[Th 1] Authentication

AUTHENTICATION

Authentication vulnerabilities allow attackers to gain access to sensitive data and functionalities.
In addition, they can expose the system to other attacks surface to perform other exploits.

What is the authentication?

Authentication is the mechanism used by websites or applications in order to verify the identity of a user
By the fact websites are exposed to anyone on the internet authentication is one of the most important mechanisms to protect in order to increase security.

Types of authentication (authentication factors)

there are basically 4 types of authentication factors:

- **something you know (knowledge factors)**
  - it involves the use of **Passwords, PINs, answers to prearranged questions**
  - in general not very secure, the attackers can steal them
- **something you have (possession factors)**
  - it involves the usage of **tokens from electronic keycards, smart cards, and physical cards**
  - for example, a PIN sent to our mobile phone
  - not so secure, the attackers can steal them
- **something you are (static inherence factors)**
  - it involves **static biometrics like fingerprint, retina, and face**
  - example fingerprint on mobile phone
  - sometimes the software/hardware used to perform the check can give false positive or negative examples
- **something you do (dynamic inherence factors)**

    - it involves **dynamic biometrics like voice patterns, handwriting characteristics, and typing rhythm**
    - example, sometimes recaptcha takes traces about our mouse usage etc, and doesn't ask for proof that we are not robots
    - sometimes the software/hardware used to perform the check can give false positive or negative examples

The best solution in order to increase the overall security is the **multifactor authentication.**

- is the strategy of using different factors
- note that we have to use different typologies of factors and not two times the same.

Authentication vs authorization (authN vs authZ)

The **authentication (AuthN) verifies who the user claims to be.** Basically is used to check that the user who claims to be "Alice" is the same person who created the "alice" account.

The **authorization (AuthZ) verifies what a user is allowed to do.** So basically after "alice" is logged in, what are the permissions she has?

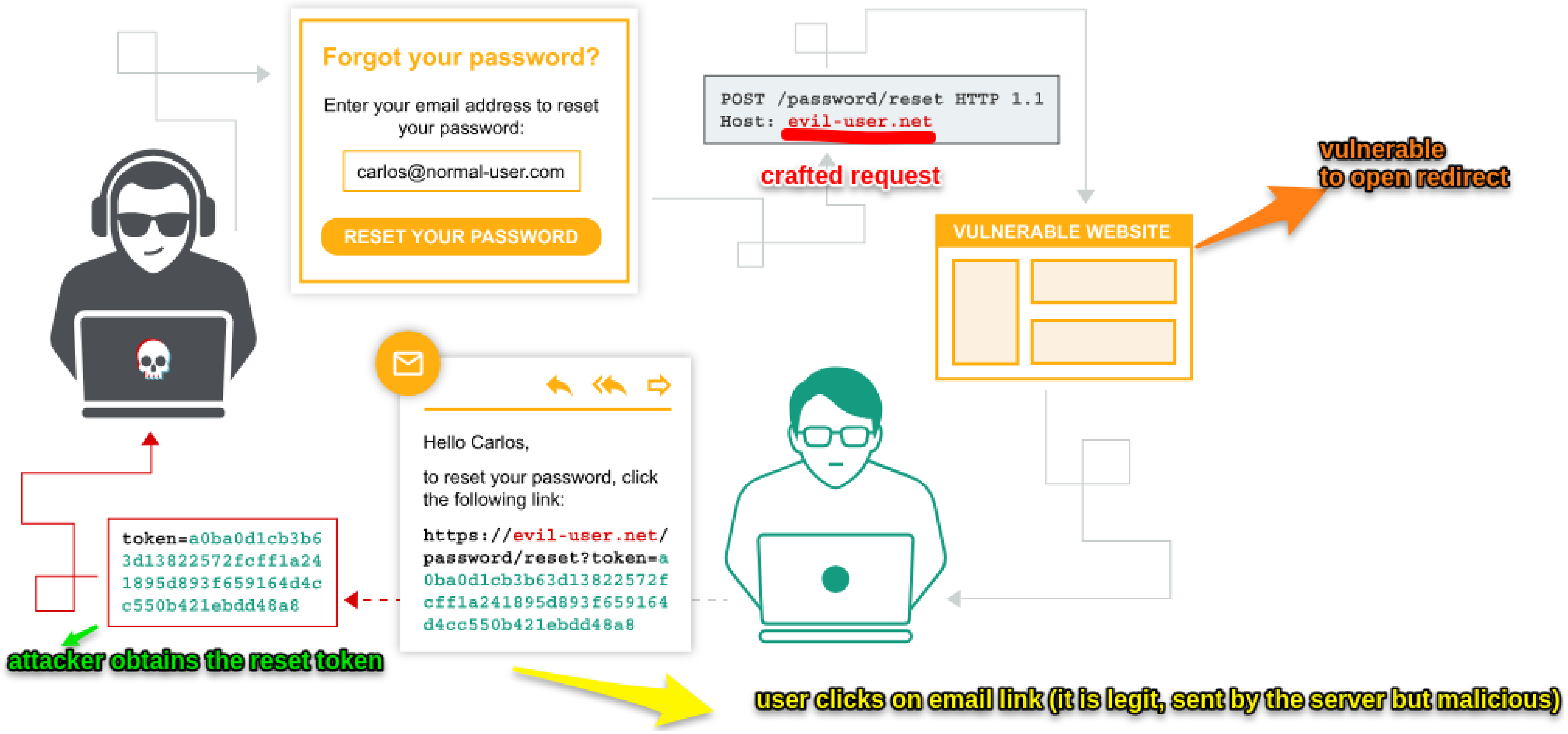
- she could be an admin and so has the access to delete other user profiles, or maybe not

How do the authentication vulnerabilities arise?

Generally, authentication vulnerabilities derive from the fact that:

- authentication mechanisms are weak because they are not able to protect against brute force
- logic flaws or poor coding in the implementation of the authentication that allows bypassing (this is called broken authentication)

ATTACK EXAMPLE


Forgot your password?
Enter your email address to reset your password:
carlos@normal-user.com
RESET YOUR PASSWORD
crafted request
vulnerable to open redirect
VULNERABLE WEBSITE
Hello Carlos,
to reset your password, click the following link:
attacker obtains the reset token
user clicks on email link (it is legit, sent by the server but malicious)

```
POST /password/reset HTTP 1.1
Host: evil-user.net
```

```
https://evil-user.net/
password/reset?token=a
0ba0d1cb3b63d13822572f
cff1a241895d893f659164
d4cc550b421ebdd48a8
```

```
token=a0ba0d1cb3b6
3d13822572fcff1a24
1895d893f659164d4c
c550b421ebdd48a8
```

In some cases, we have websites that suffer from open redirects:

- An open redirect vulnerability occurs when an application allows a user to control a redirect or forward to another URL. If the app does not validate untrusted user input, an attacker could supply a URL that redirects an unsuspecting victim from a legitimate domain to an attacker's phishing site.

in this specific scenario, the website allows users to reset their password by sending an email with a reset link

- the link contains the host header of the HTTPs request
- a uuid for the user

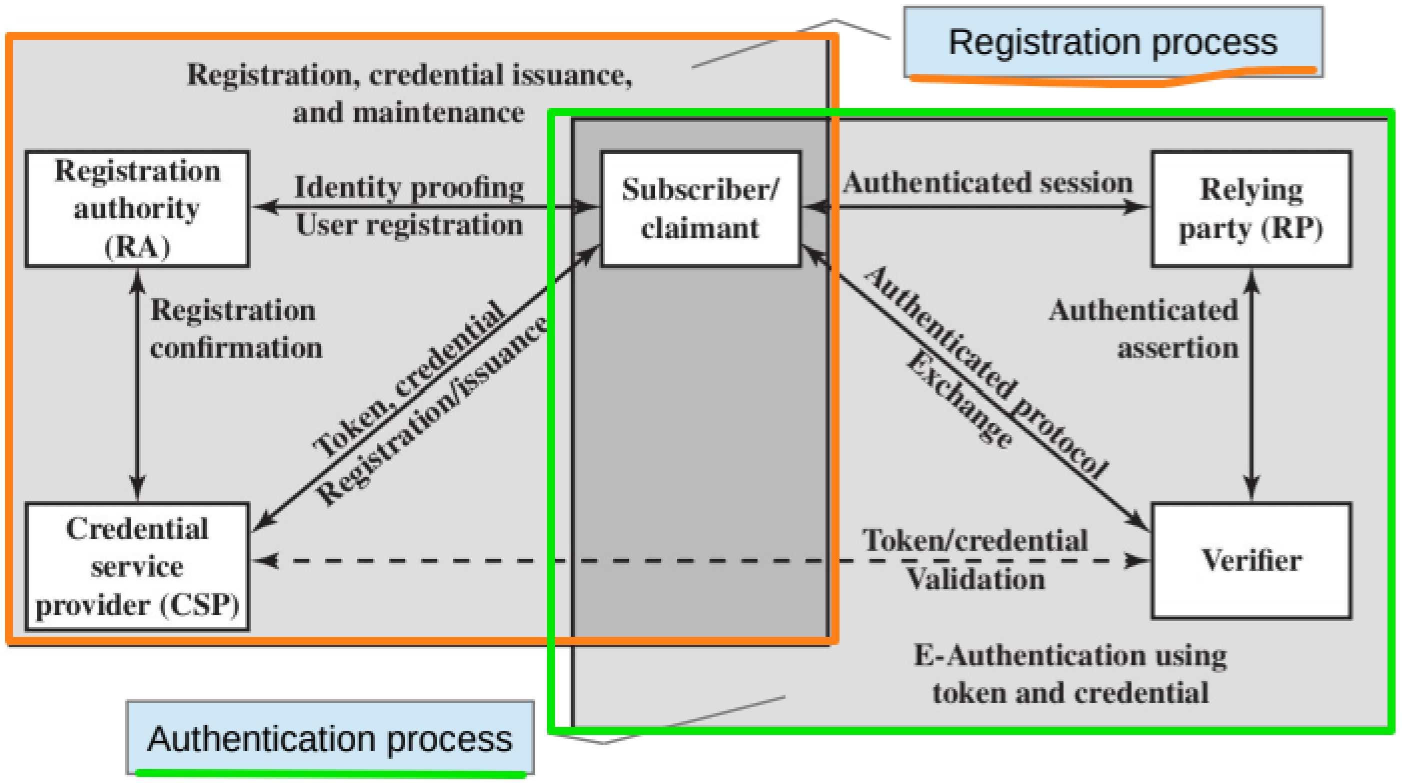
The link is unique for the user and allows the password reset

Attack:

1. the attacker tries to reset another user's password
    1. it intercepts its own request and modifies the host header inserting in its own host server
2. the server obtains the request and crafts the reset link
    1. it takes the host of the request
3. the victim will receive the email
4. the victim clicks on the link in the email
5. the attacker will obtain a request on his host that contains the user token to reset the password
6. the goes on the real reset link and paste the obtained token

Digital user authentication model (NIST SP 800-63-3)

This is a standard model used to simplify the real world:



Password-based authentication

Users provide credentials a username and a password.

The system compares the credentials with those stored and guarantees access if there is a match.

Attack strategies on password-based auth and countermeasures

Offline dictionary attacks

- we can prevent access to the password file (for example in Linux only the root can read the shadow file that contains the passwords)

- we can implement an intrusion detection measure to identify a compromise

Specific account attack
- we can lock the account after 5 failed logins (better if we lock it for some time and not forever)

Popular password attack
- we could use passwords that are not common
- we can monitor authentication requests in order to find patterns (for example I try to perform login on various accounts)

Password guessing against a single user
- we could make passwords difficult to discover (min length, a set of chars, non-known passwords)

Workstation hijacking (when someone obtains control of a workstation where we don't perform log out)
- we could invalidate sessions after a period of inactivity

Exploiting user mistakes (password written on post-its, password sharing, default passwords)
- we could use multifactor auth

Exploiting multiple password usage (same password for different websites)
- we could train the users

Hashed passwords

The important thing in password authentication is to don't save user passwords!
If a website can send you your password then it stores your password somewhere and it is a problem!
If a website can check that your new password is similar to the previous five then it stores your last five passwords and it is a problem!

DON'T STORE THE PASSWORDS

Store the hash value of the password combined with a salt value

- **a salt value is a public random value used to increase the unpredictability of the password at all**
- it is used to avoid rainbow table attacks

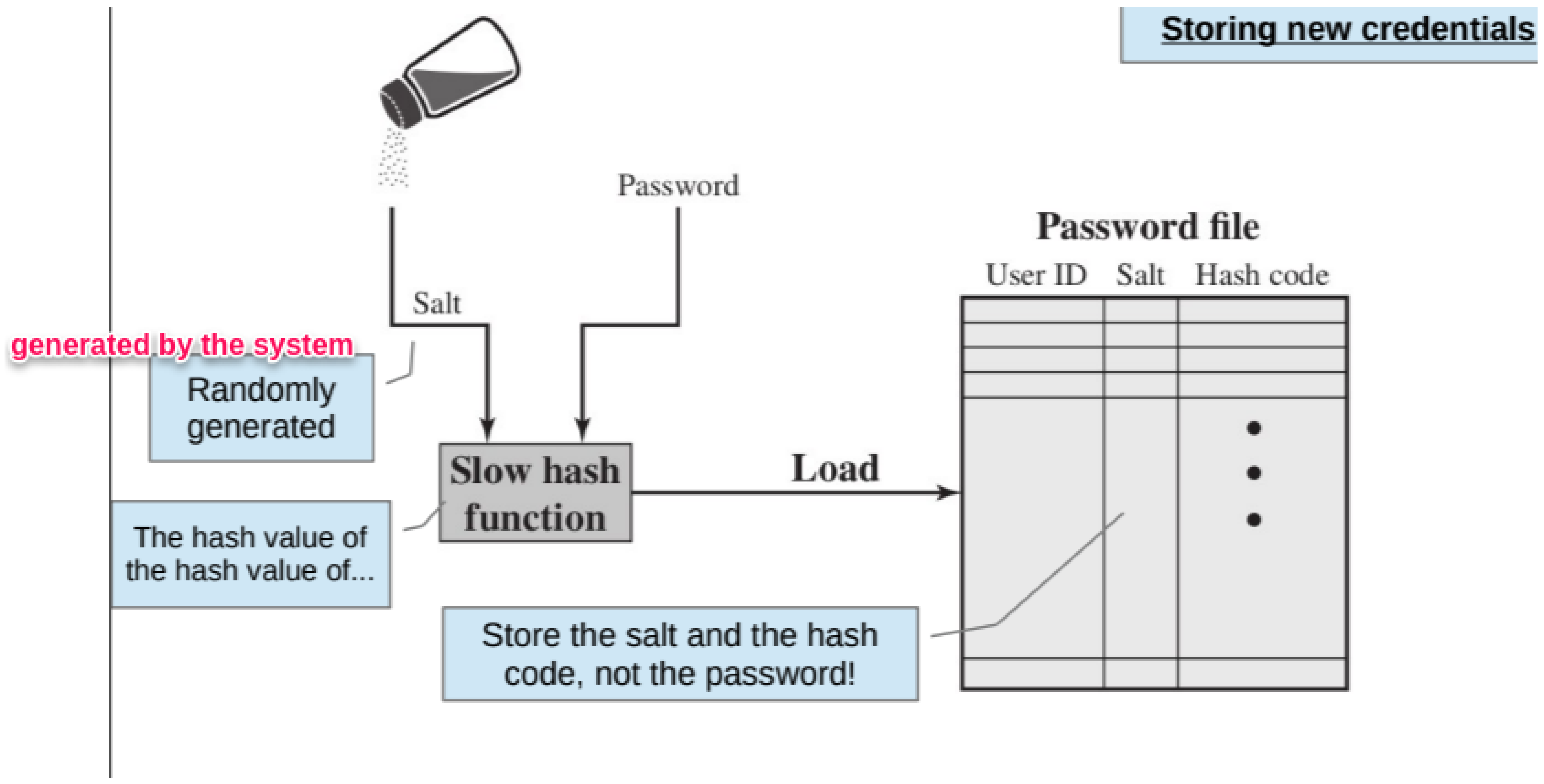
In some cases, there is another random value that is called pepper

- **a pepper value is a random value stored privately on the server**
- no one can know it
- if the server loses it then we need to reset all passwords

To make the rainbow table creation and the brute force attacks impossible to do we must prefer to store

- **the hash of the hash.... of the hash of the password**

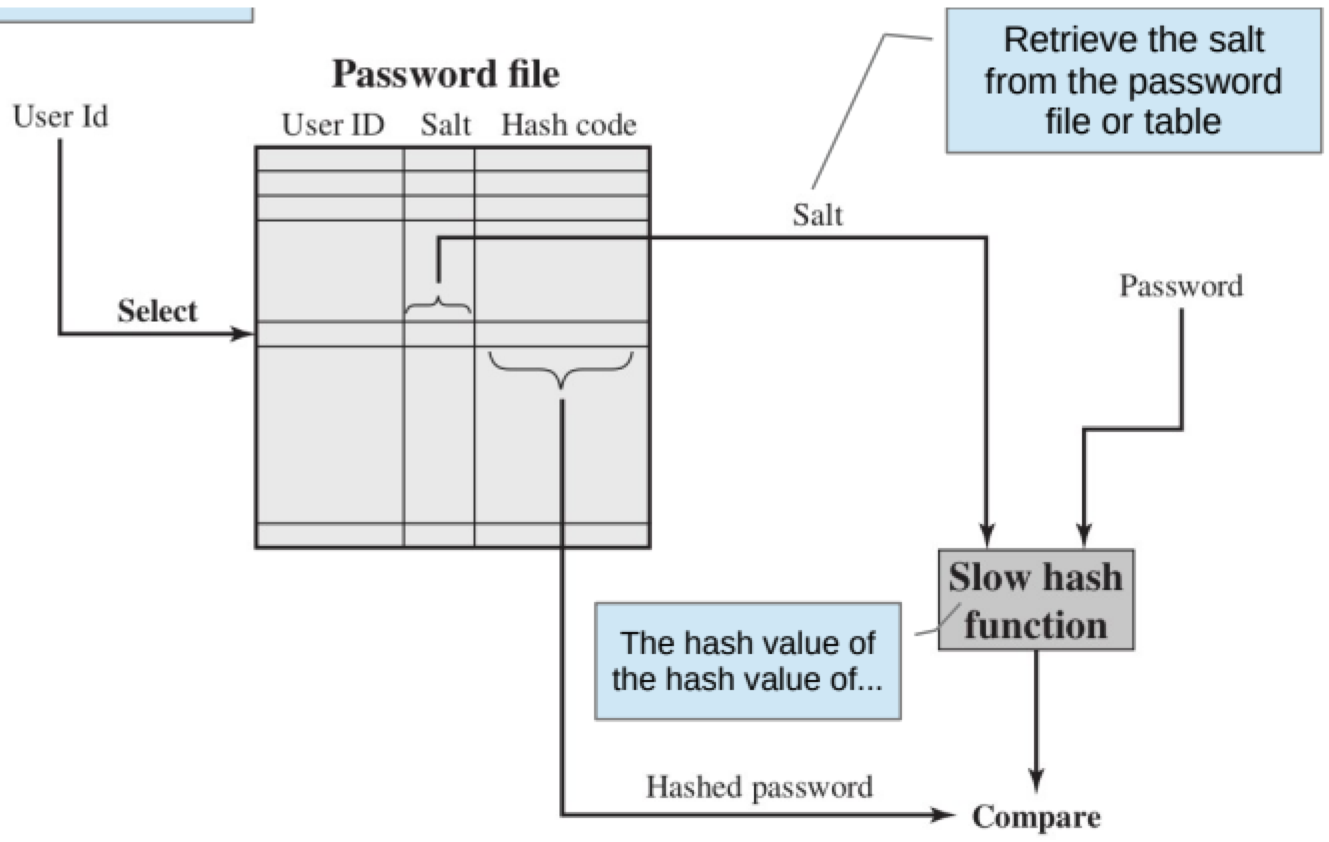
Save new credentials



Generally, a hashed password is of the type:

- 6b3a55e0261b0304143f805a24924d0c1c44524821305f31d9277843b8a10f4e
- in the initial part, we can find information about salt, how many times the hash function is used
- then we can find the hash value of the password

Verify the credentials



Salt and hash purpose

With the salt, we can have an interesting property. Basically, each user has its own salt so

- **the same password will generate two different hash values**
- **the rainbow table becomes impossible to be generated**

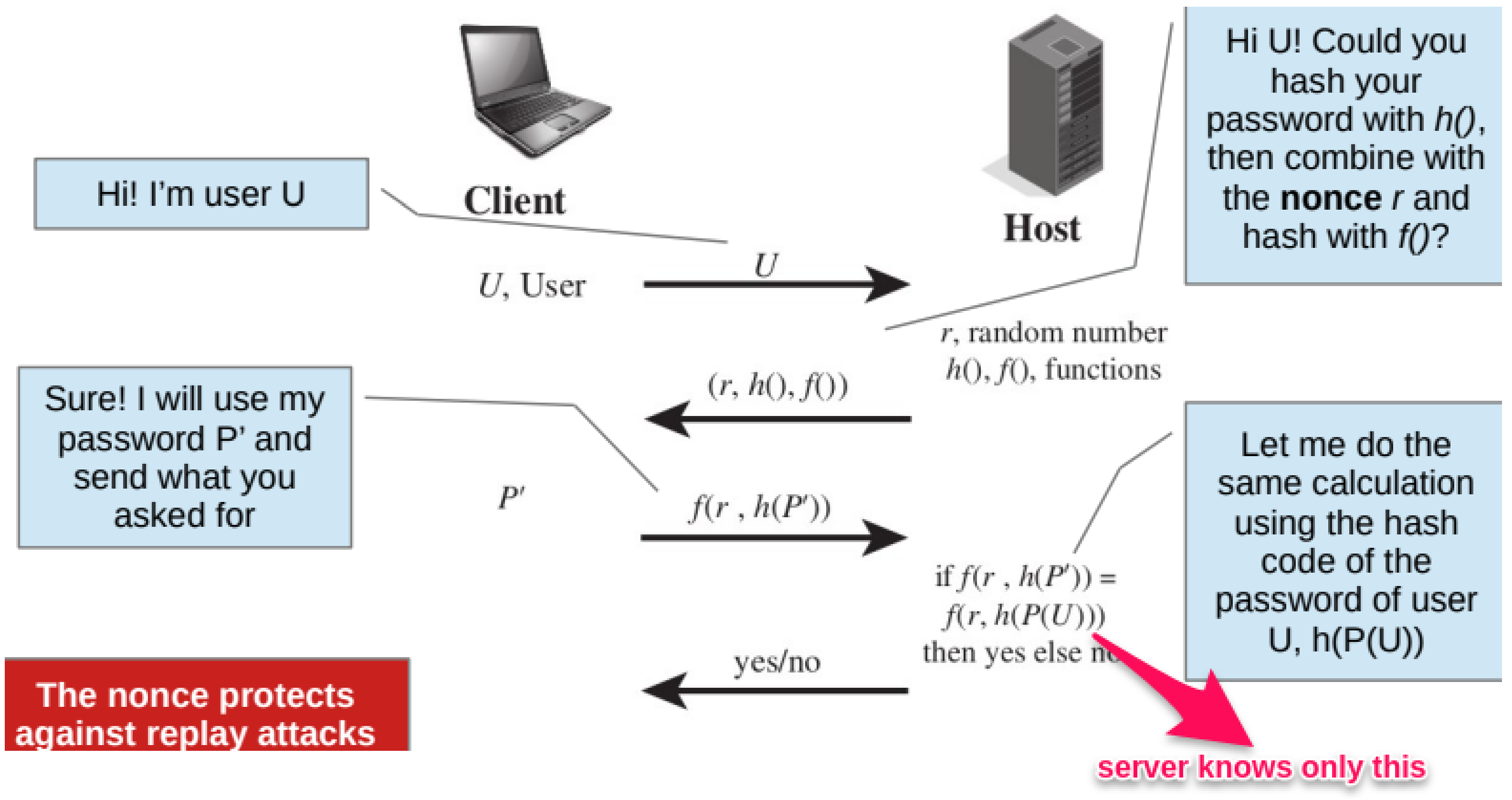
In general, **if the hash function is sufficiently slow then brute force cracking is practically impossible.**

Modern password hash/salt schemes use Bcrypt.

Crackstation (to decrypt hash values using a rainbow table)

We can use https://crackstation.net/ to try to revert hash values using rainbow tables.

Password protocol

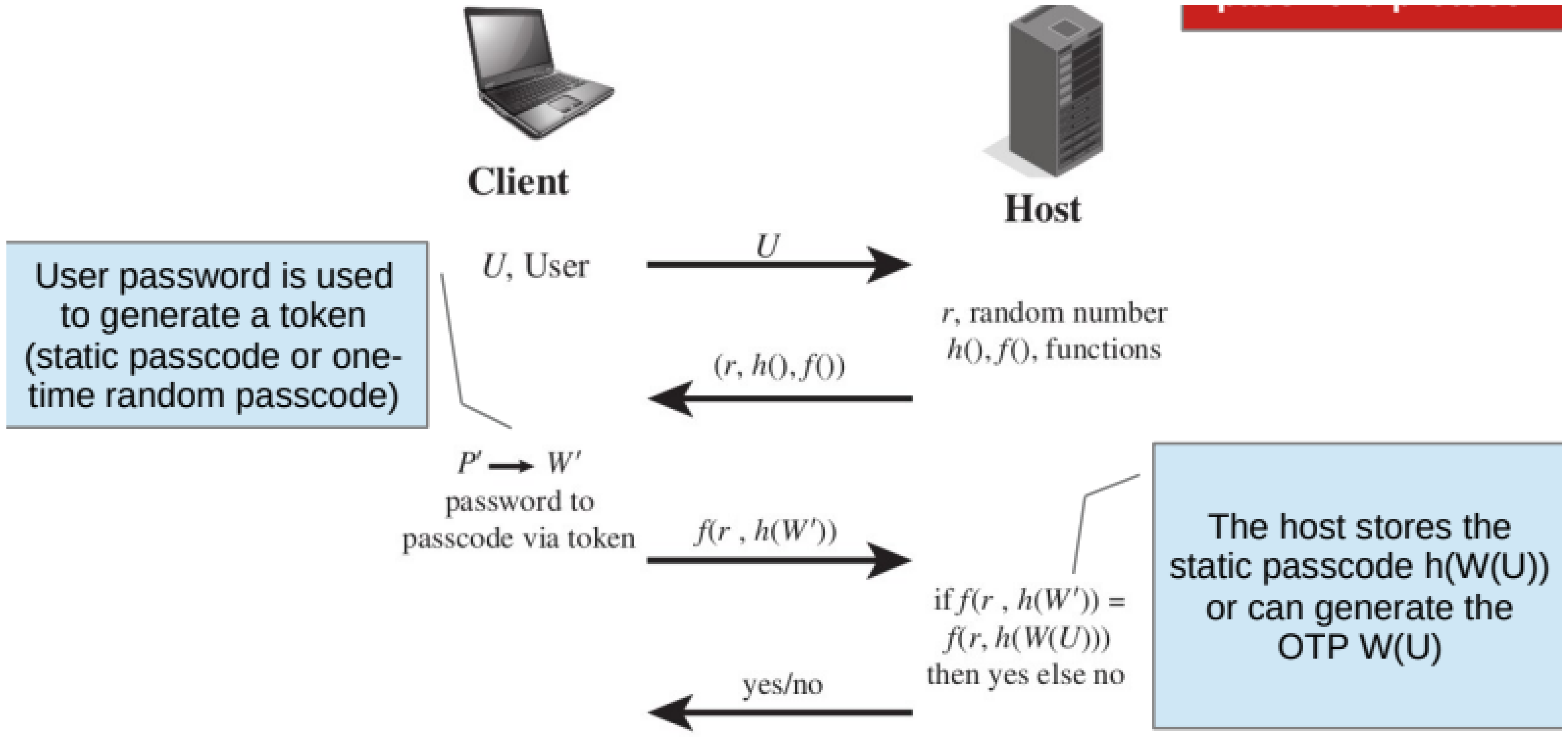


Client
Host
Hi! I'm user U
U, User
U
Hi U! Could you hash your password with h(), then combine with the nonce r and hash with f()?
r, random number
h(), f(), functions
(r, h(), f())
Sure! I will use my password P' and send what you asked for
P′
f(r , h(P′))
Let me do the same calculation using the hash code of the password of user U, h(P(U))
if f(r , h(P′)) = f(r, h(P(U))) then yes else no
yes/no
The nonce protects against replay attacks
server knows only this

- the nonce is used one time and is generated randomly each time a login request is sent in order to avoid replay attacks
- f() and h() are hash functions

Token protocol

It is pretty similar to the password protocol.


Client
Host
User password is used to generate a token (static passcode or one-time random passcode)
U, User
U
r, random number
h(), f(), functions
(r, h(), f())
P' → W'
password to passcode via token
f(r , h(W'))
if f(r , h(W')) = f(r, h(W(U))) then yes else no
The host stores the static passcode h(W(U)) or can generate the OTP W(U)
yes/no

- the idea is that for example the server knows the hash of the password of the user and from this it generates the token

In both cases, the only way to block an MITM is to use TLS or security protocol over data.

Vulnerabilities in password-based login

In websites that adopt password-based authentication users can create an account themselves or they are assigned an account by the admin.
This account is associated with a unique username and a secret password, which the user will enter in a login form to authenticate themselves.
So the security of the website is compromised if the attacker can guess or obtain these login credentials. This is because the entire security is based only on them.

Brute-force attacks

A brute force attack is an attack in which an attacker uses a system of trial and error in order to guess valid user credentials.
These attacks are typically automated using wordlists of usernames and passwords.
However, we have to consider that if the attacker knows some public information about the users for example the brute force can be refined in order to increase the efficiency.

- for example, in some companies, the employee emails are of the type name.surname@company.com

Brute force credentials are easier if usernames can be enumerated.

Username enumeration

Username enumeration is an activity in which the attacker is able to understand the usernames used to perform the website registration by looking at website behavior.
Basically, when we try to brute force usernames we need to pay attention to:

- HTTP status code
- error messages, maybe when the username does not exist we have a different error on the login form
- response time

Flawed brute-force protection

In general brute force attacks will generate a high number of wrong attempts so for this reason brute-force protection relies on this behavior.

There are two most common ways of preventing brute force attacks:

- lock the attacked account
- block the attacker's IP

But in some cases, these protection strategies are wrong implemented for example:

- if we lock the account after 5 attempts and report to the user that the account will be blocked then the attacker can enumerate the users
- if we block the IP of the attacker, in general, the IP attempt counter is reset after a successful login
    - so the attacker can perform a number of guesses followed by a successful login with an account created appositely (aòternate brute force with correct login)

Prevention

To prevent security problems we have to follow always some principles

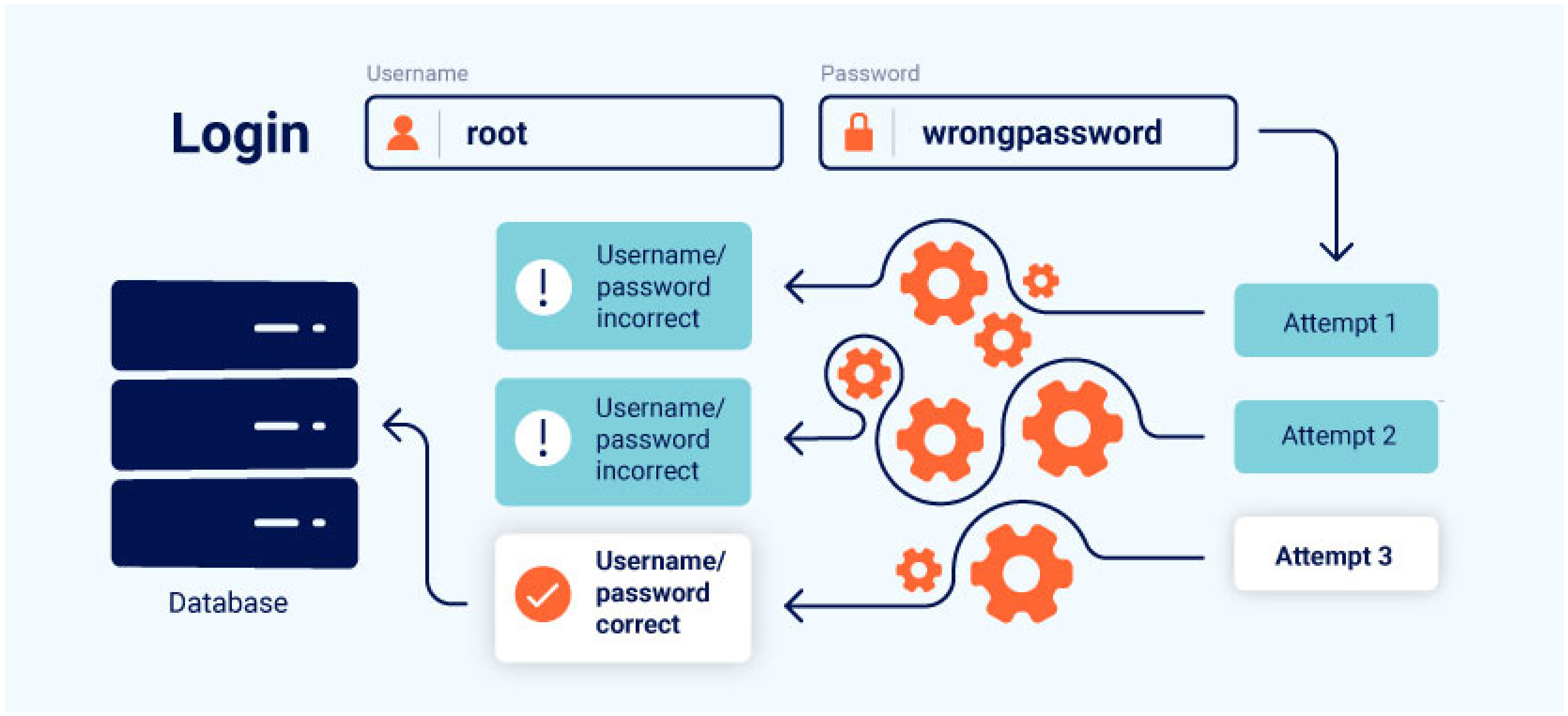
- take care about the user credentials
    - don't store passwords
    - use always HTTPS (redirect each HTTP request to HTTPS)
- don't trust user security
    - implement a good password policy
- prevent username enumeration
    - show generical and the same error messages for invalid username or password
    - or send these errors via email
- implement robust brute-force protection
    - implement CAPTCHA test with every login attempt
- check your verification logic
    - use well-known libraries don't implement AuthN by yourself
- don't forget supplementary functionalities
    - implement password reset/change properly
- implement proper multi-factor authentication

- use two different factors

BUSINESS LOGIC VULNERABILITIES

Business logic vulnerabilities are flaws in the design and implementation of an application that allows an attacker to elicit unintended behavior.
These flaws are generated by flawes assumptions about user behavior and about how the user will interact with the system.

Example:


Login
Username
root
Password
wrongpassword
Attempt 1
Attempt 2
Attempt 3
Username/ password incorrect
Username/ password incorrect
Username/ password correct
Database

In this example, we can see a business logic flaw that an attacker can use to perform an attack.
As we can see we can obtain different information by looking at the response time of the website because we have different response times for:

- incorrect username (attempt 1)

- incorrect password (attempt 2)
- correct credentials (attempt 3)

So an attacker can use it for username enumerations and then brute-force passwords.

How do business logic vulnerabilities arise?

Business logic vulnerabilities often arise because the design and development teams make flawed assumptions about how users will interact with the application. These bad assumptions can lead to inadequate validation of user input.

For example, if the developers assume that users will pass data exclusively via a web browser, the application may rely entirely on weak client-side controls to validate input.
These are easily bypassed by an attacker using an intercepting proxy.

- we can use ZAP, Javascript console, or Python requests to skip client-side validation

For this reason, when the attacker deviates from the expected user behavior the application fails to handle the situation safely.

Examples of business logic vulnerabilities

- excessive trust in client-side controls, we assume that the user will interact only with a web-client interface
- failing to handle unconventional input
- making flawed assumptions about user behavior
- domain specific flaws
- providing an encryption oracle, when the user can obtain his how encrypted input and can guess the encryption algorithm or key

Prevention

Business logic vulnerabilities are often specific to a given application

To prevent them we have to :

- understand the domain of the application
- don't assume about user behavior
- don't assume about the behavior of other parts of the application
- use DDD (domain driven development)