SSH – Secure Shell

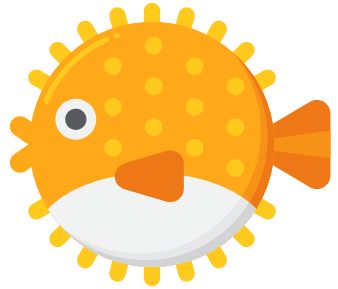
Attacks and Best Practices

Emanuel Duss <emanuel.duss@compass-security.com>

May 2026

Welcome!

- Content
  - SSH Introduction
  - Service Discovery
  - Information Disclosure
  - SSH Authentication (Host, User, SSH CAs, MFA, Security Keys / FIDO2)
  - SSH Port Forwarding
  - SSH Agent
  - SSH Session Multiplexing
  - Cryptographic Algorithms
- Emanuel Duss
  - Security Analyst / Penetration Tester
  - Compass Security, https://www.compass-security.com
  - emanuel.duss@compass-security.com
  - @compasssecurity.com | @compasssecurity
  - @emanuelduss.ch | @emanuelduss@infosec.exchange

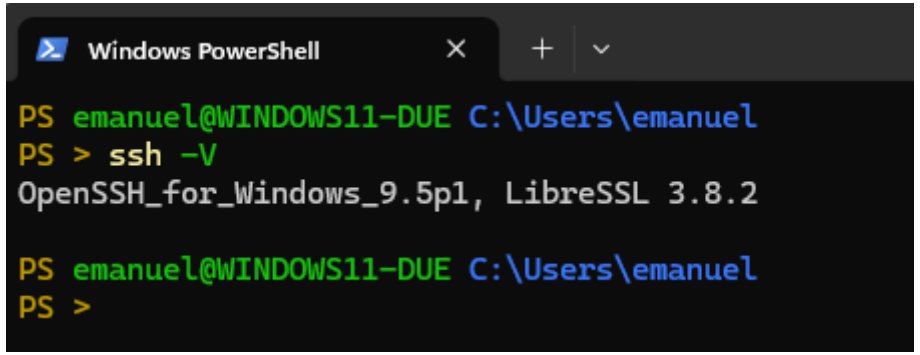


SSH Introduction

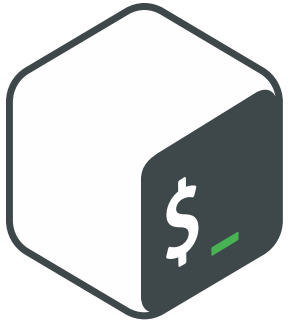
Secure Shell

- Establish authenticated & encrypted network connection to remote systems
- Used for
  - Remote login / shell access
  - Data transfer
  - Port forwarding
  - Traffic tunneling, proxying

- History
  - Replacement for plaintext protocols (`rsh`, `rlogin`, `telnet`, `ftp`, …)
  - Server / Client Architecture
  - SSH version 1 in 1995
  - SSH version 2 in 2006, standardized in various RFCs
  - OpenSSH is one implementation
  - Available on all major Linux/Unix OS
  - Available for Windows since 2017

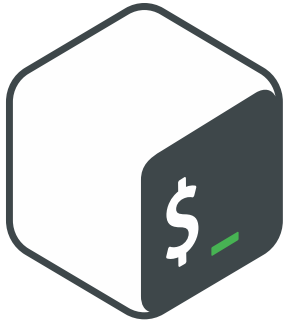


SSH Tools & Files



- Tools
  - Remote Operations: `ssh`, `scp`, `sftp`
  - Key Management: `ssh-add`, `ssh-keyscan`, `ssh-keygen`, `ssh-keysign`
  - Server side: `sshd`, `sftp-server`, `ssh-agent`
- Files
  - Server config: `/etc/ssh/sshd_config` and `/etc/ssh/sshd_config.d/`
  - Global client config: `/etc/ssh/ssh_config` and `/etc/ssh/ssh_config.d/`
  - Personal client config: `~/.ssh/config`
- Manpages
  - Everything is lovely documented!
  - `ssh(1)`, `ssh-add(1)`, `ssh-agent(1)`, `ssh-copy-id(1)`, `ssh-keygen(1)`, `ssh-keyscan(1)`, `ssh-keysign(8)`, `ssh-pkcs11-helper(8)`, `ssh_config(5)`, `sshd(8)`, `sshd_config(5)`

SSH Commands



- Establish remote shell session:

```
alice@beastie:~$ ssh puffy
Welcome to puffy.
alice@puffy:~$
```

- Execute command on remote system:

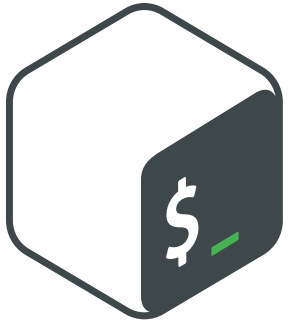
```
alice@beastie:~$ ssh puffy id
Welcome to puffy.
uid=1001(bob) gid=1001(bob) groups=1001(bob),27(sudo)
```

- Copy files remotely:

```
alice@beastie:~$ scp puffy:/etc/ssh/sshd_config .
alice@beastie:~$ scp .ssh/known_hosts puffy:.ssh/
alice@beastie:~$ scp puffy:notes aix:/tmp/
```

This is not a talk about SSH tricks and ninja magic. This would fill several other talks 🤪!

SSH Server Commands



- Show version number:

```
$ sshd -V
OpenSSH_10.0p2 Debian-5, OpenSSL 3.5.1 1 Jul 2025
```

- Parse and print SSH server configuration (use `-T` to also validate configuration)

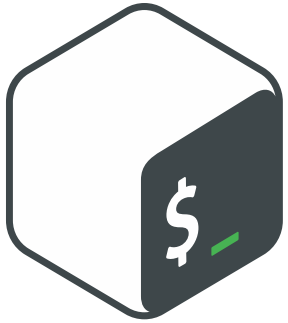
```
alice@beastie:~$ sudo sshd -G
port 22
addressfamily any
listenaddress [::]:22
[...]
```

Useful for hardening reviews!

- Show SSH server configuration for different scenarios:

```
alice@beastie:~$ /usr/sbin/sshd -G -C user=alice
[...]
GatewayPorts yes
[...]
```

SSH Server Commands



- Check validity of SSH server configuration:

```
alice@beastie:~$ /usr/sbin/sshd -t
/etc/ssh/sshd_config: line 18: Bad configuration option: ThisOptionDoesNotExist
/etc/ssh/sshd_config: terminating, 1 bad configuration options
```

- Start SSH server in foreground and print log to stdout (for debugging):

```
alice@beastie:~$ sudo /usr/sbin/sshd -D -e
debug1: Bind to port 22 on 0.0.0.0.
[...]
```

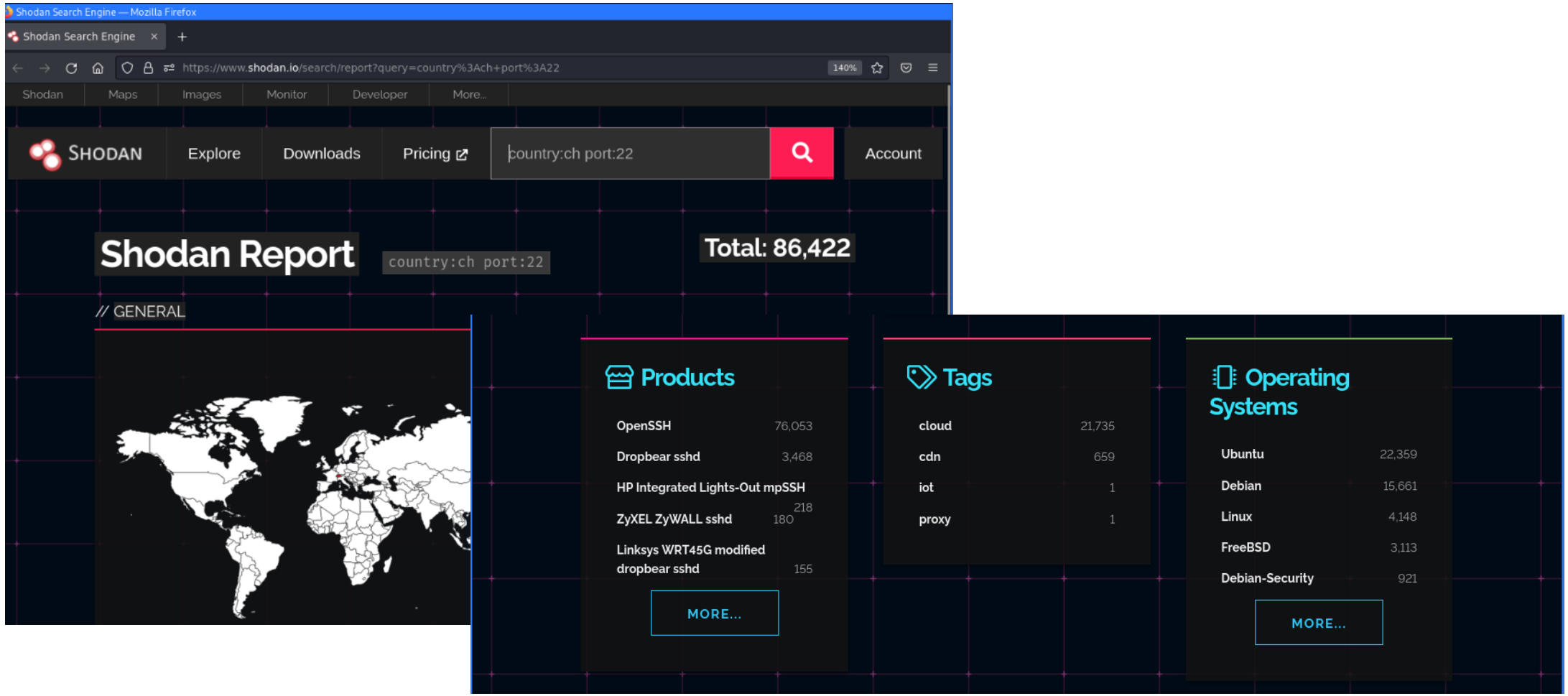
- Start in debug mode (no fork, terminates after 1 connection possible, multiple `-d` possible):

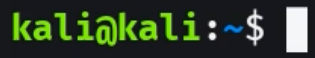
```
alice@beastie:~$ sudo /usr/sbin/sshd -d
debug1: Bind to port 22 on 0.0.0.0.
[...]
```

Service Discovery

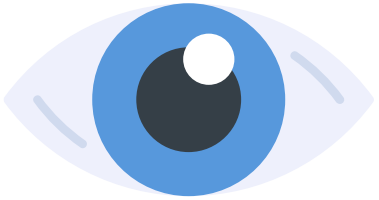
Network Discovery

- The SSH server runs on port `22/tcp` by default.
- They can easily be found.




kali@kali:~$

Service Exposure



- **Attack**
  - When exposed to the Internet or a network, external attackers can easily find your SSH servers.
  - They can then perform further attacks on this system.
- **Recommendations**
  - Only expose your servers when necessary.
  - Only expose your servers to allowed IP addresses when possible.
- **Note**
  - It’s possible to “hide” the server by using a random high port or port knocking.
  - This is security by obscurity and should not be used for security reasons.
  - Can be used to prevent non-targeted attacks from script kiddies.
  - Can be done, but security should not rely on this.
  - Instead, the system should be correctly configured and managed.
  - This includes hardening, patching, network segregation, logging, monitoring, alerting, …
  - Logging of system changes, file manipulation, user behavior, login sources, failed/successful logins, …

Information Disclosure

Information Disclosure

- The SSH version banner can be grabbed unauthenticated

```
$ ncat puffy.example.net 22
SSH-2.0-OpenSSH_10.0p2 Debian-8
```

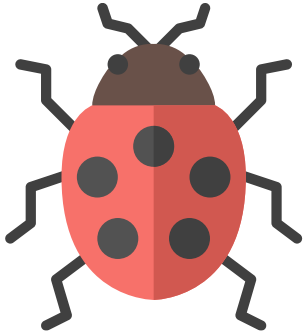
- **Attack**
  - An attacker could gain information about the system and perform targeted attacks.
- **Recommendations**
  - Hide what's possible.
  - But again: the security should not rely on hiding information!
  - Instead, patch your systems!
- The banner can't be disabled via SSH server config.
- Debian can suppress some information:

```
DebianBanner no
```

- Result

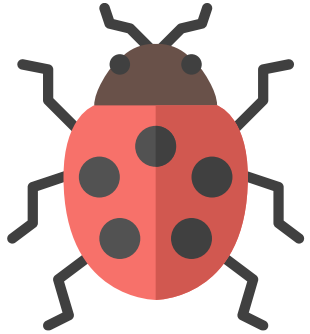
```
$ ncat debian.example.net 22
SSH-2.0-OpenSSH_10.0p2
```

Publicly Known Vulnerabilities



- OpenSSH < 5.2 (2009)
  - Recover up to 14 bits of plaintext in CBC encryption mode (success probability of 2^{-14})
- SSH SSH2 sshd <= 2.0.11 (SSH.com)
  - CVE-1999-1029: CVSSv2: 7.5 (High)
  - Attackers can perform brute-force attacks without being logged in the SSHD logfile by terminating the session before the last login attempt
  - https://marc.info/?l=bugtraq&m=92663402004280&w=2
- SSH Secure Shell for Servers 3.0.0 to 3.1.1 (SSH.com)
  - CVE-2002-1646: CVSSv2: 7.5 (High)
  - Attackers can override `AllowedAuthentications`
  - Attackers can login using a known password / perform brute-force attacks instead of using SSH keys

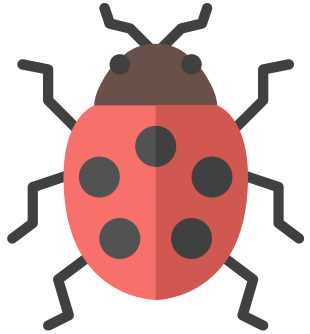
Publicly Known Vulnerabilities



- OpenSSH 6.8 to 6.9
  - CVE-2015-6565: CVSSv2: 7.2 (High),
  - Low privileged users could inject characters into another logged in user's terminal (TTY)
  - Injecting a command into an admin's terminal resulted in privilege escalation
  - Exploit PoC: https://www.openwall.com/lists/oss-security/2017/01/26/2

```
alice@dragonfly:~$ ./not_an_sshnuke /dev/pts/3
[*] Waiting for slave device /dev/pts/3
[+] Got PTY slave /dev/pts/3
[+] Making PTY slave the controlling terminal
[+] SUID shell at /tmp/sh

alice@dragonfly:~$ /tmp/sh --norc --noprofile -p
root@dragonfly:~# id
euid=0(root) groups=0(root)
```

Publicly Known Vulnerabilities



- OpenSSH Client 5.4 to 7.1
  - CVE-2016-0777 (information leak): CVSSv3: 6.5 (Medium)
  - CVE-2016-0778 (buffer overflow): CVSSv3: 8.1 (High)
  - Undocumented connection-resuming feature of the SSH client (roaming)
  - A malicious SSH server can read sensitive information from the client (e.g. private keys for logins)
  - Happens after the host key verification, so not possible to perform attack via machine-in-the-middle
  - Exploit Server: https://www.exploit-db.com/papers/39247
- OpenSSH Server 9.5p1 to 9.9p1
  - CVE-2024-6387: regreSSHion: RCE in OpenSSH's server (glibc-based Linux systems)
  - Get remote code execution (RCE) on a vulnerable server (exploit takes some hours/days)
  - Paper: https://www.qualys.com/2024/07/01/cve-2024-6387/regresshion.txt
- OpenSSH 6.8p1 to 9.9p1
  - CVE-2025-26465: MitM attack against OpenSSH's `VerifyHostKeyDNS`-enabled clients
  - Paper: https://www.qualys.com/2025/02/18/openssh-mitm-dos.txt

Information Disclosure

- A banner can be displayed to the users before successful login

```
Banner /etc/issue.net
```

- Example session

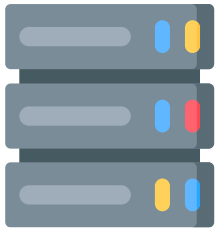
```
alice@beastie:~$ ssh puffy
**************************************************************
*      WARNING: Unauthorized access is prohibited.           *
**************************************************************

alice@puffy's password:
```

- **Attack**
  - An attacker could gain information about the system and perform targeted attacks.
- **Recommendations**
  - If this feature is used, the banner should not leak any sensitive / useful information to an attacker.

```
kali@kali:~$
```

SSH Authentication

Host Authentication



- Users must authenticate the server.
- The host has one or more host keys (ECDSA, Ed25519, RSA).
- On first connection, a host key fingerprint his shown:

```
alice@beastie:~$ ssh puffy
The authenticity of host ' puffy (203.0.113.23)' can't be established.
ED25519 key fingerprint is SHA256:aPDwXPsHTWTSebUW3jPkb4nH/lUGmvILmQsEkXKsY9c.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

- This can be accepted by typing `yes` or by pasting the fingerprint itself.
  - If accepted, host key is stored in users' known hosts file `~/.ssh/known_hosts`.
- These host keys are then trusted for future connections.
  - TOFU: Trust On First Use principle
- Users generally don’t verify this fingerprint.

Host Authentication

- Host key fingerprints can be stored in DNS (SSHFP resource record)

```
puffy.example.net IN SSHFP 4 2 5b7629b7b5906567aaf57b[...]f96079c3
```

- SSH clients can use SSHFP records this to verify host key

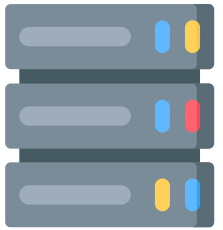
```
VerifyHostKeyDNS ask # or 'yes' to automatically accept
```

- Example session

```
alice@beastie:~$ ssh puffy
[...]
Matching host key fingerprint found in DNS.
Are you sure you want to continue connecting (yes/no)?
```

- Not always trustworthy
  - Without DNSSEC, the DNS resolver can’t verify authenticity of SSHFP record.
  - Without DNSSEC or DNS over TLS (DoT), a client can’t trust DNS resolver.

Host Authentication



- **Attack**
  - When a user accepts an arbitrary host key, the user cannot ensure the connection is established to the correct server.
  - An attacker between client & server can sniff and manipulate the network traffic (like credentials).
- **Recommendation**
  - Users should NEVER have to verify host keys themselves, since they don’t do it properly.
  - A centrally managed known hosts file should be used (default is `/etc/ssh/ssh_known_hosts`)
  - Alternatively, an SSH host CA could be used.

User Authentication

- Different types and combinations of user authentication methods
  - Host-based authentication
  - Password authentication
  - Public key authentication
  - GSSAPI (used for single sign-on like Kerberos or NTLM)
  - Keyboard interactive (via PAM, used e.g. for MFA)
  - Combination using either public key & password or public key & MFA
- Example server config (`sshd_config`)

```
AuthenticationMethods password # Password only
AuthenticationMethods publickey # Public key only
AuthenticationMethods keyboard-interactive # Authentication via PAM
AuthenticationMethods publickey,password publickey,keyboard-interactive # MFA
AuthenticationMethods publickey,publickey # Two different public keys
```

User Authentication

- **Attack**
  - When password authentication is enabled, attackers can try to online brute-force passwords.
- **Recommendation**
  - Generally, if the password is strong (long and random), password authentication is OK.
  - However, users tend to choose weak passwords.
  - Furthermore, passwords may leak through data breaches, phishing attacks, password reuse, …
  - Therefore, enforce public key authentication or MFA.
  - When using passwords, a brute-force protection should be implemented (fail2ban, SSH's PerSourcePenalties features)

```
kali@kali:~$
```

PerSourcePenalties Brute-Force Protection

- Brute-Force protection is implemented using the `PerSourcePenalties` option
  - Introduced in OpenSSH 9.8 (2024-07-01)
  - Penalties for conditions that may indicate attacks
  - Applies to the source IP address of the client
- Example server config (`sshd_config`)

```
PerSourcePenalties ↲
  crash:90    authfail:5    noauth:1    grace-exceeded:10    refuseconnection:10 ↲
  max:600    min:15
PerSourcePenaltyExemptList  10.5.23.0/24
```

Penalty durations in seconds

Crashing server

Failed auth

Disconnect without auth

No auth after `LoginGraceTime`

Refused by `RefuseConnection` option in `Match` block

Maximum accumulation of penalties

Enforcement after amount of penalties

These are the default values

Exclusion

PerSourcePenalties Brute-Force Protection

- Simple example to visualize the penalty behavior (for the source IP address)

```
PerSourcePenalties authfail:10 min:15
```

  - Every disconnect after failed authentication adds 10 seconds time penalty
  - Every second, the penalty time is decreased 1 second
  - It the penalty time is > 15 seconds, the client are blocked

- Example
  - The client connects, performs 3 failed logins and disconnects: penalty = 10
  - The client waits 2 seconds: penalty = 8
  - The client connects, performs 3 failed logins and disconnects: penalty = 18
  - The penalty is now > 15
  - The client has to wait 18 seconds until the client can connect again

- Reloading the SSH daemon drops current penalties

PerSourcePenalties Brute-Force Protection

Server log:

```
00:03 Failed password for bob from 10.23.23.5 port 40874 ssh2
00:06 Failed password for bob from 10.23.23.5 port 40874 ssh2
00:09 Failed password for bob from 10.23.23.5 port 40874 ssh2
00:10 Connection closed by authenticating user bob 10.23.23.5 port 40874 [preauth]

00:13 Failed password for bob from 10.23.23.5 port 50523 ssh2
00:16 Failed password for bob from 10.23.23.5 port 50523 ssh2
00:17 Connection closed by authenticating user bob 10.23.23.5 port 50523 [preauth]

00:20 Failed password for bob from 10.23.23.5 port 60042 ssh2
00:21 Connection closed by authenticating user bob 10.23.23.5 port 60042 [preauth]

00:21 srclimit_penalise: 10.23.23.5/32: activating ipv4 penalty of 20 seconds for penalty: failed
authentication
drop connection #0 from [10.23.23.5]:55116 on [10.23.23.10]:22 penalty: failed authentication
```

0 + 10 = 10

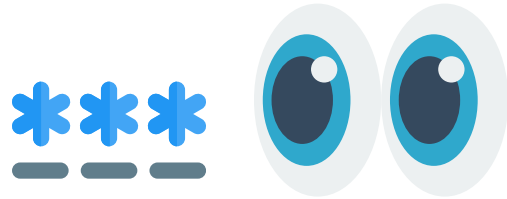
10 - 7 + 10 = 13

13 - 4 + 10 = 20

The client cannot connect again:

```
$ ssh alice@puffy
Connection closed by 10.23.23.10 port 22

$ ncat puffy 22
Not allowed at this time
```

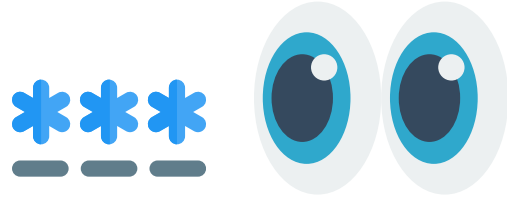
Password Sniffing as Root



- From RFC 4251: SSH Protocol Architecture:

```
RFC 4251                SSH Protocol Architecture                January 2006

[...]

9.4.5.  Password Authentication

   The password mechanism, as specified in the authentication protocol,
   assumes that the server has not been compromised.  If the server has
   been compromised, using password authentication will reveal a valid
   username/password combination to the attacker, which may lead to
   further compromises.

[...]
```

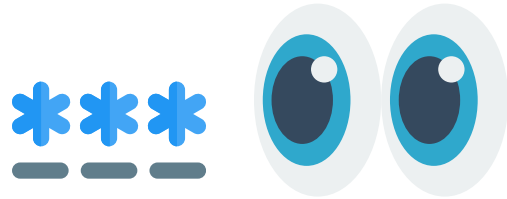
Password Sniffing as Root



- **Attack**
  - An attacker with root access on the server can read the password when a user authenticates.
  - If this password is valid on another system (when the same password is configured or when LDAP is used), an attacker can use the password and use it for lateral movement.
  - Common scenarios
    - Server owner who is admin on only one system.
    - External partner who has admin access on only some systems.
- **Recommendation**
  - Enforce public key authentication
  - Never reuse-passwords (not possible for LDAP authentication but for local accounts)
  - OTP-based MFA still leaks password and OTP but may be an option when the token is different on every server

```
9.4.5.  Password Authentication
[...]
   This vulnerability can be mitigated by using an alternative form of
   authentication.  For example, public key authentication makes no
   assumptions about security on the server.
[...]
```

Password Sniffing as Root



- Attacker

```
bob@tux:~$ sudo strace -p "$(pgrep -f /usr/sbin/sshd)" -f -e trace=write
strace: Process 19531 attached
strace: Process 19594 attached
[...]
[pid 19595] write(5, "\0\0\0\4alice", 8) = 8
[...]
[pid 19595] write(5, "\0\0\0\10P@ssw0rd", 12) = 12
[...]
^C
bob@tux:~$ ssh alice@puffy
alice@puffy's password: ********
Welcome to puffy.
```

- User

```
alice@beastie:~$ ssh alice@tux
alice@tux's password: ********
```

```
root@linux-srv-02:~#
```

Session Sniffing as Root

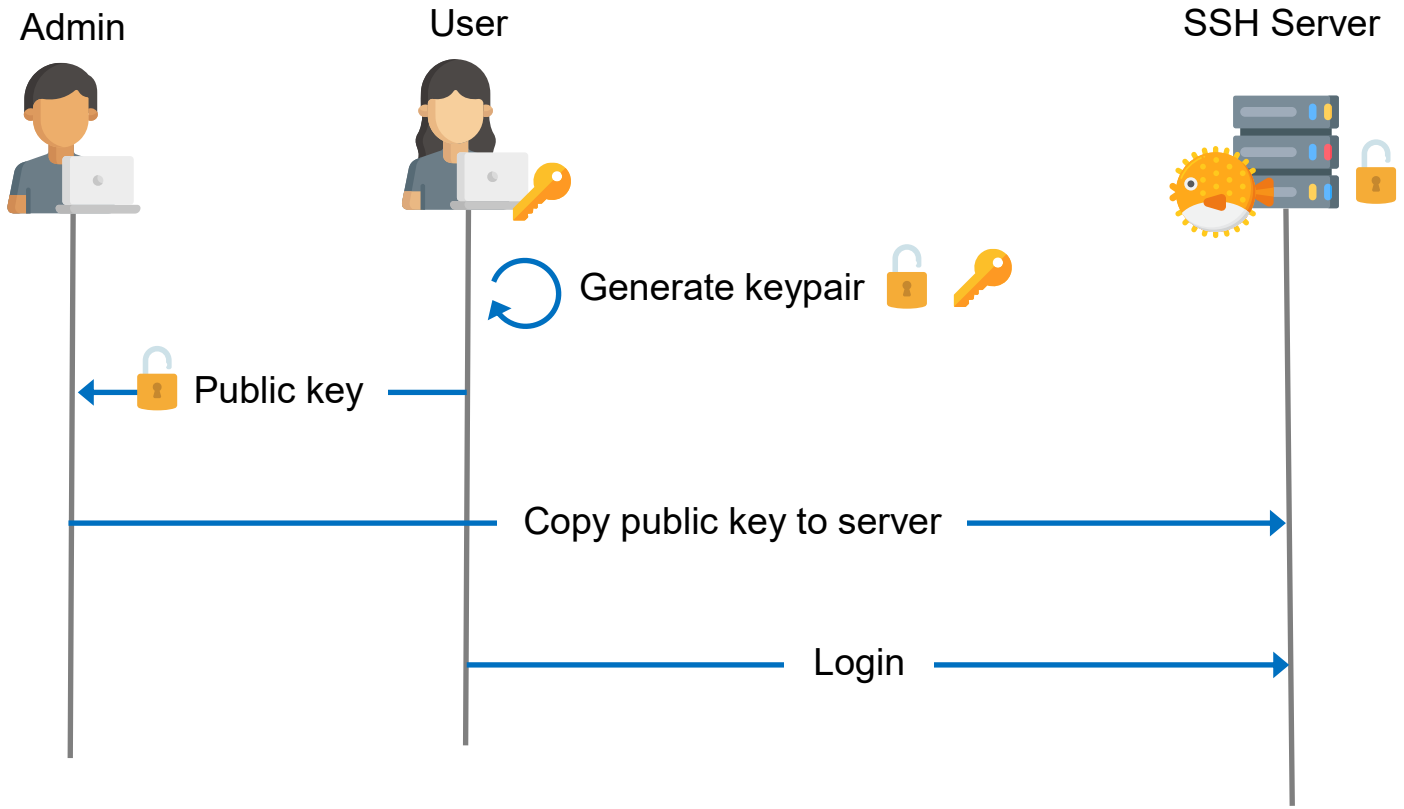
- The `root` user can by design do everything on a system.
- Tools like `sshspy` can show the terminal of logged in users in real-time
  - It's a small `bash` script which uses `strace` to get all the information.
  - https://github.com/InfosecMatter/Scripts/blob/master/sshspy.sh
  - For newer SSH versions, change the regex on line 8 to `sshd.*@`

- **Attack**
  - An attacker with root access on the server can see/read everything other users do on the system.
  - This also includes typed passwords.
- **Recommendation**
  - Keep in mind who is `root` on the system and act accordingly.
  - Never store data / process information / type passwords on untrusted systems.


root@server:~
#

Public Key Authentication

- Passwordless authentication
- User generates keypair
- Private key on client
- Public key on target server
- Login using key



Public Key Authentication

- Key pairs are stored in users's `~/.ssh` directory.
- Different algorithms are available (ECDSA, Ed25519, RSA).
- Private keys can be encrypted using a passphrase
- Private keys can be stored on secure devices
  - Smart Cards (PKCS #11)
  - Hardware Keys / Hardware Authenticators / FIDO2 keys (e.g. YubiKey, Nitrokey, …)
  - TPM
- Public keys are stored in the authorized keys file on the target server
- By default, two authorized keys files are used (`sshd_config`):

```
AuthorizedKeysFile
    Specifies the file that contains the public keys used for user authentication
    [...] The default is ".ssh/authorized_keys .ssh/authorized_keys2".
```

- Instead of distributing lots of keys, an SSH user CA could be used.

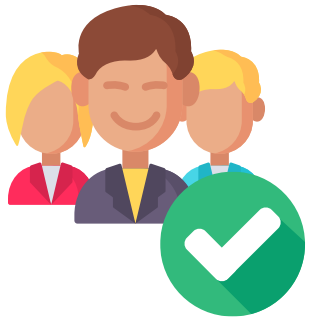
Public Key Authentication

- **Attacks**
  - An attacker who can perform privilege escalation on a host where private keys are stored can steal them.
  - If these are not passphrase protected, these can be used for authentication.
  - An attacker can try to offline brute-force the private key passphrase.
  - The authorized keys files can be used as a “backdoor” (attacker places own key for persistence)
- **Recommendation**
  - Keys should not be stored on systems where other user’s have access to (e.g. jump hosts, source code repositories, scripts, network shares, …).
  - Keys should be protected using a strong passphrase or placed on a secure device.
  - Explicitly define the authorized keys file
  - The authorized keys files should be centrally managed and monitored.
- Example server config (`sshd_config`)

```
AuthorizedKeysFile .ssh/authorized_keys
```

```
bob@linux-srv-01:~$
```

Allowed Users & Groups



- By default, all users are allowed to login if they have a login method configured.
  - Users with a password / SSH keys configured
- **Attacks**
  - RCE in a web application → change password via «`echo alice:P@ssw0rd | chpasswd`» → shell
  - Arbitrary file write in a web application → write own SSH keys to `~/.ssh/authorized_keys` → shell
- **Recommendation**
  - Shared accounts should generally not be used → disable root login via SSH.
  - Only authorized users/groups should be able to establish an SSH connection.
  - Restrict SSH access to explicitly allowed users or groups.
- Example server config (`sshd_config`)

```
AllowUsers alice bob
AllowGroups sysadmins ssh-users
PermitRootLogin no # Implicit, but enforce it even if root is in allowed group
```

Public Key Information Leakage

- Public keys are public (as the name says 😉).
- Without the private key, you can't use them to authenticate.
- Keys might be exposed where you don't expect.
- E.g. all GitHub user keys are public:

```
$ curl https://github.com/emanuelduss.keys
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIHUpSBIZZ8EJy6hGGF0x9uypjJhLPuZNRFeYEIZtyKT4
```

- Maybe also in other public repositories or internal LDAP directory
- Also, you send your public key(s) to every server you try to authenticate.
  - By default: `id_rsa`, `id_ecdsa`, `id_ecdsa_sk`, `id_ed25519`, `id_ed25519_sk`, `id_dsa` in `~/.ssh/` & all keys loaded into the SSH agent.
- The user's public key is sent encrypted over the network after the host authentication.
  - A machine-in-the-middle attacker can't read the public key.

Public Key Information Leakage

- If you login to arbitrary servers, you do expose your public key.
- PoC by Filippo Valsorda: https://words.filippo.io/ssh-whoami-filippo-io/
- It checks if your sent public key is on GitHub and shows your username:

```
$ ssh whoami.filippo.io
    +-------------------------------------------------------------------+
    |              _o/ Hello Emanuel Duss!                              |
    |                                                                   |
    |  Did you know that ssh sends all your public keys to any server   |
    |  it tries to authenticate to?                                     |
    |                                                                   |
    |  We matched them to the keys of your GitHub account,              |
    |  @emanuelduss, which are available via the GraphQL API            |
    |  and at https://github.com/emanuelduss.keys                       |
    |                                                                   |
    |  -- Filippo (https://filippo.io)                                  |
    |                                                                   |
    |  P.S. The source of this server is at                             |
    |  https://github.com/FiloSottile/whoami.filippo.io                 |
    +-------------------------------------------------------------------+
```

Public Key Information Leakage

- It’s possible to verify if a public key can be used to login or not, even without the private key:

```
alice@beastie:~$ ssh -v -i key.pub root@puffy
[...]
debug1: Offering public key: key.pub ED25519
SHA256:L619XZboqfh8ui85GqTBRPCpwkrxECR3WOoIagTWeno explicit
debug1: Authentications that can continue: publickey
debug1: No more authentication methods to try.
[...]

alice@beastie:~$ ssh -v -i key.pub alice@puffy
[...]
debug1: Offering public key: key.pub ED25519
SHA256:L619XZboqfh8ui85GqTBRPCpwkrxECR3WOoIagTWeno explicit
debug1: Server accepts key: key.pub ED25519
SHA256:L619XZboqfh8ui85GqTBRPCpwkrxECR3WOoIagTWeno explicit
[...]
```

Public Key Information Leakage

- This process can be automated:

```
$ sudo nmap -p 22 --script ssh-publickey-acceptance --script-args
'ssh.usernames={"root", "alice"}, publickeys={"./id_rsa1.pub", "./id_rsa2.pub"}' puffy

Nmap scan report for puffy (10.5.23.42)
22/tcp open  ssh     syn-ack
| ssh-publickey-acceptance:
|   Accepted Public Keys:
|_    Key ./id_rsa1 accepted for user alice
```

- Use Case
  - You find 50 passphrase encrypted SSH keys during an internal pentest
  - The pentest ends tomorrow and you only want to crack keys which are useful for you.
  - Which one do you want to crack?

Public Key Information Leakage

- **Attacks**
  - An attacker can get access to your public key when you login on an attacker-controlled system.
- **Recommendation**
  - Use different keys for different services (e.g. internal systems, external systems, external partners, 3rd party services, …) and always only use one to connect.
  - Since public keys are meant to be public, the security should not rely on this.
  - Also good to know for OPSEC reasons in red teamings

- This can be done via the following SSH config (`~/.ssh/config`):

```
Host github.com
  IdentityFile .ssh/id_ed25519_github
Host *.example.net
  IdentityFile .ssh/id_ed25519_internal
Host *
  IdentityFile .ssh/id_ed25519_external
```

```
emanuel@x1:~
$
```

Private Key Information Leakage

- You can specify a comment during key generation (default is `username@hostname`):

```
$ ssh-keygen -t ed25519 -C mykey
Generating public/private ed25519 key pair.
[...]
```

- The comment is stored inside the public key file:

```
$ cat .ssh/id_ed25519.pub
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIBPhRbyUNQirYCo6GrODJ++Jl/MUtTIPW1dBafg8vLu+ mykey
```

- The comment is also stored inside the private key as well and can be shown:

```
$ rm .ssh/id_ed25519.pub
$ ssh-keygen -y -f .ssh/id_ed25519
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIBPhRbyUNQirYCo6GrODJ++Jl/MUtTIPW1dBafg8vLu+ mykey
```

- This is usually no problem, since private keys should be kept private.
- But keep this in mind when you generate private keys which someone else could read (e.g. for trainings, CTF challenges, OPSEC during red team engagements, …)

SSH Certificate Authorities (CAs)

- Default setup
  - For every host, the users must trust the SSH host key (configure or just accepting it)
  - For every user, the hosts must trust the SSH public key (configure public key on the hosts)
- SSH CA
  - Sign keys to produce certificates
  - Can be used for user and/or host authentication.
- Trust
  - To trust a certificate, the CA public key must be trusted by the SSH server / client

SSH Host CA

- On the CA server, generate SSH host CA public & private key:

```
trent@ca:~$ ssh-keygen -t ed25519 -f ca/ca_host_key
```

- Install the public key in the global known hosts file on all clients (`/etc/ssh/ssh_known_hosts`):

```
@cert-authority *.example.net ssh-ed25519 AAAAC3NzaC1lZ[...]xxu8==
```

- Copy the servers public key to the CA server:

```
trent@ca:~$ scp puffy:/etc/ssh/ssh_host_ed25519_key.pub keys/host-puffy.pub
```

- Sign the public host key:

```
trent@ca:~$ ssh-keygen -s ca/ca_host_key -I puffy -h -n puffy.example.com keys/host-puffy.pub
```

- Copy the signed public host key back to the server:

```
trent@ca:~$ scp keys/host-puffy-cert.pub puffy:/etc/ssh/ssh_host_ed25519_key-cert.pub
```

- Install on SSH server (`sshd_config`):

```
HostCertificate /etc/ssh/ssh_host_ed25519_key-cert.pub
```

- Clients now automatically trust all signed host keys.

SSH User CA

- On the CA server, generate SSH user CA public & private key:

```
trent@ca:~$ ssh-keygen -t ed25519 -f ca/ca_user_key
```

- Copy the public key to the SSH servers:

```
trent@ca:~$ scp ca/ca_user_key.pub puffy:/etc/ssh/ca_user_key.pub
```

- On the SSH servers, trust the SSH user CA for authentication (`sshd_config`):

```
TrustedUserCAKeys /etc/ssh/ca_user_key.pub
```

- Copy the public key of a user to the CA:

```
trent@ca:~$ scp puffy:/home/alice/.ssh/id_ed25519.pub userkeys/alice.pub
```

- Sign the public key of a user:

```
trent@ca:~$ ssh-keygen -s ca/ca_user_key -I alice -n alice userkeys/alice.pub
```

- Copy the signed public host key back to the server:

```
trent@ca:~$ scp userkeys/alice-cert.pub puffy:/home/alice/.ssh/id_ed25519-cert.pub
```

- The key can now be used to login.

SSH User CA

Example certificate:

```
alice@puffy:~$ ssh-keygen -L -f ~/.ssh/id_ed25519-cert.pub
attacker-cert.pub:
        Type: ssh-ed25519-cert-v01@openssh.com user certificate
        Public key: ED25519-CERT SHA256:SypV+Pjx65Jy/t5YL1tuE0xhB3SQCF22UAqB2siJCsU
        Signing CA: ED25519 SHA256:Tjd72wG6HJCtWuKYUEi/Lcrd3b5vLaWhzQMTP39hQiY (using ssh-
ed25519)
        Key ID: "something"
        Serial: 0
        Valid: forever
        Principals:
                alice
Critical Options: (none)
        Extensions:
                permit-X11-forwarding
                permit-agent-forwarding
                permit-port-forwarding
                permit-pty
                permit-user-rc
```

Could be multiple usernames

SSH User CA

- It’s possible to define which users are allowed for certificate authentication.
- So not every user can use their certificate on every server that trusts the CA.
- Configure which users are trusted by the CA in the SSH server config:

```
AuthorizedPrincipalsFile /etc/ssh/auth_principals
```

- Configure allowed users:

```
alice@puffy:~$ echo "alice" | sudo tee /etc/ssh/auth_principals
alice@puffy:~$ echo "bob" | sudo tee -a /etc/ssh/auth_principals
```

- User `alice` and `bob` (principal in the certificate) are now allowed to login.

SSH User CA

- It's possible to define which key can login as which user.
- Define `AuthorizedPrincipalsFile` in the SSH server config (`sshd_config`):

```
AuthorizedPrincipalsFile /etc/ssh/auth_principals/%u
```

- Define which principal (username from the certificate) is allowed to login as which user:

```
alice@puffy:~$ sudo mkdir /etc/ssh/auth_principals/
alice@puffy:~$ echo "alice" | sudo tee /etc/ssh/auth_principals/admin
alice@puffy:~$ echo "bob" | sudo tee -a /etc/ssh/auth_principals/admin
```

- User `alice` and bob (principal in the certificate) are now allowed to login as `admin`:

```
alice@tux:~$ ssh admin@puffy
Welcome to puffy.
alice@puffy:~$
```

SSH CAs

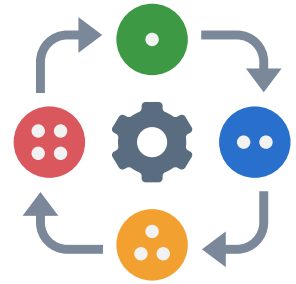
- **Attacks**
  - If attackers can compromise the SSH CA, they can sign arbitrary host / user keys.
  - This can be used to compromise the entire infrastructure, depending on how broadly the CA is trusted.
- **Recommendation**
  - SSH CA keys should have a strong passphrase.
  - SSH CAs should be well protected.
  - This includes a hardened system, limited access, offline system if possible, logging & monitoring, …

- **Attacks**
  - If an attacker can compromise a single SSH certificate, it can be used forever.
- **Recommendation**
  - Revoke compromised certificates.
  - Limit the validity interval (expiration date)

```
trent@ca:~$ ssh-keygen -s ca_user_key -I alice -n alice -V -4w:+4w userkeys/alice.pub
```

```
root@linux-srv-04:~#
```

SSH CA Integration in Infrastructure



- SSH CAs can nicely be integrated into an infrastructure.
- Example for host CA
  - Every newly installed host lets automatically sign their host keys during installation process
  - User's never have to update their `known_hosts` or accept untrusted keys.
- Example for user CA
  - Admins can easily give other users temporary access to systems.
- Example for user CA
  - User must authenticate using MFA on a webapp
  - After authentication, user can upload public key to sign it
  - The user can download and use the certificate to login.
  - After 12 hours, the SSH certificate is expired, and the user must authenticate again.
- Example for user CA
  - An external partner can get SSH key with limited usage after MFA and 4-eyes-principle confirmation.

Multi Factor Authentication using OTPs

- One example of Multi Factor Authentication (MFA) using public keys + authenticator app (One Time Password, OTP)
- SSH server configuration (`sshd_config`)

```
UsePAM yes
PasswordAuthentication no
KbdInteractiveAuthentication yes
AuthenticationMethods publickey,keyboard-interactive
```

This means public key AND keyboard-interactive

- Install `libpam-google-authenticator`
- For every, user, generate OTP configuration (creates `~/.google_authenticator`):

```
alice@puffy:~$ google-authenticator
```

Contains OTP secret and backup codes

- PAM config for SSH (`/etc/pam.d/ssh`)

```
# Standard Un*x authentication.
#@include common-auth
auth required pam_google_authenticator.so nullok
```

Multi Factor Authentication using OTPs

- Example session

```
alice@beastie:~$ ssh -v puffy
[...]
debug1: Authentications that can continue: publickey
debug1: Next authentication method: publickey
debug1: Offering public key: /home/carol/.ssh/id_ed25519 ED25519
SHA256:/qM8Kw1JwTx/ijOG6k1Z2ILe/l2/K0lyAr0/zUGLqW8
debug1: Server accepts key: /home/carol/.ssh/id_ed25519 ED25519
SHA256:/qM8Kw1JwTx/ijOG6k1Z2ILe/l2/K0lyAr0/zUGLqW8
Authenticated with partial success.
debug1: Authentications that can continue: keyboard-interactive
debug1: Next authentication method: keyboard-interactive
Verification code: 500230
Welcome to puffy.
alice@puffy:~$
```

Multi Factor Authentication using OTPs

- **Attack**
  - An attacker with root access on the server can
    - read the password when a user authenticates and
    - Extract the secret of the OTP file
  - If the same password and OTP seed is used on another system, an attacker can use this information for lateral movement.
- **Recommendation**
  - Use public key authentication instead of passwords.
  - Use a different OTP configuration for every user on every system.
  - Use another MFA method which is not vulnerable (like FIDO2)

Multi Factor Authentication: FIDO2

- FIDO2 is an open authentication standard
- FIDO authenticators contains cryptographic keys inside hardware
  - e.g. YubiKey, Nitrokey, TPM, Smartphone
- Native support in OpenSSH versions ≥ 8.2p1
- Key types: `ecdsa-sk` or `ed25519-sk` (these can also be passphrase protected)
- Keys consist of two parts
  - Key file on disk
  - Per-device key file on authenticator
- Requires authorization for some operations by touching them
- For key generation, the authenticator requires authentication (PIN or biometrics)

Multi Factor Authentication: FIDO2

- Discoverable / resident keys
  - Resident key is stored on authenticator (private key protected by the authenticator key, can be copied from the key)
  - Only the authenticator is required to login
  - Easier to use on multiple computers
- Non-discoverable / non-resident
  - Private key stored in `~/.ssh`, protected using authenticator
  - Key file on disk and FIDO authenticator is required to login

MFA: FIDO2 with Discoverable Key

- Generate discoverable key:

```
alice@beastie:~$ ssh-keygen -t ed25519-sk -O resident -O application=ssh:alice-work
Generating public/private ed25519-sk key pair.
You may need to touch your authenticator to authorize key generation.
Enter PIN for authenticator:
You may need to touch your authenticator again to authorize key generation.
Enter file in which to save the key (/home/alice/.ssh/id_ed25519_sk):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/alice/.ssh/id_ed25519_sk
Your public key has been saved in /home/alice/.ssh/id_ed25519_sk.pub
[...]
```

- New key entry on authenticator:

```
alice@beastie:~$ ykman fido credentials list
Enter your PIN:
ssh:alice-work 0000000000000000000000000000000000000000000000000000000000000000 openssh
```

MFA: FIDO2 with Discoverable Key

- Private key is stored in ~/.ssh:

```
alice@beastie:~$ ls -l .ssh/id_ed25519_sk*
-rw------- 1 alice alice 529 Mar 28 14:29 .ssh/id_ed25519_sk
-rw------- 1 alice alice 157 Mar 28 14:29 .ssh/id_ed25519_sk.pub
```

- Private key can only be accessed with key material on the authenticator (tied to authenticator).

- Login using the passphrase protected private key and the authenticator :

```
alice@beastie:~$ ssh puffy
Enter passphrase for key '.ssh/id_ed25519_sk’:
Confirm user presence for key ED25519-SK
SHA256:uQwkqxPZO1bTj3ARWxa/6EL6PdQemQQ2X4pViE/je1w
User presence confirmed
Welcome to puffy.
alice@puffy:~$
```

MFA: FIDO2 with Discoverable Key

- The resident key can be downloaded to another client (authenticator PIN is required):

```
alice@dragonfly:~$ ssh-keygen -K
Enter PIN for authenticator:
You may need to touch your authenticator to authorize key download.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Saved ED25519-SK key ssh:alice-work to id_ed25519_sk_rk_alice-work
```

New passphrase

- This key can then again be used to login:

```
alice@beastie:~$ ssh puffy
Enter passphrase for key '.ssh/id_ed25519_sk':
Confirm user presence for key ED25519-SK
SHA256:uQwkqxPZO1bTj3ARWxa/6EL6PdQemQQ2X4pViE/je1w
User presence confirmed
Welcome to puffy.
alice@puffy:~$
```

MFA: FIDO2 with Discoverable Key

- Instead of copying the key, the key can also be loaded into the SSH agent:

```
alice@dragonfly:~$ ssh-add -K
Enter PIN for authenticator:
Resident identity added: ED25519-SK SHA256:kl63KizwgVqe4RtCPhiMqnExygduOTMQdqLJRJfXKZg
```

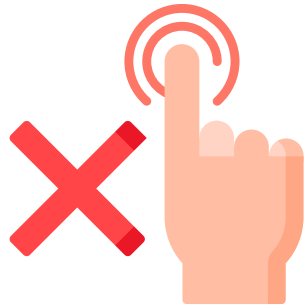
MFA: FIDO2 with Discoverable Key

- **Attacks**
  - An attacker with
    - a) knowledge of the authenticator key PIN
    - b) physical access to the authenticator
  - can use the authenticator to an own machine and use it to authenticate.
- **Recommendation**
  - For higher security, discoverable keys should not be used.
  - Instead, non-discoverable / non-resident keys should be used.
- Generate non-resident keys:

```
alice@beastie:~$ ssh-keygen -t ed25519-sk -O application=ssh:alice-work
```

- The generated private keys are protected via the authenticator key.
- They are not stored on the authenticator itself and must be copied manually to other systems
  - `~/.ssh/id_ed25519_sk` and `~/.ssh/id_ed25519_sk.pub`

MFA: FIDO2 Without Touch



- By default, the authenticators require user presence (key touch) for every key access.
- A user can generate a key which does not require user presence:

```
alice@beastie:~$ ssh-keygen -t ed25519-sk -O resident -O no-touch-required -O
application=ssh:alice-work
```

- By default, SSH servers require user presence.
- A user can overwrite this in their personal authorized keys file:

```
alice@puffy:~$ cat .ssh/authorized_keys
no-touch-required sk-ssh-ed25519@openssh.com
AAAAGnNrLXNzaC1lZDI1NTE5QG9wZW5zc2guY29tAAAAIEvUpHBQeQCE4OuuTnTijntxMFdknEzPD06tKkfa88M
nAAAADnNzaDphbGljZS13b3Jr alice@beastie
```

- The user can then login without touch:

```
alice@beastie:~$ ssh puffy
Welcome to puffy.
alice@puffy:~$
```

MFA: FIDO2 Without Touch

- **Attacks**
  - An attacker with
    - a) access to the private key protected by the authenticator
      (no passphrase set, weak passphrase set, loaded into SSH agent)
    - b) code execution on the client
    - c) authenticator plugged in
  - can establish an SSH connections without the user noticing.
- **Recommendation**
  - The server should enforce user presence.
- The server can enforce user presence (`sshd_config`):

```
PubkeyAuthOptions touch-required
```

- The user's cant override this anymore in the authorized keys file.

MFA: FIDO2 Without User Authentication

- **Attacks**
  - An attacker with
    - a) access to the private key protected by the authenticator
      (no passphrase set, weak passphrase set, loaded into SSH agent)
    - b) code execution on the client
    - c) authenticator plugged in and physical access to the authenticator
  - can establish an SSH connections using the private key and by touching the authenticator.
- **Recommendation**
  - The server should enforce user authentication on every authenticator access (PIN/biometrics).
- The server can enforce user authentication (`sshd_config`):

```
PubkeyAuthOptions verify-required
```

- The user's cant override this anymore in the authorized keys file.

MFA: FIDO2 With User Authentication

- A user then has to generate keys which require authentication:

```
alice@beastie:~$ ssh-keygen -t ed25519-sk -O verify-required
  -O application=ssh:alice-work
```

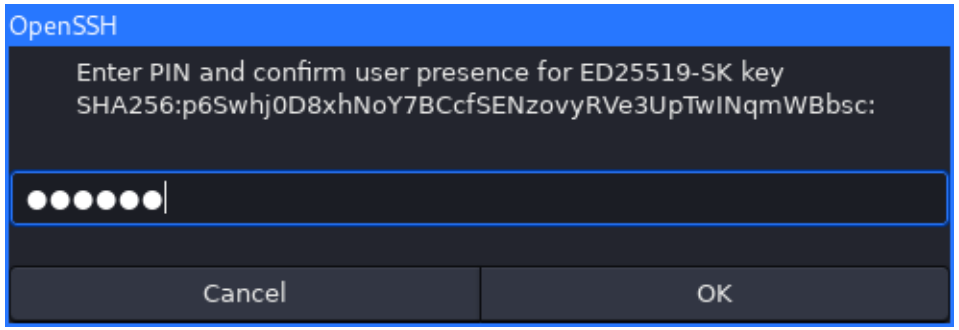
- The user then must enter the key passphrase, the authenticator PIN and touch the FIDO key:

```
alice@beastie:~$ ssh puffy
Enter passphrase for key '/home/alice/.ssh/id_ed25519_sk':
Confirm user presence for key ED25519-SK
SHA256:11vUu+qDwFaIOZvBgODlzmcr5d60+7ljmzxJp/8KxAc
Enter PIN for ED25519-SK key /home/alice/.ssh/id_ed25519_sk:
Confirm user presence for key ED25519-SK
SHA256:11vUu+qDwFaIOZvBgODlzmcr5d60+7ljmzxJp/8KxAc
User presence confirmed
Welcome to puffy.
alice@puffy:~$
```

MFA: FIDO2 With User Authentication

- Example session on another client using resident keys & `ssh-agent` (requires `ssh-askpass`):

```
alice@beastie:~$ ssh-add -K
Enter PIN for authenticator:
Resident identity added: ED25519-SK SHA256:p6Swhj0D8xhNoY7BCcfSENzovyRVe3UpTwINqmWBbsc

alice@beastie:~$ ssh puffy
```



ssh-askpass

+

No passphrase required, since private key file is not copied to machine.

```
Welcome to puffy.
alice@puffy:~$
```

MFA: FIDO2

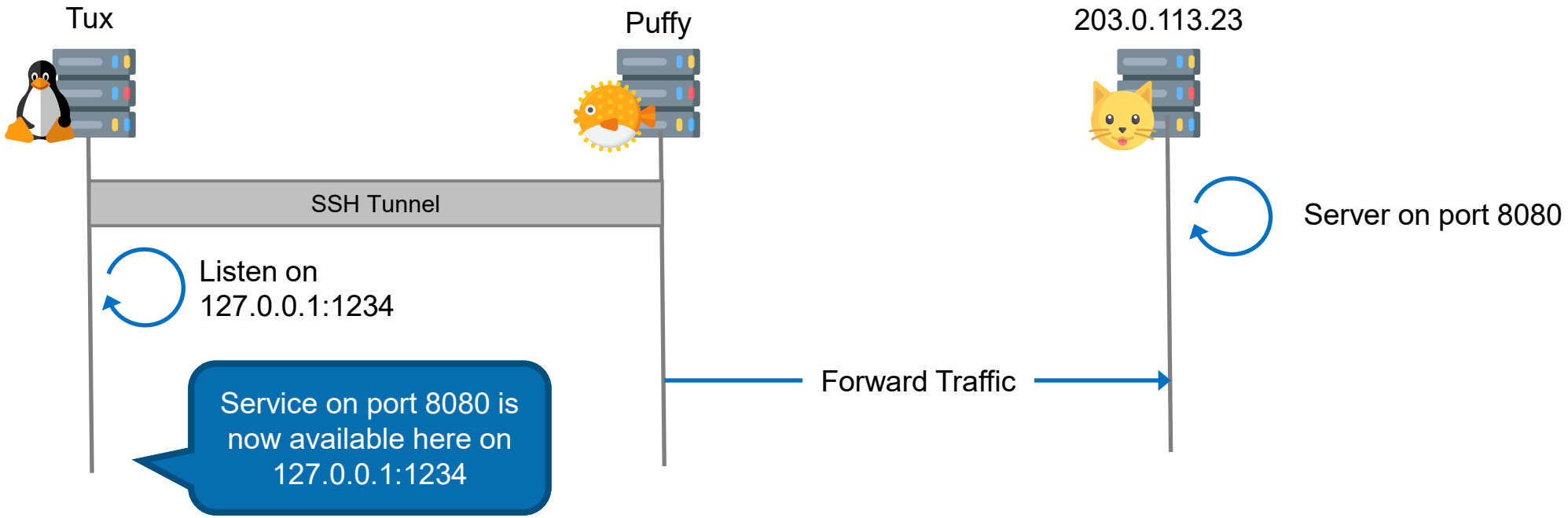
- Lot of different methods and options how FIDO2 authenticators can be configured
- Most important: you have MFA
  - Better than only a password or only a private key
  - Possession Factor: You need the authenticator itself to be plugged in
  - Knowledge Factor: You need the authenticator's PIN
- Without FIDO2
  - An attacker with root on your system can copy the private key file and sniff the passphrase
  - When using FIDO2 authenticators, the private keys cannot be copied
  - The authenticator is always required
- Adjust options and usage to your security requirements
  - E.g. if you need to be flexible: use discoverable keys, but always require authentication via PIN
  - E.g. For high secure environments: use non-discoverable keys, use strong key passphrase, always require authentication via PIN

Port Forwarding / Tunneling

Local Port Forwarding

- SSH can be used to forward / tunnel TCP ports
- Local port forwarding
  - Forward an incoming connection on the local system to the remote server
  - Start a new TCP listener on the local system (binds by default on loopback interface 127.0.0.1/::1)

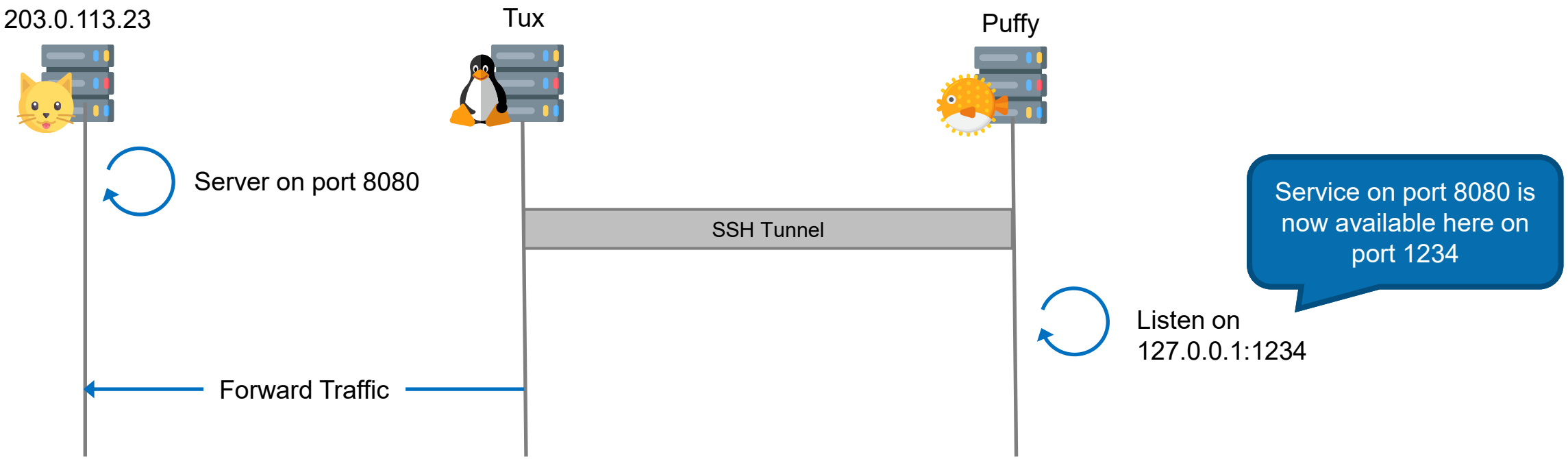
```
alice@tux:~$ ssh -L 1234:203.0.113.23:8080 puffy
```



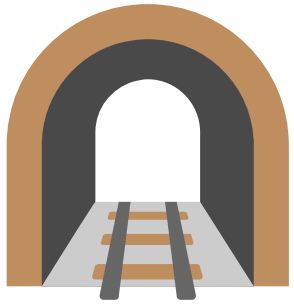
Remote Port Forwarding

- Remote port forwarding
  - Forward an incoming connection on the remote system to the local client
  - Start a new TCP listener on the remote system (binds by default on loopback interface 127.0.0.1/::1)

```
alice@tux:~$ ssh -R 1234:203.0.113.23:8080 puffy
```



Remote Port Forwarding



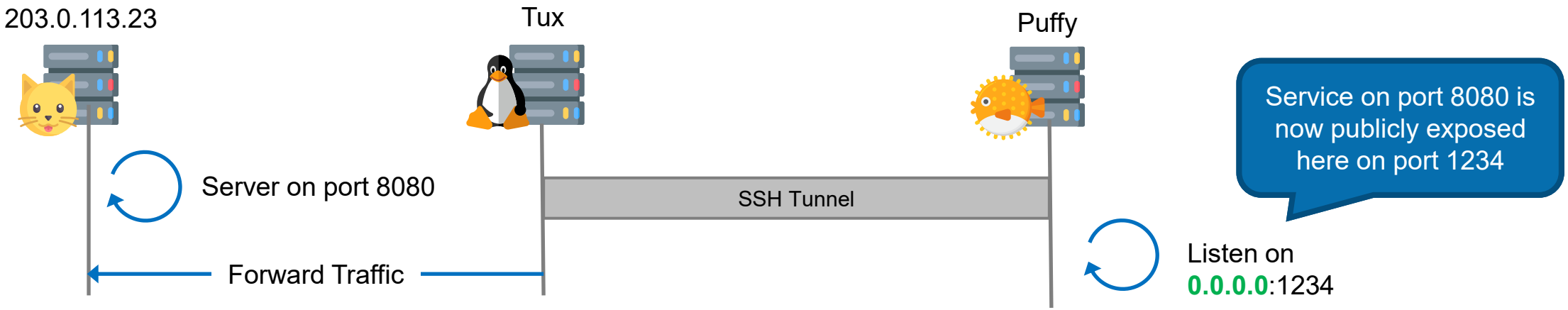
- By default, the new TCP listener binds to loopback interface (`127.0.0.1`, `::1`)
- The server can relax this restriction:

```
GatewayPorts clientspecified # or yes
```

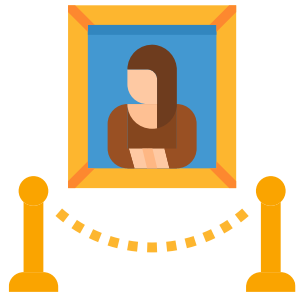
- It's then possible to specify other interfaces

```
alice@tux:~$ ssh -R 0.0.0.0:1234:203.0.113.23:8080 puffy
```

- Only then, external systems (non-loopback) can connect to remote forwarded ports.



Unintentional Service Exposure



- **Attacks**
  - If the `GatewayPorts` option in the SSH server config is set to `yes`,
  - all remotely forwarded ports automatically bind to all interfaces (`0.0.0.0`, `[::]`).
  - This could expose forwarded ports unintentionally, if users are not aware of this setting.
- **Recommendation**
  - The `GatewayPorts` option should not be set to `yes`.
  - Either not set, set to `no` (default) or set to `clientspecified`.
  - If `clientspecified` is used, default is still loopback, but users can overwrite.

- Example server config (`sshd_config`):

```
GatewayPorty clientspecified
```

SOCKS Proxy

- Create SOCKS proxy on the local host for tunneling traffic through remote host:

```
alice@tux:~$ ssh -D 1080 -N puffy
alice@tux:~$ ss -ltpn
State  Recv-Q Send-Q Local Address:Port  Peer Address:PortProcess
[...]
LISTEN  0  128  0.0.0.0:1080  0.0.0.0:*  users:(("ssh",pid=650009,fd=4))
[...]
```

SOCKS server listening on 1080

- You can now access services through the SOCKS proxy:

```
alice@tux:~$ cat /etc/proxychains.conf
[...]
proxy_dns
socks5 127.0.0.1 1080
alice@tux:~ $ proxychains curl -v https://example.net
[...]
```

Proxychains config

Tunnel traffic via SOCKS

Useful Use-Case for Pentests

- Situation
  - You got a notebook of a customer for an internal pentest
  - The internal pentest is performed remotely using the VPN client on the notebook
  - The notebook has all the latest and fancy anti-malware / EDR software installed

- Poor analyst's problem
  - You can't use your `$TOOLS` from your Kali VM or on the customer's notebook 😢

- Solution: SSH to the rescue! 🐡 🎉
  - Connect the notebook to your testing network where your testing VM is
  - Use SSH from the notebook to create a SOCKS proxy on your attacker machine
  - You can then access the corporate network from your attacker machine

Useful Use-Case for Pentests

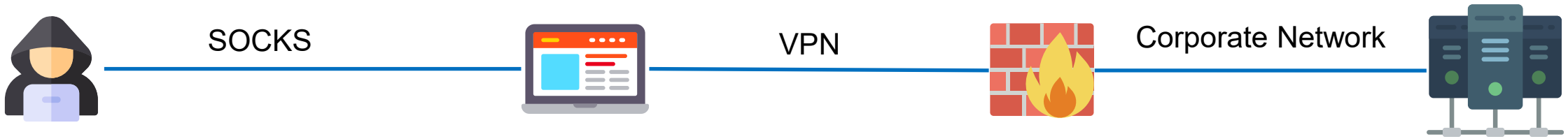
- Connect notebook to own network and execute:

```
PS domainuser@notebook C:\> ssh -R 1080 -N kali
```

- New SOCKS proxy on your attacker kali:

```
attacker@kali:~ $ sudo ss -ltpn sport = 1080 | cat
State  Recv-Q Send-Q Local Address:Port Peer Address:PortProcess
LISTEN 0      128        127.0.0.1:1080      0.0.0.0:*    users:(("sshd",pid=8169,fd=9))
LISTEN 0      128            [::1]:1080         [::]:*    users:(("sshd",pid=8169,fd=7))
```

- Created tunnel:

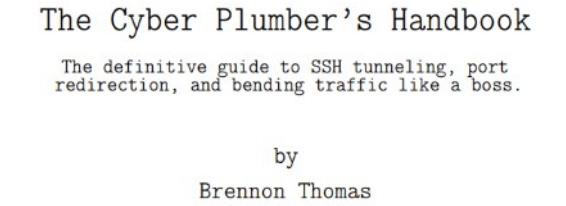


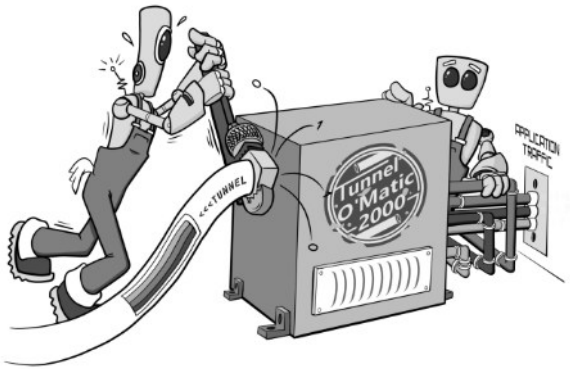
- You can now access the customer's network (SOCKS limitations apply):

```
attacker@kali:~ $ proxychains nxc smb dc.example.net -u alice -p s3cret -d example.net
[...]
```

Useful Use-Case for Pentests

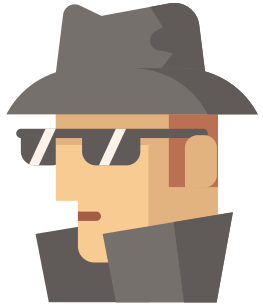
- If you are not familiar with tunneling techniques,there is the Cyber Plumber's Handbook
  - SSH, proxychains, netcat basics
  - Port forwarding
  - SOCKS proxying
  - Netsh
  - Meterpreter SOCKS, port forwarding, routing
- https://github.com/opsdisk/the_cyber_plumbers_handbook





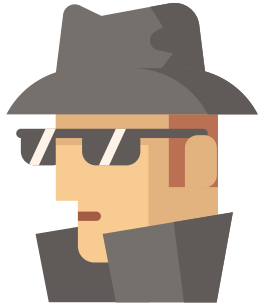
SSH Agent

SSH Agent



- With passphrase protected keys, the key must be unlocked for each connection.
- To address this, keys can be loaded once into a so-called SSH agent.
  - Loading the key requires the passphrase.
- SSH agent is a process running in the background on the user's client.
- Holds private keys used for public key authentication.
- The key can then be used without entering the passphrase again

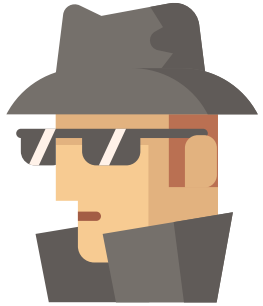
SSH Agent



- Uses environment variables to connect to the agent socket.
- Example for Linux

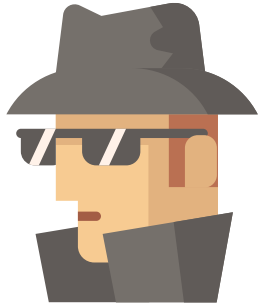
```
alice@beastie:~$ eval $(ssh-agent)
Agent pid 1318

alice@beastie:~$ env | grep ^SSH
SSH_AUTH_SOCK=/tmp/ssh-PmBPRK9DcVkb/agent.2305
SSH_AGENT_PID=1318

alice@beastie:~$ ls -la /tmp/ssh-TUKDBryLDJUV/agent.1347
srw------- 1 alice alice 0 Feb 21 13:14 /tmp/ssh-TUKDBryLDJUV/agent.2305
```

SSH Agent



```
alice@beastie:~$ ssh puffy
Enter passphrase for key '/home/alice/.ssh/id_ed25519':
^C

alice@beastie:~$ ssh-add
Enter passphrase for /home/alice/.ssh/id_ed25519:
Identity added: /home/alice/.ssh/id_ed25519 (alice@beastie)

alice@beastie:~$ ssh-add -l
256 SHA256:4CbWpsIxO1X+xvhHAZwVyPU50dRyV8i0skV2S09G21g alice@beastie (ED25519)

alice@beastie:~$ ssh puffy
Welcome to puffy.
alice@puffy:~$
```

SSH Agent



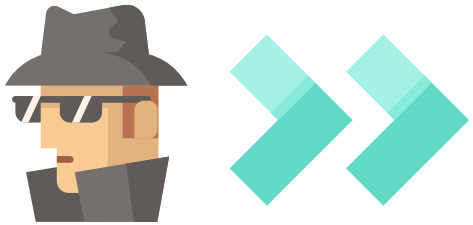
- Example for Windows

```
PS > Get-Service ssh-agent | Set-Service -StartupType Automatic
PS > Get-Service ssh-agent
[...]
Running  ssh-agent          OpenSSH Authentication Agent

PS > ssh-add ~\.ssh\id_ed25519
256 SHA256:4CbWpsIxO1X+xvhHAZwVyPU50dRyV8i0skV2S09G21g alice@beastie (ED25519)

PS > ssh puffy
Welcome to puffy
alice@puffy:~$
```

- PuTTY also has an SSH Agent (pageant)

SSH Agent Forwarding

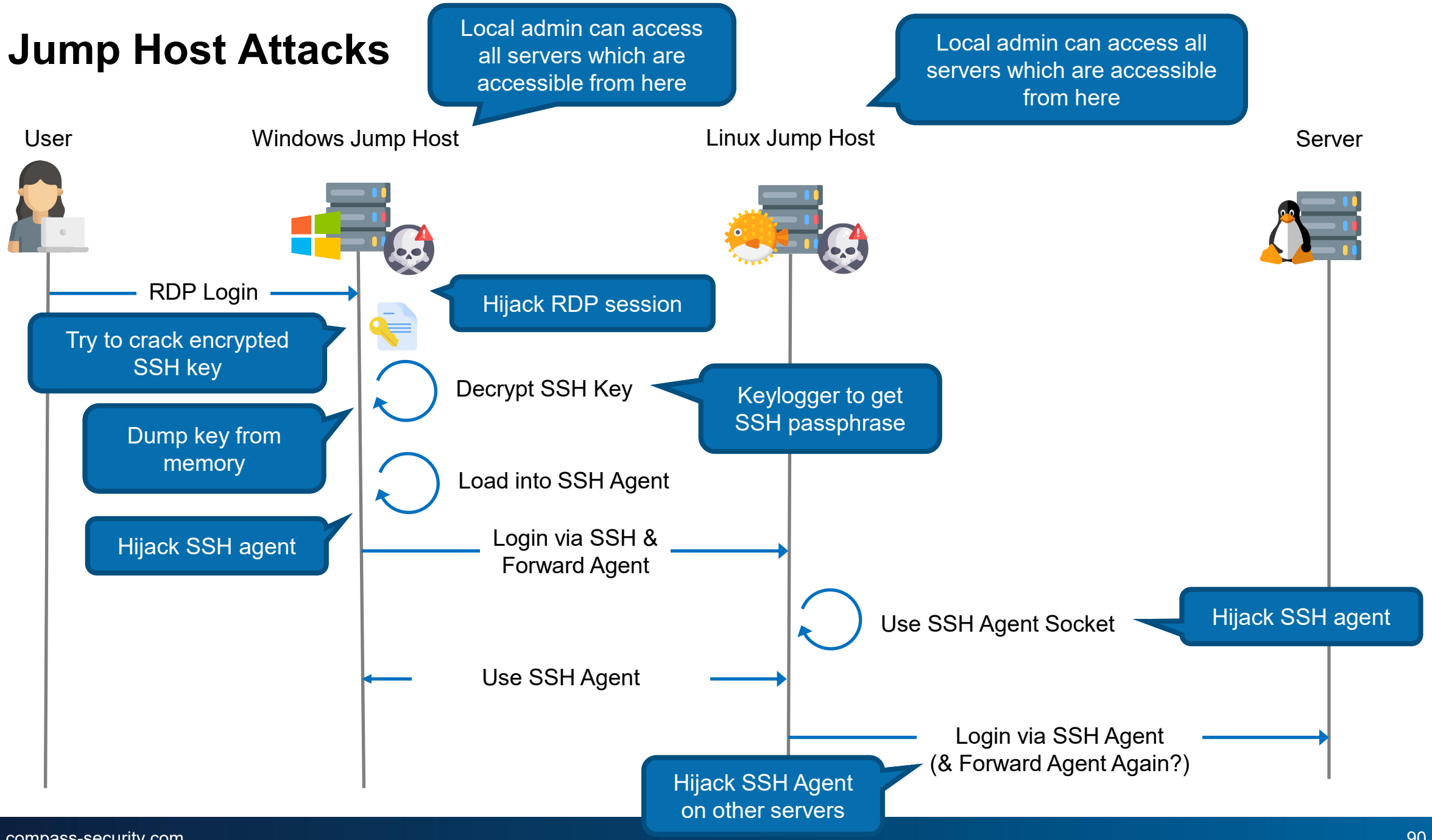
- SSH agent can be forwarded to a remote server
- This makes the loaded keys available on the remote server.



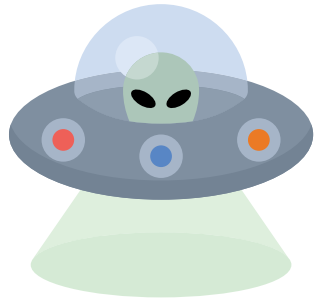
```
alice@beastie:~$ ssh -A jumphost
Welcome to jumphost.

alice@jumphost:~$ echo $SSH_AUTH_SOCK
/tmp/ssh-10bpIHPZjF/agent.1365

alice@jumphost:~$ ls -l $SSH_AUTH_SOCK
srwxr-xr-x 1 alice alice 0 Feb 21 13:22 /tmp/ssh-10bpIHPZjF/agent.1365

alice@jumphost:~$ ssh-add -l
256 SHA256:4CbWpsIxO1X+xvhHAZwVyPU50dRyV8i0skV2S09G21g alice@beastie (ED25519)

alice@jumphost:~$ ssh puffy
Welcome to puffy.
alice@puffy:~$
```

Comment specified during key generation

Jump Host Attacks



SSH Agent Hijacking



- Example

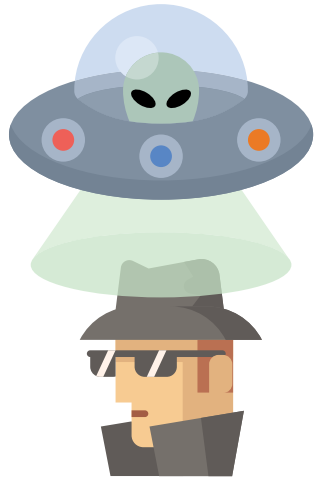
```
external-partner@aix:~$ ssh puffy
external-partner@puffy: Permission denied (publickey).
external-partner@aix:~$ ssh -l alice puffy
alice@puffy: Permission denied (publickey).

external-partner@aix:~$ sudo -i
root@aix:~# find / -type s -ls 2>/dev/null
/tmp/ssh-10bpIHPZjF/agent.2305

root@aix:~# export SSH_AUTH_SOCK=/tmp/ssh-10bpIHPZjF/agent.2305
root@aix:~# ssh-add -l
256 SHA256:4CbWpsIxO1X+xvhHAZwVyPU50dRyV8i0skV2S09G21g alice@beastie (ED25519)

root@aix:~# ssh -l alice puffy
Welcome to puffy.
alice@puffy:~$
```

SSH Agent Hijacking

- **Attacks**
  - Whoever has access to the SSH agent socket, can use it to authenticate (but not obtain key material).
    - Low privileged users who can perform privilege escalation.
    - External partners with admin access to only one machine.
    - Sysadmins with only admin access on limited machines.

```
root@linux-srv-04:~#
```

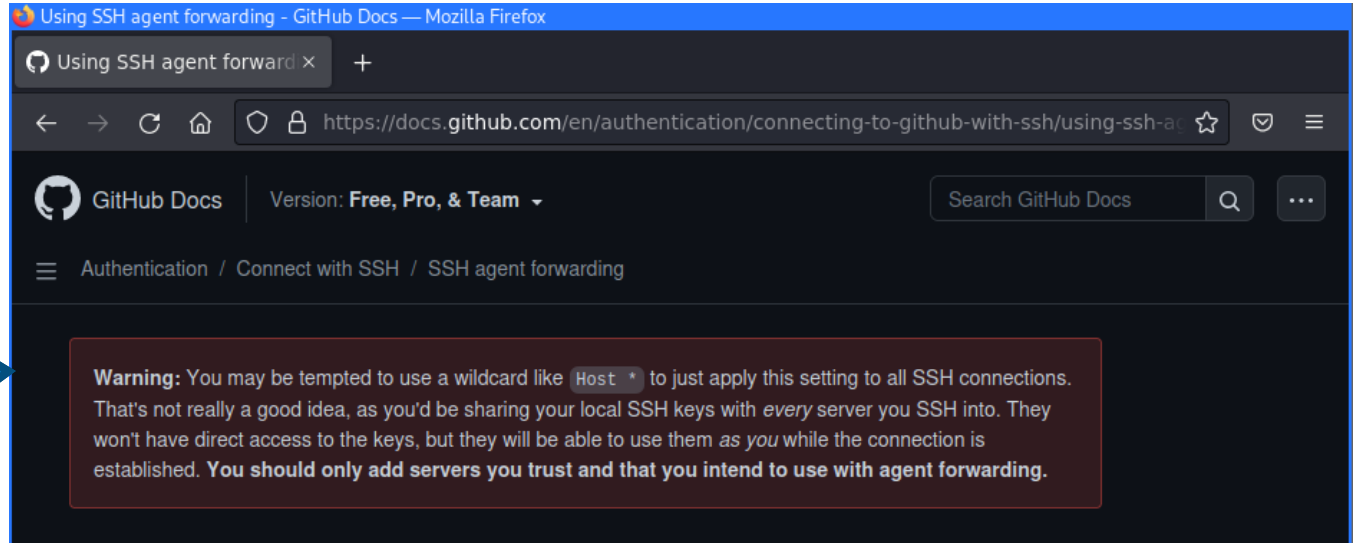
SSH Agent Hijacking



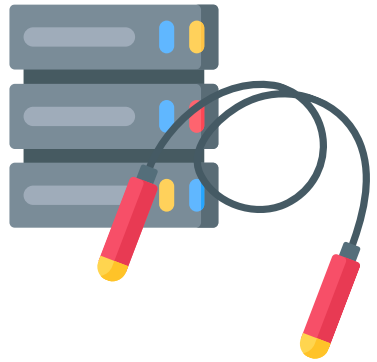
- **Recommendation**
  - Again: Don’t store private keys on jump hosts.
  - Again: Encrypt private keys using a passphrase.
  - Don’t use SSH agent forwarding
  - Explicitly deny SSH agent forwarding on the server
  - Use SSH jump proxy feature `ProxyJump`
    (Connect stdio on the client to a single port forward on the server.)
  - Don’t allow interactive login on jump proxy
- Example server config

```
AllowAgentForwarding no
```

GitHub also warns from using this feature.



SSH Jump Proxies



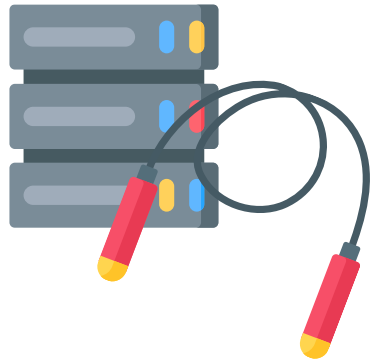
- Example session

```
alice@beastie:~$ ssh-add
Enter passphrase for /home/alice/.ssh/id_ed25519:
Identity added: /home/alice/.ssh/id_ed25519 (alice@beastie)

alice@beastie:~$ ssh-add -l
256 SHA256:4CbWpsIxO1X+xvhHAZwVyPU50dRyV8i0skV2S09G21g alice@beastie (ED25519)

alice@beastie:~$ ssh -J jumphost puffy
Welcome to puffy.
alice@puffy:~$
```

- It's possible to use multiple jump hosts:

```
alice@beastie:~$ ssh -J jumper,bouncy puffy
Welcome to puffy.
alice@puffy:~$
```

SSH Jump Proxies



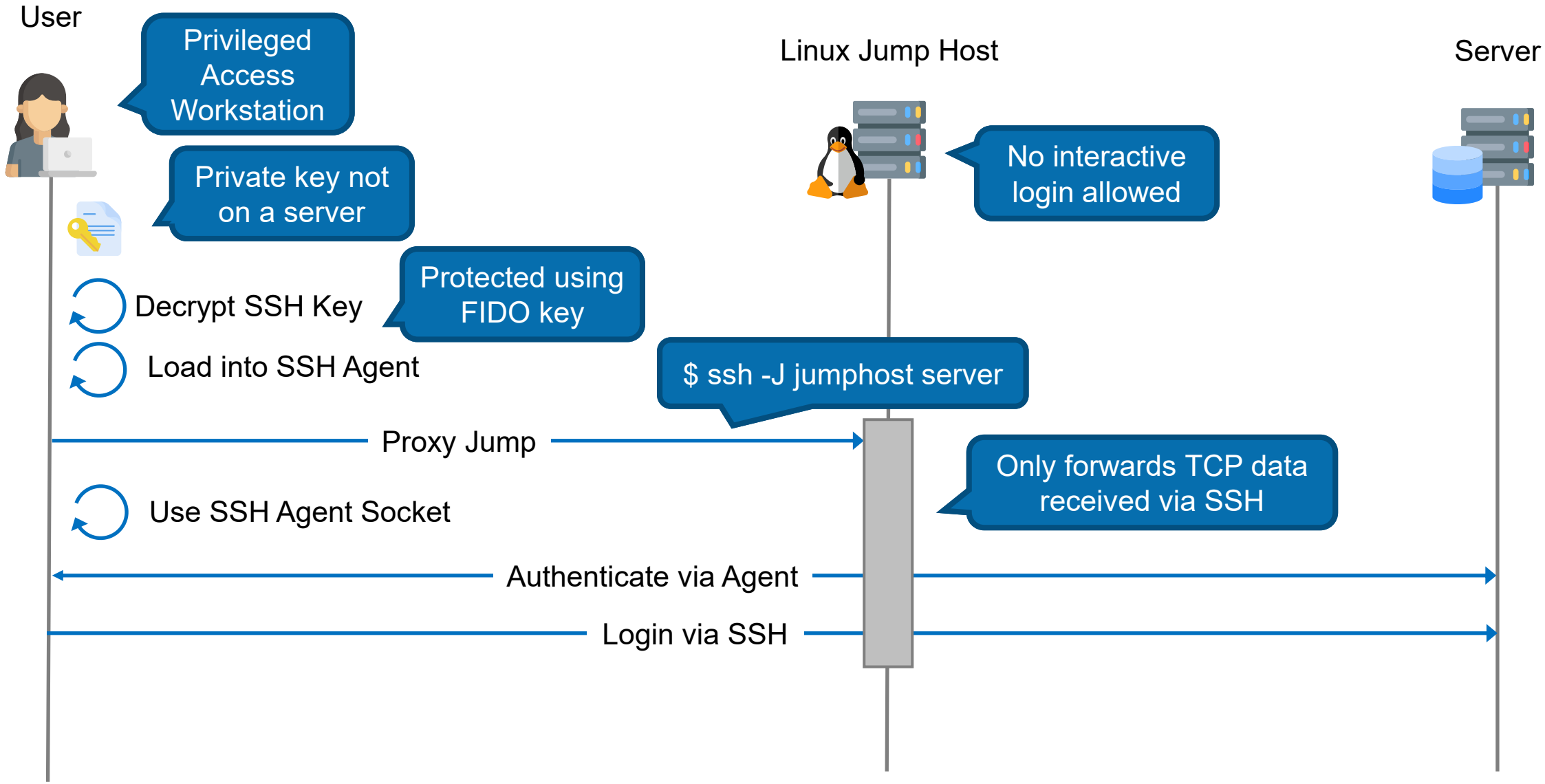
- Example client config

```
Host puffy linux-srv-?? aix-srv-??
    HostName %h.example.net # Add domain for internal systems

Host *.example.net !jumphost.example.net
     ProxyJump jumphost.example.net # connect via the JumpHost
```

- Example session

```
alice@beastie:~$ ssh puffy
Welcome to puffy.
alice@puffy:~$
```

Remediation



Older Systems

- The `ProxyJump` option is available since OpenSSH 7.2.
- Older systems can use the `ProxyCommand` option.
- Use SSH's "netcat mode" (available since 5.3) as a `ProxyCommand`

```
alice@beastie:~$ ssh -o ProxyCommand="ssh -W %h:%p jumphost" puffy
Welcome to puffy.
alice@puffy:~$
```

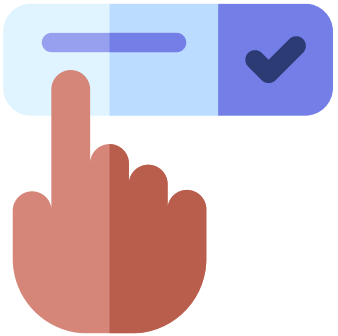
- Otherwise use netcat in the `ProxyCommand`:

```
alice@beastie:~$ ssh -o ProxyCommand="ssh jumphost nc %h %p" puffy
Welcome to puffy.
alice@puffy:~$
```

But I must use SSH keys on remote systems…

- In some situations, you need the authentication material on another system
  - Example: You want to clone a Git repository which requires SSH keys on a remote system
- Dangerous solutions
  - SSH keys should not be copied to the remote system (can be stolen)
  - SSH agent forwarding should not be used (SSH agent hijacking)
- Since `git` has to be executed on the remote system, `ProxyJump` can't be used
- Possible solutions so limit attack surface (not completely eliminated)
  - Harden SSH agent: SSH agent user confirmation
  - Use FIDO keys that require user-presence confirmation
  - SSH agent restrictions

SSH Agent Confirmation



Require confirmation on each usage (`-c`):

```
$ ssh-add -c ~/.ssh/id_ed25519
Enter passphrase for /home/alice/.ssh/id_ed25519 (will confirm each use):
Identity added: /home/ alice /.ssh/id_ed25519 (alice@beastie)
The user must confirm each use of the key
```

Login:

```
alice@beastie:~$ ssh puffy
```

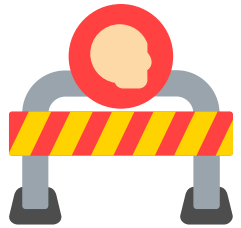
ssh-askpass

Source/destination host is not shown.

```
Welcome to puffy.
alice@puffy:~$
```

This can be used to protect keys when agent is forwarded to remote untrusted systems. Attackers can hijack and hope you accept their usage.

SSH Agent Restrictions



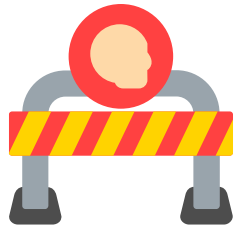
It’s possible to add SSH agent restrictions:

```
$ ssh-add -h "jenkins-dev" ~/.ssh/id_rsa
$ ssh-add -h "puffy" -h "puffy>alice@aix" ~/.ssh/id_ed25519
```

This only allows:

- Use the agent from the origin host to authenticate on jenkins-dev as any user with the key `id_rsa`
- Use the agent from the origin host to authenticate on `puffy` as any user with the key `id_ed25519`
- Use the agent on `puffy` to authenticate on `aix` as user `alice` with the key `id_ed25519`

SSH Agent Restrictions



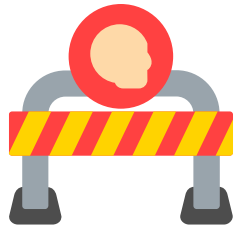
```
alice@beastie:~$ ssh-add -l
3072 SHA256:Gebp6qXAsO9nboiykm1jvEye+V/dCjE4kE0vqn1YGkc alice@development (RSA)
256 SHA256:j/PQO3tpaXNRVJYYt0dWql/5V4V0D6fFt/FWUW723eQ alice@beastie (ED25519)

alice@beastie:~$ ssh -v -A puffy
[...]
debug1: get_agent_identities: agent returned 1 keys
[...]
Welcome to puffy.
```

Only 1 key available to authenticate to this host

```
alice@puffy:~$ ssh-add -l
error fetching identities for protocol 1: agent refused operation
256 SHA256:j/PQO3tpaXNRVJYYt0dWql/5V4V0D6fFt/FWUW723eQ alice@beastie (ED25519)
```

Only 1 key available for furhter authentication

SSH Agent Restrictions



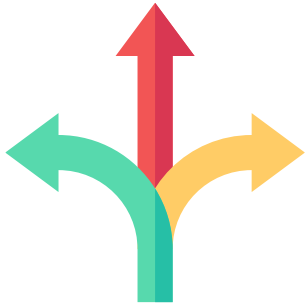
- Compatibility: Requires OpenSSH ≥ 8.9 on both the client & server
- Failsafe: If destination host does not support it, keys can't be used at all
- Restrictions based on hostnames (according to the `known_hosts` file where SSH agent is running)
- Future ideas
  - Specify entire forwarding chain in single argument (`"puffy>aix>dragonfly"`)
  - Make keys available only on specific forwarded host (e.g. not on origin host)
  - Key permissions (key can e.g. only be used for git commit signing but not authentication, …)
  - Can be combined with SSH agent user confirmation
  - Show path information (which host authenticates to which system)
    - for SSH confirmation dialogue
    - for FIDO touch/PIN request dialogue
- See documentation for more information: https://www.openssh.com/agent-restrict.html

CVE-2023-38408: RCE on Client via forwarded SSH Agent

- Fantastic research by Qualys Security Advisory team
  - https://www.qualys.com/2023/07/19/cve-2023-38408/rce-openssh-forwarded-ssh-agent.txt
- When a client forwards the SSH agent to a server, the server can load/unload shared libraries from `/usr/lib*` on the client.
- Qualys researchers could turn this to execute arbitrary code on the client.
- Preconditions
  - Specific libraries must be present on the client (victim)
  - Client (victim) must forward SSH agent to attacker-controlled server
- Fixed in OpenSSH 9.3p2 (released on 2023-07-19)

SSH Session Multiplexing

SSH Session Multiplexing

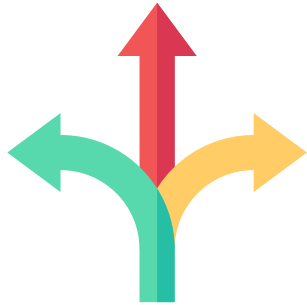


- It's possible to reuse one TCP connection for multiple SSH sessions.
- Only establish one TCP connection and authenticate once on the server.
- Faster, because further SSH sessions will use the already established SSH session.
- Example Use Case: Speed up connections via jump proxy

```
Host jumphost.example.net
    ControlMaster auto
    ControlPath ~/.ssh/cm-%r-%h-%p
    ControlPersist 0
Host *.example.net !jumphost.example.net
    ProxyJump jumphost.example.net # connect via the JumpHost
```

- Example Use Case: Speed up running multiple Ansible Playbooks

SSH Session Multiplexing



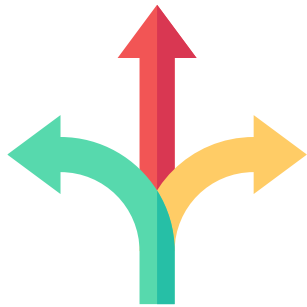
- Establishing first session

```
alice@beastie:~$ time ssh puffy true
real    0m1.000s
```

- Control socket exists

```
alice@beastie:~$ ssh -O check puffy
Master running (pid=49960)
alice@beastie:~$ ls -l .ssh/cm-alice-puffy-22
srw------- 1 alice alice 0 Feb 23 14:41 .ssh/cm-alice-puffy-22
```

SSH Session Multiplexing



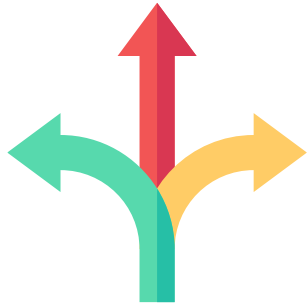
- Establishing second session

```
alice@beastie:~$ time ssh puffy true
real    0m0.080s
```

- Terminating control socket:

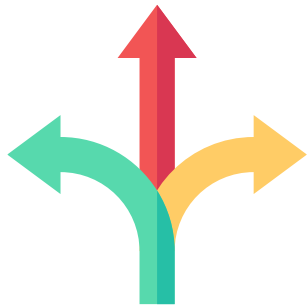
```
alice@beastie:~$ ssh -O stop puffy
Stop listening request sent.

alice@beastie:~$ ssh -O check puffy
Control socket connect(/home/alice/.ssh/cm-alice-puffy-22): No such file or directory
```

SSH Session Multiplexing



- **Attacks**
  - Whoever has access to the SSH control socket, can use it to reuse the connection and establish a new SSH session
    - Low privileged users who can perform privilege escalation.
    - External partners with admin access to only one machine.
    - Sysadmins with only admin access on limited machines.
  - Since the connection is already authenticated, no further authentication is required
    - No need for passwords, private keys, passphrase for keys
    - Even bypasses MFA (because the socket is already authenticated)

SSH Session Multiplexing



- Example

```
external-partner@aix:~$ ssh puffy
external-partner@puffy: Permission denied (publickey).
external-partner@aix:~$ ssh -l alice puffy
alice@puffy: Permission denied (publickey).

external-partner@aix:~$ sudo -i
root@aix:~# find / -type s -ls 2>/dev/null
/home/alice/.ssh/cm-alice-puffy-22

root@aix:~# ssh -l alice -S /home/alice/.ssh/cm-alice-puffy-22 puffy
alice@puffy:~$
```

```
root@linux-srv-05:~#
```

SSH Session Multiplexing

- **Recommendation**
  - Generally, don’t use SSH control sockets.
  - Only allow one session per connection on the server to deny control sockets.
  - If you have to, only use it on systems you trust.
- Example server config:

```
MaxSessions 1
```

Cryptographic Algorithms

Cryptographic Algorithms

- SSH supports various cryptographic algorithms
  - Host Key
  - Key Exchange
  - Encryption
  - Message Authentication
- Recent OpenSSH servers use sane default, but some ciphers are “more secure” than others.
- The Internet is full of recommendations / guides and tools.

Cryptographic Algorithms

- **Attacks**
  - If weak algorithms are used, attackers who can intercept your communication could decrypt or even manipulate it.
  - This is however not that easy as it sounds, especially for non-state/nation-level attackers.
- **Recommendation**
  - Audit your SSH config and only enable secure cryptographic algorithms.
  - SSH-Audit Hardening Guide: https://www.ssh-audit.com/hardening_guides.html
  - Tool to audit your SSH config: `ssh-audit`

Cryptographic Algorithms

```
$ ssh-audit linux-srv-01
# general
(gen) banner: SSH-2.0-OpenSSH_9.1p1 Debian-2
(gen) software: OpenSSH 9.1p1
(gen) compatibility: OpenSSH 8.5+, Dropbear SSH 2018.76+
(gen) compression: enabled (zlib@openssh.com)

# key exchange algorithms
(kex) sntrup761x25519-sha512@openssh.com  -- [warn] using experimental algorithm
                                          `- [info] available since OpenSSH 8.5
(kex) curve25519-sha256                   -- [info] available since OpenSSH 7.4, Dropbear SSH 2018.76
(kex) curve25519-sha256@libssh.org        -- [info] available since OpenSSH 6.5, Dropbear SSH 2013.62
(kex) ecdh-sha2-nistp256                  -- [fail] using weak elliptic curves
                                          `- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62
(kex) ecdh-sha2-nistp384                  -- [fail] using weak elliptic curves
                                          `- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62
(kex) ecdh-sha2-nistp521                  -- [fail] using weak elliptic curves
                                          `- [info] available since OpenSSH 5.7, Dropbear SSH 2013.62
(kex) diffie-hellman-group-exchange-sha256 (2048-bit) -- [info] available since OpenSSH 4.4
(kex) diffie-hellman-group16-sha512       -- [info] available since OpenSSH 7.3, Dropbear SSH 2016.73
(kex) diffie-hellman-group18-sha512       -- [info] available since OpenSSH 7.3
(kex) diffie-hellman-group14-sha256       -- [info] available since OpenSSH 7.3, Dropbear SSH 2016.73
```

Cryptographic Algorithms

```
# host-key algorithms
(key) rsa-sha2-512 (3072-bit)          -- [info] available since OpenSSH 7.2
(key) rsa-sha2-256 (3072-bit)          -- [info] available since OpenSSH 7.2
(key) ecdsa-sha2-nistp256              -- [fail] using weak elliptic curves
                                       `- [warn] using weak RNG could reveal the key
                                       `- [info] available since OpenSSH 5.7, Dropbear 2013.62
(key) ssh-ed25519                      -- [info] available since OpenSSH 6.5

# encryption algorithms (ciphers)
(enc) chacha20-poly1305@openssh.com -- [info] available since OpenSSH 6.5
                                       `- [info] default cipher since OpenSSH 6.9.
(enc) aes128-ctr                       -- [info] available since OpenSSH 3.7, Dropbear SSH 0.52
(enc) aes192-ctr                       -- [info] available since OpenSSH 3.7
(enc) aes256-ctr                       -- [info] available since OpenSSH 3.7, Dropbear SSH 0.52
(enc) aes128-gcm@openssh.com           -- [info] available since OpenSSH 6.2
(enc) aes256-gcm@openssh.com           -- [info] available since OpenSSH 6.2

# fingerprints
(fin) ssh-ed25519: SHA256:W3Ypt7WQZWeq9XueVDqTfJVzIaly/4KkYSwFzvlgecM
(fin) ssh-rsa: SHA256:CjyhXmy2WEHJu6Pr/O85XG6Kh41SL8pCyZgi/ZR3BoM
```

Cryptographic Algorithms

```
# message authentication code algorithms
(mac) umac-64-etm@openssh.com          -- [warn] using small 64-bit tag size
                                       `- [info] available since OpenSSH 6.2
(mac) umac-128-etm@openssh.com         -- [info] available since OpenSSH 6.2
(mac) hmac-sha2-256-etm@openssh.com    -- [info] available since OpenSSH 6.2
(mac) hmac-sha2-512-etm@openssh.com    -- [info] available since OpenSSH 6.2
(mac) hmac-sha1-etm@openssh.com        -- [warn] using weak hashing algorithm
                                       `- [info] available since OpenSSH 6.2
(mac) umac-64@openssh.com              -- [warn] using encrypt-and-MAC mode
                                       `- [warn] using small 64-bit tag size
                                       `- [info] available since OpenSSH 4.7
(mac) umac-128@openssh.com             -- [warn] using encrypt-and-MAC mode
                                       `- [info] available since OpenSSH 6.2
(mac) hmac-sha2-256                    -- [warn] using encrypt-and-MAC mode
                                       `- [info] available since OpenSSH 5.9, Dropbear SSH 2013.56
(mac) hmac-sha2-512                    -- [warn] using encrypt-and-MAC mode
                                       `- [info] available since OpenSSH 5.9, Dropbear SSH 2013.56
(mac) hmac-sha1                        -- [warn] using encrypt-and-MAC mode
                                       `- [warn] using weak hashing algorithm
                                       `- [info] available since OpenSSH 2.1.0, Dropbear SSH 0.28
```

Post-Quantum Cryptography

- PQC protects against “store now, decrypt later” attacks
- "cryptographically-relevant" quantum computers can break asymmetric cryptography (RSA/ECC)
- Post-quantum safe cryptographic algorithms will are resistant to these quantum computers
- OpenSSH
  - OpenSSH 9.0 (April 2022) supports post-quantum kex exchange
  - Default since OpenSSH 10.0 (April 2025)
  - Warning is shown since OpenSSH 10.1 (October 2025) if no PQC key exchange was used
- Used algorithms by OpenSSH are hybrid
  - combination of normal key exchange and post-quantum key exchange algorithms
  - Just in case the these post-quantum algorithms have flaws or are broken at some time
  - It adds another layer of security, you don’t loose anything by using it

Post-Quantum Cryptography

If no post-quantum key exchange is used, you'll see a warning
(after host authentication, before user authentication):

```
alice@beastie:~$ ssh puffy
The authenticity of host 'puffy (203.0.113.23)' can't be established.
ED25519 key fingerprint is SHA256:aPDwXPsHTWTSebUW3jPkb4nH/lUGmvILmQsEkXKsY9c.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added puffy' (ED25519) to the list of known hosts.

** WARNING: connection is not using a post-quantum key exchange algorithm.
** This session may be vulnerable to "store now, decrypt later" attacks.
** The server may need to be upgraded. See https://openssh.com/pq.html

Authorized uses only. All activity may be monitored and reported.
alice@puffy's password:
```

That’s it

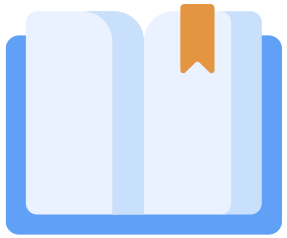
Recap

- Content
  - SSH Introduction
  - Service Discovery
  - Information Disclosure
  - SSH Authentication (Host, User, SSH CAs, MFA, Security Keys / FIDO2)
  - SSH Port Forwarding
  - SSH Agent
  - SSH Session Multiplexing
  - Cryptographic Algorithms

- Feedback / Questions
  - Mail: emanuel.duss@compass-security.com / me@emanuelduss.ch
  - Bluesky: @emanuelduss.ch
  - Mastodon: @emanuelduss@infosec.exchange

References

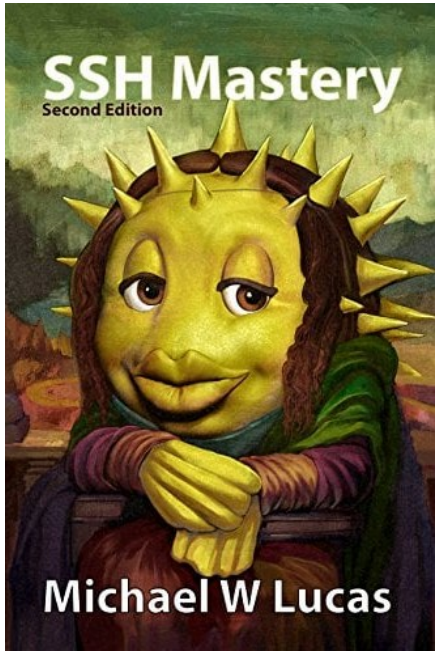
References



- OpenSSH: https://www.openssh.org/
- Standards
  - SSH Protocol Architecture, RFC 4251, 2006: https://datatracker.ietf.org/doc/html/rfc4251
  - Secure Shell (SSH) Protocol Parameters, IANA, 2005: https://www.iana.org/assignments/ssh-parameters/ssh-parameters.xhtml
  - Using DNS to Securely Publish Secure Shell (SSH) Key Fingerprints, RFC 4255, 2006: https://datatracker.ietf.org/doc/html/rfc4255
- Documentation
  - https://www.openssh.com/agent-restrict.html
  - https://www.openssh.org/pq.html
- Manpages
  - `sshd_config(5)`: `DebianBanner`, `Banner`, `VerifyHostKeyDNS`, `AuthenticationMethods`, `AuthorizedKeysFile`, `PermitRootLogin`, `AllowUsers`, `AllowGroups`
  - `ssh_config(5)`: `GlobalKnownHostsFile`, `UserKnownHostsFile`, `HashKnownHosts`
  - `ssh(1)`: `AUTHENTICATION`, `ssh-keygen(1)`, `ssh-agent(1)`, `ssh-add(1)`

References

- General
  - SSH Mastery. 2nd Edition. Michael W Lucas. 2018.
- Session spying
  - https://www.infosecmatter.com/ssh-sniffing-ssh-spying-methods-and-defense/
- FIDO Keys
  - https://developers.yubico.com/SSH/Securing_SSH_with_FIDO2.html
- Jump Proxy
  - https://www.redhat.com/sysadmin/ssh-proxy-bastion-proxyjump
  - https://wiki.gentoo.org/wiki/SSH_jump_host
  - https://www.cyberciti.biz/faq/create-ssh-config-file-on-linux-unix/
- Other interesting information about SSH
  - BornHack 2023. Sexy SSH Hacks. Nicolai Søborg.
    https://media.ccc.de/v/bornhack2023-56142-sexy-ssh-hacks




COMPASS
SECURITY