Following wide community consultation with both academia and industry, 19 Knowledge Areas (KAs) have been identified to form the scope of the CyBOK (see diagram below). The Scope document provides an overview of these top-level KAs and the sub-topics that should be covered under each and can be found on the project website: https://www.cybok.org/.

We are seeking comments within the scope of the individual KA; readers should note that important related subjects such as risk or human factors have their own knowledge areas.

It should be noted that a fully-collated CyBOK document which includes issue 1.0 of all 19 Knowledge Areas is anticipated to be released in October 2019. This will likely include updated page layout and formatting of the individual Knowledge Areas.
Web and Mobile Security

Sascha Fahl

September 2019

INTRODUCTION

The purpose of this Knowledge Area is to provide an overview of security mechanisms, attacks and defences in modern web and mobile ecosystems. This overview is intended for use in academic courses in web and mobile security, and to guide industry professionals who are interested in an overview of web and mobile security.

Web and mobile security have a long history, and their impact on overall information security is tremendous due to the sheer prevalence of web and mobile applications. Covering both web and mobile security, this Knowledge Area emphasises the intersection of their security mechanisms, vulnerabilities and mitigations. A more detailed description of what is covered by this Knowledge Area and what is not will be given below.

In the 1990s, web and mobile security had a strong focus on server-side and infrastructure security. Web browsers were mostly used to render and display static websites without dynamic content. At those times, security mostly affected server-side code (cf. Software Security Knowledge Area) and components. The focus on the server-side prevailed even with the rise of early scripting languages such as Perl and PHP. However, web content became more dynamic, and server-side security had to address injection attacks. Similar to the web, early mobile devices had somewhat limited functionality and were mostly used to make calls or send SMS. Mobile security back then focused on access control, calls and SMS security.

Modern web and mobile platforms led to notable changes. A significant amount of web application code is no longer executed on the server-side but runs in the browser. Web browser support for Java, Adobe Flash, JavaScript and browser plugins and extensions brought many new features to the client, which prompted a drastic change of the attack surface on the web. New types of attacks came up (e.g., Cross-Site Scripting Attacks) and Adobe Flash is known for being an attractive target for attackers. Hence, Google Chrome recently disabled the Adobe Flash plugin by default [1]. Similar to the development of web browsers, mobile devices became smarter and more feature-rich. Smartphones and tablets are equipped with all kinds of sensors, including motion, GPS and cameras. They have extensive compute power, storage capacity and are connected to the Internet 24-7. Modern Android and iOS devices run full-blown operating systems (cf. Operating Systems and Virtualisation Security Knowledge Area) and mobile applications have access to all the devices’ resources and sensors, and process highly sensitive user information. The rise of powerful and feature-rich clients makes them promising and attractive targets for attackers, just as was the case with servers and infrastructure in earlier days.

Modern web and mobile ecosystems are the primary drivers for the rise of appification and the "there is an app for everything" motto sums up many of the technological and security developments in recent years. Web and mobile users are connected to the Internet 24-7 and interact with a variety of websites, mobile applications and mobile web applications. The appification trend resulted in millions of apps ranging from simple flashlight apps to online social network apps, from online banking apps to mobile and browser-based games. Appification also sparked the merging of technologies and
security mechanisms used in web and mobile applications. Both web and mobile ecosystems are client-server oriented. Web browsers and mobile applications communicate with backend services to exchange information. The most prevalent technologies are web-focused. Client-server communication is mostly based on the Hypertext Transfer Protocol (HTTP) and its secure extension HTTPS. Both web-browsers and mobile applications tend to primarily exchange Hypertext Markup Language (HTML) documents and make extensive use of the JavaScript programming language, both on the server- and client-side. Webification describes the conversion to web technologies.

The sheer amount of applications in modern web and mobile ecosystems also impacted the software distribution model, which moved away from decentralised channels such as website downloads to centralised models. Centralised application stores allow developers to publish, advertise and distribute their software, while users can download new apps and app updates.

This Knowledge Area focuses on the appification trend and an introduction to the core technologies of the webification phenomenon. Figure 1 provides an overview of the involved entities and their interaction.

![Figure 1: Web and Mobile Ecosystem](image)

After introducing core technologies and concepts, we describe important security mechanisms and illustrate how they differ from non-web and non-mobile ecosystems. Software and content isolation are crucial security mechanisms and aim to protect apps and websites from malicious access. While isolation is understood in relation to traditional operating systems (cf. Operating System & Virtualisation Security Knowledge Area), differences for web and mobile platforms will be outlined.

Modern web and mobile platforms introduced new forms of access control (cf. AAA KA) based on permission dialogs. This Knowledge Area gives a brief introduction to access control, discussing web and mobile characteristics. As mentioned, web and mobile applications make extensive use of the HTTP and HTTPS protocols. Hence, we will discuss the web public key infrastructure (PKI) and HTTPS and extend the transport layer security (TLS) section in the Network Security Knowledge Area. While authentication is generally part of the AAA Knowledge Area, this Knowledge Area covers web and mobile-specific authentication aspects. In particular, we will address passwords as the most common authentication mechanism on the web and for mobile devices.

Finally, we address frequent software updates as a crucial security measure. While software up-
dates are equally important for traditional computer systems, the centralisation of web and mobile ecosystems, introduces new challenges and opportunities.

The following sections focus on web and mobile-specific client and server-side security aspects. However, we will focus on web and mobile-specific characteristics and will not address common software vulnerabilities (cf. Software Security Knowledge Area) and operating system security (cf. Operating System & Virtualisation Security Knowledge Area) in general. Section 2 first covers phishing and clickjacking attacks and defenses. Both affect web and mobile clients and exploit human difficulties in correctly parsing URLs and HTML documents. Since feature-rich web and mobile clients store much sensitive data, we will discuss client-side storage security issues and mitigations. Finally, Section 2 discusses physical attacks on mobile clients, including smudge attacks and shoulder surfing. While they are unique to mobile devices, these attacks also affect mobile web applications and mobile web browsers. After discussing the client-side specific aspects of web and mobile security, Section 3 addresses server-side challenges, starting with an overview of frequent injection attacks that affect web and mobile ecosystems. We discuss SQL and command injection attacks that allow malicious users to manipulate database queries to storage backends of web applications and commands that are executed. In addition to manipulated SQL queries and command parameters, malicious users can upload specially crafted files to backend servers through browsers and mobile applications. The server-side security section closes with a discussion of Cross-Site Scripting and Cross-Site Request Forgery attacks and common server-side misconfigurations that might lead to vulnerable service backends.

Overall, the discussion of client- and server-side security challenges aim to serve as the underlining of the natural split between entities in web and mobile ecosystems. Additionally, the chosen aspects illustrate the difference between the web and mobile world from other ecosystems.

Due to its focus on the intersection of both web and mobile security, this Knowledge Area does not cover aspects that are unique for either web or mobile and therefore omits mobile device security, mobile network (i.e., 2G/3G/4G/5G) security (see Physical Layer and Telecommunications Security Knowledge Area), and mobile malware. For some details and background information on these topics, the reader is referred to the Hardware Security, Malware and Network Security Knowledge Area. We also do not address side-channel attacks against mobile devices, web servers, and clients. However, the concept of and examples for side-channel security is covered in the Hardware Security Knowledge Area.

CONTENT

1 Web and Mobile Security

This section describes fundamental mechanisms of modern web and mobile platforms that affect security. The information in this section is intended to serve as a foundation to better assess the security challenges in the following sections. Similar to other software products and computer systems, mobile operating systems and applications and web browsers as well as web servers may contain exploitable bugs. General purpose software vulnerabilities are discussed in the Software Security Knowledge Area.

1There are only a limited number of widely used web browsers and application stores.
1.1 Appification

Over the last couple of years, the rise of mobile devices and ubiquitous Internet access has changed the way software is produced, distributed and consumed, altering how humans interact with computer devices and software installed on the devices. While regular Internet browsers have been the dominant way of accessing content on the web in the pre-mobile era, the concept of appification significantly changed the way users access content online [7]. Appification describes the phenomenon of moving away from a web-based platform to access most digital tools and media online with a web-browser through mobile applications with highly specialised, tiny feature sets. As mobile devices grew to become the primary interface for web access worldwide [2], the number of apps rose enormously over the last decade. “There is an app for everything” became the mantra of appified software ecosystems and produced multiple applications for all sorts of use cases and areas of application. Applications range from flashlights to online banking and social networking to messaging. There is almost no use case that is not addressed in an app. Many apps look like native local desktop or mobile applications. However, they are often (mobile) web applications (e.g., the Facebook or Spotify apps) that communicate with back end services and outsource computation and storage tasks to client. The shift towards appification had a significant impact on web and mobile security. In the pre-appification era, web browsers were primarily used to serve and render mostly static HTML websites. Hence, security efforts focused on web servers and applications in the backend. However, modern browsers and mobile devices developed complex and feature-rich environments in appified ecosystems, creating more security challenges on the client-side. The rise of appification also impacted the developer landscape. In the pre-appification era, software development was mostly dominated by experienced developers. Due to the more extensive tool and framework support the market entrance barrier is lower in appified ecosystems, which attracts more and sometimes unprofessional and inexperienced developers. This has negative consequences for web and mobile security in general (cf. Human Factors KA).

The Rise of the Citizen Developer

The application trend attracts many non-professional software developers called citizen developers. Many of them do not have a software engineering education but make use of the high number of simple APIs and tools available to build apps for different platforms. Oltrogge et al. [37] found that the adoption of easy-to-use online application generators (OAG) to develop, distribute and maintain apps has a negative impact on application security. Generated apps tend to be vulnerable to reconfiguration and code injection attacks and rely on an insecure infrastructure.

1.2 Webification

The rise of modern web and mobile platforms brought up another phenomenon. Many of the applications are not native applications written in compiled programming languages such as Java or Kotlin and C/C++ (e.g. for Android apps) or Objective-C and Swift (e.g. for iOS apps). Instead, they are based on web technologies including server-side Python, Ruby, Java or JavaScript scripts and client-side JavaScript. In addition to conventional web applications targeting regular web browsers, mobile web applications are more frequently built using these web technologies. In particular, mobile web application make heavy use of the JavaScript language.

This section gives a brief introduction to the most essential technologies including Uniform Resource Locators (URLs), the Hypertext Transfer Protocol (HTTP), the Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and the JavaScript programming language. For more detailed information, we suggest reading [38].
Uniform Resource Locators. Uniform Resource Locators (URLs) [8] are a core concept in the web. A URL is a well-formed and fully qualified text string that addresses and identifies a resource on a server. Address bars in modern browser UIs use the URLs to illustrate the remote address of a rendered document. A fully qualified absolute URL string consists of several segments and contains all required information to access a particular resource. The syntax of an absolute URL is: `scheme://credentials@host:port/resourcepath?query_parameters#fragments`. Each segment has a particular meaning (cf. Table 1).

<table>
<thead>
<tr>
<th>Segment</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme:</td>
<td>○</td>
<td>Indicates the protocol a web client should use to retrieve a resource. Common protocols in the web are http: and https:</td>
</tr>
<tr>
<td>//</td>
<td>○</td>
<td>Indicates a hierarchical URL as required by[8]</td>
</tr>
<tr>
<td>credentials@</td>
<td>●</td>
<td>Can contain a username and password that might be needed to retrieve a resource from a remote server.</td>
</tr>
<tr>
<td>host</td>
<td>○</td>
<td>Specifies a case-insensitive DNS name (e.g. cybok.org), a raw IPv4 (e.g. 127.0.0.1) or IPv6 address (e.g. [0:0:0:0:0:0:0:1]) to indicate the location of the server hosting a resource.</td>
</tr>
<tr>
<td>:port</td>
<td>●</td>
<td>Describes a non-default network port to connect to a remote server. Default ports for HTTP are 80 and for HTTPS 443.</td>
</tr>
<tr>
<td>/resourcepath</td>
<td>○</td>
<td>Identifies the resource address on a remote server. The resource path format is built on top of Unix directory semantics.</td>
</tr>
<tr>
<td>?query_parameters</td>
<td>●</td>
<td>Passes non-hierarchical parameters to a remote server, such as server-side script input parameters.</td>
</tr>
<tr>
<td>#fragment</td>
<td>●</td>
<td>Provides instructions for the browser. In practice, it is used to address an HTML anchor element for in-document navigation.</td>
</tr>
</tbody>
</table>

Table 1: URL segments.

Hypertext Transfer Protocol. The Hypertext Transfer Protocol (HTTP) is the most widely used mechanism to exchange documents between servers and clients on the web. While HTTP is mostly used to transfer HTML documents, it can be used for any data. Although HTTP/2.0 [3] is the newest protocol revision, the most widely supported protocol version is HTTP/1.1 [4]. HTTP is a text-based protocol using TCP/IP. An HTTP client initiates a session by sending an HTTP request to an HTTP server. The server returns an HTTP response with the requested file.

The first line of a client request includes HTTP version information (e.g. HTTP/1.1). The remaining request header consists of zero or more name: value pairs. The pairs are separated by a new line. Common request headers are User-Agent – these include browser information, Host – the URL hostname, Accept – and carries all supported document types, Content-Length – the length of the entire request and Cookie – see Section [1.8]. The request header is terminated with a single empty line. HTTP clients may pass any additional payload to the server. Although the payload can be of any type, clients commonly send HTML payload to the server, e.g. to submit form data. The HTTP server responds to the request with a response header followed by the requested payload. The response header contains the supported protocol version, a numerical status code, and an optional, human-readable status message. The status notification is used to indicate request success (e.g. status 200), error conditions (e.g. status 404 or 500) or other exceptional events. Response headers might also contain Cookie headers – cf. Section [1.8]. Additional response header lines are optional. The header ends with a single empty line followed by the actual content of the requested resource. Similar to the request payload, the content may be of any type but is often an HTML document.
Although **cookies** are not part of the original HTTP RFC [4], they are one of the most important protocol extensions. Cookies allow remote servers to store multiple `name=value` pairs in the client storage. Servers can set cookies by sending a `Set-Cookie: name=value` response header and consume them by reading a client’s `Cookie: name=value` request header. Cookies are a popular mechanism to maintain sessions between clients and servers and to authenticate users.

HTTP is request-response based and neatly fits unidirectional data transfer use cases. However, for better latency and more effective use of bandwidth, bidirectional network connections are needed. Bidirectional connections not only allow clients to pull data from the server, but also the server to push data to the client at any time. Therefore, the **WebSocket** protocol [39] provides a mechanism on top of HTTP. WebSocket connections start with a regular HTTP request that includes an `Upgrade: WebSocket` header. After the WebSocket handshake is completed both parties can send data at any time without having to run a new handshake.

**Hypertext Markup Language.** The Hypertext Markup Language (HTML) [9] is the most widely used method to produce and consume documents on the web. The most recent version is HTML5. The HTML syntax is fairly straightforward: A hierarchical tree structure of tags, `name=value` tag parameters and text nodes form an HTML document. The Document Object Model (DOM) defines the logical structure of an HTML document and rules how it is accessed and manipulated. However, competing web browser vendors introduced all sorts of custom features and modified the HTML language to their wishes. The many different and divergent browser implementations resulted in only a small portion of the websites on the Internet adhering to the HTML standard’s syntax. Hence, implementations of HTML parsing modes and error recovery vary greatly between different browsers.

The HTML syntax comes with some constraints on what may be included in a parameter value or inside a text node. Some characters (e.g., angle brackets, single and double quotes and ampersands) build the blocks of the HTML markup. Whenever they are used for a different purpose, such as parts of substrings of a text, they need to be escaped. To avoid undesirable side effects, HTML provides an entity encoding scheme. However, the failure to properly apply the encoding to reserved characters when displaying user-controlled information may lead to severe web security flaws such as cross-site scripting (cf. Section 3).

**Cascading Style Sheets.** Cascading Style Sheets (CSS) [6] are a consistent and flexible mechanism to manipulate the appearance of HTML documents. The primary goal of CSS was to provide a straightforward and simple text-based description language to supersede the many vendor-specific HTML tag parameters that lead to many inconsistencies. However, similar to divergent HTML parsing implementations, different browsers also implement different CSS parsing behavior. CSS allows HTML tags to be scaled, positioned or decorated without being limited by the original HTML markup constraints. Similar to HTML tag values, values inside CSS can be user-controlled or provided externally, which makes CSS crucial for web security.

**JavaScript.** JavaScript [10] is a simple yet powerful object-oriented programming language for the web. It runs both client-side in web browsers and server-side as part of web applications. The language is meant to be interpreted at runtime and has a C-inspired syntax. JavaScript supports a classless object model, provides automatic garbage collection and weak and dynamic typing. Client-side JavaScript does not support I/O mechanisms out of the box. Instead, some limited predefined interfaces are provided by native code inside the browser. Server-side JavaScript (e.g., Node.js [5]) supports a wide variety of I/O mechanisms, e.g., network and file access. The following discussion will focus on client JavaScript in web browsers. Every HTML document in a browser is given its JavaScript execution context. All scripts in a document context share the same sandbox (cf. Section 1.4). Inter-context communication between scripts is supported by browser-specific APIs. However, execution contexts are strictly isolated from each other in general. All JavaScript blocks in a context are executed individually and in a well-defined order. Script processing consists of three phases:
Pars**ing** validates the script syntax and translates it to an intermediate binary representation for performance reasons. The code has no effect until parsing is completed. Blocks with syntax errors are ignored, and the next block is parsed.

**Function Resolution** registers all named, global functions the parser found in a block. All registered functions can be reached from the following code.

**Execution** runs all code statements outside of function blocks. However, exceptions may still lead to execution failures.

While JavaScript is a very powerful and elegant scripting language for the web and mobile, it brings up new challenges and security issues such as Cross-Site Scripting vulnerabilities (cf. Section 3.1).

### 1.3 Application Stores

Application stores are centralised digital distribution platforms that organise the management and distribution of software in many web and mobile ecosystems. Famous examples are the Chrome web store for extensions for the Chrome browser, Apple’s AppStore for iOS applications, and Google Play for Android applications. Users can browse, download, rate and review mobile applications or browser plugins and extensions. Developers can upload their software to application stores that manage all of the software distribution challenges, including the provision of storage, bandwidth and parts of the advertisement and sales. Before publication, most application stores deploy application approval processes for testing reliability, adherence to store policies, and for security vetting [12] [13].

Most of the software available in ecosystems that have application stores are distributed through the stores. Only a few users side-load software (i.e. install software from other sources than the store). Application stores allow providers to control which applications are available in their stores, which leads to banning particular applications. In the past, some application stores were criticised for censorship activities. However, the deployment of security vetting techniques helped to significantly reduce the amount of malicious software available in stores [12] and to reduce the number of applications that suffer from vulnerabilities due to the misuse of security APIs by developers [14] [15]. Deployed security vetting techniques include static and dynamic analysis applied to application binaries and running instances of applications. In addition to security vetting techniques, application stores require applications to be signed by developer keys. Application signing does not rely on public key infrastructures known from the web. Instead, developers are encouraged to use self-signed certificates. Trust between application developers and their users is established due to trust on first use mechanisms. Developers are required to sign application updates with the same key to prevent malicious updates [17] [16]. Application stores not only allow developers and users centralised access to software publication, distribution and download, they also enable users to rate and review published applications. User rating and reviews are intended to help other users make more informed download decisions, but they also have a direct connection to application security.

### Impact of User Ratings and Reviews on Application Security

Nguyen et al. [40] conducted a large scale analysis of user reviews for Android applications and the reviews’ impact on security patches. They found that the presence of security and privacy related user reviews for applications are contributing factors to future security-related application updates.

### 1.4 Sandboxing

Both modern mobile and browser platforms make use of different sandboxing techniques to isolate applications and websites and their content from each other (cf. *Operating Systems and Virtualisation Knowledge Area*). The goal is to protect them from each other and the platform against malicious
applications and sites. Major web browsers and mobile platforms implement isolation on a process level. Each application or website runs in its process. By default, the isolated processes cannot interact with each other and cannot share resources. On mobile platforms, application isolation helps to protect apps against other malicious apps on the same device. In browsers, site isolation serves as a second line of defence as an extension to the same-origin-policy.

**Application Isolation.** Modern mobile platforms provide each application with their sandbox running in a dedicated process. Mobile platforms take advantage of underlying operating system process protection mechanisms for application resource identification and isolation. Application sandboxes run on the kernel-level and enforce security through standard operating system facilities, including user and group IDs. The sandboxes prevent applications from accessing each other by default and only provide limited access to operating system resources.

**Content Isolation.** Content isolation is one of the major security assurances in modern browsers. The main idea is to isolate documents based on their origins so that they cannot interfere with each other. The same-origin-policy (SOP) was introduced in 1995 and affects JavaScript and its interaction with a document's DOM, network requests and local storage (e.g., cookies). The core idea behind SOP is that two separate JavaScript execution contexts are only allowed to manipulate a document's DOM if there is an exact match between the document host and the protocol, DNS name and port numbers. Cross-origin manipulation requests are not allowed. Table 2 illustrates sample SOP validation results. Similar to JavaScript-DOM-interaction, the SOP limits the JavaScript XMLHttpRequest capabilities. XMLHttpRequests allow JavaScript scripts to issue HTTP requests to the origin of the host document.

<table>
<thead>
<tr>
<th>Originating document</th>
<th>Accessed document</th>
<th>Browser behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.cybok.org/">https://www.cybok.org/</a></td>
<td><a href="https://books.cybok.org/">https://books.cybok.org/</a></td>
<td>Host mismatch</td>
</tr>
<tr>
<td><a href="http://www.cybok.org/">http://www.cybok.org/</a></td>
<td><a href="https://www.cybok.org/">https://www.cybok.org/</a></td>
<td>Protocol mismatch</td>
</tr>
<tr>
<td><a href="https://www.cybok.org/">https://www.cybok.org/</a></td>
<td><a href="https://www.cybok.org:10443/">https://www.cybok.org:10443/</a></td>
<td>Port mismatch</td>
</tr>
</tbody>
</table>

Table 2: SOP validation examples.

One major flaw of SOP is that it relies on DNS instead of IP addresses. Attackers who can intentionally change the IP address of a DNS entry can circumvent SOP security guarantees.

Since code that enforces the same-origin-policy occasionally contains security bugs, modern browsers introduced a second line of defence. To offer better protection, websites are rendered in their own processes that run in a sandbox. Sandboxing websites is meant to prevent powerful attacks such as stealing cross-site cookies and saved passwords.

1.5 Permission Dialog Based Access Control

The primary purpose of permission systems in modern mobile and web platforms is to protect the privacy of their users by controlling access to resources. The control of access to resources on a traditional computer system requires the accurate definition of all involved security principals and the protected resources in the system. Finally, an access control system requires a non-bypassable and trusted validation mechanism for any access request (reference monitor) and sound security policies that cover all access requests. Based on the security policies, the reference monitor can decide whether it grants access or denies it.

Modern mobile and web platforms deviate from conventional computer systems in multiple ways:

---

2 The protocol, DNS name and port number triple is called origin.
The Security Principals. Traditional computer systems are primarily multi-user systems with human users and processes running on their behalf.

Modern mobile and web platforms extend conventional multi-user systems. The human user of the system is not the only one considered a security principal. Additionally, systems consider all involved developers that have their applications installed on the system as security principals.

The Reference Monitor. Typically, conventional computer systems implement reference monitoring as part of the operating system (OS), e.g., the file system and network stack. User-level processes can then extend this OS functionality and implement their access control mechanisms.

Like conventional computer systems, modern mobile and web platforms build on top of OS low-level access control mechanisms. Additionally, the extensive frameworks on top of which applications are developed and deployed, provide extended interfaces. Modern platforms use interprocess communication (IPC) for privilege separation and compartmentalisation. Access control mechanisms on calling processes are used to protect IPC interfaces.

The Security Policy. In conventional computer systems, a process can have different privilege levels: It can run as the superuser, as a system service, with user-level privileges or with guest privileges. All processes that share the same privilege level have the same set of permissions and can access the same resources in the system.

Modern mobile and web platforms make a clear distinction between system and third-party applications: Access to security- and privacy-critical resources is only granted to designated processes. In contrast, third-party applications have by default no access to critical resources. Instead, application developers may request permissions from a set commonly available to all third-party applications.

A more thorough discussion of access control mechanisms can be found in the Authentication, Authorisation and Access Control Knowledge Area.

Different Permission Approaches. Mobile and web platforms implement different permission approaches. First, platforms distinguish different privilege levels. A common distinction is two levels (e.g., as implemented on Android): normal (e.g., access to the Internet) and dangerous permissions (e.g., access to the camera or microphone). While application developers have to request both normal and dangerous permissions to grant their applications access to the respective resources, the levels differ for application users. Normal permissions are granted silently without any application user interaction. However, whenever applications require dangerous permissions, the underlying mobile or web platform presents users with permission dialogs. While earlier Android versions showed users a list of all the necessary permissions of an application at install time, modern mobile platforms and browsers present permission dialogs at runtime. A permission dialog usually is shown the first time an application requests access to the corresponding resource. Application users can then either grant the application access to the resource or deny access. Modern permission-based access control systems allow greater flexibility and control for both developers and users.

Permission Dialogs: Attention, Comprehension and Behaviour

While permission dialogs theoretically allow for greater flexibility and control, in practice they tend to have serious limitations. Porter Felt et al. found that Android applications developers tend to request more permissions for their applications than needed [19]. Hence, applications request access to more resources than strictly necessary, which violates the least-privilege principle. Similar to developers, end-users struggle with permission dialogs. Porter Felt et al. [20] found that they often do not correctly understand permission dialogs and ignore them due to habituation (cf. the Human Factors KA).
The web PKI and the HTTPS [21, 22] protocol play a central role in modern mobile and web platforms, both of which are based on client-server architectures. In the web, web servers or applications exchange information with browsers. On mobile platforms, apps exchange information with backend (web) servers. In both cases, the secure extension of the Hyper Text Transfer Protocol HTTPS is mostly used for secure network connections between clients and servers. To establish secure network connections, the web public key infrastructure is used. Using the web PKI and X.509 certificates, clients and servers can authenticate each other and exchange cryptographic key material for further encrypted information transport. This document will not provide further details on how the authentication process and the key exchange procedures work in detail (cf. Network Security KA). Rather, it gives an overview of specific aspects of web and mobile platforms.

HTTPS is the most widely deployed secure network protocol on the web and mobile. It overlays HTTP on top of the TLS protocol to provide authentication of the server and integrity, and confidentiality for data in transit. While HTTPS offers mutual authentication of servers and clients based on X.509 certificates, the primary use is the authentication of the accessed server. Similar to TLS, HTTPS protects HTTP traffic against eavesdropping and tampering by preventing man-in-the-middle attacks. Since HTTPS encapsulates HTTP traffic, it protects URLs, HTTP header information including cookies and HTTP payload against attackers. However, it does not encrypt the IP addresses and port numbers of clients and servers. While HTTPS can hide the information exchanged by clients and servers, it allows eavesdroppers to learn the top-level domains of the websites browsers that users visit, and to identify the backend servers that mobile apps communicate with.
Both web browsers and mobile apps authenticate HTTPS servers by verifying X.509 certificates signed by certificate authorities (CAs). Browsers and mobile apps come with a list of pre-installed certificate authorities. A pre-installed certificate authority list in modern browsers and on modern mobile platforms typically contains hundreds of CAs. To be a trusted HTTPS server certificate, it needs to be signed by one pre-installed CA.

Modern browsers present their users a warning message (e.g., see Figure 4) when the server certificate could not be validated. The warning messages are intended to indicate a man-in-the-middle attack. However, common reasons for warning messages are invalid certificates, certificates that were issued for a different hostname, network errors between the client and server and errors on the client such as misconfigured clocks \[23\]. In most cases, browser users can click-through a warning message and visit a website even if the server certificate could not be validated \[24\]. Browsers use colored indicators in the address bar to display the security information for a website. Websites loaded via HTTP, websites that load content over an HTTP connection \[5\] and sites that use an invalid certificate but for which the user clicked through a warning are displayed insecure. HTTPS websites with a valid certificate are displayed securely (e.g., see Figure 4). In contrast, mobile app users cannot easily verify whether an application uses the secure HTTPS protocol with a valid certificate. No visual security indicators similar to those used in browsers are available. Instead, they have to trust application developers to take all the necessary security measures for HTTPS connections.

As of 2019, most of the popular websites support HTTPS, and the majority of connections from clients to servers in the web and mobile applications use HTTPS to protect their users against man-in-the-middle attacks. To further increase the adoption of HTTPS, server operators are encouraged to use HTTPS for all connections and deploy HTTP Strict Transport Security (HSTS) \[26\]. Additionally, browser users can install extensions and plugins to rewrite insecure HTTP URLs to secure HTTPS URLs \[27\] if possible and mobile application frameworks make HTTPS the default network protocol for HTTP connections.

Using HTTPS does protect the payload against attackers but does not preserve metadata (e.g.,

---

4 See the Knowledge Area on Network Security for details on the validation process.
5 Called mixed content.
which websites a user visits). Please refer to the *Privacy and Online Rights Knowledge Area* for more information, including private browsing and the TOR network.

### Rogue Certificate Authorities and Certificate Transparency

The web PKI allows every trusted root certificate authority to issue certificates for any domain. While this allows website operators to freely choose a CA for their website, in the past CAs were discovered to have issued fraudulent certificates for malicious purposes. One of the most prominent examples is the DigiNotar CA in 2011 [41]. It issued fraudulent certificates for multiple websites including Google’s Gmail service. Nobody has been charged for the attack. However, DigiNotar went bankrupt in 2011. Certificate transparency [42] was introduced to fight fraudulent certificate issuance. Certificate transparency provides a tamper proof data structure and monitors all certificate issuance processes of participating CAs. While it cannot prevent fraudulent certificate issuance, it drastically reduces the time to their detection. Clients can verify the correct operation of the certificate transparency providers and should only connect to websites that use X.509 certificates that include a signed certificate timestamp. Certificate transparency is supported by most major certificate authorities and browser vendors.

### 1.7 Authentication

Authentication in the web and on mobile platforms is an important security mechanism designed to enable human users to assert their identity on web applications, mobile devices or mobile apps. This section will not give a detailed overview of concepts of authentication (cf. *AAA Knowledge Area*) but will focus on authentication mechanisms and technologies relevant for web and mobile platforms.

![Basic HTTP Authentication exchange](image)

**HTTP Authentication.** In the HTTP context, authentication generally refers to the concept of verifying the identity of a client to a server, e.g., by requiring the client to provide some pre-established secrets such as username and password with a request. This section highlights two widely used authentication methods on the web, the Basic HTTP authentication, and the ubiquitous HTTP Form-based HTTP authentication.

Basic HTTP authentication [28] is a mechanism whose results are used to enforce access control to resources. It does not rely on session identifiers or cookie data. Nor does the Basic HTTP authentication scheme require the setup of dedicated login pages, as all major browsers provide an integrated login form. A server can trigger this authentication form by sending a response header containing the “HTTP 401 Unauthorized” status code and a “WWW-Authenticate: Basic” field. Credentials entered into this form by the client are combined with a “:” ("Username:Password"), Base64 encoded for transit (“VXNlc3NhbWU6UGFzc3dvcnQK”), and added as Authorisation header to the next request.
Authorization: Basic VXNlcm5hbWU6UGFzc3dvcmQK). An example exchange between server and client is shown in Figure 5. The Basic authentication scheme is not secure, as the credentials are transmitted after a simple Base64 encoding, which is trivial to reverse. Hence, login credentials are transmitted in plain text across the network, which allows man-in-the-middle attacks. Therefore, Basic HTTP authentication should not be used without additional enhancements that grant confidential network traffic such as HTTPS.

Form-based HTTP authentication in which websites use a form to collect login credentials is a widely prevalent form of authentication in modern web and mobile applications. For this scheme, an unauthenticated client trying to access restricted content is shown an HTML-based web form that prompts for their credentials. The client then submits the entered credentials to the server (e.g., in a POST request). The server validates the form data and authenticates the client on successful validation. Similar to Basic authentication, Form-based authentication exposes user credentials in plain text if not protected by HTTPS.

**Mobile Device Authentication.** Mobile devices deploy a variety of authentication mechanisms to unlock devices, grant legit users access, and protect their data from illegitimate users. The most common mechanisms for mobile device unlocking are passwords, PINs, patterns and biometric features.

Users can use common alphanumeric passwords, including special characters. However, since mobile device authentication is a frequent task [25], many users tend to unlock their mobile device using numerical PINs. Android devices also support unlock patterns (see Figure 6). Instead of choosing a password or PIN, users can pick an unlock pattern from a 3x3 grid.

Modern mobile devices allow users to authenticate using biometric features, including fingerprint and facial recognition.

![Android Unlock Patterns](image)

**Android Unlock Patterns**

Similar to passwords (see Section 1.9) device unlock patterns suffer from multiple weaknesses. Uellenbeck et al. [43] conducted a study to investigate users’ choices of 3x3 unlock patterns. They found empirical evidence that users tend to choose biased patterns, e.g., users typically started in the upper left corner and selected three-point long straight lines. Hence, similar to regular passwords (cf. Human Factors Knowledge Area) the entropy of unlock patterns is rather low. In addition to users choosing weak unlock patterns, the mechanism is vulnerable to shoulder surfing attacks (see Section 2.3). As a countermeasure, De Luca et al. [44] propose to use the back of a device to authenticate users.

**1.8 Cookies**

Web servers can assign stateful information to clients by providing them HTTP cookies [29]. Cookie information (e.g., IDs of items added to the shopping cart in an online shop) is stored on the client.
Cookies allow clients and servers to include their unique session identifiers in each HTTP request-response, avoiding the need for repeated authentication. Session cookies expire when the session is closed (e.g., by the client closing the browser) but persistent cookies only expire after a specific time.

Cookie-based authentication allows clients to re-establish sessions every time they send requests to the server with a valid cookie. Cookie-based session management is vulnerable to the hijacking of session identifiers [45]. Hijackers who post valid session cookies can connect to the attacked server with authenticated privileges of the victim.

Cookies can also be used to track users across multiple sessions by providers. This behaviour is generally jeopardising user privacy (cf. Adversarial Behaviours KA and Privacy and Online Rights Knowledge Area).

1.9 Passwords and Alternatives

Passwords are the most widely deployed mechanism to let users authenticate to websites and mobile applications and protect their sensitive information against illegitimate access online. They are the dominant method for user authentication due to their low cost, deployability, convenience and good usability. However, the use of passwords for most online accounts harms account security [30]. Since humans tend to struggle memorising many different complicated passwords, they often choose weak passwords and re-use the same password for multiple accounts. Weak passwords can easily be guessed by attackers offline or online. Re-used passwords amplify the severity of all password attacks. One compromised online account results in all other accounts protected with the same password become vulnerable.

Online service providers deploy various countermeasures to address security issues with weak passwords and password re-use:

Password Policies. Password policies are rule sets to encourage users to choose stronger passwords. Some password policies also address the memorability issue. To support stronger passwords, most rules address password length and composition, blacklists and the validity period of a password [31, 32].

Password Strength Meters. Password strength meters (PSMs) pursue the same goal as password policies and aim to encourage the choice of stronger passwords (cf. Human Factors Knowledge Area). PSMs typically provide visual feedback or assign passwords scores to express password strength (see Figure 7) [33].

![Figure 7: A password strength meter](image)

However, addressing weak passwords and password re-use by deploying restrictive policies or PSMs only has a limited effect on overall password security [15]. Hence, service providers can use extensions for plain passwords to increase authentication security.

Multi-Factor Authentication. Instead of only requiring one factor (e.g., a password), multi-factor authentication systems require users to present multiple factors during the authentication process [48].
Example: Transaction on ATMs

A good non-web example is the use of ATMs: One can only carry out a transaction after giving two factors – a bank card and the corresponding PIN.

In the web, passwords are often complemented with a second factor for two-factor authentication (2FA). Most commonly, the second factor is mobile device-based. In addition to a password, users need to have their device on hand to receive a one-time token to authenticate successfully.

**OAuth.** While not being an authentication mechanism itself (cf. AAA KA), Open Authorisation (OAuth) can be used for privacy-friendly authentication and authorisation for users against third-party web applications. OAuth uses secure tokens instead of requiring users to provide login credentials such as usernames and passwords. Popular examples of OAuth providers are social network providers such as Google, Facebook and Twitter. On behalf of their users, they provide access tokens that authorise specific account information to be shared with third-party applications.

1.10 Frequent Software Updates

Frequent software updates are a fundamental security measure and particularly crucial for web and mobile platforms. This section discusses the different components in the web and mobile ecosystems that require regular updates, the different update strategies, and their pros and cons. Traditionally, browser and mobile device updates required their users to install updates whenever new versions were available manually. Users had to keep an eye on software updates and were responsible for downloading and installing new releases. This approach was error-prone and resulted in many outdated and insecure deployed software components.

Most of the critical components on modern web and mobile platforms have short release cycles. Web browsers, including Google Chrome and Mozilla Firefox, implement auto-update features and frequently push new versions and security patches to their users.

Similar to automatic browser updates, mobile platforms provide automatic application updates for third-party apps. While this approach generally results in quicker updates and the more timely distribution of security patches, automatic mobile application updates are only enabled by default for devices connected to a WiFi. Devices connected to a cellular network (e.g., 3G/4G) do not benefit from automatic application updates by default. This update behaviour leads to most third-party application updates being installed on mobile devices within a week. Automatic third-party application updates work well on mobile devices. Mobile operating system update behaviour heavily depends on the platform. In particular, many non-Google Android devices suffer from outdated and insecure operating system versions.

Overall, modern web and mobile platforms recognised the disadvantages of non-automatic software updates and now provide automatic or semi-automatic platforms or application updates in most cases.

Outdated Third Party Libraries

While frequent software updates are crucial in general, updates of third party libraries is a particularly important security measure for software developers who need to patch their own code and distribute updates, while also tracking vulnerabilities in libraries they use and updating them for better security. Derr et al. conducted a measurement study of third party library update frequencies in Android applications and found that a significant number of developers use outdated libraries, exposing their users to security issues in the affected third party libraries. Lauinger et al. conducted a similar study for JavaScript libraries in web applications and also found many websites that include outdated and vulnerable libraries.
2 Client Side Vulnerabilities and Mitigations

This section covers attacks and their countermeasures with a focus on the client-side. It discusses issues of both modern web browsers and mobile devices. The illustrated security issues highlight aspects that dominated the discussion of the last years. We focus on attacks that exploit weaknesses in the interaction process of web and mobile clients and their users. We then discuss challenges resulting from the trend of storing more and more information on the client-instead of server-side. Finally, we address physical attacks that do not focus on exploiting software or human vulnerabilities, but weak points that result from using mobile devices.

2.1 Phishing & Clickjacking

This section presents two prevalent issues that exploit user interface weaknesses of both web and mobile clients. Phishing and clickjacking rely on issues humans have with properly verifying URLs and the dynamic content of rendered HTML documents.

**Phishing.** Phishing attacks are fraudulent attacks that aim to steal sensitive information, including login credentials and credit card numbers from victim users [50]. Common types of phishing attacks use email, websites or mobile devices to deceive victims. Attackers disguise themselves as trustworthy parties and send fake emails, show fake websites or send fake SMS or instant messages. Fake websites may look authentic. Attackers can use successfully stolen login credentials or credit card numbers to impersonate victims and access important online accounts. Successful phishing attacks may result in identity theft or loss of money.

Attackers commonly forge websites that appear legit to trick users into believing they are interacting with the original website. To initiate a phishing attack, attackers plant manipulated links on users via Email, a website or any other electronic communication. The manipulated link leads to a forged website that appears to belong to the spoofed organisation behind the website in question. Attackers often spoof online social media, online banking or electronic payment provider websites. They trick victims into following manipulated links using misspelled URLs, subdomains or homograph attacks.

**Example: Phishing URL**

In the following example URL https://paymentorganization.secure.server.com, it appears that the URL points to the secure.server section of the paymentorganization website. However, in fact the link leads the paymentorganization.secure section of the server.com website.

To make forged websites look even more authentic, some phishers alter a browser’s address bar by replacing the original address bar with a picture of the legitimate URL or by replacing the original address bar with a new one. Address bar manipulation attacks require the use of JavaScript commands. Phishing attacks that involve manipulated URLs and address bars are even harder to detect in mobile browsers since the address bar is not visible during regular browsing. Website phishing is one of the most frequent attacks. For human users, it tends to be problematic to spot phishing URLs and websites [49].

Therefore, common countermeasures are anti-phishing training and public awareness campaigns [49] that try to sensitize users and teach them how to spot phishing URLs. Modern browsers deploy technical security measures, including blacklists and visual indicators that highlight the top-level domain of a URL.

**Clickjacking.** In a clickjacking attack, attackers manipulate the visual appearance of a website to trick users into clicking on a fake link, button, or image. Clickjacking is also known as a *user interface*
redress attack and belongs to the class of confused deputy attacks [51]. Using transparent or opaque layers over original websites, attackers fool their victims. While victims believe they click on the overlay element, the original website element is clicked on. Attackers can thus make their victims trigger arbitrary actions on the original website. The attack website uses an iFrame to load the target website and can make use of the absolute positioning features of iFrames for correct visual alignment. Thus, it is hard for victims to detect the attack elements over the original website. Clickjacking attacks are particularly dangerous when victims have already logged in to an online account or visit configuration or settings websites. In those cases, an attacker can trick the victim into performing actions on a trusted site when the victim is already logged in. One of the most prominent clickjacking attacks was one targeting the Adobe Flash plugin settings page [52]. Attackers used invisible iFrames to trick their victims into changing the plugin’s security settings and permitting the attackers to access the microphone and camera of their victims’ machines.

Clickjacking attacks can be used to launch other attacks against websites and their users, including Cross-Site Request Forgery and Cross-Site Scripting attacks (see Section 3.1) [51].

A clickjacking attack is not a programming mistake but a conceptual problem with JavaScript. Hence, detection and prevention are not trivial. Detecting and preventing clickjacking attacks can be done both server- and client-side. Web browser users can disable JavaScript and iFrames to prevent clickjacking attacks. However, since this would break many legitimate websites, different browser plugins (e.g., NoScript [53]) allow the controlled execution of JavaScript scripts on behalf of the user. In order to contain the impact of clickjacking attacks, users should log out of online accounts when leaving the website. In order to prevent clickjacking attacks on the server-side, website developers need to make sure that a website is not frame-able, i.e. a website does not load if it is inside an iFrame. Websites can include JavaScript code to detect whether a website has been put into an iFrame and break out of the iFrame. This defence technique is called FrameBusting [54]. However, since users might have disabled JavaScript, this method is not reliable. The recommended server-side defence mechanism is to set a proper HTTP response header. The X-FRAME-OPTIONS header can be set to DENY which will prevent a website being loaded inside an iFrame.

Clickjacking attacks affect both desktop and mobile web browsers.

<table>
<thead>
<tr>
<th>Phishing and Clickjacking on Mobile Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phishing and Clickjacking are not limited to browsers and the web. Mobile application users are susceptible for both attacks. Aonzo et al. [61] find that it is possible to trick users into an end-to-end phishing attack that allows attackers to gain full UI control by abusing Android’s Instant App feature and password managers to steal login credentials. Fratantonio et al. [62] describe the Cloak &amp; Dagger attack that allows a malicious application with only two permissions (cf. Section 1.5) to take control over the entire UI loop. The attack allows for advanced clickjacking, keylogging, stealthy fishing and silent phone unlocking.</td>
</tr>
</tbody>
</table>

### 2.2 Client Side Storage

Client-side storage refers to areas that a browser or operating system provides to websites or mobile applications to read and write information. The storage is local to the client and does not require server-side resources or an active Internet connection. At the same time, malicious users may manipulate stored information. Hence, client-side storage areas need to be protected from malicious access. This section describes common client-side storage areas and their protection mechanisms.

**Client Side Storage in the Browser.** Historically, client-side browser storage was only used to store cookie information (see Section 1.8). However, due to their simple design and limited capacity, cookies cannot be used to store large or complex amounts of information. With the rise of HTML5, more powerful and feature-rich alternatives for client-side storage in the browser came up. These
include WebStorage [55], which is similarly to cookies and allows to store key-value pairs, and IndexedDB [56], which serves as a database in the vein of noSQL databases and can be used to store documents, other files and binary blobs. JSON Web Tokens [57] is another secure storage mechanism for encrypted and signed JSON Objects. Servers can send JSON Web Tokens to the client and verify them for future use.

As mentioned, the primary security issue with client-side storage mechanisms is that malicious users can manipulate them. To guarantee integrity for sensitive information (e.g., session information), developers are advised to follow the JSON Web Tokens approach and cryptographically sign the data stored on the client and verify it upon retrieval.

In addition to information integrity, a second important aspect of the WebStorage and IndexedDB storage is that stored information is not automatically cleared after users leave a website. To store information in a session-like fashion, web application developers are advised to rely on the sessionStorage object of the WebStorage API [58].

**Client Side Storage in Mobile Applications.** In mobile applications, handling client-side storage security also depends on the type of information and storage mechanism, e.g., private storage of an application or public storage such as an SD card. Most importantly, data should be digitally signed and verified (cf. Cryptography Knowledge Area) for both browser and mobile client storage purposes. It is recommended that developers sign sensitive information and apply proper user input sanitisation. This is particularly relevant for shared storage such as SD-cards that do not use secure access control mechanisms. Instead, proper access administration mechanisms are provided for storage areas that are private to an application.

<table>
<thead>
<tr>
<th>Sensitive Information Leaks in Android Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enck et al. [63] investigated the security of 1,100 popular Android applications. Amongst other things, they found that a significant number of apps leaked sensitive user information to publicly readable storage locations such as log files and the SD card.</td>
</tr>
</tbody>
</table>

**2.3 Physical Attacks**

Instead of attacking web or mobile applications’ code, physical attacks aim to exploit bugs and weak points that result from using a device or client. We focus on two representative examples below.

**Smudge attacks.** In a smudge attack, an attacker tries to learn passwords, PINs or unlock patterns entered on a touchscreen device mostly affecting mobile devices. The main problem with entering sensitive unlock information through a touchscreen is the leak of a considerable amount of unlocking information through the oily smudges that users’ fingers leave behind when unlocking a device. Using amateur cameras and image processing software, an attacker can recover the grease trails and infer unlock patterns, passwords, and pins [59]. To run a smudge attack, an attacker needs a clear view of the target display.

**Shoulder Surfing.** Shoulder surfing is a physical attack where an attacker tries to obtain sensitive information such as passwords, PINs, unlock patterns, or credit card numbers [60]. For a shoulder surfing attack, an attacker needs a clear view of the target display. The attacker can mount a shoulder surfing attack either directly by looking over the victim’s shoulder or from a longer range by using dedicated tools such as cameras or telescopes. Shoulder surfing attacks are particularly dangerous for mobile device users during device unlock or web users in public spaces such as trains, railways, and airports.
3 Server Side Vulnerabilities and Mitigations

This section discusses aspects of server-side security. It provides details for common aspects of server security, including well-known vulnerabilities and mitigations. The section discusses root causes, illustrates examples, and explains mitigations. The aspects discussed below are central for the web and mobile and dominated much of the security discussion in this area in the past.

3.1 Injection Vulnerabilities

Injection attacks occur whenever applications suffer from insufficient user input validation so that attackers can insert code into the control flow of the application (cf. Software Security Knowledge Area). Prevalent injection vulnerabilities for web and mobile applications are SQL and Shell injections [38]. Due to inadequate sanitisation of user input, requests to a database or shell commands can be manipulated by an attacker. Such attacks can leak or modify information stored in the database or issue commands on a system in ways developers or operators have not intended. The main goal of injection attacks is to circumvent authentication and expose sensitive information such as login credentials, personally identifiable information, or valuable intellectual property of enterprises.

Injections vulnerabilities can be addressed by adequately sanitising attacker-controlled information and deploying proper access control policies. The goal of input sanitisation is to filter invalid and dangerous input. Additionally, strict access control policies can be implemented to prevent injected code from accessing or manipulating information [64].

**SQL-Injection.** SQL-injection attacks refer to code injections into database queries issued to relational databases using the Structured Query Language (SQL). Many web and mobile applications allow users to enter information through forms or URL parameters. SQL injections occur if such user input is not filtered correctly for escape characters and then used to build SQL statements. Enabling attackers to modify SQL statements can result in malicious access or manipulation of information stored in the database.

**Example: SQL Injection attack**

The statement below illustrates the vulnerability.

```java
vuln_statement = " 'SELECT * FROM creditcards WHERE number = ' + user_input + " ; '"
```

The intention of the statement is to retrieve credit card information for a given user input. An example for an expected input 123456789.

However, the statement above allows malicious values for the `user_input` variable. An attacker might provide `’ OR ’1’=’1` as input which would render the following SQL statement:

```java
vuln_statement = " 'SELECT * FROM creditcards WHERE number = " + user_input + " OR ’1’=’1; ""
```

Instead of pulling detailed credit card information only for one specific credit card number, the statement pulls information for all credit cards stored in the database table. A potential web application with the above SQL injection vulnerability could leak sensitive credit card information for all users of the application.

Consequences of the above SQL injection vulnerability might be directly visible to the attacker if all credit card details are listed on a results page. However, the impact of an SQL injection can also be hidden and not visible to the attacker.
blind SQL injections do not display the results of the vulnerability directly to the attacker (e.g., because results are not listed on a website). However, the impact of an attack might still be visible through observing information as part of a true-false response of the database. Attackers might be able to determine the true-false response based on the web application response and the way the web site is displayed.

One way to mitigate SQL injection attacks is with the use of prepared statements. Instead of embedding user input into raw SQL statements (see above), prepared statements use placeholder variables to process user input. Placeholder variables are limited to store values of a given type and prohibit the input of arbitrary SQL code fragments. SQL injections attacks would result in invalid parameter values in most cases and not work as intended by an attacker. Also, prepared statements are supported by many web application development frameworks at the coding level using object-relational mapping (ORM) interfaces. ORMs do not require developers to write SQL queries themselves but generate database statements from code. While prepared statements are an effective mitigation mechanism, a further straightforward way is to escape characters in user input that have a special meaning in SQL statements. However, this approach is error-prone, and many applications that apply some form of SQL escaping are still vulnerable to SQL injection attacks. The reasons for mistakes are often incomplete lists of characters that require escaping. In case escaping is used, developers should rely on functions provided by web application development frameworks (e.g., the `mysqli_real_escape_string()` function in PHP) instead of implementing their own escaping functionality.

**Command Injections.** This type of injection attack affects vulnerable applications that can be exploited to execute arbitrary commands on the host operating system of a web application. Similar to SQL injection attacks, command injections are mostly possible due to insufficient user input validation. Vulnerable commands usually run with the same privileges as the host application.

An example of a command injection attack is a web application that converts user-provided images using a vulnerable image command line program. Providing malicious input (e.g., a file path or a specially crafted support vector graphic that includes malicious code) might allow attackers to exploit insufficient input validation and extend the original command or run additional system commands.

A mitigation for command injection attacks is to construct the command strings, including all parameters in a safe way that does not allow attackers to exploit malicious string input. In addition to proper input validation due to escaping, following the principle of least-privilege and restricting the privileges of system commands and the calling application is recommended. The number of callable system commands should be limited by using string literals instead of raw user-supplied strings. In order to further increase security, regular code reviews are recommended, and vulnerability databases (e.g., the CVEs database) should be monitored for vulnerabilities. Finally, if possible, executing system commands should be avoided altogether. Instead, the use of API calls of the respective development framework is recommended.

**User Uploaded Files.** Similar to command parameters, files provided by users such as images or PDFs have to be handled with care. Malicious files can affect command execution on the host operating system of the server, overload the host system, trigger client-side attacks, or deface vulnerable applications.

---

**Example: Online Social Network**

An example application could be an online social network that allows users to upload their avatar picture. Without proper mitigation techniques in place, the web application itself might be vulnerable. A malicious user could upload a `.php` file. Accessing that file might prompt...
the server to process it as an executable PHP file. This vulnerability would allow attackers to both execute code on the server with the permissions of the PHP process and also control the content served to other users of the application.

To prevent attacks through user-uploaded files, both meta-data including file names and paths and the actual content of user-uploaded files need to be restricted and filtered. Filenames and paths should be constructed using string literals instead of raw strings and proper mime-types for HTTP responses used whenever possible.

Files that are only available for download, can be protected by sending a Content-Disposition HTTP response header [69]. Another successful mitigation for the above issue is to serve files from a different domain. If the domain is not a subdomain of the original domain, the SOP [1, 4] prevents cookies and other critical information from being accessible to the malicious file. Additionally, JavaScript and HTML files are protected to the SOP as well.

**Local File Inclusion.** This type of vulnerability is a particular form of the above command injection or user-uploaded files vulnerabilities [38]. For example, attackers can exploit a command injection, use a malformed path in a database or a manipulated filename. The file path resulting from one of these vulnerabilities can be crafted to point to a local file on the server, e.g., a .htaccess or the /etc/ shadow file. A vulnerable web application might then access the maliciously crafted file path and instead of loading a benign file, reading and sending the content of the attacked file and e.g. leaking login credentials in the /etc/ shadow file.

In addition to sanitisation of file path parameters such as leading / and .. in user input, the application of the least privilege principle is recommended. A web application should be executed with minimal privileges and so that it cannot access sensitive files.

**Cross Site Scripting (XSS).** Cross Site Scripting (XSS) [70] attacks are injection vulnerabilities that allow attackers to inject malicious scripts (e.g., JavaScript) into benign websites. They can occur whenever malicious website users are able to submit client scripts to web applications that redistribute the malicious code to other end-users. Common examples of websites that are vulnerable to XSS attacks are message forums that receive user content and show it to other users. The primary root cause for XSS vulnerabilities is web applications that do not deploy effective input validation mechanisms. Untrusted and non-validated user-provided data might contain client-side scripts. Without proper user input validation, a formerly user-provided malicious JavaScript script might be distributed to other users and manipulate the website they are visiting or steal sensitive information. In an XSS attack, the client browser cannot detect the malicious code, since it is sent from the original remote host, i.e. *same-origin-policy* based security measures are ineffective. We distinguish two types of XSS attacks:

- **stored** In a stored XSS attack the malicious script is permanently stored on the target server (e.g. in a database) and distributed to the victims whenever they request the stored script for example as part of a comment in a message forum. Stored XSS attacks are also called permanent or Type-I XSS.

- **reflected** In a reflected XSS attack, the malicious script is not permanently stored on the target server, but reflected off the server to the victims. Malicious scripts in reflected attacks are distributed through different channels. A common way of delivering a malicious script is to craft a link to the target website. The link contains the script and clicking the link executes the malicious script in the website's script execution context. Reflected XSS attacks are also called non-permanent or Type-II XSS.

Preventing both types of XSS attacks requires rigorous user input validation and escaping on the server-side. The most effective way of input validation is a whitelist approach, which denies any input
that is not explicitly allowed. For proper and secure entity encoding, the use of a security encoding library is recommended, since writing encoders is full of pitfalls.

XSS detection is error-prone and can be difficult. The most promising approach is the use of code reviews.

**Cross Site Request Forgery.** Cross Site Request Forgery (CSRF) \cite{71} attacks mislead victims into submitting malicious HTTP requests to remote servers. The malicious request is executed on behalf of the user and inherits their identity and permissions. CSRF attacks are so dangerous because most requests to remote servers include credentials and session information associated with a user’s identity, including session cookies. Authenticated users are particularly attractive victims for attackers since remote servers can hardly distinguish benign from malicious requests as long as they are submitted from the victim’s machine. CSRF attacks do not easily allow attackers to access the server response for the malicious request. Therefore, the main goal of a CSRF attack is to trick victims into submitting state-changing requests to remote servers. Attractive targets are requests that change the victim’s credentials or purchase something.

---

**Example: Online Banking**

In the following online banking scenario Alice wishes to transfer 50 EUR to Bob using an online banking website that is vulnerable to a CSRF attack. A benign request for an authenticated user Alice for the mentioned scenario could be similar to `GET https://myonlinebank.net/transaction?to=bob&value=50`.

In a first step, an attacker can craft a malicious URL such as `https://myonlinebank.et/transaction?to=attacker&value=50` and replace the intended recipient of the transaction with the attacker’s account. The second step for successful CSRF attack requires the attacker to trick Alice into sending the malicious request, e.g. by sending a SPAM email containing the request. However, CSRF attacks are not limited to HTTP GET requests but do also affect POST requests, e.g. by crafting malicious `<form>` tags.

Many misconceptions lead to ineffective countermeasures. CSRF attacks cannot be prevented by using secret cookies because all cookies are sent from a victim to the remote server. Also, the use of HTTPS is ineffective as long as the malicious request is sent from the victim, because the protocol does not matter and the use of POST requests for sensitive information is insufficient since attackers can craft malicious HTML forms with hidden fields. To effectively prevent CSRF attacks, it is recommended to include randomised tokens in sensitive requests, e.g. by adding them to the request headers. The tokens must be unique per session and generated with a secure random number generator to prevent attackers from predicting them. Servers must not accept requests from authenticated clients that do not include a valid token.

---

### 3.2 Server Side Misconfigurations & Vulnerable Components

A web application stack consists of multiple components, including web servers, web application frameworks, database servers, firewall systems, and load balancers and proxies. Overall, web application security highly depends on the security of each of the involved components.

A single insecure component is often enough to allow an attacker access to the web application and further escalate their attack from the inside. This is why deploying and maintaining a secure web application required more than focusing on the code of the app itself. Every component of the web application stack needs to be configured securely and kept up to date (see Section \cite{1,10}).
The Heartbleed Vulnerability

A famous example of a critical vulnerability that affected many web application stacks in 2014 is Heartbleed [72]. Heartbleed was a vulnerability in the widely used OpenSSL library and caused web servers to leak information stored in the webservers' memory. This included TLS certificate information such as private keys, connection encryption details, and any data the user and server communicated, including passwords, usernames, and credit card information [73]. To fix affected systems, administrators had to update their OpenSSL libraries as quickly as possible and ideally also revoke certificates and prompt users to change their passwords.

As previously discussed, the principle of least privilege can reduce a web application stack's attack surface tremendously. Proper firewall and load balancer configurations serve as examples:

**Firewall.** To protect a webserver, a firewall should be configured to only allow access from outside where access is needed. Access should be limited to ports like 80 and 443 for HTTP requests via the Internet and restricting system configuration ports for SSH and alike to the internal network (cf. Network Security Knowledge Area).

**Load Balancers.** A widely deployed component in many web application stacks is load balancers. Load balancers can control HTTP traffic between servers and clients and provide additional access control for web application resources. They can be used to direct requests and responses to different web servers or ports, balance traffic load between multiple web servers and protect areas of a website with additional access control mechanisms. The most common approach for controlling access is the use of .htaccess files. They can restrict access to content and instruct load balancers to require additional authentication. .htaccess files are not provided by the original web server, for example using local file inclusion, to prevent the leakage of sensitive information such as static authentication credentials.

Load balancers can also serve for rate limiting purposes. They can limit request size, allowed request methods and paths or define timeouts. The main use of rate-limiting is to reduce the potentially negative impact of denial of service attacks on a web server and prevent users from spamming systems, as well as restrict and prevent unexpected behavior.

Additionally, load balancers can be used to provide secure TLS connections for web applications. When managing TLS, load balancers serve as a network connection endpoint for the TLS encryption and either establish new TLS connections to the application service or connect to the web application server using plain HTTP. If the web application server is not hosted on the same machine, using plain network connections might leak information to the internal network. However, if the web application server does not provide HTTPS itself, using a load balancer as a TLS endpoint increases security.

**HTTPS Misconfigurations**

One cornerstone of web and mobile security is the correct and secure configuration of HTTPS on web servers. However, Holz et al. [75] found that a significant number of popular websites deploy invalid certificates with incomplete certificate chains, issued for the wrong hostname or expired lifetime. In a similar study Fahl et al. [76] confirmed these findings and also asked website operators for the reasons to deploy invalid certificates. Most operators were not aware of using an invalid certificate or used one on purpose because they did not trust the web PKI. Krombholz et al. [77, 78] conducted a set of studies and found that operators have difficulties with correctly configuring HTTPS, or they harbour misconceptions about the security features of HTTPS.
Databases. Similar to load balancers and firewalls, many web application stacks include databases to store user information permanently. Often, databases are operated as an additional service that is hosted on another server. The application server interacts with the database through libraries and APIs. While on the application server-side, it is important to prevent injections, errors in the implementation of database libraries or coarse permissions that are required by the application can lead to vulnerabilities.

To reduce the attack vector, most database systems provide user management, to limit users’ privileges to create, read, delete or modify entries in tables and across databases. This way one database per application can be created and particular users with read-only permissions can be used by the application server.

An important aspect of increasing database security is the decision on how to store data. Encrypting data before storage in the database can help. However, especially for passwords or other information that only needs to be counter-checked, hashing before storage can tremendously increase security. In the case of a data leak, the sensitive information remains unreadable.

<table>
<thead>
<tr>
<th>Password Leaks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developers tend to store plain passwords, credit card information or other sensitive information in databases instead of encrypting or hashing them (cf. Human Factors Knowledge Area). Hence, many leaks of password databases or credit card information put users at risk [79]. Modern browsers and password managers help users to avoid passwords that were part of a previous data breach [80].</td>
</tr>
</tbody>
</table>

Conclusion

As we have shown, web and mobile security is a diverse and broad topic covering many areas. This Knowledge Area emphasised an intersectional approach by exploring security concepts and mechanisms that can be found in both the web and mobile world. It therefore builds upon and extends the insights from other Knowledge Areas, in particular the Software Security, Network Security, Human Factors, Operating System & Virtualisation Security, Privacy and Online Rights, AAA and Physical Layer & Telecommunications Security KAs.

We showed that due to the ubiquitous availability and use of web and mobile applications and devices, paying attention to their security issues is crucial for overall information security. We discussed web technologies that build the core of both web and mobile security, outlined their characteristics and illustrated how they are different from other ecosystems.

Later on, we split the discussion into client- and server-side aspects. In particular, this Knowledge Area has focussed on attacks and defences that were prevalent in web and mobile clients and servers and that dominated discussions over the last years.

Further Reading

The following resources provide a deeper insight into web and mobile security as well as guidance and recommendations for preventing and handling the vulnerabilities presented and discussed above.

The OWASP Project & Wiki

The Open Web Application Security Project (OWASP) is an international not-for-profit charitable organisation providing practical information about application and web security. It funds many projects including surveys like the OWASP TOP 10, books, CTFs and a wiki containing in-depth descriptions, recommendations and checklists for vulnerabilities and security measurements. The core wiki can be found at https://www.owasp.org/.
Mozilla Developer Network

An all-encompassing resource provided by Mozilla covering open web standards, including security advice and cross platform behaviour for Javascript APIs, as well as a HTML and CSS specifications. It can be found at [https://developer.mozilla.org](https://developer.mozilla.org)

Android Developers

The official documentation for the Android development ecosystem, including security advice for client side storage, webviews, permissions, Android databases and network connections. Includes information for outdated operating system versions and the Google Play Update process. Available at [https://developer.android.com](https://developer.android.com)

REFERENCES


