ENPM 809W
Introduction to Secure Software Engineering

Gananand Kini

Lecture 8

Authentication and Authorization related security bugs - Defenses

MITRE | SOLVING PROBLEMS FOR A SAFER WORLD™

Outline

- **Secure Authentication**
- **Password based authentication and two-factor authentication**
  - Have lockout periods
  - Don’t store passwords in the application memory (Protect secrets in user memory)
- **Secure Authorization**
- **Defining Roles**
- **Client-side versus server-side enforcement of security**
- **Authorization and Spheres of Control**
  - Ambient authority and confused deputy problem
  - Look at https://fsharpforfunandprofit.com/posts/capability-based-security//

Secure Authentication

Storing passwords: Things to avoid

- **When storing password information to authenticate an external user:**
- **Never store passwords as clear text (see Sony)**
  - Attackers can masquerade as any user on your system & many others (password reuse)
  - German chat platform Knuddels.de (“Cuddles”) fined €20K simply for storing user passwords in plain text*

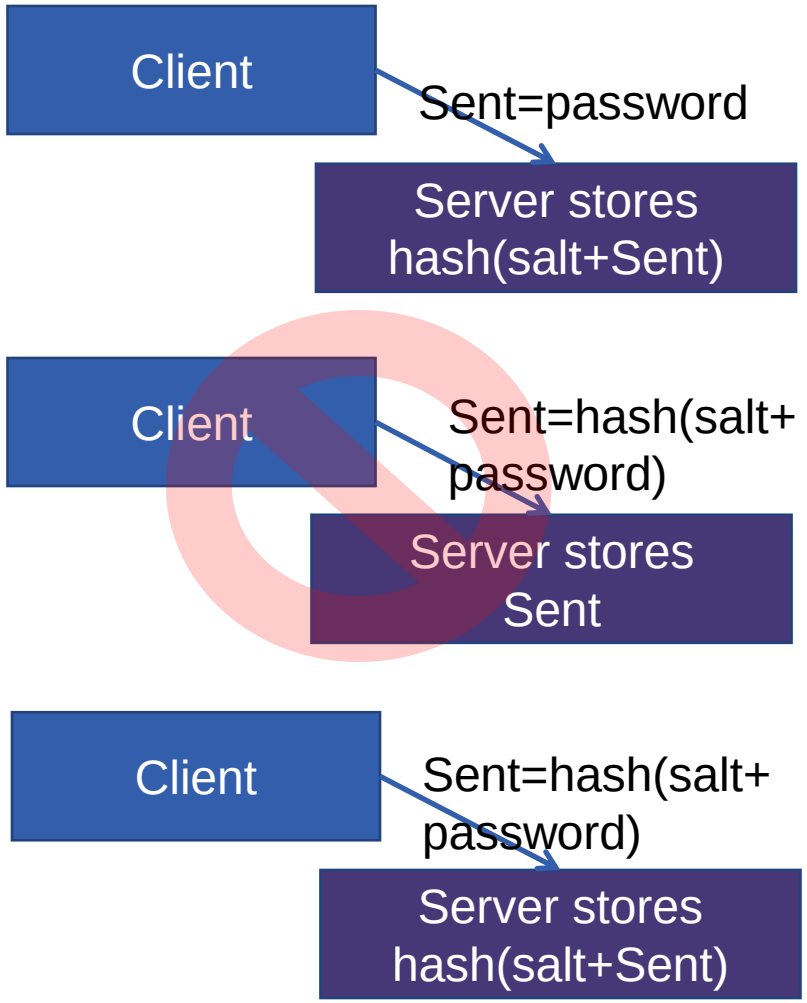
“By storing the passwords in clear text, the company knowingly violated its duty to ensure data security in the processing of personal data“*

- **Storing as simple hashes helps, but not much**
  - Attackers can pre-calculate hashes of likely passwords (“Rainbow table”), then compare
  - Attackers can easily see who has same passwords.

* “'Cuddly' German chat app slacking on hashing given a good whacking under GDPR: €20k fine PLAIN TEXT passwords showed up on file-hosting site” by Richard Chirgwin, 23 Nov 2018, The Register, https://www.theregister.co.uk/2018/11/23/knuddels_fined_for_plain_text_passwords/

Beware of client hashing password

- **Usual approach: Client sends normal password over encrypted channel**
- **Usually wrong for *only* client to create & send salted hash**
  - If attacker modifies/replaces client, modified client can just repeat the hashed value on server & get in
- **Okay to hash on *both* sides**
  - Hashing on client end means that attacker-controlled server can't see client's original password (bonus!) & stops revealing password length
  - Hashing on server end means that attacker-controlled client can't use stored hash to log in to this or other servers (usual purpose for hashed passwords)



Storing passwords: Basics of per-user salted hashes

- **Instead, store passwords as (iterated) per-user salted hashes**
  - Hash(user "salt" + user password) for *that* user
  - Since different users have different salts, precomputing fails
  - Attacker can't easily see if two users use same passwords
  - Salts need to have enough random bits

  NIST SP 800-132 requires 128+ bits & use of a cryptographically secure pseudo-random number generator (it lists specific ones) for salt
  - Salts don't need to be *encrypted*, just random
  - Use secure hash functions (SHA-512, not SHA-1 or MD5)
- **On user log in, just repeat process**
  - See if the result is the same as stored salted hash
- **Prevent social engineering attacks via "Can you tell me my password?" – instead offer to reset the password after identity verfication.**
- **To slow down guessing attacks, use iteration (key derivation)…**

Key Derivation (Iteration) Functions

- **“Key derivation functions” computes a *derived* key by iterating**
  - Repeatedly (iteratively) uses cryptographic hash, cipher, or HMAC, along with original data + salt, to generate *derived* key
  - *Intentionally slower* so brute-force attacks (password cracking) is hard
- **PBKDF2 (Password-Based Key Derivation Function 2)/RFC 2898 common**
  - Widely used: RSA Laboratories' Public-Key Cryptography Standards (PKCS) #5 v2.0, “Recommendation for Password-Based Key Derivation” NIST Special Publication 800-132, RFC 8018 (2017)
  - Relatively weak against GPUs and special hardware (ASICs)
- **Bcrypt**
  - Battle-tested
  - Stronger than PBKDF against special hardware (more RAM) but not strong
- **Argon2 (newer, especially strong at countering GPU & special hardware)**
  - Newer, winner of the Password Hashing Competition in July 2015
  - Usually use Argon2id variant, a hybrid of Argon2d (counters GPU cracking) & Argon2i (resists side-channel attacks)
  - For new starts Argon2id is generally recommended
  - Drawbacks?

Storing passwords: The bare minimum

- **An authenticator (e.g., server) must *never* store passwords in the clear. They must be *at least*:**
  - Per-user salted hashes, each salt a different **random** value
  - Use iterated key derivation (key stretching) function
  - Of those functions, PBKDF2 is okay but weak; better to use bcrypt, Argon2id, or maybe scrypt (consult latest info!)
  - You are *negligent* if you store password data for later authentication of external users in anything less
  - Prof. David A Wheeler's opinion
  - **Do not hash passwords yourself**; use an algorithm designed for it (and generally you shouldn't be implementing this algorithm!)
- **SANS "Securing Web Application Technologies" (SWAT) Checklist, Data protection section, agrees**
  - It requires that you "store user passwords using a strong, iterative, salted hash" (e.g., PBKDF2, bcrypt, Argon2id, scrypt)

What about passwords in memory?

- You are able to encrypt and limit the amount of time sensitive information stays around in dotnet.
- Use DataProtection APIs to protect sensitive data accepted by the software system.

Password policies

- **Have a good password policy for users!**
- **.NET provides easy ways of ensuring password policies for maintaining identity policies:**

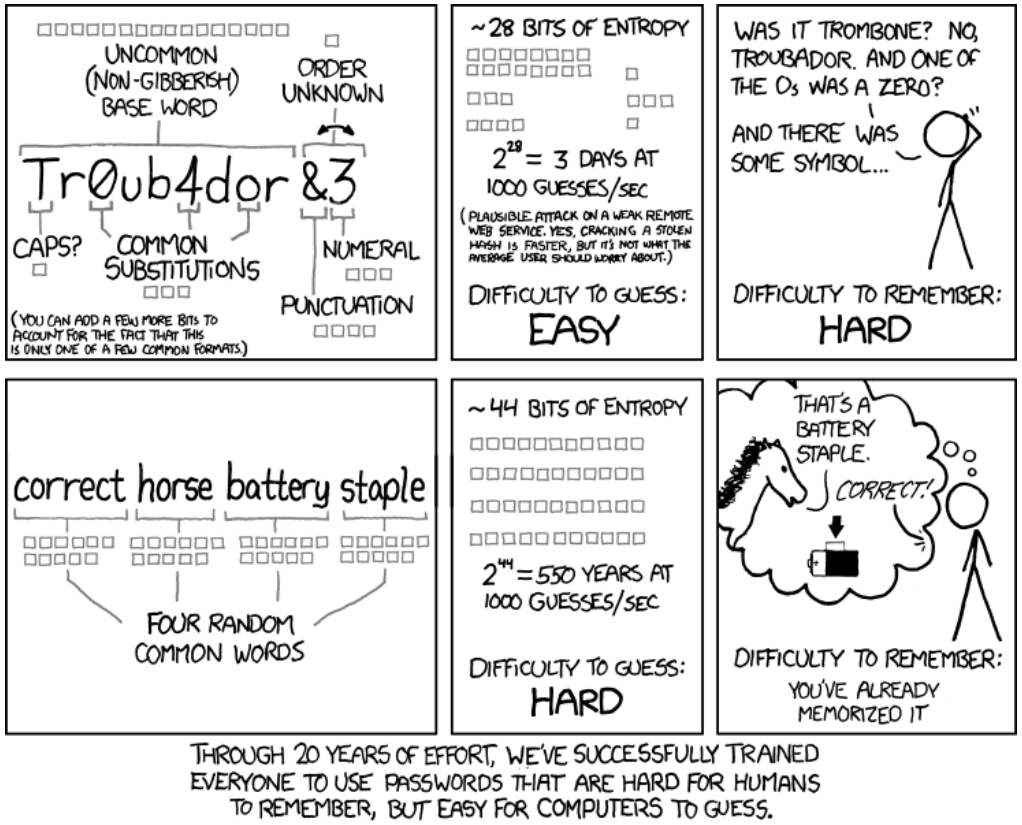
```
public class Startup {

    public void ConfigureServices(IServiceCollection
services) {

        // Add Identity services to the services container.

services.AddDefaultIdentity<ApplicationIdentityDbContext,
ApplicationUser, IdentityRole>(Configuration,

            o => {
                o.Password.RequireDigit = true;
                o.Password.RequireLowercase = true;
                o.Password.RequireUppercase = true;
                o.Password.RequireNonLetterOrDigit = true;
                o.Password.RequiredLength = 15;
            });
    }
}
```



Sources:
1. XKCD: Password Strength. https://xkcd.com/936/. Retrieved July 10, 2021.

What should you use?

- **Crypto algorithms often broken over time*; support switching algorithms**
- **Shared-secret encryption – do *not* use DES (key is too short) or 3DES**
  - For now, use (at least) AES
- **Hashing ("fingerprinting") – use SHA-2/SHA-3, do *not* use MD5 or SHA-1**
  - SHA-2's SHA-256 or SHA-512 are useful & widely available, but some concerns
  - Consider (also) using SHA-3; at least make it easy to switch to SHA-3
- **Public key**
  - Usually RSA (patents expired); again, make it easy to switch
  - Elliptic key cryptography has smaller key sizes, but patent issues lurk
  - Password storage: Argon2id; plausible alternatives are bcrypt & PBKDF2
  - Randomness - do *not* use most "random" functions (easily guessed by attacker)
  - Use cryptographic PRNGs, e.g., Linux /dev/random, Java SecureRandom()
  - Use existing good implementation, don't implement crypto yourself

Protect credentials

- ***Never* hardcode credentials (e.g., passwords) into code – put credentials in separate place (CWE-259)**
- **Don't check live credentials into a version control system – they stick around**
- **Encrypt credentials if you send them over network – never send them in the clear**
- **Attackers hunt for credentials!**
  - NSA "hunts sysadmins" when attacking networks, in particular, they hunt for the "credentials of network administrators and others with high levels of network access and privileges that can open the kingdom to intruders…. [including looking for] hardcoded passwords in software or passwords that are transmitted in the clear" [Zetter2016]

"NSA Hacker Chief Explains How to Keep Him Out of Your System" by Kim Zetter, 2016-01-28, Wired, a summary of a talk by Rob Joyce, chief of the NSA's Tailored Access Operations (TAO), http://www.wired.com/2016/01/nsa-hacker-chief-explains-how-to-keep-him-out-of-your-system/.

In case secrets are leaked on source control repositories …

- **Step 1: Revoke the secret in use. If being used by an organization, let senior developers at organization know and try to revoke use of the secret.**
- **Step 2: Try to remove the leak. Either delete the repository or make it private.**
- **Step 3. Rewrite git history and remove the offending file containing the secret. Can use BFG repo cleaner (Git only) to clean git history.**
- **Step 4: Check your access logs for the service that was using the leaked secret to see if any attackers are attempting to use the leaked one.**

Sources:
1. Mackenzie Jackson. Exposing secrets on Github: What to do after leaking credential and API keys. March 24, 2020. https://blog.gitguardian.com/leaking-secrets-on-github-what-to-do/. Retrieved July 10, 2021.

Excessive Logins Example: No restrictions

```
1   int validateUser (char *host, int port)
2   {
3     int isValidUser = 0;
4
5     char username[USERNAME_SIZE];
6     char password[PASSWORD_SIZE];
7
8     while (isValidUser == 0)
9     {
10      if (getUserInput(username,USERNAME_SIZE) == 0) error();
11      if (getUserInput(password,PASSWORD_SIZE) == 0) error();
12
13      isValidUser = AuthenticateUser (username, password);
14    }
15
16    return(SUCCESS);
17  }
```

The validateUser() method will continuously check for a valid username and password without any restriction on the number of authentication attempts made.

Excessive Logins Example: Potential Fix

```
1   int validateUser (char *host, int port)
2   {
3     int isValidUser = 0;
4     int count = 0;
5
6     char username[USERNAME_SIZE];
7     char password[PASSWORD_SIZE];
8
9     while ((isValidUser == 0) && (count < MAX_ATTEMPTS))
10    {
11      if (getUserInput(username,USERNAME_SIZE) == 0) error();
12      if (getUserInput(password,PASSWORD_SIZE) == 0) error();
13      isValidUser = AuthenticateUser (username, password);
14      count++;
15    }
16
17    if (isValidUser) return(SUCCESS);
18    else return(FAIL);
19  }
```

Secure Authorization

Secure Authorization

- **Following the principle of least privilege, only provide the minimal authorization required to only entities that need it!**
- **Put a lifetime on the authorization allowed time window such that it only lasts for the operation that it was designed for.**
- **Over doing authorization could allow attackers to take advantage and lockout legitimate users.**

Useful to think of everything as a cycle…

- During design you typically examine data flows and identify any cycles of data from user to software system and back and analyze which of those data flows need to be secured and what properties are necessary for the security objectives to be met.

- Authorization should cover most of the data flows that update or touch the state of the software system.

- Protect all accessible resources with authorization gates. Does the user or entity have sufficient authorization or privilege to access the resource?

Hide Sensitive Information (1)

- **Hide sensitive information (e.g., private, personally-identifying information, passwords, …)**
  - In transit (input & output)
  - At rest (stored)
- **In transit – web-based applications**
  - Typically use https: (HTTP on top of SSL or TLS)
  - Don’t allow GET to submit information – encoded in Request-URI, which is often logged
- **Encrypt any storage**
  - Doesn’t help if attacker breaks into the application
  - Does defend against someone who gets storage device without encryption keys
- **Encrypt passwords (with salted hashes – explain later)**

Hide Sensitive Information (2)

- **Don’t send it, if you don’t have to**
  - E.G., create special “local” ids when sending to other sites
  - Translate back
  - Make it hard for external sites to reveal info about your users, data, etc.

Authorization failure example: Authorization leak

```
1   my $q = new CGI;
2
3   my $message_id = $q->param('id');
4
5   if (!AuthorizeRequest(GetCurrentUser())
6   {
7      ExitError("not authorized to perform this function");
8   }
9
10  my $Message = LookupMessageObject($message_id);
11
12  print "From: " . encodeHTML($Message->{from}) . "<br>\n";
13  print "Subject: " . encodeHTML($Message->{subject});
14  print "\n<hr>\n";
15  print "Body: " . encodeHTML($Message->{body}) . "\n";
```

While the program properly exits if authentication fails, it does not ensure that the message is addressed to the user. As a result, an authenticated attacker could provide any arbitrary identifier and read private messages that were intended for other users.

Authorization failure example: Authorization Leak Fix

```
1   my $q = new CGI;
2
3   my $message_id = $q->param('id');
4
5   my $Message = LookupMessageObject($message_id);
6
7   if (AuthorizeRequest(GetCurrentUser(), $Message))
8   {
9      print "From: " . encodeHTML($Message->{from}) . "<br>\n";
10     print "Subject: " . encodeHTML($Message->{subject});
11     print "\n<hr>\n";
12     print "Body: " . encodeHTML($Message->{body}) . "\n";
13  }
14  else
15  {
16     ExitError("not authorized to view message");
17  }
```

Authorization failure example: Client side authorization

Client

```
1   $customerid = AskForCustomerId();
2   $address = AskForAddress();
3
4   writeSocket($sock, "$user AUTH $customerid\n");
5   $resp = readSocket($sock);
6
7   if ($resp eq "authorized")
8   {
9      # request is authorized, go ahead and update the info!
10     writeSocket($sock, "$user CHANGE-ADDRESS $customerid $address\n");
11  }
12  else {  print "ERROR: You're not authorized to change customer's address.\n"; }
```

Server

```
1   ($user, $cmd, $customerid, $address) = ParseRequest($sock);
2
3   if ($cmd eq "AUTH")
4   {
5      $result = AuthorizeRequest($user, $customerid);
6      writeSocket($sock, "$result\n");
7   }
8
9   elsif ($cmd eq "CHANGE-ADDRESS")
10  {
11    if (validateAddress($address)) {
12       $res = UpdateAddress($customerid, $address);
13       writeSocket($sock, "SUCCESS\n");
14    }
15    else { writeSocket($sock, "FAILURE -- address is malformed\n"); }
16  }
```

An attacker can bypass authentication by just sending a CHANGE-ADDRESS command.

Authorization failure example: Client-side authorization Fix

Client

```
1   $customerid = AskForCustomerId();
2   $address = AskForAddress();
3
4   writeSocket($sock, "$user CHANGE-ADDRESS $customerid $address\n");
5   $resp = readSocket($sock);
6
7   if ($resp ne "success")
8   {
9      error("$resp\n");
10  }
```

Server

```
1   ($user, $cmd, $customerid, $address) = ParseRequest($sock);
2
3   if ($cmd eq "CHANGE-ADDRESS")
4   {
5      $result = AuthorizeRequest($user, $customerid);
6      if ($result eq "authorized")
7      {
8         if (validateAddress($args))
9         {
10           $res = UpdateDatabaseRecord($customerid, $address);
11           writeSocket($sock, "SUCCESS\n");
12        }
13        else { writeSocket($sock, "FAILURE - malformed address\n"); }
14     else { writeSocket($sock, "FAILURE - not authorized\n"); }
15  }
```

Authorization failure example: Client-side authorization Fix

Client

```
1   $customerid = AskForCustomerId();
2   $address = AskForAddress();
3
4   writeSocket($sock, "$user CHANGE-ADDRESS $customerid $address\n");
5   $resp = readSocket($sock);
6
7   if ($resp ne "success")
8   {
9      error("$resp\n");
10  }
```

Server

```
1   ($user, $cmd, $customerid, $address) = ParseRequest($sock);
2
3   if ($cmd eq "CHANGE-ADDRESS")
4   {
5      $result = AuthorizeRequest($user, $customerid);
6      if ($result eq "authorized")
7      {
8         if (validateAddress($args))
9         {
10           $res = UpdateDatabaseRecord($customerid, $address);
11           writeSocket($sock, "SUCCESS\n");
12        }
13        else { writeSocket($sock, "FAILURE - malformed address\n"); }
14     else { writeSocket($sock, "FAILURE - not authorized\n"); }
15  }
```

Third party authorization

- Keeping with the principle of least privilege, only keep authorizations that are necessary.
- This especially applies to third party services in a software system as well.
- Issues in cloud environments where keys are kept around without expiry and no lifecycle management is performed for third party services.
- Revoke authorizations for all un-necessary or expired services.

Sources:

Defining Roles

- Roles have to have non-overlapping functions
- Typically used to separate responsibilities
- Apply the separation of duties security principle
- Overlapping roles can lead to transitive authorization weaknesses.
- Beware of the confused deputy problem.
- Use of tokens for authorization…

Secure Federation

Federation

- Giving a user the ability to use the same credentials across multiple services.
- Great when software system developers don’t want to store user credentials, because why bother when <Insert Big Software Company Here> already does?
- Typically involves one main Identity Provider (IP - that vouches for the identity of the user) or Asserting Party (AP)
- The other security domains that then trust the IP/AP are called the Relaying Parties (RP) or Service Providers (SP).
- The IP/AP typically will generate an Security Token Service (STS) token, which then gets passed to these Relaying parties or Service Providers.
- Example: OAuth authorization server and OpenID Connect. OpenID Connect is used to authenticate users, while OAuth is used to authorize services access to user’s resources. OpenID Connect extends OAuth by adding an ID token in addition to OAuth’s authorization token.

OAuth2 Roles

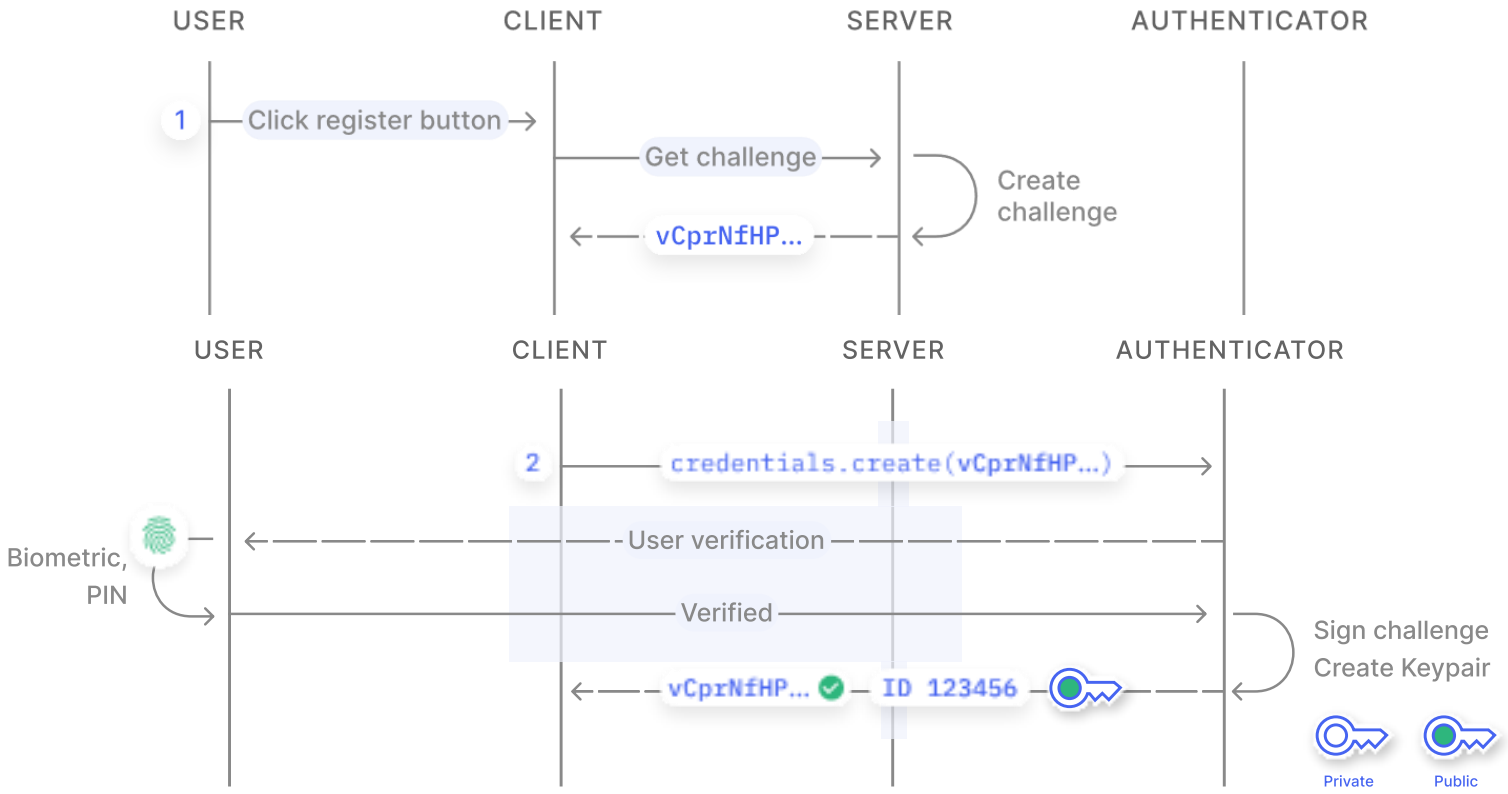
- Resource Owner – The owner of the data
- Resource Server – The server hosting the data (requires the access token and validates scope rules)
- Authorization Server – The Identity Provider or Authentication server
- Client / Application – The application or service provider that is requesting access to the resource or data.
- Scopes – More akin to roles within the software system. Uses dot separated strings. For example, contacts.readonly, contacts.importonly etc.

Secure Federation

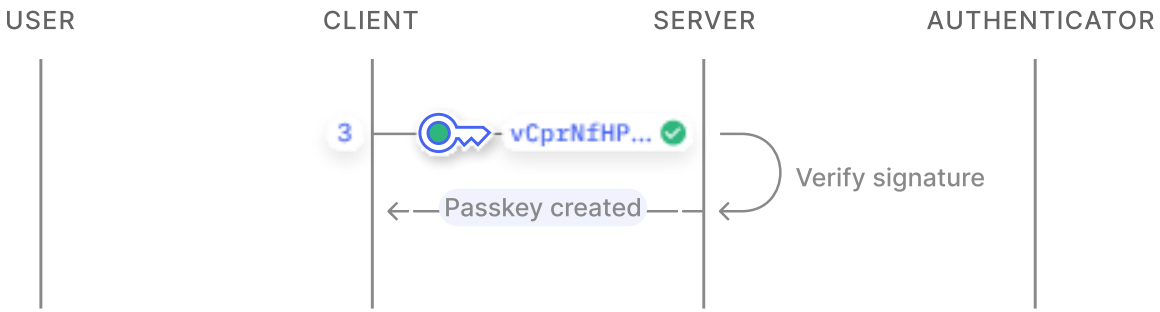
- Ensure you are encrypting all data in transit using TLS or equivalent between client, resource server and the authorization server.
- Validate all the tokens on the Resource server since they are generated on the authorization server.
- Check all principal identities using appropriate cryptography.
- Ensure you are only defining the scope to only the required level and enforce that scope on the resource server.
- Good management of consumer key and consumer secret.

FIDO Passkeys

- https://www.passkeys.io/technical-details



FIDO Passkeys contd…



What could go wrong here?

How to mitigate?

Are the following a good idea?

- Enforce assurance of registered authenticators
- Make risk decisions based on FIDO2 MDS metadata
- Something else?

Next time …

- **Session Management related bugs - Attacks**