Chapitre 2

Web Security Threats and Defenses

Business Training

1

Outline

- **Introducing the OWASP Project**
- **OWASP - Top 10 security issues**
  - A1- Injection - Never trust user input!
  - A2- Broken authentication
  - A3-Sensitive data exposure
  - A4- XML External Entities (XEE)
  - A5- Broken Access Control
  - A6- Security Misconfiguration
  - A7- Cross-site scripting (XSS)
  - A8- Insecure Deserialization
  - A9- Using components with known vulnerabilities
  - A10- Insufficient Logging & Monitoring
- **Understanding CSRF (Cross Site Request Forgery)**

2

Introducing the OWASP Project

3

What is OWASP ?

- **OWASP is the Open Web Application Security Project** https://www.owasp.org
  - It came online on December 1st, 2001
- **They're a not-for-profit worldwide charitable organization focused on improving the security of software on the web**
- **The mission of OWASP is to make application security visible so that people and organizations can make informed decisions about application security risks**
- **OWASP is frequently a reference point for security specialists and developers alike**

Flagship projects

- **The OWASP Flagship designation is given to projects that have demonstrated strategic value to OWASP and application security as a whole.**
  - **Free Tools**
    - OWASP Zed Attack Proxy
    - OWASP Web Testing Environment Project
    - OWASP OWTF
    - OWASP Dependency Check
  - **Code OWASP ModSecurity Core Rule Set Project**
    - OWASP CSRFGuard Project
    - OWASP AppSensor Project
  - **Documentation OWASP Application Security Verification Standard Project**
    - OWASP Software Assurance Maturity Model (SAMM)
    - OWASP AppSensor Project
    - OWASP Top Ten Project
    - OWASP Testing Guide Project

OWASP – Top 10 security issues

OWASP Top Ten (2017)

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML External Entities (XEE)
5. Broken Access Control
6. Security Misconfiguration
7. Cross-site scripting (XSS)
8. Insecure Deserialization
9. Using components with known vulnerabilities
10. Insufficient Logging & Monitoring

7

Top 10, From 2013 to 2017

OWASP Top 10 - 2013		OWASP Top 10 - 2017
A1 – Injection	→	A1:2017-Injection
A2 – Broken Authentication and Session Management	→	A2:2017-Broken Authentication
A3 – Cross-Site Scripting (XSS)	↘	A3:2017-Sensitive Data Exposure
A4 – Insecure Direct Object References [Merged+A7]	U	A4:2017-XML External Entities (XXE) [NEW]
A5 – Security Misconfiguration	↘	A5:2017-Broken Access Control [Merged]
A6 – Sensitive Data Exposure	↗	A6:2017-Security Misconfiguration
A7 – Missing Function Level Access Contr [Merged+A4]	U	A7:2017-Cross-Site Scripting (XSS)
A8 – Cross-Site Request Forgery (CSRF)	X	A8:2017-Insecure Deserialization [NEW, Community]
A9 – Using Components with Known Vulnerabilities	→	A9:2017-Using Components with Known Vulnerabilities
A10 – Unvalidated Redirects and Forwards	X	A10:2017-Insufficient Logging&Monitoring [NEW,Comm.]

8

Security doesn't end with the Top 10

- **We're going to look at the Top 10 risks in great detail**
  - The definitions of the risks
  - How they're exploited
  - Multiple ways of mitigating them
- **Security goes well beyond just technology implementation though**
  - Business processes may pose risks
  - Social engineering may still exploit people risks
  - Other technology risks remain outside the scope of this course
- **This is a rapidly evolving landscape**
  - Attackers are in an arms race to outsmart builders
  - New risks and attack vectors are constantly emerging
  - Stay vigilant!

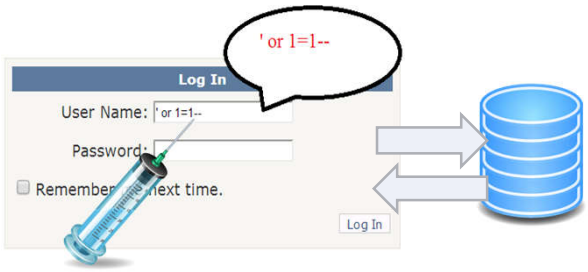
9

A1- Injection

10

Understanding SQL Injection

- **SQL injection is the placement of malicious code in SQL statements, via web page input.**
  - It usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will unknowingly run on your database.
  - **Sample SELECT Query without SQL Injection that will not work**
    SELECT * FROM UserDetails WHERE UserName=Tom';
  - **Sample SELECT Query without SQL Injection that will work**
    SELECT * FROM UserDetails WHERE UserName=**Tom' or '1'='1'**;



11

How to prevent ?
Quick Answer: Never trust user input!

- **Preventing injection requires keeping data separate from commands and queries.**
  - The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
    - Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
  - Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
  - For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
    - Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
    - Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

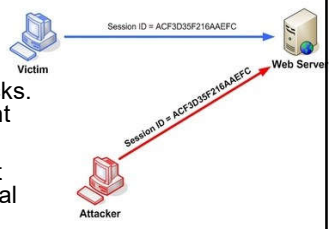
12

A2- Broken Authentication and Session Management

13

Understanding Broken authentication

- **This vulnerability gives the possibility for a hacker to impersonate other users. Once he takes possession of the identity of a victim, all the features that can be used by the victim will now be available to the attacker.**



- **Main causes**
  - The application does not handle brute force attacks. It's a type of attack with which the malicious agent gets a password by trying all the combinations
  - The application allows for weak passwords: short passwords, using dictionary words, without special characters or characters
  - The Session ID is sent to each page between the client and the server through a cookie, an address parameter, or an invisible form field.This means that the ID session is stored on the client side. Which would get it easily to impersonate a user
  - The exposure of the session ID in a URL.
  - No expiration mechanism is con fi gured for the session: The session is always open and the data it contains are always available

14

How to prevent ?

- Encrypt the data in transit using an SSL certificate. As the name implies, an SSL (secure socket layer) is a digital certificate that encrypts information sent between a web server and web browser.
- Where possible, implement multi-factor authentication to prevent automated, credential stuffing, brute force, and stolen credential re-use attacks.
- Do not ship or deploy with any default credentials, particularly for admin users.
- Implement weak-password checks, such as testing new or changed passwor
- Ensure registration, credential recovery, and API pathways are hardened against account enumeration attacks by using the same messages for all outcomes.
- Limit or increasingly delay failed login attempts. Log all failures and alert administrators when credential stuffing, brute force, or other attacks are detected.
- • Use a server-side, secure, built-in session manager that generates a new random session ID with high entropy after login. ds against a list of the top 10000 worst passwords.

15

A3- Sensitive Data Exposure

16

Understanting Sensitive Data Exposure

- **This is a vulnerability that occurs when the web application does not protect data considered to be sensitive, whether they are at rest or in transit.**
  - This can generate the disclosure of certain information that a hacker could exploit for attacks. These sensitive data may vary by application but are generally passwords, national credit card numbers or identifiers or source code of the page and the identifier of a session.
- **Rather than directly attacking crypto, attackers steal keys, execute man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's client, e.g. browser.**
- **Example Attack Scenarios**
  1. **Scenario #1**: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.
  2. **Scenario #2**: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g. at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie.

17

How to prevent

- **Do the following, at a minimum, and consult the references:**
  - Classify data processed, stored, or transmitted by an pplication. Identify which data is sensitive according to privacy laws, regulatory requirements, or business needs.
  - Apply controls as per the classification.
  - Don't store sensitive data unnecessarily. Discard it as soon as possible or use PCI DSS compliant tokenization or even truncation. Data that is not retained cannot be stolen.
  - Make sure to encrypt all sensitive data at rest.
  - Ensure up-to-date and strong standard algorithms, protocols, and keys are in place; use proper key management.
  - Encrypt all data in transit with secure protocols such as TLS with perfect forward secrecy (PFS) ciphers, cipher prioritization by the server, and secure parameters. Enforce encryption using directives like HTTP Strict Transport Security (HSTS).
  - Disable caching for responses that contain sensitive data.
  - Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2.
  - Verify independently the effectiveness of configuration and settings

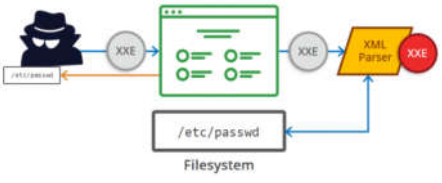
18

A4- XML External Entities (XEE)

19

Understanding XML External Entities

- **This vulnerability is a new entry in the top 10 of 2017 of OWASP. It got the 4th position mainly because of its ease of detection and the severity of the damage it can cause**
- **An XML External Entity attack is a type of attack against an application that parses XML input.**
  - This attack occurs when XML input containing a reference to an external entity is processed by a weakly configured XML parser. This attack may lead to the disclosure of confidential data, denial of service, server side request forgery, port scanning from the perspective of the machine where the parser is located, and other system impacts.
- **Main causes**
  - App accepts and interprets XML obtained from an external source.
  - App does not check the contents of XML documents before executing them.
  - App interprets the declared external entities and the DTDs in this application.



20

How to prevent

- **Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:**
  - Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
  - Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
  - Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'.
  - Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
  - Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.
  - Use mail server content scanning and filtering
  - Never use unfamiliar USBs
  - Keep your software and operating system updated
  - Backup your data

A5- Broken Access Control

Understanding Broken Access Control

- **Access control, sometimes called authorization, is how a web application grants access to content and functions to some users and not others.**
  - These checks are performed after authentication, and govern what 'authorized' users are allowed to do. Access control sounds like a simple problem but is insidiously difficult to implement correctly.
  - A web application's access control model is closely tied to the content and functions that the site provides. In addition, the users may fall into a number of groups or roles with different abilities or privileges.
- **Bad access control allow malicious agents to access sensitive data on the page, and to obtain administrator rights to change it.**
  - This vulnerability is quite common and exploitable, and its damage is severe because it can give full access to the application
- **Example attach scenarios**
  - An attacker simply force browses to target URLs.
  - Admin rights are required for access to the admin page.
    - http://example.com/app/getappInfo
    - http://example.com/app/**admin_getappInfo**
  - If an unauthenticated user can access either page, it's a flaw. If a non-admin can access the admin page, this is a flaw.

How to prevent

- **Developer training is essential to identify and mitigate XXE. Besides that, preventing XXE requires:**
  - Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
  - Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
  - Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'.
  - Implement positive ("whitelisting") server-side input validation, filtering, or sanitization to prevent hostile data within XML documents, headers, or nodes.
  - Verify that XML or XSL file upload functionality validates incoming XML using XSD validation or similar.

A6- Security Misconfiguration

25

Understanding Security Misconfiguration

- **Attackers will often attempt to exploit unpatched flaws or access default accounts, unused pages, unprotected files and directories, etc to gain unauthorized access or knowledge of the system.**
- **Is the Application Vulnerable?**
  - The application might be vulnerable if the application is:
    - Missing appropriate security hardening across any part of the application stack, or improperly configured permissions on cloud services.
    - Unnecessary features are enabled or installed (e.g. unnecessary ports, services, pages, accounts, or privileges).
    - Default accounts and their passwords still enabled and unchanged.
    - Error handling reveals stack traces or other overly informative error messages to users.
    - For upgraded systems, latest security features are disabled or not configured securely.
    - The security settings in the application servers, application frameworks (e.g. Struts, Spring, ASP.NET), libraries, databases, etc. not set to secure values.
    - The server does not send security headers or directives or they are not set to secure values.

26

How to prevent

- **Secure installation processes should be implemented, including:**
  - A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically, with different credentials used in each environment. This process should be **automated** to minimize the effort required to setup a new secure environment.
  - A **minimal platform** without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
  - A **task to review and update the configurations** appropriate to all security notes, updates and patches as part of the patch management process. In particular, review cloud storage permissions (e.g. S3 bucket permissions).
  - A segmented application architecture that provides effective, secure separation between components or tenants, with segmentation, containerization, or cloud security groups.
  - Sending security directives to clients, e.g. **Security Headers**.
  - An **automated process** to verify the effectiveness of the configurations and settings in all environments

27

A7- Cross-site scripting (XSS)

28

Understanding Cross-site scripting (XSS)

- **Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted websites.**
  - An attacker can use XSS to send a malicious script to an unsuspecting user. The end user's browser has no way to know that the script should not be trusted, and will execute the script. Because it thinks the script came from a trusted source, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser and used with that site. These scripts can even rewrite the content of the HTML page
- **Cross-Site Scripting (XSS) attacks occur when:**
  - Data enters a Web application through an untrusted source, most frequently a web request.
  - The data is included in dynamic content that is sent to a web user without being validated for malicious content.
- **Stored and Reflected XSS Attacks**
  - Stored XSS Attacks
    - Stored attacks are those where the injected script is permanently stored on the target servers, such as in a database, in a message forum, visitor log, comment field, etc. The victim then retrieves the malicious script from the server when it requests the stored information.
  - Reflected XSS Attacks
    - Reflected attacks are those where the injected script is reflected off the web server, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request. Reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other website.

29

How to prevent

- **Preventing XSS requires separation of untrusted data from active browser content. This can be achieved by:**
  - Using frameworks that automatically **escape** XSS by design, such as the latest Ruby on Rails, React JS. Learn the limitations of each framework's XSS protection and appropriately handle the use cases which are not covered.
  - Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities. The OWASP Cheat Sheet 'XSS Prevention' has details on the required data escaping techniques.
  - Applying **context-sensitive encoding** when modifying the browser document on the client side acts against DOM XSS. When this cannot be avoided, similar context sensitive escaping techniques can be applied to browser APIs as described in the OWASP Cheat Sheet 'DOM based XSS Prevention'.
  - Enabling a **Content Security Policy (CSP)** is a defense-in-depth mitigating control against XSS. It is effective if no other vulnerabilities exist that would allow placing malicious code via local file includes (e.g. path traversal overwrites or vulnerable libraries from permitted content delivery networks)

30

A8- Insecure Deserialization

31

Understanding Insecure Deserialization

- **An insecure deserialization vulnerability exists when an application doesn't properly secure this process.**
  - If a deserialization implementation is left to its default settings, an application can have little to no control over what data is deserialized. In the most extreme cases, this can include any incoming serialized data from any source, with no verification or precautions.
- **Conceptually, this is very similar to the XML External Entities (XXE) risk, especially since XML is a format used for serialization. We've already looked at the vulnerabilities of XML specifically, but insecure deserialization applies to a wider range of data formats. Some of the more common serialization formats include JSON, XML, BSON and YAML.**

32

How to prevent

- **The only safe architectural pattern is not to accept serialized objects from untrusted sources or to use serialization mediums that only permit primitive data types.**
- **If that is not possible, consider one of more of the following:**
  - Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
  - Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of
  - classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.
  - Isolating and running code that deserializes in low privilege environments when possible.
  - Logging deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
  - Restricting or monitoring incoming and outgoing network connectivity from containers or servers that deserialize.
  - Monitoring deserialization, alerting if a user deserializes constantly

33

A9- Using components with known vulnerabilities

34

Understanding Components with Known Vulnerabilities

- More and more apps are using pre-existing components rather than being coded completely from scratch.
  - Web applications often need fast turnaround, and with the quantity of open-source components available, there's no reason not to make use of them. Analysis indicates that 96% of applications make at least some use of open-source components
- The problem is exacerbated by the tendency for software developers to use open source components often developed without any quality control process
  - Under pressure to deliver at speed, these components are not sufficiently checked before use. The result can be new websites and applications with deeply embedded vulnerabilities unknown to the application operator.

35

How to prevent

- **There should be a patch management process in place to:**
  - **Remove unused dependencies**, unnecessary features, components, files, and documentation.
  - **Continuously inventory the versions of both client-side and server-side components** (e.g. frameworks, libraries) and their dependencies using tools like versions, DependencyCheck,retire.js, etc. Continuously monitor sources like CVE and NVDfor vulnerabilities in the components. Use software composition analysis tools to automate the process. Subscribe to email alerts for security vulnerabilities related to components you use.
  - **Only obtain components from official sources over secure links**. Prefer signed packages to reduce the chance of including a modified, malicious component.
  - **Monitor for libraries and components that are unmaintained or do not create security patches for older versions**. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue

36

A10- Insufficient Logging & Monitoring

37

Understanding Insufficient Logging & Monitoring

- **Rather than being a vulnerability in itself, Insufficient logging and Monitoring is an OWASP category that covers the lack of various best practices that could in turn prevent or damage control security breaches.**
  - The category includes everything from unlogged events, logs that are not stored properly and warnings where no action is taken within reasonable time.
- **When OWASP made this a top 10 vulnerability, the category became part of the list based on a industry survey rather than quantifiable data, so it is unclear how many systems are affected. However, there are always improvements to be made, and logging and monitoring is something that everyone should always have in mind.**
- **In 2016, the average detection rate for an attack was 191 days. Had the breaches been detected earlier the impact could be drastically minimised.**
  - When a security breach is not discovered in time, the attackers have time to escalate the attack further into the system. It also means they can use the stolen data for malicious purposes for a longer time.

38

Understanding Insufficient Logging & Monitoring

- **How to discover Insufficient Logging and Monitoring ?**
  - From an outsider perspective, Insufficient Logging and Monitoring is really hard to detect. The logs should only be exposed internally, so whether or not logging and monitoring best practices are implemented is not something an outsider can determine.

How to prevent

- **As per the risk of the data stored or processed by the application:**
  - Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis.
  - Ensure that logs are generated in a format that can be easily consumed by a centralized log management solutions.
  - Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
  - Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion.
  - Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later.
- **There are commercial and open source application protection frameworks such as OWASP AppSensor, web application firewalls such as ModSecurity with the OWASP ModSecurity Core Rule Set, and log correlation software with custom dashboards and alerting.**

Understanding Cross-Site Request Forgery (CSRF)

41

Understanding Cross Site Request Forgery (CSRF)

- **Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated.**
- **CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.**
  - With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing.
    - If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

42

Understanding Cross Site Request Forgery (CSRF)

- **How does the attack work?**
  - Let us consider the following example: Alice wishes to transfer $100 to Bob using the bank.com web application that is vulnerable to CSRF. Maria, an attacker, wants to trick Alice into sending the money to her instead. The attack will comprise the following steps:
    1. building an exploit URL or script
    2. tricking Alice into executing the action with social engineering
  - GET scenario
    - If the application was designed to primarily use GET requests to transfer parameters and execute actions, the money transfer operation might be reduced to a request like:

      GET http://bank.com/transfer.do?acct=BOB&amount=100 HTTP/1.1
    - Maria now decides to exploit this web application vulnerability using Alice as her victim. Maria first constructs the following exploit URL which will transfer $100,000 from Alice's account to her account. She takes the original command URL and replaces the beneficiary name with herself, raising the transfer amount significantly at the same time:

      http://bank.com/transfer.do?acct=MARIA&amount=100000
  - The social engineering aspect of the attack tricks Alice into loading this URL when she's logged into the bank application. This is usually done with one of the following techniques:
    - sending an unsolicited email with HTML content
    - planting an exploit URL or script on pages that are likely to be visited by the victim while they are also doing online banking

Understanding anti-forgery tokens

- **CSRF attacks work because they're predictable**
- **The attack is merely reconstructing a request adhering to the same structure as a legitimate one (path and parameters)**
- **To mitigate this risk, we can add randomness via a CSRF token**
- **A token is a random string known to both the legitimate page where the form is and to the browser via a cookie**

Understanding Cross Site Request Forgery (CSRF)

- **What makes a CSRF attack possible?**
  - Authenticated sessions are persisted via cookies
  - The cookie is sent with every request to the domain
  - The attacking site recreates a legitimately formed request to the target site
  - Although the request has a malicious payload (query string parameters or post data)
  - The victim's browser is tricked into issuing the request
  - For all intents and purposes, the target website views it as a legitimate request

45

How to prevent

- **Preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.**
  - The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is more prone to exposure.
  - The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs a greater risk that the URL will be exposed to an attacker, thus compromising the secret token.
  - OWASP's CSRF Guard can automatically include such tokens in Java EE, .NET, or PHP apps. OWASP's ESAPI includes methods developers can use to prevent CSRF vulnerabilities.
  - Requiring the user to reauthenticate, or prove they are a user (e.g., via a CAPTCHA) can also protect against CSRF.

46