# SVS - Tutorials

eo shiru

July 11, 2019

## Contents

**1 Nmap, Attack Targets, HTTP Headers & Statusses** **1**

**2 SQL Injection** **3**

**3 Cross-Site-Scripting (XSS)** **4**

**4 Hashing, Keys and Encryption** **7**

**5 SSL/TLS (+ HTTP, SSH)** **10**

**6 Kerberos** **12**

6.0.1 Kerberos Procedure . . . . . 12

## 1 Nmap, Attack Targets, HTTP Headers & Statusses

The following activities are usually performed to take control over a network device:

| Activity | Goals | Defence mechanisms |
|---|---|---|
| Network analysis | Finding potential attack targets | Forbid ping responses, VLAN, disconnect end devices |
| Target System scanning | Identifying of software and services on target system | Avoid using standard ports, honeypots, disable unused servies, firewalls |
| Break-in | Take-over of control of target system | Block senders with unusal behavior, use strong password, update software |
| Exploitation | Exploits the weaknesses or failures of a system or an application to obtain privileges | Patch system and applications to fix weak points |

- tools for
  - target system scanning → nmap
    - scan a single IP → `nmap 192.168.1.1`

    - scan specific IPs → `nmap 192.168.1.1 192.168.2.1`
    - scan a range → `nmap 192.168.1.1-254`
    - `-sV --version-all` for service detection
    - flag `-A` or `-O` for OS detection
    - reverse DNS: `nmap -v -sn 192.168.0.0/24 # sucht alle rechner mit der IP 192.168.0.*`
    - services on host: `nmap -T4 -O -F tan.informatik.tu-chemnitz.de`
  - network analysis → Cain & Abel, Wireshark
  - break-in → Hydra

- Example IT entities which can become a target for an attack: Users, Applications, Operating Systems, End devices, Networks

**Which IT entities can become a target of attack?**

| Class | Objects | Examples |
|---|---|---|
| Network infrastructure | Router, Switches, Connections | Cable break, Flooding, Sniffing, MAC-Spoofing, Routing/Switching-Tables |
| End devices | Clients (PC, Laptops, Smartphones, IoT), Server, Proxies, Gateways | Physical takeover / destruction |
| Operating systems | Protocols, Libraries, Data stock (user & rights management, certificates) | SYN-Flooding, updates restraint, ping-of-death, Rootkits, Exploits, Virus, Worms |
| Applications and services | DNS/Mail/Web services, Browser, Firewalls, FTP, Database, Web apps | XSS, CSRF, Brute-Force, SQL-Injection, Dictionary attack, Port scanning |
| Users | Laziness, Inattention, Ignorance (Social Engineering) | Password guessing, phishing |

- "HTTP/1.0, includes the specification for a Basic Access Authentication scheme. This scheme is not considered to be a secure method of user authentication (unless used in conjunction with some external secure system such as SSL), as the user name and password are passed over the network as cleartext."
  - headers: `WWW-Authenticate: <type> realm=<realm>` response header defines the authentication method that should be used to gain access to a resource and `Authorization: <type> <credentials>` from request (user) to authenticate
    - Bei dem WWW-Authenticate: Basic Verfahren, werden Benutzername und Passwort unverschlüsselt an den Server übertragen. Dabei werden die Zugangsdaten in dem HTTP Header Authorization in der folgenden Form übertragen zB `Authorization: Basic base64_encode(Benutzername:Passwort)` d.h. Benutzername und Passwort wird mit einem Doppelpunkt verbunden und mit Base64 codiert (nicht verschlüsselt!).

Codierung ist nur eine andere Darstellung der Quelldaten und kann mit Kenntnis des Verfahren ohne Probleme codiert und decodiert werden

- The `Host` request header (`Host: <host>:<port>`) specifies the domain name of the server (for virtual hosting), and (optionally) the TCP port number on which the server is listening. If no port is given, the default port for the service requested (e.g., "80" for an HTTP URL) is implied (eg `Host: developer.cdn.mozilla.net`)
- The `Accept` request HTTP header advertises which content types, expressed as MIME types, the client is able to understand. Using content negotiation, the server then selects one of the proposals, uses it and informs the client of its choice with the `Content-Type` response header.

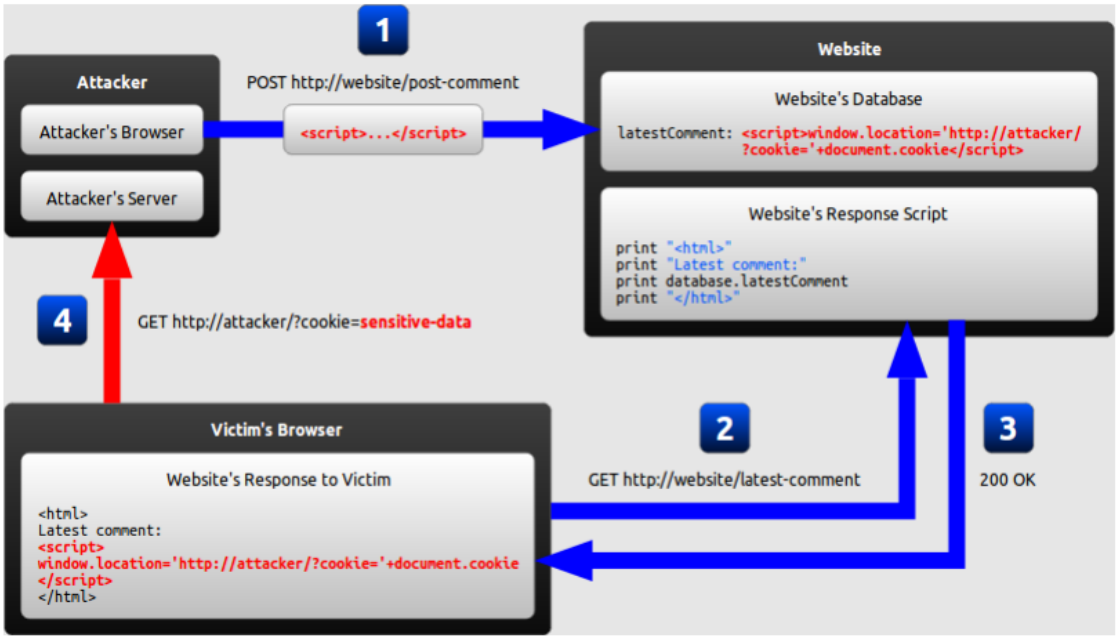
1xx (Informational): The request was received, continuing process
2xx (Successful): The request was successfully received, understood and accepted
3xx (Redirection): Further action needs to be taken in order to complete the request
4xx (Client Error): The request contains bad syntax or cannot be fulfilled
5xx (Server Error): The server failed to fulfill an apparently valid request

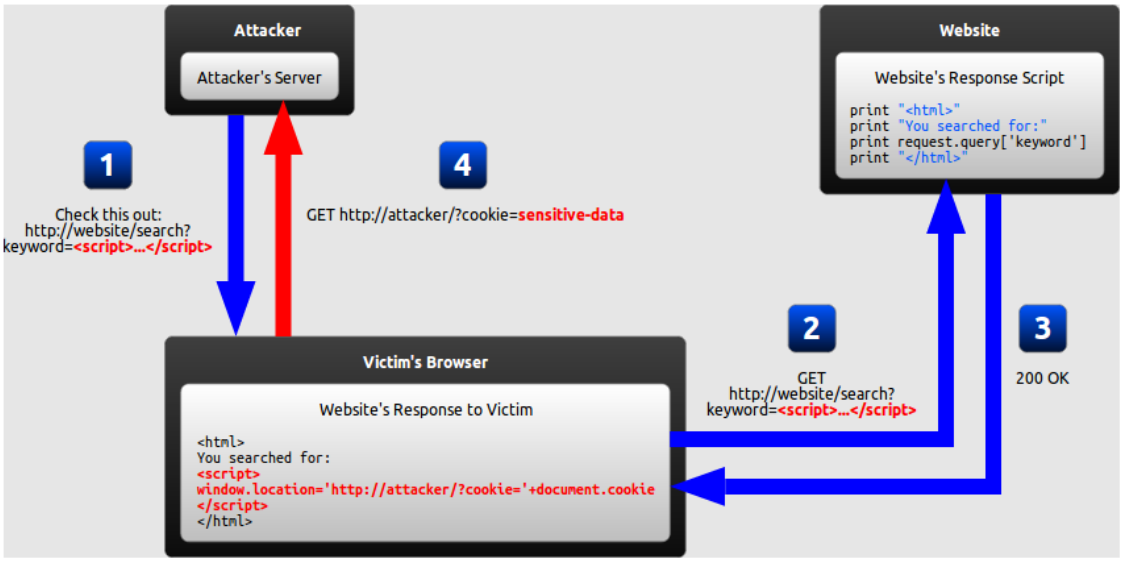
## 2 SQL Injection

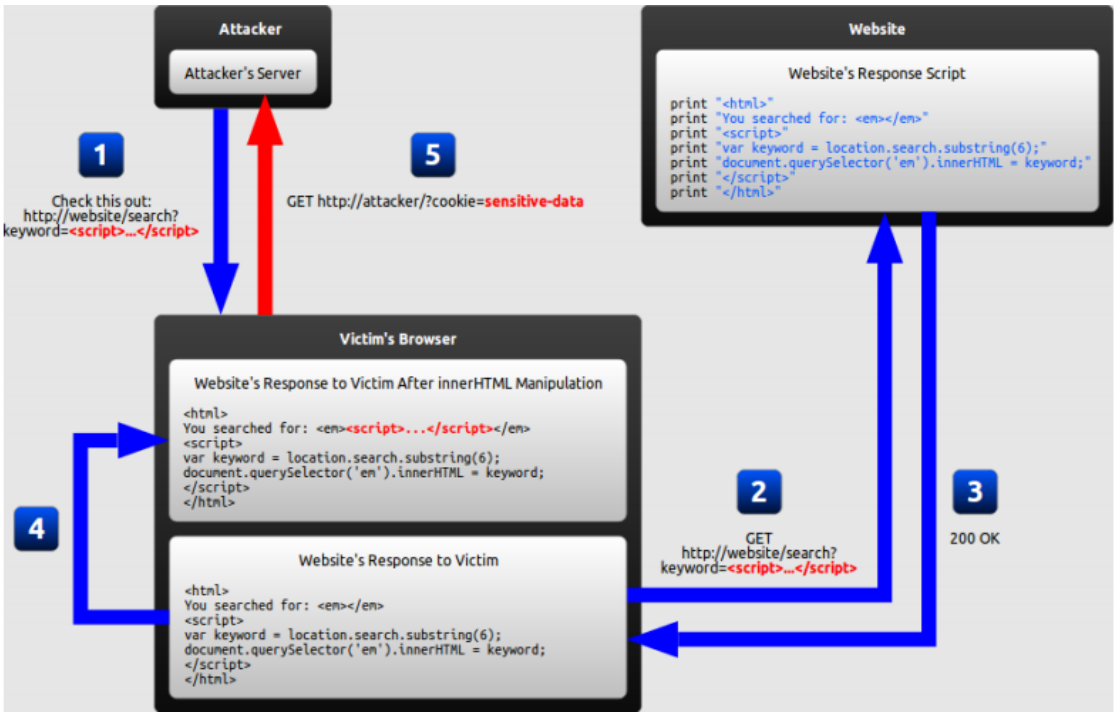
- What is Session Hijacking?
  - The Session Hijacking attack consists of the exploitation of the web session control mechanism, which is normally managed for a session token.
- How do you get a Session-Token?
  - Predictable Session Token
  - Session Sniffing
  - Client-side attacks (XSS)
- set "session" cookie via JS: `document.cookie="session=John";`
- What is SQL-Injection?
  - A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application
- What can be done with SQL Injection?
  - ...read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system
- a SQL query can be injected via form data, url parameters, cookies
- SQL injection example (not sanitized form fields):
  - one input field 'name': enter Max you get: `SELECT * FROM personen WHERE name = 'Max'`, enter Max' OR name LIKE '% and you get `SELECT * FROM personen WHERE name = 'Max' OR name LIKE '%'` which gives you all persons
  - two inputs name and password: enter user1 in the name input field and pass1' OR username LIKE '% in the password field to get all users with their respective passphrases

- SQL-Injection Protection:
  - Filter or mask special chars
  - Check input values for properties
  - Black-/Whitelisting of parameters
  - Seperation of Data and SQL-Queries
  - Fine Distinction of user rights
  - Disable unused DB functionalites
  - Apply Web Application Firewalls

# 3 Cross-Site-Scripting (XSS)






- What is Cross-Site Scripting (XSS)?
  - Cross-Site Scripting attacks are a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites
- What is achievable with XSS?
  - Spy out data (incl. Cookie / Session variables)
  - Altering website
  - Phishing
- Which defense methods against XSS do you

know?

- Validate user inputs
- Encode output
- Use HttpOnly flag for cookies
- Deactivate JavaScript in the browser
- Web Application Firewalls
- Example input form:
  - insert <script>JSCODE</script> to do whatever you want
- What is the difference between stored, reflected and DOM-based XSS attacks?

  - Stored XSS Attacks = Malicious code stored on server side – forums, guestbooks etc.
  - Reflected XSS Attacks = Malicious code delivered to the client, but not stored on the server side
  - DOM-based Attacks = Malfunction of normal code behavior via manipulated parameters

- Example XSS Input:

```
<script type="text/javascript">
function cookie_lesen() {
  document.cookie="username=John Doe;" + document.cookie;
  alert("Deine Cookies:\n\n" + document.cookie);
  // hier k nnte man nat rlich die Kekse an ein fremdes System
     schicken...
}
</script>

<a onclick="cookie_lesen()" href="#">cookies</a>

<script type="text/javascript">
function farbe_aendern() {
  var h1 = document.getElementsByTagName("h1")[0];
  h1.setAttribute("style", "color:red;");
}
function links_aendern() {
  var links = document.getElementsByTagName("a");
  var i;
  for (i = 0; i<links.length;i++) {
    links[i].href = "http://google.de/";
  }
}
farbe_aendern();
links_aendern();
</script>
```

- Logout via JS:
  - invalidate cookies → document.cookie += "; expires=Thu, 01 Jan 1970 00:00:01 GMT;"
  - location.reload()
- The same-origin policy is a critical security mechanism that restricts how a document or script loaded from one origin can interact with a resource from another origin (origin = protocol + port + host). It helps isolate potentially malicious documents, reducing possible attack vectors.
  - the first iFrame has same domain and port, the second iFrame refers to another domain and the access is therefore restricted by the same-origin-policy

```
var h1_first_frame = window.frames[0].document.getElementsByTagName("h1
    ")[0].innerHTML;
 window.frames[1] // undefined
```

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

For example:

```
<img src="http://meine-bank.example.com/?logout=1">
// or
<img src="http://meine-bank.example.com/transfer-money?from=123&to=321&
    amount=1024">
// or
<href="https://www-user.tu-chemnitz.de/~cole/svs/03/session.php?logout
    =1">Schaut mal, tolle Sache hier</a>
```

# 4 Hashing, Keys and Encryption

- Where to use one-way hash functions?
  - Transmit error detection
  - Fast data access
  - Identification/Comparison of secrets
- Criteria of a good one-way hash function:
  - Pre-image resistance, irreversibility
  - Second pre-image resistance, collision resistance
  - Efficient calculable
  - High dispersion  order preservation* SVS-Tutorial$_{06.pdf}$
- Example: Encode message "VSR" using the ASCII table and hash function $f(s) = (\sum_{i=1}^{\text{length}(s)} s_i)$
  - f("VSR") = (86 + 83 + 82) mod 7 = 251 mod 7 = 6

Example: Create MD5 hash of "VSR": `echo -n "VSR" | md5sum` for eg sha265 use `sha265sum` or via gpg: `echo -n "VSR" | gpg --print-md md5`

Caesar Cipher

- Advantages

  - Fast Algorithms
  - Easy Hardware implemenation
- Disadvantages
  - requires secure channel
  - requires key administration
- Security Risks
  - Frequency Analysis
  - Brute-Force
  - concealment of approach not key

Example: Encrypt message M = "DE" using RSA encryption with public key ($e = 7$, $N = 77$)

1. Turn M into **one** number m which is smaller than N: convert to numbers D $\rightarrow$ 4, E $\rightarrow$ 5 (positions in alphabet) $\rightarrow 4 + 5 = 9$
2. Compute ciphertext c: $c = m^e \mod N$
3. \$c = $9^7 \mod 77 = 37 Sendctopartner$

4. Recover m from c using via private key ($d = 43$, $N = 77$): $m = c^d \mod N$
5. $m = 37^43 \mod 77 = 9$

Using gpg

- `gpg --full-generate-key` to generate key-pair (RSA)
- send public key to recipient
  - via a file: `gpg --armor --output mypubkey.gpg --export your.name@yourdomain.com`
  - via public key server: `gpg --list-secret-keys` to find out public key id which stands next to "sec" and then export via `gpg --send-keys KEYID` and note the GPG server
- encrypting a file: `gpg --output myfile.txt.gpg --encrypt --recipient your.friend@your myfile.txt`
- decrypting a file: `gpg --output myfile.txt --decrypt myfile.txt.gpg`

signing a message so that the recipient can verify that it is indeed by the sender:

- private key = (d=27, N=55); public key = (e=3, N=55); message m = "DE" $\rightarrow$ 9
- $s = m^d \mod N$ = $9^{27} \mod 55 = 4\$ send message(9) and signature(4) to recipient$

- recipient verifies the message via $m = s^e \mod N$
- $m = 4^3 \mod 55 = 9 \rightarrow$ correct!

Alice sends $m$ through the public channel and $h(m)$ through the *integrity$ channel where $h$ is a cryptgraphic hash function

**authenticity** via verifying keys

- symmetric: secret key $K$ is generated and sent to Alice & Bob, Alice sends (m, MAC(m, K)) through public channel and Bob verifies via VERIF$(m, K)$; MAC is a hash function parameterized by K
- asymmetric: a key pair ($K_s$, $K_p$) is generate dby Alice and she sends the public key $K_p$ to Bob via public channel and she sends $(m, \text{SIGN}(m, K_s))$ through public channel and Bob verifies via VERIF$(m, K_p)$

**confidentiality** via permutation (encryption & decryption)

- symmetric: secret K is generated and sent to Alice & Bob, Alice sends $c = E(m, K)$ and Bob decrypts via $m = D(c, K)$
- asymmetric: key pair ($K_s$, $K_p$) is generated by Bob and he sends public key $K_p$ to Alice who then sends $c = E(m, K_p)$ to Bob who decrypts this via $m = D(C, K_s)$

X.509 is a standard defining the format of public key certificates, X.509 certificates are used in many Internet protocols, including TSL/SSL which is the basis for HTTPS

- contains a public key, an identity (hostname, organization, individual) and is either signed by a certificate authority or self-signed

In public key infrastructure (PKI) systems, a certificate signing request (also CSR or certification request) is a message sent from an applicant to a certificate authority in order to apply for a digital identity certificate. It usually contains the public key for which the certificate should be issued, identifying information (such as a domain name) and integrity protection (e.g., a digital signature)

create self signed cert via openssl: `openssl req -new -newkey rsa:4096 -x509 -sha256 -days 365 -nodes -out MyCertificate.crt -keyout MyKey.key`

**Symmetrische Verfahren**

- Integritätssicherstellung mit Hash. Wichtiges Kriterium der Hash's: Kollisionsfreiheit.
- Authentizität: Woher weiß Bob, dass Nachricht von Alice kommt? Niemand kennt privaten Schlüssel.

**Asymmetrische Verfahren**

- Kein Austausch von Schlüsseln; jeder generiert sich ein Schlüsselpaar. Jeder Kommunikationspartner hat die öffentlichen Schlüssel von den Anderen. Wenn Alice an Bob eine Nachricht schreibt, verwendet sie den Public-Key zum Verschlüsseln und Bob verwendet seinen Private-key zum entschlüsseln.

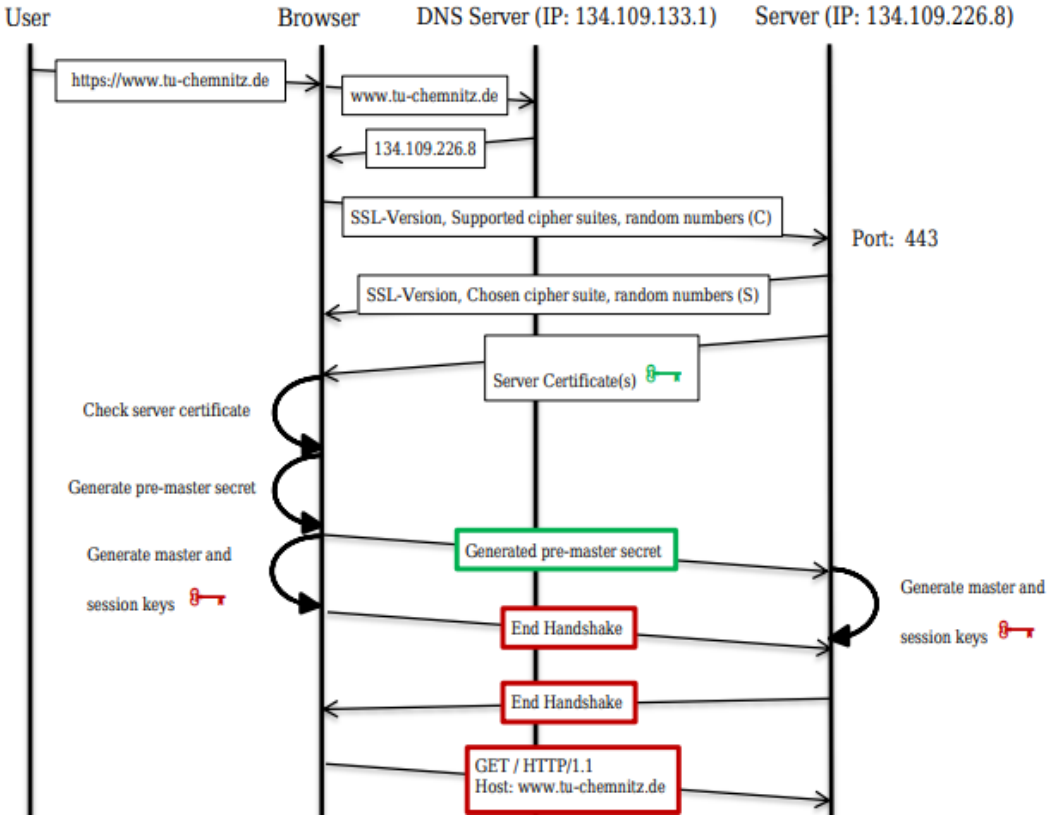
**Vertraulichkeit** = Private-Key ist nur dem Sender bekannt.

**Integrität** = Man kann blind versuchen die Nachricht zu manipulieren, z.B. Nachrichtenteil anhängen (der zweite Teil wird dann einfach wieder mit dem Public-Key verschlüsselt). Darum sollte man die Nachricht vor dem Versenden mit einem Hash versehen. Hashwert wird mitverschlüsselt.

**Authentizität** = Bob bekommt Nachricht mit Hash (von Alice). Woher weiß er, wer die Nachricht versendet hat?

- Signatur wird erstellt mit Private-Key von Alice über Hashwert der Nachricht.
- Bob kann Signatur mit Public-Key von Alice entschlüsseln und Authentizität feststellen.

# 5 SSL/TLS (+ HTTP, SSH)




- Hostname wird an DNS Server übertragen
- IP-Adresse kommt zurück
- Port: 443
- Server sendet Zufallszahl $\text{Random}_S$, gewählte Cipher Suite, SSL-Version
- Überprüfung des Zertifikats
  - Anhand des Zertifikatspeichers des Browsers, bspw. bei selbst signierten Zertifikaten
  - Signatur von Zertifikat, wird geprüft, ob öffentlicher Schlüssel im Browser bekannt ist
- Auf Basis von den 3 Zufallszahlen wird auf Client-Seite auch Primär- und Sizungs-Schlüssel generiert
- Server-End-Handshake, als Antwort auf Client-End-Handshake
- GET / HTTP/1.1, Host: www.tu-chemnitz.de
- Which goals does SSL/TLS have?
  - confidentiality
  - authenticity
  - integrity

- Which risks exist despite of usage of SSL/TLS?
  - out-of-scope
- How does the server decide which certificate should be shown if several virtual hosts exist?
  - Server Name Indication (SNI)
  - Wildcard Certificates
  - Multidomain-Certificates
- HTTP Digest Authentication
  - Pro:
    - prevent phishing
    - no password storage at apps
    - prevents chosen-plaintext attacks
    - prevents replay attacks
  - Con:
    - many security options are optional
    - man-in-the-middle attack
    - prevents strong hash algorithms
  - What would a client send as a response to the following server message, if his username would be „Max“ and his password - „Secure123“?
    - Request (Client):

```
GET /index.html HTTP/1.1
Host: localhost
```

    - Response (Server):

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Digest realm="Secured␣Area",
nonce="aer95b7fg2dd2hhe8b11d0f6f7afb0c14v"
Content-Length: 0
```

    - Client:
      - HA1 = MD5(username:realm:password)
      - HA2 = MD5(method:digestURI)
      - response = MD5(HA1:nonce:HA2)
- SSH Public Key Authentication
  - create key: `ssh-keygen -t rsa -C "my_email@example.com"`
  - transfer to server: `cat ~/.ssh/id_rsa.pub | ssh username@server.address.com 'cat » ~/.ssh/authorized_keys'` (ssh agent needs to be running for this)
    - alternative: `ssh-copy-id -i .ssh/id_rsa.pub {user}@{server}`
  - test authentication via `ssh {user}@{server}`

# 6 Kerberos

### 6.0.1 Kerberos Procedure

**Description**

- The client authenticates itself to the Server (AS) which forwards the username to a key distribution center (KDC). The KDC issues a ticket-granting ticket (TGT), which is time stamped and encrypts it using the ticket-granting service's (TGS) secret key and returns the encrypted result to the user's workstation. This is done infrequently, typically at user logon; the TGT expires at some point although it may be transparently renewed by the user's session manager while they are logged in.
- When the client needs to communicate with another node ("principal" in Kerberos parlance) to some service on that node the client sends the TGT to the TGS, which usually shares the same host as the KDC. Service must be registered at TGT with a Service Principal Name (SPN). The client uses the SPN to request access to this service. After verifying that the TGT is valid and that the user is permitted to access the requested service, the TGS issues ticket and session keys to the client. The client then sends the ticket to the service server (SS) along with its service request.

**User Client-based Logon**

- A user enters a username and password on the client machine(s). Other credential mechanisms like pkinit (RFC 4556) allow for the use of public keys in place of a password.
- The client transforms the password into the key of a symmetric cipher. This either uses the built-in key scheduling, or a one-way hash, depending on the cipher-suite used.

**Client Authentication**

- The client sends a cleartext message of the user ID to the AS (Authentication Server) requesting services on behalf of the user. (Note: Neither the secret key nor the password is sent to the AS.)

The AS checks to see if the client is in its database. If it is, the AS generates the secret key by hashing the password of the user found at the database (e.g., Active Directory in Windows Server) and sends back the following two messages to the client:

- Message A: Client/TGS Session Key encrypted using the secret key of the client/user.
- Message B: Ticket-Granting-Ticket (TGT, which includes the client ID, client network address, ticket validity period, and the client/TGS session key) encrypted using the secret key of the TGS.

Once the client receives messages A and B, it attempts to decrypt message A with the secret key generated from the password entered by the user. If the user entered password does not match the password in the AS database, the client's secret key will be different and thus unable to decrypt message A. With a valid password and secret key the client decrypts message A to obtain the Client/TGS Session Key. This session key is used for further communications with the TGS. (Note: The client cannot decrypt Message B, as it is encrypted using TGS's secret key.) At this point, the client has enough information to authenticate itself to the TGS.

**Client Service Authorization**

- When requesting services, the client sends the following messages to the TGS:
- Message C: Composed of the TGT from message B and the ID of the requested service.
- Message D: Authenticator (which is composed of the client ID and the timestamp), encrypted using the Client/TGS Session Key.

Upon receiving messages C and D, the TGS retrieves message B out of message C. It decrypts message B using the TGS secret key. This gives it the "client/TGS session key". Using this key, the TGS decrypts message D (Authenticator) and compare client ID from message C and D, if they match server sends the following two messages to the client:

- Message E: Client-to-server ticket (which includes the client ID, client network address, validity period and Client/Server Session Key) encrypted using the service's secret key.
- Message F: Client/Server Session Key encrypted with the Client/TGS Session Key.

**Client Service Request**

- Upon receiving messages E and F from TGS, the client has enough information to authenticate itself to the Service Server (SS). The client connects to the SS and sends the following two messages:
  - Message E: from the previous step (the client-to-server ticket, encrypted using service's secret key).
  - Message G: a new Authenticator, which includes the client ID, timestamp and is encrypted using Client/Server Session Key.

The SS decrypts the ticket (message E) using its own secret key to retrieve the Client/Server Session Key. Using the sessions key, SS decrypts the Authenticator and compares client ID from messages E and G, if they match server sends the following message to the client to confirm its true identity and willingness to serve the client:

- Message H: the timestamp found in client's Authenticator (plus 1 in version 4, but not necessary in version 5[6][7]), encrypted using the Client/Server Session Key.

The client decrypts the confirmation (message H) using the Client/Server Session Key and checks whether the timestamp is correct. If so, then the client can trust the server and can start issuing service requests to the server. The server provides the requested services to the client.

Advantages of KDC

- User's passwords are never sent across the network, encrypted or in plain text. Secret keys are only passed across the network in encrypted form. Hence, a miscreant snooping and logging conversations on a possibly insecure network cannot deduce from the contents of network conversations enough information to impersonate an authenticated user or an authenticated target service.
- Client and server systems mutually authenticate – at each step of the process, both the client and the server systems may be certain that they are communicating with their authentic counterparts

- the tickets passed between clients and servers in the Kerberos authentication model include timestamp and lifetime information. This allows Kerberos clients and Kerberized servers to limit the duration of their users' authentication. While the specific length of time for which a user's authentication remains valid after his initial ticket issued is implementation dependent, Kerberos systems typically use small enough ticket lifetimes to prevent brute-force and replay attacks. In general, no authentication ticket should have a lifetime longer than the expected time required to crack the encryption of the ticket

Eve sniffs the traffic between editor, KDC and printer during key exchange - is she able to decrypt the sniffed data key?

- no because she doesn't have the password

After sniffing the data Eve was successful interrupting communication between editor & printer and forwards the unmodified sniffed data to the printer. Is she now able to impersonate Alice?

- yes

Eve wants to bypass KDC and access the printer directly, is this possible?

- doesnt know the printers password so she cannot create a session key and act as the KDC

Repeat the process of Kerberos authentication:

1. The operation system of Alice has Kerberos integration. Alice wants to sign in into the system. Describe how the authentication process takes place.
    - Alice enters username and OS prompts for according password
    - OS sends username (in cleartext) to KDC Authentication Server (AS)
    - AS sends TGT packet to OS which is encrypted with $P_A$
    - OS decrypts the TGT packet with the password from alice, if the password is correct this succeeds
2. Alice wants to access a Kerberos-enabled service, e.g. POP3. Does she have to re-enter her password?
    - no as long as the OS has a valid session key which is sent with TGT packet to the TGS (Ticket Granting Service)
3. How does POP3 service check if the request comes really from Alice? What are the timestamps used for?
    - identification of Alice (name) is engrained in the TGT and timestamps are used to invalidate tickets and prevent repeat attacks (additional security measure)

An organization operates a LAN (IP range 192.168.0.0/24) with a Web/FTP-server (IP 83.160.17.4), and a Firewall with dynamic packet filter (IP 220.20.117.6) (cf. illustration below). Create firewall rules, which fulfil the requirements below. Try to achieve maximum security.

- Access to the Web server should be allowed only using HTTPS (both from the internet and from the LAN).
- Access to the FTP service is allowed only from the LAN

- The administration of the Webserver should take place over SSH and only from the machines 192.168.0.6 and 192.168.0.7.
- Access from LAN to the server of an online game operator (IP 65.223.145.12) should be forbidden.
- Access to TCP-based internet services from the LAN is allowed.
- DNS requests from LAN are allowed.

| Action | Protocol | Interface | From | To | Port |
|---|---|---|---|---|---|
| Permit | TCP | any | any/any | 83.160.17.4 | 443 |
| Permit | TCP | eth2 | 192.168.0.0/24 | 83.160.17.4 | 20, 21 |
| Permit | TCP | eth2 | 192.168.0.2 | 83.160.17.4 | 22 |
| Deny | TCP, UDP | eth2 | 192.168.0.0/24 | 65.223.145.12 | any |
| Deny | any | eth2 | 192.168.0.0/24 | 83.160.17.4 | any |
| Permit | TCP | eth2 | 192.168.0.0/24 | any | any |
| Permit | UDP | eth2 | 192.168.0.0/24 | any | 53 |
| Deny | any | any | any/any | any | any |