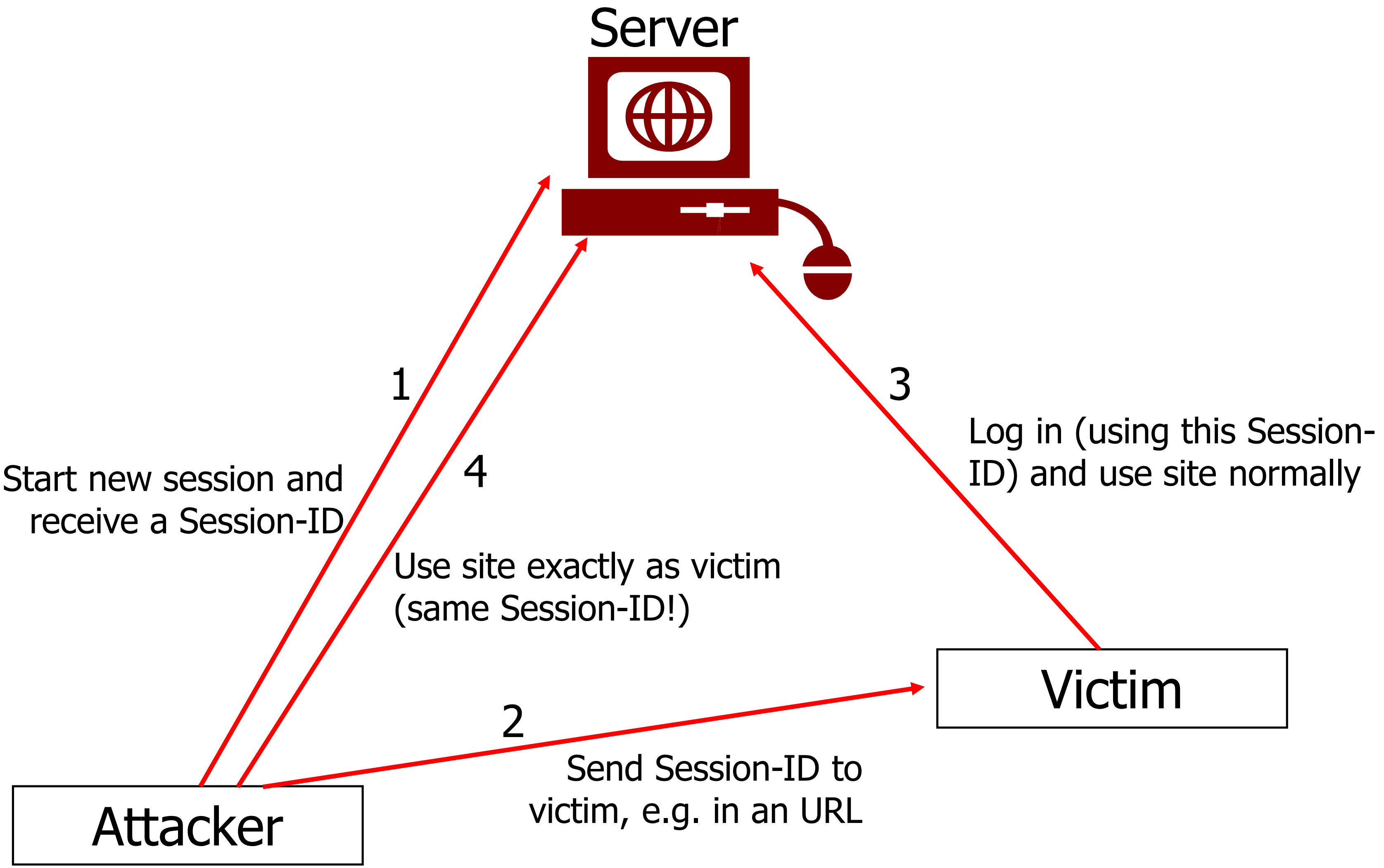
# Stealing HTTPS cookies

- Transmission is encrypted, so even MITM is not successful (directly)
- Exploitation:
  - User logs in to https://vulnerab.le/ and gets a cookie
  - User navigates to http://somewhere-el.se/
    - Note: **Un**encrypted communication!
  - MITM replies with a redirect to "vulnerab.le"
    - Original request gets lost
  - User's browser will request http://vulnerab.le, including the cookie
    - Note: redirection (→ request) is to unencrypted URL!
    - Server might automatically redirect to https; but this is too late…
  - MITM can now sniff the cookie
  - User notices that he didn't get where he wants; perhaps he tries again; now it works (not redirected again)
- See browser security features: HSTS prevents this
  - On second and further accesses – but note that here only the second request (=secure!) is (then unsuccessfully) attacked!

# Stealing HTTPS cookies

- Non-Prevention:
  - Do not accept requests via HTTP or redirect them to HTTPS
    - Reason: the attacker is not interested in the reply, all he needs is that the request is sent…
- Prevention:
  - Make sure Cookies are not sent over unencrypted connections
    - “Secure” flag set for the cookie
  - Ensure site is “locked” to HTTPS
    - Browsers might try https first/rewrite
      - Extensions or built into the browser
    - See HSTS (HTTP Strict Transport Security)

# Session fixation



Server

Victim

Attacker

1

Start new session and receive a Session-ID

2

Send Session-ID to victim, e.g. in an URL

3

Log in (using this Session-ID) and use site normally

4

Use site exactly as victim (same Session-ID!)

# Session fixation

- You get the victim to use a specific session ID
  - As you know this ID, you can access the web application exactly as the user could do
- Example:
  - Go to the desired website and start a session
    - You receive a new session ID
  - Send the ID to the victim, e.g. in a URL (URL shortener…)
  - Victim clicks on the URL and “receives” the same session ID
  - Victim logs in
  - Attacker uses the session ID to “be” logged in simultaneously
- What to do:
  - Invalidate a session before checking username + password
  - If success → authenticate and assign a **new** session ID
  - If error → send to login page (and assign **new** session ID)
- Works the same with cookies (set new ID as cookie content)!

# Sessions

- Session handling is complicated
  - Whenever the privilege level of the user **changes**, the session ID **must** be **regenerated**
  - Ideally: start a new session (see CAPTCHA re-riding!)
    - Not always possible → make sure to delete problematic content
  - Only accept the session ID in the way you sent it
    - Cookie → reject (ignore/create alert) session ID in GET/POST
  - Take care of expiration
  - Force logout by JavaScript (e.g. after timeout has expired without page load or when closing the window)
  - Use long IDs (>=128 Bit) with lots of entropy (good PRNG → You can assume half the length, so here 64 Bit “real” entropy)
  - Use meaningless session IDs, e.g. hash or random number
  - Encrypt session ID transmissions (→ TLS)
  - Restrict them: Secure (=different sessions for HTTP and HTTPS), Domain, Path, Expiry

# CAPTCHA Re-Riding

- Access control through CAPTCHA, e.g. for creating accounts
  - □ One manual solution can be reused for several requests
- Basic premises (often existing!):
  - □ Captcha is generated and solution **stored in session** (on server!)
  - □ Solution is **not removed** from session during its verification
  - □ a) Registration successful → New session-id assigned
  - □ b) Registration successful → Session-id stays the same
  - □ c) New session (=without solution) generated → No attack!
- Exploiting b):
  - □ Solve CAPTCHA manually and submit it
  - □ Monitor this “solution” request with a proxy
    - ● Contains valid session-id, form fields, and manual CAPTCHA solution
  - □ Submit request several times, replacing the user name/id/… (and all other unique values) with new values

Kalra, G., http://gursevkalra.blogspot.co.at/2012/03/captcha-re-riding-attack.html

# CAPTCHA Re-Riding

- Exploiting a):
  - Solve CAPTCHA manually and submit it
  - Monitor this “solution” request with a proxy
    - Contains valid session-id, form fields, and CAPTCHA solution
  - Submit a new request and receive a new session-id
    - Session **content** on server stays the same, only id changes
  - Replace session-id in recorded request and send again
- Preventing such problems:
  - One-time tokens may not solve the problem
    - See replacing the session-id above!
  - Remove CAPTCHA solution from session after verification
  - Create a completely new session after login - and on every single try of solving the Captcha
    - This is example c) from above – No such security problem there!

# Prevention

- Check that all credentials and session IDs are
  - stored only in encrypted/hashed form
  - secure against guessing
  - protected against overwriting
    - Creating a new account with specifying an existing user id/number
    - Change password, password recovery…
  - never placed in an URL
  - deleted on logout and expire soon
  - sent only over encrypted connections
  - renewed after a (un)successful login (try)
    - First visit → anonymous user → session ID1  
      Login → authenticated user → session ID2
  - can never be specified by users
    - “Session fixation”, e.g. getting a user to click on http://www.site.org/login.asp?session=08ag15 and logging in with this Session-ID

# Session state on client?

- The session state should always be only on the server side
  - But what about e.g. load balancing?
- Can we store the session state on the client – **securely**?
- Yes we can!
  - Everything might be stored on the server (too) → optional!
  - We send the session state to the client
    - Potentially encrypted and compressed
  - The client sends it back with the request to use
    - Hidden form fields: links → Javascript!
  - We recreate the session from the request and answer it
    - If recreating the session failed, e.g. because of tampering on the client, we retrieve the version stored on the server (e.g. in a database)
      - Or start a new session (if not stored)
    - Advantage: stored on server is only a backup and might take long to access, but usually we take the fast one from the client

# Secure session state on the client

- Ensuring the integrity of session information sent from the client
  - Digital signature: create session data, sign it, send to client; verify signature when received
    - Works, but requires a lot of computing power and might be slow
  - Hash value: create hash from session data, store it, send to client; receive session state, calculate hash, compare to locally stored
    - We need to store (and distribute – load balancing) only the hash value, but falsifying might be relatively easily possible (depends on alg.)
  - Cryptographic hash value: create session data, concatenate secret value, create hash from result, send data+hash to client; receive data+hash; concatenate secret value to data received, create hash, check for identity with received hash
    - We don’t even have to store the hash value on the server
    - Falsification is extremely difficult (secret would have to be recreated from the hash!)
    - Load balancing: systems must only share (static!) secret – nothing else!

# Server-Side-Request-Forgery

- Basic idea: requests from myself (="localhost") are always OK; no need to login; admin permissions etc
  - "Nobody can get to the console, anyway!", "The admin port is blocked by the firewall and not externally reachable"
    - When moving to the cloud, such things can change quickly…
- Problem: web applications connect to itself
  - Not a problem as such, but when the client can influence these URLs then there is an option for attacks
  - Note: works exactly in the same way for "backend" systems
- Example:
  - Pass URL as parameter in a form to the client
    - Can be leading to a back-end system, third-party server etc
  - Receive it back from the client
    - Modified by the attacker to point instead to "localhost" or "127.0.0.1"
  - Connect to the URL and retrieve data
    - Request comes from localhost → trusted/doesn't need authentication…

# Server-Side-Request-Forgery

- How to get this wrong:
  - The server has access to other resources
  - The server performs such access only itself and does not allow clients to do this directly
  - The access is performed with the rights of the server, not of the user initiating it
  - Attackers can initiate/modify the request
- Allows circumventing the WAF and/or firewall → internal request
- Very often used in combination with cloud services
  - Backend systems cannot be reached from the outside, but must be reachable from the public-facing servers
  - Public-facing servers must have access to them, i.e. access tokens
- Other targets: internal REST interfaces or local files (“file://” URLs)
- Related to unvalidated redirect&forwards and other attacks

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

# THANK YOU FOR YOUR ATTENTION!

**Michael Sonntag**
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)