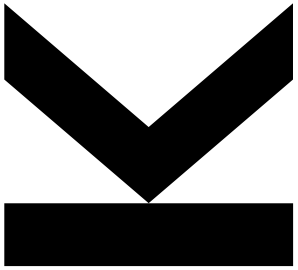
CSRF



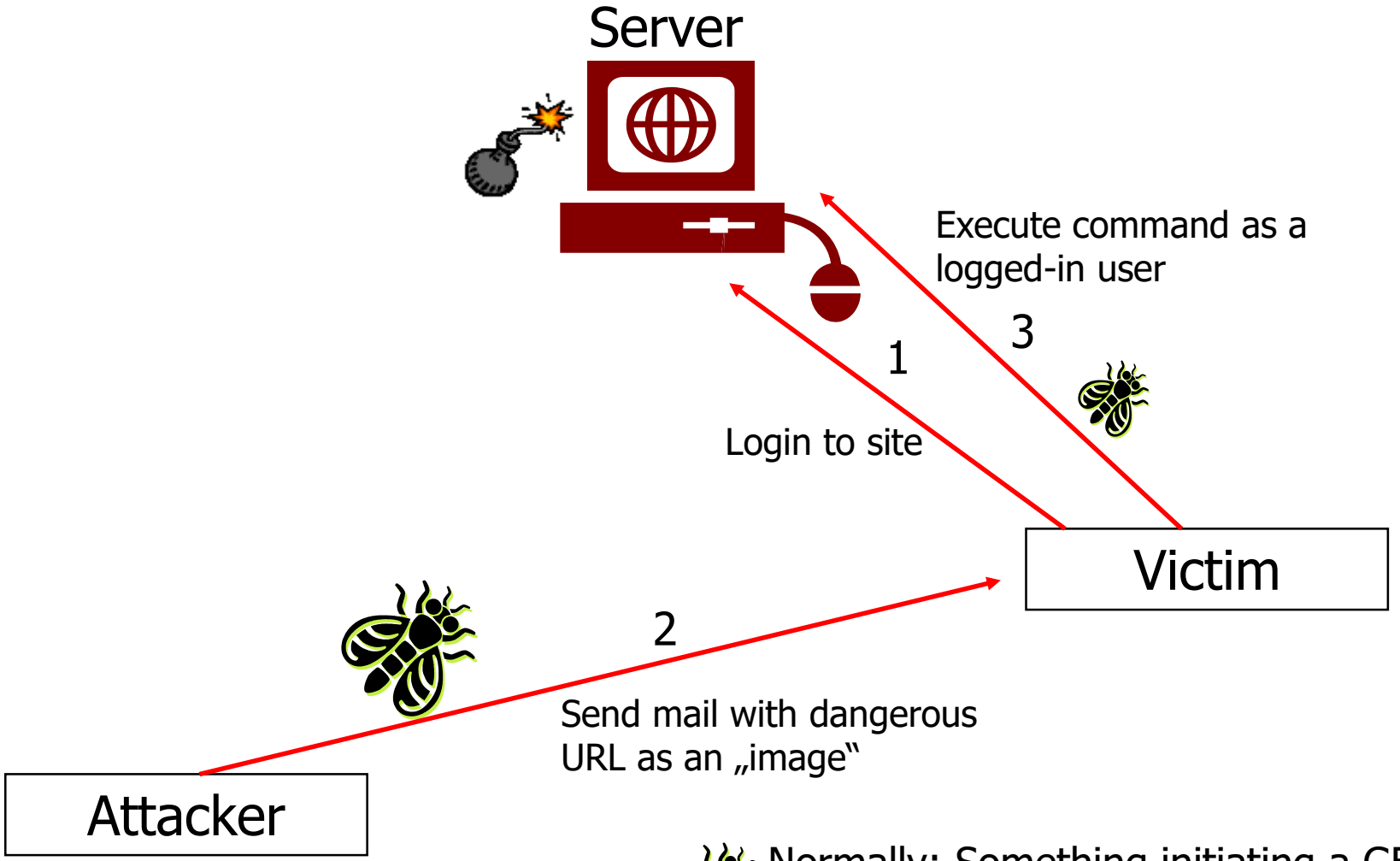
Cross-Site Request Forgery

JOHANNES KEPLER
UNIVERSITÄT LINZ

Michael Sonntag
Institute of Networks and Security

INSTITUTE
OF NETWORKS
AND SECURITY

Cross-Site Request Forgery: CSRF/XSRF



Normally: Something initiating a GET request or some JavaScript code (Webpage, link...)

Cross-Site Request Forgery

- An innocent third person is instrumented to carry out a specific attack against a web server
  - Typically this third person is **entitled** to perform some action on the web server, and is “made” to perform one he/she **does not want to do** (and **without knowing** about it)
- This is possible in two ways
  - “Social engineering”: Threats, bribery, blackmailing…
  - “Technologically”: Sending a link which seems to lead to a movie, but when clicking on it actually deletes all the records in the company’s database
- Biggest problem here: Users are performing actions which they are **entitled** to do and **must be able** to do!
  - Still, some precautions exist: At least for the second way!
  - Aim: Users should only ever perform an action if they know that they are performing one, and which one

CSRF: How does it work?

- The third party is lured to a webpage (or sent an E-Mail), on which he/she will click on a link, or where the page employs JavaScript
- The script/link inherits the third parties identity and privilege, and then executes an request
  - E.g. cookie, cached logon credentials, IP address, client-side SSL authentication…
  - The site cannot distinguish this from a real request: All the necessary credentials and permissions are ok; it originates from the correct IP; the session is “fresh”;…!
- Different forms:
  - Most dangerous: Attack stored on attacked website itself
    - Users will be logged in, most users will go there willingly
  - Less dangerous: On a random website
    - Get users to view website and perhaps initiate some action
  - Least dangerous: In an E-Mail
    - You must get the user to click on a link (→ social engineering!)

CSRF: Trivial example

- The third party is logged into the web application
- This application requires a login and stores a cookie on the client's computer, which is then used for maintaining the session state
- One legitimate action there is filling in a form (resulting in a GET request) to delete a record
  - GET /deleteRecord?id=15
- The attacker sends an E-Mail with the following link (HTML):
  - Click here for the free iPhone app!
- If the third party is logged into the application and clicks on the link, the session management cookie is sent automatically by the browser and the record is deleted
  - If the third party is not logged in, nothing problematic happens (login page shown/error message/…)

What will not necessarily help you (1)

- Using secret and very secure cookies
  - The cookie is sent, because it **should** be sent there!
  - Applies also to all other credentials, which might be cached
    - E.g. session identifiers: The request comes from the correct user - the problem is the “voluntariness”, not the “origin”!
- Accepting only POST requests
  - Attackers can use scripts (still more difficult → SOP)
  - Attackers insert hidden values in voluntarily submitted forms
    - Third person thinks that the form will do something completely different; the “additional” parameters (=input fields used to deceive the third person) submitted by the user are ignored by the application
- Multi-step transactions: Requiring several clicks/forms/…
  - As long as the sequence is known or predictable this won’t help, it just renders the attack more complex and longer
    - Series of hidden iframes submitted by JavaScript

What will not necessarily help you (2)

- CORS: You cannot read the result sent by the server – but the server might still execute the request…
  - But: If the request comes from somewhere else, i.e. has not been embedded into the site itself, browsers will check for CORS first
    - Simple requests are sent immediately
      - But this must then be a GET request, which **is forbidden** to change anything on the server: Is this really **never** used for that in practice?
    - Preflight-requests send a preflight first: This would be successful regarding the content
    - In **both** cases you should check the Origin header (if present!)
      → If it is from a different (non-acceptable) source, deny preflight-request and **do nothing** for **both variants**!
  - Note: In some cases browsers do not send the Origin header
- Samesite cookie attribute: Set it to “strict”
  - Not sent in any cross-browsing context
  - Depending on application scenario, “lax” might be necessary

What will not necessarily help you (3)

- Checking the referer header:
  - Accept only input from your own site
  - But: Stored on that page/What to do with empty referers?
    - These occur quite often (privacy!): None is sent over HTTP**S**
  - Similar to the Origin header of CORS!
  - E.g. Adobe Flash once allowed setting the referer arbitrarily
- URL rewriting: Putting the session ID into the URL
  - Session ID's cannot be guessed by the attacker
    - Really? Many other vulnerabilities (e.g. XSS) allow this!
  - Also, this opens up numerous other problems:
    - Bookmarks won't work any more
    - The (secret!) session ID is shown publicly

Attention: These things **do** help, also against CSRF, but they cannot **guarantee** security against CSRF!

CSRF: Typical attack vectors

- Images instead of links: Will be requested automatically
  - Note: Answer doesn't need to be an image!
  - GET requests to servers **other** than SOP…
- URL shorteners: Hiding the actual target
  - Makes it easier to get people to click on it
  - Some services (try to) check for such attacks
- URL spoofing: http://www.app.com@192.168.1.1
  - Link leads to site 192.168.1.1, not www.app.com (="user name")!
- Put the links in hidden frames: Result pages do not appear
- JavaScript: Can construct URLs arbitrarily
  - Note: Browser security precautions might require some kind of user intervention, e.g. getting the user to click on a button
- XSS+CSRF: Many successful attacks used XSS to obtain the token needed to work around CSRF protection
  - Also bypasses any referer checks simultaneously!

CSRF: Prevention by Nonce

- For each page a new form field value (“nonce”) is generated
  - Only if this value is present and correct, the request originated from the „correct“ page and should be honoured
    - Note: Will not protect against attacks stored on your site!
  - This token must be
    - Really random: Else attackers can predict the value and add it
      - Similar to simply guessing the session token!
    - Tied to the session: Else they fetch their own and substitute it
    - Expire soon: Limit exposure window
  - Very difficult to do manually, but can be integrated perfectly and completely into frameworks
  - Also: Make sure that there are no additional security problems
    - Browser vulnerabilities or XSS may allow extracting the token!
- This token should be secured
  - Use TLS for communication (**whole**, not only login page!)

CSRF: Prevention by Nonce

- Potential problems:
  - Open two forms in two tabs → Will both still work?
  - Bookmarking “result pages”?
  - Back button?
- Sometimes therefore only session-duration tokens
  - Like the session ID, but sent with every link and form submission (→ Cookie could be omitted then!)
  - Potential weakness: Leaking the token, especially in GET requests
    - Browser history, HTTP log files, referer headers…
    - This is only a slight problem, as several other security problems are absolutely necessary for any exploitation
- Ideal solution:
  - Send the token in POST requests only
  - Modify the application to only ever use POST requests
    - Includes clicking on a link!

Other prevention measures

- Use CAPTCHAs – for every single request
  - Similarly: Require login for each request
  - Similarly: Require one-time tokens/codes for each request
- This is very secure - but completely unusable!
  - Note: For very important or dangerous actions this might be an improved precaution (in addition to being logged in)
  - See online banking: Additional security measure for authorizing transfers (i/m/…-TANs, tokens, etc)

Other prevention measures

- Double cookie submission: Cookie with session ID is sent as a cookie (→ HTTP header) **and** as a (hidden) form value
  - Server checks if both values are the same
  - This is similar to a session-duration nonce, as it requires modifying the application to send this value with every action
  - Advantage: When dynamically added to request on sending, it always sends the current cookie, i.e. no problems with tabs!
  - Note: Potential problem with “httpOnly” flag of a cookie…
    - You don’t need to access the cookie if you can get it from the form field
    - Disable it, so you can add it to requests by JavaScript → Everyone else can then too…

Other prevention measures

- User-related prevention: Get users to…
  - always immediately log off after using the app
  - always use only a single app simultaneously
    - No tabbed browsing, no multiple browser windows
  - never switch applications (to E-Mail, another site…)
  - always enter links manually/through bookmarks
  - always check the full link on link-shortening services
  - never cache usernames/passwords
  - never allow sites to remember you (→ long-duration cookies)
  - disable JavaScript (or use plugins like NoScript)
- Problem: This is not very dependable or user-friendly…
- Never retrieve “a” parameter: Always retrieve a “GET” **or** a “POST” parameter, depending on what you expect
  - Trivial to replace POST by GET otherwise!

Prevention summary

- Users cannot prevent this in any useable way!
  - This **MUST** be protected against by the web site
  - They **CAN** mitigate the risk, but it is complex and burdensome
- It is very difficult to protect against “manually”
  - Use a web framework which does it for you
  - And take care not to subvert it
    - Creative URLs, additional features…
- CSRF is often forgotten, as compared to XSS
  - But it is very dangerous…

    …and often used
    - Advantage: Usually combined with other attacks and not “alone”

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)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at