KULeuven

Development of secure software

Mooc
Web security

Kevin Loonen

Contents

1 Is security an illusion?

1.1 Introduction

1.2 The terrifying state of affairs

Database breaches because most of them use the default settings. This makes them insecure. But even secured databases are subject to attacks: SQL injection.
Most know attacks:

1. **SQL injection** : Execution malicious SQL code to extract unauthorized modification of data.
2. **Direct object reference** : Use of an identifier to refer to an object. Ex. incrementing numeral identifier.
3. **XSS** : Use of malicious javascript code to impersonate the current user.

1.3 Why everyone is a target

Assumption that the application is not important, doesn't handle explicit information. Why would someone try to attack it?
Attackers care more than data alone.

1. **Hardware** : take advantage of computing resources. (ex mining for crypto currency)
2. **Storage and bandwidth** : host the attacker his files (illegal wares, malicious exploits, phishing pages, ...)
3. **Use as botnet** : sell botnet to execute commands (DoS)

When the application handle sensitive data (username, passwords), the attacker can steal this information, aka a **data breach**. Having this information puts the attacker in control. Use the data for financial gain:

1. **Extorting the company** : threaten to release the data.
2. **Selling personal data**
3. **Ransomware attack** : after breaking in, encrypt all the data on the server.

1.4 Approaching security from the ground up

Biggest obstacle is not technology, but the approach to security. Seen as an obstruction that gets in the way of functionality and productivity. Usualy security check right before deployment in terms of a penetration check. This will result in a report of security problems.
Some disadvantages:

1. **Costly** : To perform such a test for each release.
2. **Coverage** : Is everything tested, or just a part of the application.
3. **Fundamental problem** : Should the whole application be redesigned?

Implement various security activities:

1. Raise security awereness among developers. Knowing the common threats and being able to recognize dangerous patterns.
2. Secure coding guidelines

Have a look at OWASP, where you can see the most important attacks, and how you can defend against it. Recommended to adopt an explicit process to enumerate potentional threats. This is known as **threat modeling** or **risk analysis**.
Steps in threat modeling:

1. Enumerate potentional threats
2. Is the threat relevant and significant
3. How to mitage the threats in the application

Power of modeling lies in it's proactive nature, it let you reason about threats and defences. Therefor you can incorporate security into the core of the application's architecture.
A penetration test is a usefull activity, it helps to verify if the taken measurements are effective, and if you overlooked some aspects.

1.5 Browser security concepts

Security decisions in the browser rely on a derivative of the URL, known as the "origin". This is a triple consisting of the scheme, host and port.
Most significant security policies that uses the origin is the Same-Origin Policy, or **SOP**. States that contexts from the same origin can freely interact with each other, while context from different contexts are isolated from each other. When two windows with different origins try to interact with each other, the POC will kick in. It will prevent direct access to the contents of the window, and only exposes a limited API. A browsing context is protected against undesired access that can lead to information extraction or undesired modifications.
An attack on one application can spread out to the other application from the same origin.
Original goal of POC was regulating interactions between different browsing contexts. Now this includes browser provided storage. Another example is user-granted permissions. Cookies on the other hand, use different set of rules.

1.6 Cookies

A sever sends information to the browser, and the browser returns it on subsequent requests. These are often used to keep track of session information such as the authenticated state of the user.
A cookie is nothing more then a key-value pair. A host can only set a cookie for its registred domain and subdomains. The **Path** attribute narrows the scope of the cookie to a certain resource.
A cookie belongs only to a domain.

1.7 Towards client-centric security

Alternative browser security mechanisms:

1. Cookie security flags that restrict default behavior of cookies.

There are a lot of policies that are set by the server, but enforced by the browser. This is known as **"server-driven browser-enforced security policies"**. Examples are:

- Content Security Policy
- HTTP Strict Transport Security

These policies offer the server additional defence mechanisms. Combining these policies with existing defences enables building a layered defence strategy.
Most of the new security policies are intended as second line defence. They are not suited as a sole replacement of traditional defences.

1.8 Recap and conclusion

Vulnerabilities can be introduced in every component of the application. By common mistake, insecure dependencies or a lack of knowlegde about specific threats.
One vulnerability can compromise the whole application.

Extra reading:

- How work cookies :
  `https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies`
- Microsofts take on a secure development lifecycle :
  `https://www.microsoft.com/en-us/sdl/default.aspx`
- A different perspective on integrating security activities in the development lifecycle :
  `http://www.swsec.com/resources/touchpoints/`
- A good description of various threat modeling approaches :
  `https://www.owasp.org/index.php/Application_Threat_Modeling`

2 Securing the communication channel

2.1 Towards secure communication

Users don't think about connection to open wireless networks. Can you offer any security guarentees if your users use an insecure network? It's possible, but getting it right is harder then you can imagine.
Therefor there is HTTPS as baseline for secure applications. This prevents many network-based attacks. HTTP pages are marked as insecure and they can't use sensitive API's.
Handshake to establish HTTPS is the most crucial step. The importance of certificates to ensure the security properties of HTTPS

2.2 The dangers of an unprotected channel

Security different on a wireless network compares to a wired network.
For a **wired network** you need physical access to the cable, which means you need access to the building.
For a **wireless network** you simply have to be nearby. It's also possible to be several kilometers away to attack a wireless network. This definitely increases the risks.
Different types of networks:

1. **Open:** there is no security. Everyone can connect and all data can also be intercepted.
2. **Password:** are more secure, if at least no one else has the password. So at hotels, conferences, coffee shop, ... everyone has the password, so there are still risks.
3. **Professional:** here you have a username and a password. These networks are the most secured.

There is a flaw in WPS (Wi-Fi Protected Setup) that still can be used to gain access to the network.
There is another flaw in the Wi-Fi standard itself and this allows you to attack most devices even though they are fully secured and used all the recommended practices.
HTTPS helps against all of this. The protocol makes sure that you are talking to this website and nothing else. It encrypts your traffic. So no one can see what you are sending and it also authenticates the traffic. This means no one can modify the data or inject content into a web page.
Take away message for developpers: Use HTTPS and properly configure it. It's the best defence that we have.

2.3 The recent push for HTTPS

Browsers are the main force to push HTTPS. They first tackled the insecure login pages from HTTP. This should trigger website owners to improve their

security practices.
Now browsers only allow secure context to use sensitive features like retrieving location or accessing the webcam or microphone.
Good way to measure the quality of any website is to scan it with the **Qualys SSL Server Test**. It runs a battery of tests against the web server and reduces all the results into a single rating.
The **SSL Pulse** project uses the SSL Server Test to automate the scanning of a list of popular sites. To test from a local machine or intranet, you can use `https://testssl.sh`
Protecting data in transit has become almost mandatory for any website.

2.4 Security properties of HTTPS

Why is HHTPS more secure, and where does that security come from? There is an extra protocol added, SSL/TLS (Secure Sockets Layer / Transport Layer Security). How does it work and what does it offer.
Because TLS fits into the network stack, it's transparant to the applications using it. The TLS record encapsulate the HTTP message and ensures **confidentiality** and **integrity**. TLS is therefor suitable for other protocols as SMTP, POP and IMAP.
Security properties of SSL/TLS:

- Confidentiality: ensures that unauthorized parties cannot read the data transmitted over the network.
- Integrity: ensures that when the data is tampered with, the receiver can detect this. When this happens, the receiver no longer trusts the message and discards it.
- Authenticity: ensures that the client who's sending the message, really the client is.

The TLS protocol consists of several sub-protocols. The **handshake protocol** and the **record protocol**.
The record protocol ensures the confidentiality and integrity of the transmitted data.
The handshake protocol handles establishing a new connection. During this phase, the browser and server negotiate the algorithms and parameters that will be used later on. This protocol also ensures authenticity. The browser verifies first the identity of the server before establising a secure channel. This is crucial to avoid main in the middle attacks. If the attacker can impersonate the server, confidentiality and integrity are useless.
A few limitations of TLS. The attacker can still observe the communication. From this data, the attacker can derive which server you are connecting to. So it does not ensure privacy.
Even encrypted messages are subject to traffic analysis techniques. Traffic analysis of encrypted data can leak information about unencrypted contents.

2.5 Keys, certificates and ciphers

HTTPS gets its security properties from cryptographic building blocks such as encryption, HMACs and dignital signatures. Each version of TLS specification introduces new cryptographic algorithms. Each algorithm server a particular purpose, and the combination is referred to as a **cipher suite**.
Confidentiality is achieved using a symmetric-key algorithm. A message is encrypted with a specific key. Since the algorithm is symmetric, decrypting the message only works with the same key.
Integrity come from using an HMAC function. This function uses a secret key to calculate a chechsum for a specific message. To verify a chechsum, the receiver recalculates the chechsum on the received message using the same key. By comparing the chechsums, the receiver can be sure that the message has not been tampered with.
Browser and server need to have access to the same keys. These keys come from the **pre-master secret (PMS)**. Use asymmetric-key algorithms to send the pre-master secret in a secure way over an unsecured channel.
The asymmetric-key algorithms use a key pair, that consists of a private and a public key. To exchange the PMS, the browsers uses the public key of the webserver, and the webserver decrypts the message with its private key.
Knowing all the public keys is not practical. The server can send it's public key in the first step of the handshake. Therefor we need authenticity. This is ensured by using certificates. A certificate associates a specific public key, and hence the associated private key with a specific domain. The browser verifies the validity of the certificate and checks if the domain matches the certificate.
The security of an HTTPS connection now depends entirely on the authenticity property. To impersonate a legitimate server, the attacker needs a valid certificate and the associated key pair.

2.6 Common misconceptions about HTTPS

A common misconception is that HTTPS is only relevant for sensitive content, such as login forms or online payments. When an application sends an HTTP request, the user becomes vulnerable to a variety of attacks.
A second misconception is that HTTPS has significant performance impact, and servers won't be able to handle it. CPUs come with support for AES and TLS has undergone significant performance tuning.
Another misconception states that certificates are expensive, and a nightmare to configure. There are services that offer free certificates for everyone. They can even provide a tool support to automate the whole process (request, install, renew).
You can only run one HTTPS application per IP address. Almost every client supports a TLS extension called **Server Name Indication (SNI)**.

2.7 Perfect Forward Secrecy

In a classical HTTPS deployment, depend the three main properties on the secure exchange of the pre-master secret. If someone is listening, they can't decrypt the message. There is only one important disadvantage with this type of key exchange. It cannot guarentee confidentiality towards the future.
An attacker can record the entire HTTPS conversation, including the handshake. If the attacker comes in possession of the server's private key, he can decrypt the pre-master secret, derive the shared keys, and decrypt the entire conversation. Disclosure of the private key can happen by mistake or by stealing it from the server.
Instead deploy a key exchange algorithm that support perfect forward secrecy. The most common algorithm is the **Diffie-Hellman key**. Is capable of establishing a shared secret over an insecure channel without encryption. A possible attacker can never reconstruct the shared secret. Only directly involved parties can calculate the pre-master secret.
How does DH work? Browser and server start with a same value, each of them add something random to that value and exchange those values with eachother. Then when they received eachothers random generated value, they add their own random value. Now they share the same shared secret which they can use. This method is still vulnerable against Man In The Middle attacks. To ensure authenticity, the DH protocol is combined with an asymmetric key algorithm.

2.8 Dealing with mixed content

Mixed content blocking: protect a secure page by refusing to load scripts and styles over an insecure HTTP connection. If we load resources over HTTP, the security guarentees no longer hold.
Distinction between:

- Passive content: content that is only displayed, and thus only pose a limited threat (Images, audio, video). The developer sees a warning.
- Active content: content has full acces to the page, and this poses a significant risk (scrypts, styles, iframes, objects).

Fixing is straight forward, all resources are loaded over HTTPS.
When you have a lot of mixed content in years' worth of archives, it's harder than it seems. Advice: take it slow. Start with a section of the site, the keep adding more and more. For this you can leverage **Content Security Policy (CSP)**. CSP allows you to control what content can be loaded on your pages. CSP can send reports when a given policy is violated.

2.9 Partial HTTPS deployments are not the answer

An attacker can modify an URL that will go to a secure HTTPS part of the website, to just the HTTP version. The user won't notice this, and the attacker can extract the password. This is the danger of a public HTTP page. It allows

an attacker to get a foothold within the application.
There are different kind of attacks that can be performed on this manner:

- Preventing the upgrade to HTTPS
- Phishing
- Social engineering

2.10 Redirecting HTTP to HTTPS

Turning off support for HTTP is a straightforward way to force use of HTTPS. From a security perspective, this seems like a good idea. For usability, this will have a significant impact on the application. All exisiting URLs that point to an HTTP page will stop working (search engines, links external website will break).
Make use of a redirect mechanism to send all traffic from the HTTP version to the HTTPS version. When the server receives a call for an HTTP page, it instructs the browser instead, to load the HTTPS version of the page.
Why do browsers not use HTTPS to send the first request? Because it is hard to address. Not all applications support HTTPS out of the box. Possible solution is to try again over an HTTP connection. But how can the browser tell the difference between a legitimate error and an attack?

2.11 Enabling Strict Transport Security

The redirect from HTTP to HTTPS is a crucial step in any HTTPS deployment. But the redirect itself remains vulnerable to network-based attacks. By executing an **SSL stripping attack**, the attacker intercept the request and prevents the redirect. Instead the attacker fetches the HTTPS page and serves it over HTTP. Only way of stopping the attack, is adding additional security policies, the **HTTP Strict Transport Security policy (HSTS)**. With HSTS, a web application instruct the browser to use HTTPS by default for a specified period. The server configures the HSTS policy by sending a response header. "Strict-Transport-Security" has two parameters: max-age and includeSubdomains. Set a lifetime based on how frequente a user will visite the website (2 weeks, 1 year ...). The includeSubdomains flag can be dangerous. If a legacy HTTP service is still running, it will become unavailable to all browsers that have seen the HSTS policy.
How can we protect the user for the first visit of the application. Inform the browser up front that the application wants to use HTTPS. Browsers support a preload list of sites that want this type of behavior. Before you can add yourself to the list, you need to meet several requirements (`hstspreload.org`:

1. add the preload flag (show your consent of being added to the list)
2. need to have the includeSubdomains flag enabled for your entire domain
3. need to have a sensible HSTS configuration (at least eighteen weeks)

2.12 Practial deployment scenarios

Several questions rise once you start deploying HTTPS:

- Run different HTTPS sites on one server?
- Where do you store the sensitive cryptographic material?
- What happens if you use a third-party service for taffic optimization?

It was the case that one server was bound to one HTTPS application. The TLS handshake does not support sending the domain name of the application up front to get the right certificate.
However there is now a TLS extension known as **Server Name Indication (SNI)**. With SNI enabled, the client includes the domain name of the application it is connection to in the first step of the handshake.
The capablitiy of running multiple HTTPS site on one IP address enables a broad range of practical deployment scenarios. Deploying a reverse proxy service as a dedicated TLS endpoint. Establishes secure communication with a client. The proxy forwards all request to internal services. This has several benefits:

- deployment of internal services becomes a lot more practical. (no worry about public-facing TLS configuration)
- significant security benefit. (isolation of the cryptographic material on a seperate host or container -¿ out of reach for an attacker)

2.13 Analyzing the trust model behind HTTPS

For authenticity there is a need for a whole ecosystem. Certificates are issued by a Certificate Authority (CA). The CA signs the certificate with its private key. The browser can verify the certificate by using the CA's public key. How can the browser know whether a key belongs to a CA or not? There are differnet levels of CAs:

1. **Intermediate CA**: signing end user certificates
2. **Higher-level CA**: prove the legitimacy of its key material
3. **...**: (several levels of Intermediate CAs)
4. **Root CA**: no higher authority to vouch for the validity of their key. Browsers have a list of hardcoded root CAs. These CAs are trusted by default by browsers and operating systems.

If an attacker attempts to request a certificate for an existing domain, we expect the CA to deny this request. Otherwise, the attacker can use this certificate to impersonate that application. This does not only violates authenticity but also undermines confidentiality and integrity.

How to verify whether a request is legitimate? Most common way is to verify whether the requester is in control of the domain (by domain validation). Different types of validation:

- Domain validation (basic domain ownership verification):
  - sending a code to a reserved email address
  - place a particular response at a specific location on the web server
- Organization validation (no fixed set of validation rules)
- Extended validation (extensive validation of the business and the certificate request)

2.14 The fragility of the certificate ecosystem

The ecosystem surrounding certificates is quite fragile. The unconditional trust in the root CAs. Browsers trusts hundreds of root CAs. Any CA can issue a certificate. CAs have mistakenly issued certificates before. Even worse, attackers have compromised CAs, enabling them to issue fraudulent certificates.
Relevant technologies to address the problem.

1. Certificate Transparency (CT). All CAs are required to publish all certificates they issue into a public log (Detect issuing fraudulent certificate).
2. Certificate Authority Authorization (CAA). Allows a domain owner to limit the number of CAs that are allowed to issue certificates for a domain. Stops other CAs from mistakinly issuing a certificate for a domain. Configurable by DNS records that CAs have to check.

Determine which key the server can use. Imposible for an attacker to use his own key pair in combination with a fraudulent certificate.

- HTTP Public Key Pinning (HPKP). Hard to get right.
- DNS-based Authentication of Name Entities (DANE)

Trust model of HTTPS hinges on the legitimacy of a certificate.

2.15 Certificate Transparency

Problem with fraudulent certificates:

- issued by a real CA, so is accepted by all browsers
- detection is slow and mostly accidental

This weakn the security of all HTTPS deployments. Process to request a certificate based on CT:

1. administrator request a certificate

2. CA creates the certificate and submits it to the certificate transparency log service
3. the log publishes the certificate and return a signed certificate timestamp (SCT)
4. CA sends the certificate to the administrator

Browser checks the certificate and the SCT. CT depends on two assumptions:

- domain owners need to setup log monitoring so they can detect fraudulent certificates
- browsers need to mandate the presence of an SCT, to force CAs to publish their certificates in a log.

Supported ways to send the SCT to the browser:

1. CA can embed the SCT information into the certificate (requires more effort from the CA side. generate a precertificate, submit it and generate a final certificate)
2. use a TLS extension to deliver the SCT information (a lot more complicated)
3. send SCT information along with OCSP responses (requires little extra effort by any involved party)

OCSP stapling: a TLS extension adds OCSP information to the handshake, this proves to the browser that the certificate has not been revoked. Witch OCSP stapling, the server fetches an OCSP response from the CA. This response only remains valid for a few days. Needs to be refreshed frequently.

2.16 Recap and conclusion

Several challenges with establishing a secure communication channel. HTTPS offers secure communication between the browser and the web application. HTTPS offers three important security properties: confidentiality, integrating and authenticity (CIA).
Redirect HTTP to HTTPS, but watch out for SSL Stripping. To prevent this, deploy Strict Transport Security policy.
Legitimacy of the certificate is crucial and come down to the trustworthiness of CAs. Therefor we use Certificate Transparency to log everything and detect fraudulent certificates.
Using HTTPS in combination with Strict Transport Security is the baseline for building secure web applications.

Extra reading:

- An overview of steps to take to prepare the move towards HTTPS : `https://online.marketing/guide/https/`

- An explanation of how the SSL server test comes to its score :
  https://www.ssllabs.com/projects/rating-guide/
- An overview of SSL/TLS Deployment Best Practices :
  https://www.ssllabs.com/projects/best-practices/
- The free OpenSSL cookbook containing lots of practical details about configuring TLS :
  https://www.feistyduck.com/books/openssl-cookbook/
- Moxie Marlinspike's entertaining talk on Authenticity in TLS :
  https://vimeo.com/32912604
- The account of Wired's upgrade to HTTPS, part1, part2 and the final report :
  https://www.wired.com/2016/04/wired-launching-https-security-upgrade/
  https://www.wired.com/2016/05/wired-first-big-https-rollout-snag/
  https://www.wired.com/2016/09/wired-completely-encrypted/
- F5 labs' 2016 TLS Telemetry Report, documenting the evolution of various SSL/TLS features :
  https://f5.com/Portals/1/PDF/labs/R065%20-%20REPORT%20-%20The%202016%20TLS%20Telemetry%20Report.pdf
- The 'Black Tulip' report on the compromise of DigiNotar :
  https://www.rijksoverheid.nl/binaries/rijksoverheid/documenten/rapporten/2012/08/13/black-tulip-update/black-tulip-update.pdf
- A detailed explanation of what happens during the initialization of an HTTPS connection :
  http://www.moserware.com/2009/06/first-few-milliseconds-of-https.html
- Facebook's tool to subscribe to Certificate Transparency logs :
  https://www.facebook.com/notes/protect-the-graph/introducing-our-certificate-transparen
  1811919779048165/

3 Preventing unauthorized access

3.1 Access control in web applications

How do you keep track of the authentication state during the users's session? Many applications accept fraudulent operations in the user's name.
Three main points of the chapter:

- Authentication: proper way to store credentials + benefits of multi-factor authentication
- Session management
- Authorization: enforce proper permissions on access to data or operations. Needs to ensure that actions carried out in the user's name are intentional.

3.2 Introducing state into your application

HTTP is based on requests and responses. Subsequent request are not related to each other. HTTP is therefor **stateless**. Authentication and authorization are challenging in a stateless protocol.
How to propagate an authenticated user to the next request? The browsers uses **Basic Authentication**. It makes appear a window for the client to give it's credentials, which will be send over encoded in **Base64**. But it can be undone very easily.
Drawbacks of Basic Authentication:

- only provides the identity of the user, no additional info
- username and password are send with each request
- no decent credential management in the browser
- no UI integration with the web application: the login happens in a popup window

Now web applications use a custom authentication form in combination with session management

3.3 The truth about passwords

Passwords are insecure, inefficient, ... Better make use of alternative authentication like hardware tokens or mobile applications.
What is the common problem with passwords, and how to make them more secure?
If you have the password, you can use it to get access to content. There are several attacks to guess the password.

- using personal information about the victim (children, pet)

- dictionary attacks (actual words or word combinations)
- using lists of commonly used passwords
- phishing attack: lead the user to a malicious website that looks like an innocent one. User authenticates on that website, and the attacker steals the user's credentials
- hacking the application and stealing the database. Passwords are often stored in an insecure way. Amplified because people use the same password for multiple accounts (`https://haveibeenpwned.com/`)

Best way to improve the security for passwords, is to use a **password manager** which stores credentials for various accounts. The user don't have to remember all his passwords. This enables to use a unique and complex password for every account. This will eliminate a couple of security problems.

1. Guessing attacks irrelevant: passwords long and random
2. No need to reuse passwords across accounts
3. Password managers increase resilience against phishing attacks. Most managers use the autocomplete features in the browser. The manager finds the account linked to the URL and offers to login with the click of a button.

If you don't trust password managers, you can use the following technique: divide accounts into trust levels, and use a different password for each level (seperate sensitive sites and unimportant sites). Only one level can be compromised.

3.4 Insecure password storage

Passords often seen as weak form of authentication and comes from how users handle passwords (reusing passwords across applications)
Every application can suffer from a data breach. Adhere a defens-in-depth strategy to minimize the impact of a breach. A crucial aspect is storing passwords. How not to store passwords:

- as plain text
- hashed passwords. some issues arize:
  - two users take the same password. the hash will be the same.
  - hacker pre-compute the hashes for passwords. Compare this list with the stolen data, and see which hashes are equal. (**Rainbow table**)

One way to prevent attacks against hashed passwords is adding a **salt**. This is a long random string that will be added to the password, to make the hash unique, even if two users have the same password. Rainbow tables are now

ineffective because they don't take salt into account.
But salting and hashing is still weaker than you would expect. Problem is the use of hashing algorithms. Designed to be fast, but enables brute force attacks.

3.5 Secure password storage

To prevent brute force attack you should use a dedicated password hashing function. These are cryptographic functions that perform a certain amount of iterations to calculate the output. Are expensive to execute and withstand brute-force attacks by design. All of these functions also use a salt.
BCRYPT is a popular password hashing function. There are several parts in the output:

1. indicates which algorithm has generated
2. specifies a cost parameter (makes the function more expensive to execute)
3. contains the salt and the generated hash

What to do when you already have an application running that stores passwords insecurely?

- Gradual upgrade: takes place during authentication. Application checks how the password is stored in the database. Update when the hash is calculated with a legacy algorithm. Replace old hash with the new generated hash.
- Upgrade in one go: use unsalted hashed as input for your password hashing function. Database contains now BCRYPT hashes of *insert hash function here* of the plaintext password. Keep track which password use the double hash function and which the single. The double hash functions will be filtered out to single when the user logs in.

3.6 Preventing enumeration attacks

The attacker need an enumeration attack to get a valid username. Determine if the username exists in the application.
Places where an application can leak information whether an username exists or not:

- **Login form**: depending on the error message, the attacker can know if the username exists. (**Invalid** username or password vs **Unknown username or password**)
- **Account recovery**: typically request an email address. It let the user know if the address is not found.
- **Registration form**: when choosing a username or email address discloses it when it already exists.

Prevent enumeration attacks. Avoid leaking information about the existence of an account.

- **Authentication form**: use the same error message for invalid and unknown user names.
- **Account recovery**: refrain from giving the user immediate feedback about the existence of an email address. Instead lookup if an email address exists.
  - It exists: send a recovery mail
  - If not: send an email to the user about potentional account recovery.

  Give a mesage that the user could have used another email address, or he can register a new account.
- **Registration form**: bit more tricky.
  - Simple case: application uses email addresses as a unique identifier. Use the same mechanism as in **recovery**.
  - More complicated: application uses arbitrary identifiers such as usernames. Needs to be unique, but are chosen by the user, which can lead to enumeration attacks. Limit risks by requiring to registrate with an email address first. Limits the attempts to pick a user name. Attacker learns a tiny amount of the application during the attack.

The are two common strategies to prevent brute force attacks:

1. block an user account after a certain number of failed authentications. But how will you distinguish a forgetful user from an attacker? What happens when an attacker locks all the accounts in your application? (**lockout policy**)
2. slow down authentication attemps after a couple of failures. After 3 attemps, the application waits for a second. After 4, it waits 3 seconds,...

3.7 Beyond password-based authentication

A password is a knowledge-based authentication factor. There are other means of authentication:

- **physical device**: mobile phone, smart card, usb key.
- **inherent to the user (bio-metrics)**: fingerprints, retina scans.
- **behavior and context**: user behavior and user location.

Multi-factor authentication is a combination of a knowledge-based factor with a second factor. Most common multi-factor authentication on the web are:

- SMS-based verification codes. Application send a random number to the users phone. The users enters this number during the authentication. Weaknesses of this form of authentication:
  - phone represents both factors, and is a weak point. (If the users stores his passwords on the phone)
  - real-time phishing attacks still work.
  - if the attacker can take control of the phone number, he can receive the verification code. Ex. by tricking the phone company into transfering the number. Abuse the unreliable cellular protocols to intercept verification codes.

  This is no longer recommended to use.

- U2F (Universal Second Factor) security keys. Most popular is the usb security key. Register the key first with an account. Insert the key and touching the device. The device will use an embedded secret to sign the challenge proving that it is the same device as before. Touching the device indicates the presence of a user. It's harder to trick the device in signing the secret. The origin of the authentication is part of the signature and is therfor not suspectufull to phishing.

Outsourcing authentication to thrid parties, process is know as **social login**. Deligating authentication often depends on OAuth2 or OpenId Connect.
A third party provides you with the user's identity. With this the application can couple the session with an existing account within the application.

3.8 Server-side session management

Is the standard of keeping a state in a web application depends on storing information in a session object on the server. Such object has an unique identifier (**SID**) which is shared with the client. The server can tie multiple request to the same session.
Strategies for including the SID in every request:

- store the SID in a cookie. The browser automaticly attaches the cookie on every request. Results in a robust session management mechanism.
- include the SID in every URL in every page. This is less recommended. Higher level of complexity and higher risk of leaking the SID to third parties.

The information stored in a session object is crucial for making authorization decisions throughout the application.
The security of this session management mechanism depends on the secrecy of the session identifier. If the attacker obtains the user SID, he can take control of the user's session. Having control of the session, means having full control of the application in the name of the user.
Two weaknesses that can result in the disclosure of the SID:

- insecure generation of a new SID. If predictable in some way, an attacker can calculate past and future identifiers (**Brute-forcing the SID**). Applications that use a custom algorithm are often vulnerable. Instead use a secure random number generator.

- insecure transmission or storage of the SID. Can be stolen from HTTP requests. Another attack is **cross-site scripting**.

3.9 Securing session cookies

Session hijacking is the most critical threat against cookie-based session management.
Several attack and defences against session hijacking:

- eavesdropping on the network. Exposes the SID if the application uses HTTP. But making your application run entirely over HTTPS doesn't solve the problem. An attacker can manipulate a website so that it would send a HTTP request to the server, which will include the SID.
  The server needs to implement additional security properties. Mark the SID cookie as secure so that the browser will only send it over HTTPS. The flag stops network-based session hijacking attacks.
  The application uses HTTPS and Strict Transport Security, the attack would also not have been possible.

- stealing the SID with javascript. When an attacker can control a piece of javascript running on a page of the application, he can abuse this to access the cookies.
  Prevent javascript-based session hijacking by setting the "HttpOnly" flag. This prevents script-based access to a cookie.

3.10 Alternative session management mechanisms

What happens if the application is replicated over multiple servers?

- sticky sessions: where all request within a session go to the same server.

- sharing session state between servers

Other way, make use of client-side session management. Session state is stored on the client instead of the server. But how does this impact the security?
Server-side session objects are considered trusted. This assumption no longer holds when it's stored on the client-side. The server has to check the integrity of the object before using any of its data.
Another difference is the level of control over active sessions. Servers have a list of active sessions and can revoke specific sessions. When it's stored on the client this is harder to control or even impossible. Sessions only seen when the client makes a request.
Many application are moving away from cookies and are using custom headers to transport the session object. This complicates things for the developer.

3.11 Authorization throughout your application

Why is hard to get access control hard in a web application? Think about the assets that you have, the data you want to protect, what can the users do and then protect all the entry points. If you miss one, the whole application is at risk.
Follow certain rules: never trust the client. Don't do access control on the client. Do it for every request on the server side and only use server side trusted data. Client side access control is usefull for usability but not for security.
Access management: reason which of your users / clients should be able to access which data, and configure all of your application correctly to enforce your security policies.
Advice for access control for smaller applications:

- Do access control on the server
- Have basic web security. Proper session management, HTTPS Transport Security, ...
- Put access control in the code on the server side. Reuse framework who will help you do access control right.

Access control is beside authentication and authorization also **audit**. This is also refered as **triple A**. With audit you allow that actions proceed, but you log every action. Afterwards you check every action and roll back all the actions that are not allowed for that user. Additionaly you punnish the user for doing illegal actions. You can compare audit with breaking the glass in case of fire. You allow the actions, but afterwards you evaluate if it was needed.
Take away message:

- Don't underestimate the importance of access control for the security of your application.
- Think about which data you want to protect, what kind of users that you have.
- Don't start from scratch to implement this. Make use of coding frameworks that already offer the basics.

3.12 Intentional and unintentional requests

One of the most important aspects of authorization is deciding whether a request is legitimate. An example of an attack is **Cross-Site Request Forgery (CSRF)**.
The attacker puts malicious code on a webpage which contains a hidden form which will send a post request to an application. The way that browsers handle such request, it will automaticly attach the cookie to that request. The server cannot identify the difference between a legitimate request and a CSRF.
How can a server protect itself from a CSRF attack? The server attaches to the

form a hidden CSRF token for that user. When the server receives a request, it will check if the token belongs to that user.
The attacker can still send forms to the server, but the only way to obtain the user specific token is to extract it from the page of the victim's browser. Reading a page from a different origin is denied by the **Same Origin Policy (SOP)**.
Browsers can defend against this attack by setting the "SameSite" cookie flag. Cookies should only be used within the same site. This flag is for domains, not origins.
Two options for the SameSite cookie:

1. Strict: never sent across domains
2. Lax: present on top-level GET requests

3.13 Direct access to objects

What are direct object references and what makes them insecure? The application uses an ID to retrieve certain information. The attacker can modify that ID to retrieve all the other data from the application. This is a problem because some nodes can be marked as private, but can still be retrieved by having the right ID.
A possible solution would be to place proper authorization checks in place, e.g. check if the given node is public, if not check if the user who's requesting has the ownership of the node.
Another way to tackle the problem is the use of indirect object references. The server has map that relate the indirect objects with direct objects.

3.14 Recap and conclusion

To make a sensible authorization decision, a web application needs to know who the user is. This is the result from the authentication procedure. To keep track of this, we need session management.
Most important points:

- Making passwords more secure, to avoid enumeration and brute-force attacks
- Make use of Multi-factor authentication
- Cookie attributes and prefixes to make them more secure to use
- Avoid common authorization problems (insecure direct object reference, CSRF)

Most important lesson in web applications is preventing unauthorized access is about authentication, session management and authorization.

Extra reading:

- An overview of the biggest data breaches over time :
  http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks
- A detailed account of how Dropbox goes out of their way to store your passwords securely :
  https://blogs.dropbox.com/tech/2016/09/how-dropbox-securely-stores-your-passwords/
- A blog post about the advantages of the YubiKey :
  https://www.yubico.com/2015/11/why-yubikey-wins/
- Practial advice on addressing the top challenges with implementing U2F-based multi-factor authentication :
  https://www.yubico.com/2017/01/top-considerations-implementing-fido-u2f/
- The FIDO U2F security reference, which clearly gives an overview of the threats U2F counters :
  https://fidoalliance.org/specs/fido-security-ref-ps-20150514.pdf
- The full article on Insecure Direct Object References :
  http://www.securitee.org/files/toddlerhack_hackin9.pdf
- A simplified yet practical explanation of OAuth 2.0 :
  https://aaronparecki.com/2012/07/29/2/oauth2-simplified
- A deep dive into OAuth and OpenID Connect :
  http://nordicapis.com/api-security-oauth-openid-connect-depth/

4 Securely handling untrusted data

4.1 The problem with untrusted data

What is untrusted data in web applications? User input, thisis the most apparent source of untrusted data. Handling this kind of data unsecurly, can introduce vulnerabilities such as sequal injection, command injection and cross-site scripting.
User data not the only kind of data that is untrusted. All dynamic data is untrusted in one way or another.
Server-side injection: sequal injection and command injection. Client-side injection: cross-site scriping, injecting plain html. Client injection is harder to defend against.
Advanced client-side defences such as isolating dangerous content and impose additional restrictions (**Content Security Policy (CSP)**).

4.2 The root cause of injection attacks

The lack of context leads to injection vulnerabilities:

- SQL injection, the database can't distinguish between code and data.
- Command injection
- Code injection
- LDAP injection
- Xpath injection

4.3 A decade of mitigating injection vulnerabilities

What can go wrong with SQL injection? The attacker controls the database. He cannot only get the information, but also delete, alter, add information. The cost for a code mistake can run up high, depending on which moment of the development cycle the mistake was injected and when the mistake has been noticed.
The main part there are injection possibilities, is that developpers do not follow the guidelines for not introducing such problems into the code. Main problem lies with young developers that copy-paste code into their application. They can bring in vulnerabilities while these were not present/needed in the other application.
How does a static analysis tool help to prevent SQL injection? Static analysis is look into the code but not executing it, and do some deep analysis on the source or binary code. The problem lies in that it find potentional errors, but does not fix them. Developpers are not always convinced that certain errors are important to fix. Therefor developpers need to know about security problems and their importance and how to fix them. Secondly tool support, don't want

the developer figure himself out how to write secure code. Because he can make a mistake and it takes a lot of effort.
Take a way message: for developers, be involved, make sure that the tools that they give you to write secure code, are actually tools that you want to use on a day to day basis. The tools have to be engaging tools and do the stuff that they are made for and get your job done faster.

4.4 Command injection vulnerabilities

The attacker can add commands to an URL, the shell that will execute the command, has no context information to distinguish between data and code.
A potentional command injection vulnerability exisits every time untrusted data ends in an external command. Untrusted data can come from:

- users
- cookies
- HTTP headers

4.5 Preventing command injection

1. Strict input validation. Domain names have a well known structure and a limited set of characters. Reject all input that does not comply, we reduce already the risk of command injection.
2. Encode dangerous characters. This is an emplisit mechanism to ensure that there is no confusion between data and code.
3. Use safe API's. More explicit than encoding, and specify command and parameters separately. This preserves context information until execution.

4.6 SQL injection

Modify the SQL code that is executed by the database. Examples are:

- Data extraction: getting hand on sensitive user information
- Data modification: would you be able to detect changes?
- Data destruction

One of the most dangerous problems in web applications, because of the consequences and prevalence of SQL injection. There are several ways to influent the structure of a query.

- Seperate different queries with semicolons
- Inject a **union** statement to combine both queries in one data set. This attack is well suited to steal data.

- Insertion of a boolean clause. Disable filtering by appending a boolean clause that is always true. Well suited for leaking information and by passing authorization checks.
- Insert a comment symbol (–). Everthing after that symbol is ignored. Additional constraints are not checked up against.

4.7 Preventing SQL injection

Has the same problems as command injection.

1. Input validation. Enforce strict constraint where possible. But this is not enough (e.g. No use of special characters will cause that people from France, Ireland can't register)
2. Use prepared statements with variable binding. With data binding, the database can distinguish between code from data. This doesn't work for table and column names.
3. Use the untrusted data to select a value from a whitelist of trusted values for setting column/table names.
4. Encode special characters to make them harmless. But this is hard to get right, because every database system has a particalur set of characters. Encoding should only be used as a last resort.

Untrusted data can come from:

- Users
- Cookies
- HTML headers

4.8 Traditional XSS attacks

Cross-Site Scripting (XSS) has two common attack scenarios.

1. Reflected XSS. Send the payload to the server as part of the requested data. The server incorporates the payload into the HTML page of the response. The browsers sees the data and mistakes it for code.
2. Stored XSS. Inject the payload into the applications database. When the user requests a page that uses this data, the browser will execute the payload.

What are the consequences of a XSS attack? Allows the attacker to get a foodhold in the applications browsing context (e.g. Popping up an alert dialog)

- Defacement of a page (used my politically motivated hackers).

- Stealing sensitive information stored in cookies or browser storage. Can escalate into a session hijacking attack.
- Attacker has the full power of JavaScript to inject a more elaborate payload (e.g. key loggers, network scanners, social engineering attacks).

4.9 Common defenses against XSS attacks

1. Strict input validation. Prevent dangerous characters or strings like:
   - <
   - >
   - <script>
2. Output encoding. Encode dangerous characters to their harmless counterpart. But this doesn't always work. If the injection will happen directly in the CSS, JavaScript, HTML attribute.
3. Context-sensitive output encoding. Dangerous characters are still encoded, but the difference is that the context determines which characters are dangerous.

What if the user provided data is rich text data for example for an HTML editor. This can contain legitimate but harmless HTML tags. Context-sensitive output encoding will break functionality. Therefor we can use **sanitization**. It parses the input and analyzes its contents. It removes potential dangerous parts, but keeps the rest intact.

4.10 DOM-based XSS attacks

XSS is not only a server-side problem, JavaScript can also modify the contents of a page. JavaScript code needs also the proper context information. Client-side XSS attacks are know as DOM-based XSS attacks.
Inject script code by addapting the URL, which uses a part of the URL to insert it into the page. (e.g. The script code uses the identifier in a fragment of the URL to link different components. This fragment is intended for client-side only and is not sent to the server as part of the URL.). Server side defences are useless against DOM-based XSS attacks.
How can we eliminate these vulnerabilities?

- User proper DOM APIs. Offer safe functions to create and insert elements. They offer a way to put data inside an element without having a confusing between data and code.
- Context-sensitive encoding of untrusted data
- Client-side sanitization libraries. Best sanitization library at this moment is **DOMPurify**.

4.11 Alternative injection attack vectors

Other types of content that can be injected:

- HTML
- CSS
- SVG
- Flash

Consequences of HTML injections attacks:

- Data exfiltration: use HTML tags to gain hidden information from the webpage. It can send hidden data (security tokens, personal information) elements of the website to an external website linked in the HTML tag.
- Hijack relative URLs: inject a base tag, so that you can control all the relative links on the webpage.
- Form functionality: injecting hidden input fields to overwrite legitimate form values. Or overwrite form tags or buttons, the attacker can redirect an entire form.

How to prevent HTML injection:

- Avoid confusion between data and code
- **Context-sensitive output encoding**
- Using a sanitizer is tricky, since the content seems harmless

What can an attacker do with CSS code?

- Defacement of a HTML page.
- Executing of JavaScript code from the CSS context. (Not supported anymore in any browser)
- CSS selectors. Probe the contents of individual HTML elements or attributes.

4.12 HTML5 Sandboxing

Reduce the inpact of injection attacks is to force context isolation. The Same-Origin Policy can be used to isolate the main application from untrusted parts. Load a component from a different origin, if that component contains malicious code, it won't have an impact on the main application context.
To make this work, the application needs to be split in different components so that they can be deployed in different origins. But the malicious code still runs unrestricted in its behavior.

With the HTML5 sandbox attribute, allows you to impose additional constraints to an iframe (disbale executing scripts, prevent submissions of forms). It isolates untrusted content.
The sandbox attribute provides the following aspects:

- Content within has a unique origin
- Scripts are not executed
- Forms can not sumbitted
- Navigation of external context is not allowed
- Popups are not allowed
- Plugin content (Java, Flash) is not executed
- ...

This increases the security but limit the usability of the sandboxing mechanism. But these restrictions can be modified by adding extra directives to the value of the attribute.
The real power lies in to isolate content in an unique origin. Gives the benefit of origin based isolation without the hasle of using different origins.
Combining both the "allow-sameorigin" and "allow-scripts" enable a bypass attack on the sandbox. Allow-scripts permits to run arbitrary script code and the allow-sameorigin allows permission to the application's browsing context. General rule, use one or another, but never both together.

4.13 Content Security Policy

CSP is better suited for isolating specific blocks of content. A CSP policy defines the intended behavior of a page and prevents actions that violate these intentions. Specify for each resource from where they can originate.

- Stopping XSS attacks: by default CSP does not execute inline script locks. Remote script files are only loaded if they are whitelisted.
- Stopping CSS attacks: prevents inline styles from being executed.
- Static content: browser checks a whitelist for images, fonts, media elements ...

If a directive is not specified, it will take the value from the default directive.
Enable the report directive to see if you misconfigured some directives, or to spot attacks against your application.
CSP has two modes:

1. blocking
2. report only: a violation does not result in the blocking of the resource, the browser loads the resource and sends a report of the detected violation.

4.14 Recap and conclusion

Key to success is developer training and proper tool support. Most attacks make use of lack of context information where the application can't distinguish between code and data. Therefor we can use proper context-sensitive output encoding and sanitization. Furthermore browsers provide additional security measures like CSP and sandboxing.
Always think about the source and destination of data in your application. Data can only be handled securely if the proper context information is available.

Extra reading:

- A post discussing command injection in Java and .NET:
  `https://www.denimgroup.com/resources/blog/2009/05/command-injection-in-java-80-proven-`
- A very clear overview of XSS attacks and defenses, ideal to sharpen you understanding of XSS:
  `https://excess-xss.com/`
- A paper describing various attacks that do not need JavaScript at all: Scriptless Attacks - Stealing the Pie Without Touching the Sill:
  `https://www.nds.rub.de/media/emma/veroeffentlichungen/2012/08/16/scriptlessAttacks-ccs2012.pdf`
- All kinds of content injection attacks, without depending on the execution of JavaScript:
  `http://lcamtuf.coredump.cx/postxss/`
- Google's CSP Evaluator tool to check the security of you CSP policy:
  `https://csp-evaluator.withgoogle.com/`
- A blog post series on deploying CSP at Dropbox:
  `https://blogs.dropbox.com/tech/tag/content-security-policy/`
- GitHub's well-documented CSP journey:
  `https://githubengineering.com/githubs-csp-journey/`
- Strict-dynamic, presented by Google engineers at AppSec 2016:
  `https://www.youtube.com/watch?v=uf12a-0AluI`

5 Conclusion

- The Same-Origin Policy is the basis for security. Is the foundation for other countermeasures (CSRF, sandboxing)
- Every application should use HTTPS with Strict Transport Security
- Authentication, Session management and Authorization
- Consider all data to be untrusted