Security - Examples

Usage of Security Tools

As we learned the basics of information security, we can understand the applications that use the security tools.

1. Password
2. SSH
3. SSL/TLS
4. HTTPS
5. GitHub Personal Access Tokens (PATs)

1. Password Storage

Servers do not store passwords in their original form. Instead, they:

- Create hash and salt values from the password
- Store only the hash and salt values
- Even when password files are hacked, hackers need to decrypt the original passwords from the hash values

Password	Hash
123456	e10adc3949ba59abbe56e057f20f883e
password	5f4dcc3b5aa765d61d8327deb882cf99
12345	827ccb0eea8a706c4c34a16891f84e7b
12345678	25d55ad283aa400af464c76d713c07ad
football	37b4e2d82900d5e94b8da524fbeb33c0
qwerty	d8578edf8458ce06fbc5bb76a58c5ca4
1234567890	e807f1fcf82d132f9bb018ca6738a19f
1234567	fcea920f7412b5da7be0cf42b8c93759
princess	8afa847f50a716e64932d995c8e7435a
1234	81dc9bdb52d04dc20036dbd8313ed055
login	d56b699830e77ba53855679cb1d252da
welcome	40be4e59b9a2a2b5dffb918c0e86b3d7
solo	5653c6b1f51852a6351ec69c8452abc6
abc123	e99a18c428cb38d5f260853678922e03
admin	21232f297a57a5a743894a0e4a801fc3

2. SSH (Secure Shell Protocol)

SSH is a cryptographic protocol that provides secure access to remote systems over unsecured networks.

- Runs by default on **TCP port 22**
- The **server (host)** listens for connection requests
- The **client** authenticates and, once verified, gains a secure shell session

How SSH Works

1. **Build an Encrypted Channel (using Diffie–Hellman)**

   > "We need to talk privately."
   > Both sides exchange keys using the **DH algorithm** to create a shared **session key** for encryption.

2. **Authenticate the User (Password or Public/Private Key)**

   > "We can now talk privately — who are you?"
   > After the secure tunnel is built, SSH verifies your identity using either:

   - a **password**, or a **public/private key pair**.

- **DH** → protects communication.
- **Authentication** → confirms identity.

1. DH → protects communication.

Step 1: Initial Connection (Protocol Negotiation)

1. The **server** lists supported encryption algorithms and protocol versions.
2. The **client** selects one that it also supports.
3. Both sides agree on this protocol and begin establishing a secure connection.

Step 2: Server Identity Verification (Public Key Download)

1. On the **first connection**, the client automatically downloads the server's **public key**.
2. SSH displays a message such as:

```
The authenticity of host 'serverip' can't be established.
RSA key fingerprint is SHA256:xxxx
Are you sure you want to continue connecting (yes/no)?
```

3. If you answer yes, the key is saved to:
   `~/.ssh/known_hosts`
4. On future connections, SSH compares the presented key with the stored one to detect tampering.

Example: GitHub’s Host Keys (stored in ~/.ssh/known_hosts)

```
github.com ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIOMqqnkVzrm0SdG6UOoqKLsabgH5C9okWi0dh2l9GKJl
github.com ssh-rsa    AAAAB3NzaC1yc2EAAAADAQABAAABgQCj7ndNxQowgcQnjshcLrqPEiiphnt...
github.com ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBEmKSENjQEezOmxkZMy...
```

Hint: How can the client know the server is not fake?
(Verification)

1. The **client** sends a **random challenge** (based on session data) to the server as part of the authentication handshake.
2. The **server** signs this challenge with its **private key** to prove it owns the key.
3. The **client** verifies the signature using the **stored public key** from `~/.ssh/known_hosts`.
4. If verification succeeds, the client confirms:

   > "This is the same trusted server I connected to before."

Step 3: Key Exchange (Diffie–Hellman → Shared Symmetric Key)

1. The server’s public key authenticates the host.
2. Once trust is verified, Diffie–Hellman generates a shared secret — the session key.
3. This symmetric key encrypts all further communication, ensuring confidentiality and integrity.

2. Authentication → confirms identity.

Once the encrypted tunnel is ready, SSH asks:

> "Now that our channel is secure, who are you?"

At this stage, you must **authenticate** — either by:

- **Password** → you prove identity with something you *know*, or
- **Private key/Public Key** → you prove identity with something you *have*.

Method 1: SSH Authentication Using Password

1. The user enters a **username** and **password**.
2. These credentials are sent **through the encrypted tunnel**, so even if intercepted, they cannot be read by third parties.
3. The server verifies the credentials and grants access.

Simple but weaker — vulnerable to brute-force and guessing attacks.

Method 2: Better Authentication Using Public Key

We can generate and use SSH keys to enable simple and secure user authentication:

Step 1. Generate a pair of keys on the local machine:

```
ssh-keygen -t rsa
```

- Creates two files:
  - Private key (id_rsa) – kept secret on your local machine.
  - Public key (id_rsa.pub) – safe to share.

Step 2. Copy the generated public key to the server:

```
ssh-copy-id user@serverip
```

The public key is copied server location.

```
~/.ssh/authorized_keys
```

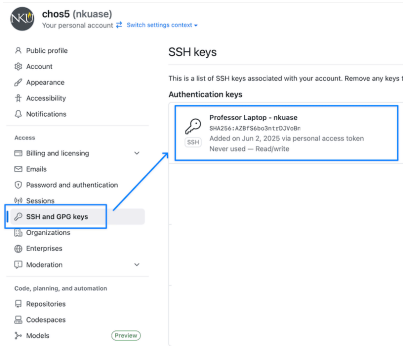
Step 3. Log in to the remote server using SSH without giving the password

1. **Server → Client:**
   The server generates a random challenge and **encrypts it with your public key**.
2. **Client → Server:**
   Your SSH client **decrypts** the challenge using your **private key** and sends back a **digital signature** — a cryptographic proof that it owns the private key.

3. **Server Verification:**
   The server verifies the signature using your **public key**.
   - ✅ If valid → you're authenticated (no password needed).
   - ❌ If invalid → access is denied.
4. **Secure Session:**
   After authentication, SSH negotiates a **session key** and switches to **symmetric encryption** (faster) for all subsequent communication.

Accessing GitHub Private Repository

To clone GitHub's private repository, we need to store our public key to GitHub.



Then we can clone your private repository:

```
git clone git@github.com:<your_user>/<your_repo>.git
```

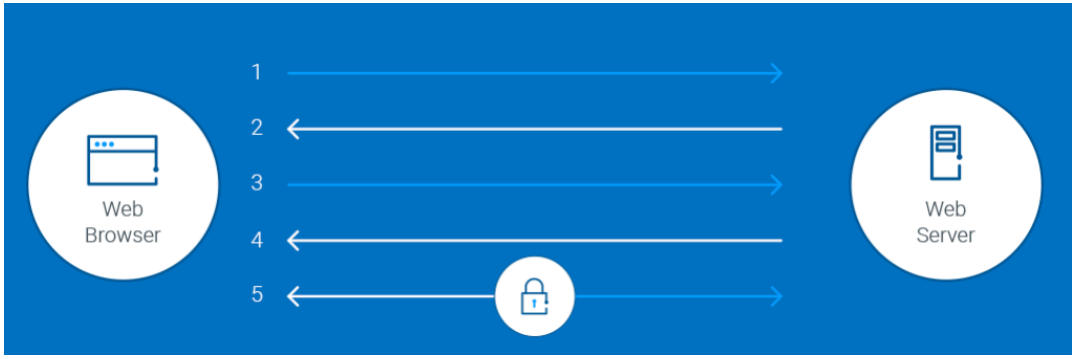
This uses your SSH keys for secure authentication and doesn’t require password prompts

3. SSL (Secure Sockets Layer)/TLS

The goal: **use a public key to share a symmetric key** between the browser and the server.

- **SSL uses hybrid encryption** — combining **public key** and **symmetric key** methods.
  - The **public key** encrypts the one-time **session key**.
  - The **session key** then encrypts all further communication (fast symmetric encryption).
- **SSL certificates** build trust by verifying the website's identity.

How SSL Works



1. **Browser → Server:**
   Browser connects to an HTTPS site and asks for identification.

2. **Server → Browser:**
   Sends its **SSL certificate**, containing its **public key**.

3. **Browser Checks:**
    - Confirms the certificate is from a trusted CA
    - Ensures it’s valid and matches the website name
4. **Browser → Server:**
    - Creates a **session key**
    - Encrypts it with the server’s **public key**
    - Sends it to the server
5. **Server → Browser:**
    - Decrypts the session key using its **private key**
    - Confirms receipt and starts secure communication

TLS (Transport Layer Security)

- **TLS** is the newer, stronger version of SSL.
- Works almost the same way, using encryption to protect data.
- The term “SSL” is often used loosely to mean “SSL/TLS.”

HTTPS (HTTP Secure)

- **HTTPS** = **HTTP + TLS** (or formerly **SSL**)
- Enables **secure communication** between browsers and web servers
- **Encrypts all data** exchanged, preventing eavesdropping or tampering

GitHub Personal Access Tokens (PATs)

A **Personal Access Token (PAT)** is like a **password replacement** for GitHub.

- It's a secret key you generate to authenticate yourself when using GitHub through **command-line tools (git, curl, VSCode, Docker, etc.)** instead of logging in via the browser.

Think of it as:

> "This token represents *you* when accessing GitHub via API or Git."

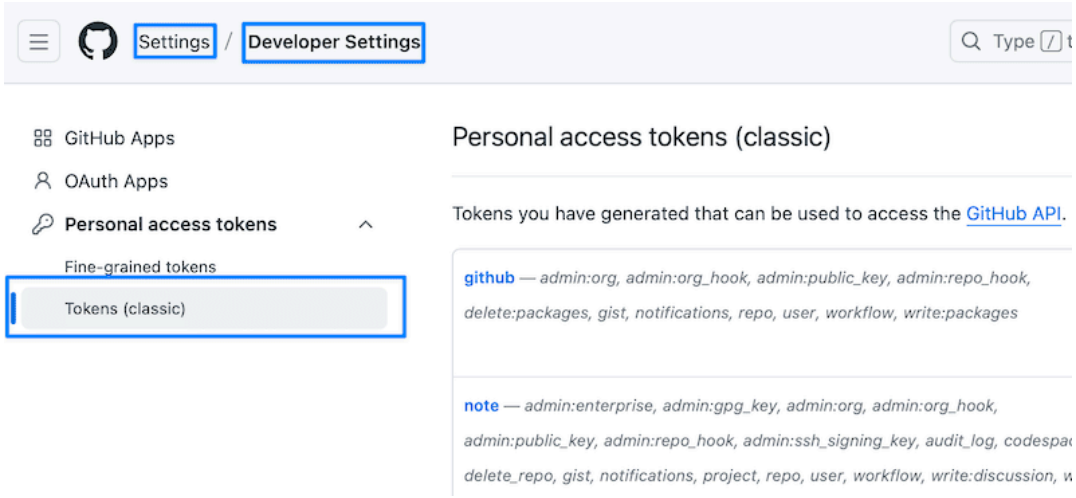
Why Tokens Instead of Passwords?

GitHub **no longer allows passwords** for Git operations and API access: Instead, you must use **tokens** because they are:

- ✅ **Revocable** – you can delete them anytime
- ✅ **Scoped** – limited to certain actions (read, write, repo, etc.)
- ✅ **Time-limited** – can automatically expire
- ✅ **More secure** – not your main GitHub password

How to use PATs Work

Step 1: Generate PAT from Settings -> Developer Settings

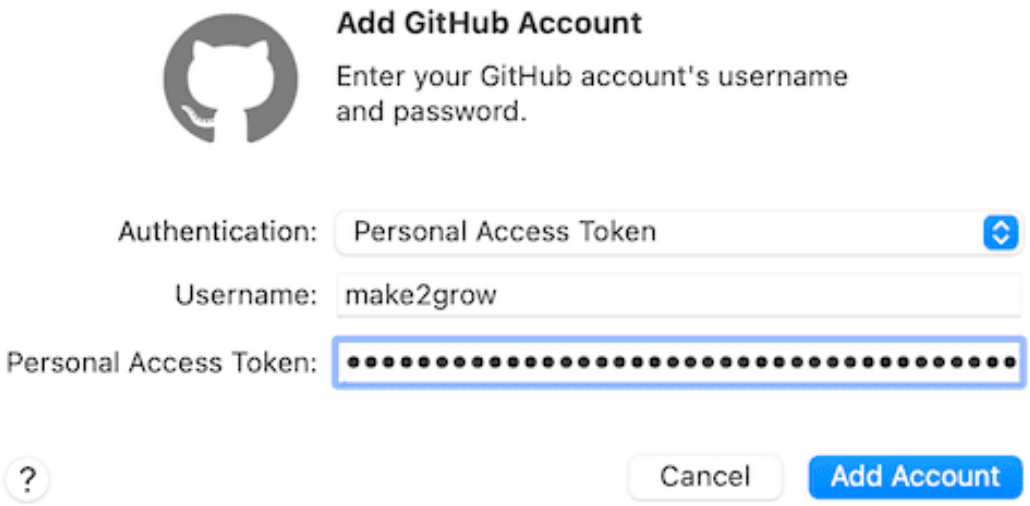


Step 2: Example Usage in Terminal

```
git clone https://oauth2:<YOUR-TOKEN-HERE>@github.com/<your_user>/<your_repo>.git
```

- Replace `<YOUR-TOKEN-HERE>` with your GitHub personal access token, `<your_user>` with your username, and `<your_repo>` with the repository name.

Step 2: Usage in Git Applications (ie., Git Tower)



PAT and the CIA Triad

CIA Principle	Relation to PAT	Explanation
Confidentiality	**Strongly related**	PAT replaces passwords, reducing exposure and protecting credentials.
Integrity	**Partially related**	Ensures that only authorized actions (within token scope) are performed.