Practical Privacy-Preserving Authentication for SSH

Lawrence Roy
Stanislav Lyakhov
Yeongjin Jang
Mike Rosulek

Oregon State University

USENIX Security 2022
ia.cr/2022/740
2022-08-12
SSH client  SSH server

should I authenticate with pub key 6c6c6568...?  no

no  yes

signature problem:
server can fingerprint client:
I refuse all advertisements
I can configure client to send only "correct" key

problem:
client can probe server:
I observe someone else's pub key, observe response
I pre-emptive signatures possible (in principle)

problem:
server sees which key was used:
I and can prove it!
authentication not deniable
fundamental to protocol

problem:
server can act as honeypot:
I accept any key, even ones never seen before
fundamental to protocol
SSH client | SSH server
---|---
should I authenticate with pub key 6c6c6568...? | no

should I authenticate with pub key 73616664...? | no

Problem:
- Server can fingerprint client: I refuse all advertisements, learn all keys.
- I can configure client to send only "correct" key.

Problem:
- Client can probe server: I observe someone else's pub key, observe response.
- I pre-emptive signatures possible (in principle).

Problem:
- Server sees which key was used: I and can prove it!
- Authentication not deniable, fundamental to protocol.

Problem:
- Server can act as honeypot: I accept any key, even ones never seen before, fundamental to protocol.
SSH client

should I authenticate with pub key 6c6c6568...?  

no

should I authenticate with pub key 73616664...?  

no

...  
no

yes
SSH client            SSH server

should I authenticate with pub key 6c6c6568...?  

no  

should I authenticate with pub key 73616664...?  

no  

:::

yes  

signature
**SSH client**

should I authenticate with pub key `6c6c6568...`?  

no

should I authenticate with pub key `73616664...`?  

no

... 

**SSH server**

**problem:** server can fingerprint client:

- refuse all advertisements $\Rightarrow$ learn all keys

**problem:** server can act as honeypot:

- accept any key, even ones never seen before  

fundamental to protocol
SSH client

should I authenticate
with pub key
6c6c6568
...

no

should I authenticate
with pub key
73616664
...

no

Filippo Valsorda https://words.filippo.io/ssh-whoami-filippo-io/
SSH client should I authenticate with pub key 6c6c6568? no

Should I authenticate with pub key 73616664? no

"yes"

Signature problem:
server can fingerprint client: I refuse all advertisements I learn all keys
I can configure client to send only "correct" key

Problem:
client can probe server: I over someone else's pub key, observe response I pre-emptive signatures possible (in principle)

Problem:
server sees which key was used: I and can prove it! I fundamental to protocol

Problem:
server can act as honeypot: I accept any key, even ones never seen before I fundamental to protocol

SSH client

[[kochanski:-]$ ssh whoami.filippo.io

-o/ Hello Mike Rosulek!

Did you know that ssh sends all your public keys to any server it tries to authenticate to?

That's how we know you are @rosulek on GitHub!

Ah, maybe what you didn't know is that GitHub publishes all users' ssh public keys. Myself, I learned it from Ben (benjojo.co.uk).

That's pretty handy at times :) for example your key is at https://github.com/rosolek.keys

-- @FiloSottile (https://twitter.com/FiloSottile)

P.S. The source of this server is at https://github.com/FiloSottile/whoami.filippo.io

Connection to whoami.filippo.io closed.
SSH client

should I authenticate
with pub key

6c6c6568...

no

should I authenticate
with pub key

73616664...

no

"yes signature

problem:
server can fingerprint client:
I refuse all advertisements
learn all keys
I can configure client to send only "correct" key

problem:
client can probe server:
I over someone else's pub key, observe response
I pre-emptive signatures possible (in principle)

problem:
server sees which key was used:
I and can prove it!
I authentication not deniable
I fundamental to protocol

problem:
server can act as honeypot:
I accept any key, even ones never seen before
I fundamental to protocol

[[kochanski:~]$ ssh whoami.filippo.io

.o/ Hello Mike Rosulek!

Did you know that ssh sends all your public keys to any server
it tries to authenticate to?

That's how we know you are @rosulek on GitHub!

Ah, maybe what you didn't know is that GitHub publishes all users' ssh public keys. Myself, I learned it from Ben (benjojo.co.uk).

That's pretty handy at times :) for example your key is at https://github.com/rosulek.keys

-- @FiloSottile (https://twitter.com/FiloSottile)

P.S. The source of this server is at https://github.com/FiloSottile/whoami.filippo.io

Connection to whoami.filippo.io closed.
SSH client

should I authenticate with pub key 6c6c6568...? 

no

should I authenticate with pub key 73616664...? 

no

: 

SSH server

**problem**: server can fingerprint client:

- refuse all advertisements ⇒ learn all keys
- can configure client to send only “correct” key
**SSH client**

should I authenticate with Bob’s pub key?  

**SSH server**

- yes/no

**Problem:** server can fingerprint client:
  - refuse all advertisements $\Rightarrow$ learn all keys
  - can configure client to send only “correct” key

**Problem:** client can probe server:
  - offer someone else’s pub key, observe response
  - *pre-emptive* signatures possible (in principle)
problem: server can fingerprint client:
  ▶ refuse all advertisements ⇒ learn all keys
  ▶ can configure client to send only “correct” key

problem: client can probe server:
  ▶ offer someone else’s pub key, observe response
  ▶ *pre-emptive* signatures possible (in principle)

problem: server sees which key was used:
  ▶ and can **prove it!** ⇒ authentication not deniable
  ▶ fundamental to protocol
SSH client  SSH server

should I authenticate with pub key 6c6c6568...?

yes

**problem:** server can fingerprint client:
- refuse all advertisements \(\Rightarrow\) learn all keys
- can configure client to send only “correct” key

**problem:** client can probe server:
- offer someone else’s pub key, observe response
- *pre-emptive* signatures possible (in principle)

**problem:** server sees which key was used:
- and can **prove it!** \(\Rightarrow\) authentication not deniable
- fundamental to protocol

**problem:** server can act as honeypot:
- accept *any* key, even ones never seen before
- fundamental to protocol
goals of this work

1. server & client should learn minimal information
goals of this work

1. server & client should learn minimal information
2. authenticate with respect to existing SSH keys
goals of this work

Requests by signature type

- ssh-rsa: 45.0%
- rsa-sha2: 47.0%
- ecdsa-sha2: 0.4%
- ssh-ed25519: 7.3%
- ssh-dss: 0.3%

https://github.blog/2021-09-01-improving-git-protocol-security-github/
goals of this work

1. server & client should learn minimal information
2. authenticate with respect to existing SSH keys
3. minimize reliance on per-site configuration
any **mixture** of existing RSA, ECDSA, EdDSA keys, in a single authentication attempt
our new authentication method: big picture

client
$sk_1, sk_4, sk_9$  

our protocol

server
$pk_1, pk_2, \ldots, pk_6$

client has 3 keys, including at least one of $\{sk_1, \ldots, sk_6\}$

any mixture of existing RSA, ECDSA, EdDSA keys, in a single authentication attempt
our new authentication method: big picture

server has 6 keys, including $pk_1$ and $pk_4$

client has 3 keys, including at least one of $\{sk_1, \ldots, sk_6\}$

any mixture of existing RSA, ECDSA, EdDSA keys, in a single authentication attempt
our new authentication method: big picture

server has 6 keys, including $pk_1$ and $pk_4$

client has 3 keys, including at least one of $\{sk_1, \ldots, sk_6\}$

any mixture of existing RSA, ECDSA, EdDSA keys, in a single authentication attempt

does not depend on site-specific configuration; safe to use all keys in every authentication attempts
our new authentication method: big picture

server has 6 keys, including \( pk_1 \) and \( pk_4 \)
client has 3 keys, including at least one of \( \{sk_1, \ldots, sk_6\} \)

- any mixture of existing RSA, ECDSA, EdDSA keys, in a single authentication attempt
- does not depend on site-specific configuration; safe to use all keys in every authentication attempts
- client won’t connect unless server knows and explicitly includes one of client’s keys
technical overview

client (with $\{sk_i\}_i$):

server (with $\{pk_j\}_j$):

1. anonymous multi-KEM

$\text{ciphertext} = \text{Enc}(\text{sk}_i \cdot \text{ciphertext})$;

$\text{server}$ decrypts $\text{ciphertext}$ to $\text{plaintext}$;

$\text{ciphertext}$ hides $\text{server}$ recipients

2. private set intersection

$\text{client learns intersection}$;

$\text{server learns whether empty}$

single MKEM construction supporting RSA, ECDSA, & EdDSA

add "proof of nonempty intersection" to $[\text{RosulekTrieu21}]$
technical overview

client (with $\{sk_i\}_i$):

server (with $\{pk_j\}_j$):

$$c, \{m_j\}_j \leftarrow \text{Enc}(\{pk_j\}_j)$$

1. anonymous multi-KEM

address ciphertext to $\{pk_j\}_j$;
$sk_j$ decrypts $c$ to $m_j$;
$c$ hides $pk_j$ recipients
technical overview

client (with \(\{sk_i\}_i\)):

server (with \(\{pk_j\}_j\)):

\[c, \{m_j\}_j \leftarrow \text{Enc}(\{pk_j\}_j)\]

1. anonymous multi-KEM

address ciphertext to \(\{pk_j\}_j\);
\(sk_j\) decrypts \(c\) to \(m_j\);
\(c\) hides \(pk_j\) recipients

\[\anonymous{\mathsf{KEM}}^{\mathsf{mKEM}} = \mathsf{Dec}^1_{sk} \cdot c \cdot \mathsf{P}^{\mathsf{PSI}}_{f_{mb}} \cdot g_{bi} \]

each party has set of items;
client learns intersection;
server learns whether empty

\[\mathsf{PSI}^{\mathsf{fmb}} = \mathsf{b}_{mi} = ?\]

single MKEM construction supporting RSA, ECDSA, & EdDSA

add "proof of nonempty intersection" to \([\text{RosulekTrieu21}]\)
**technical overview**

client (with \(\{sk_i\}_i\)):

\[
\{\tilde{m}_i := \text{Dec}(sk_i, c)\}_i
\]

server (with \(\{pk_j\}_j\)):

\[
c, \{m_j\}_j \leftarrow \text{Enc}(\{pk_j\}_j)
\]

1. **anonymous multi-KEM**

address ciphertext to \(\{pk_j\}_j\);

\(sk_j\) decrypts \(c\) to \(m_j\);

\(c\) hides \(pk_j\) recipients
technical overview

client (with \(\{sk_i\}_i\)):

\[ \{\tilde{m}_i := \text{Dec}(sk_i, c)\}_i \]

server (with \(\{pk_j\}_j\)):

\[ c, \{m_j\}_j \leftarrow \text{Enc}\left(\{pk_j\}_j\right) \]

- **1. anonymous multi-KEM**
  - address ciphertext to \(\{pk_j\}_j\);
  - \(sk_j\) decrypts \(c\) to \(m_j\);
  - \(c\) hides \(pk_j\) recipients

- **2. private set intersection**
  - each party has set of items;
Technical overview

Client (with \(\{sk_i\}_i\)): \[
\{\tilde{m}_i \coloneqq \text{Dec}(sk_i, c)\}_i
\]

Server (with \(\{pk_j\}_j\)): \[
c, \{m_j\}_j \leftarrow \text{Enc}\left(\{pk_j\}_j\right)
\]

1. Anonymous multi-KEM
   - address ciphertext to \(\{pk_j\}_j\);
   - \(sk_j\) decrypts \(c\) to \(m_j\);
   - \(c\) hides \(pk_j\) recipients

2. Private set intersection
   - each party has set of items;
   - client learns intersection;
   - \(\text{PSI}\)

\[\{\tilde{m}_i\}_i \cap \{m_j\}_j\]
technical overview

client (with \( \{sk_i\}_i\)):

\[
\{\hat{m}_i := \text{Dec}(sk_i, c)\}_i
\]

server (with \( \{pk_j\}_j\)):

\[
c, \{m_j\}_j \leftarrow \text{Enc}\left(\{pk_j\}_j\right)
\]

1. anonymous multi-KEM

address ciphertext to \( \{pk_j\}_j \);
\( sk_j \) decrypts \( c \) to \( m_j \);
\( c \) hides \( pk_j \) recipients

2. private set intersection

each party has set of items;
client learns intersection;
server learns whether empty
technical overview & contributions

client (with \(\{sk_i\}_i\)):

\[
\{\hat{m}_i := \text{Dec}(sk_i, c)_i\}
\]

\[
\{\hat{m}_i\}_i \cap \{m_j\}_j \leftarrow \text{PSI}
\]

\(\cap = \emptyset?\)

server (with \(\{pk_j\}_j\)):

\[
c, \{m_j\}_j \leftarrow \text{Enc}\left(\{pk_j\}_j\right)
\]

---

1. anonymous multi-KEM

single MKEM construction supporting RSA, ECDSA, & EdDSA

2. private set intersection

each party has set of items; client learns intersection; server learns whether empty

[add "proof of nonempty intersection" to [RosulekTrieu21]]
technical overview & contributions

client (with \(\{sk_i\}_i\)):

\[
\{\hat{m}_i := \text{Dec}(sk_i, c)\}_i
\]

server (with \(\{pk_j\}_j\)):

\[
c, \{m_j\}_j \leftarrow \text{Enc}(\{pk_j\}_j)
\]

1. anonymous multi-KEM
   single MKEM construction supporting RSA, ECDSA, & EdDSA

2. private set intersection
   add “proof of nonempty intersection” to [RosulekTrieu21] PSI
technical overview & contributions

client (with $\{sk_i\}_i$):

$$\{\hat{m}_i := \text{Dec}(sk_i, c)\}_i$$

server (with $\{pk_j\}_j$):

$$c, \{m_j\}_j \leftarrow \text{Enc}(\{pk_j\}_j)$$

1. anonymous multi-KEM
   - single MKEM construction supporting RSA, ECDSA, & EdDSA
   - add “proof of nonempty intersection” to [RosulekTrieu21] PSI

2. private set intersection
   - full UC security analysis
**concrete performance (in OpenSSH):**

<table>
<thead>
<tr>
<th># of keys</th>
<th>RSA keys only</th>
<th>{EC,Ed}DSA keys only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(worst case for us)</td>
<td>(best case for us)</td>
</tr>
<tr>
<td>client</td>
<td>server</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>comm</td>
<td>time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>comm</td>
</tr>
<tr>
<td>5</td>
<td>1000</td>
<td>1200 ms 460 kB</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>214 ms 41 kB</td>
</tr>
</tbody>
</table>

2 commodity desktop computers on LAN

[github.com/osu-crypto/PSIPK-ssh](https://github.com/osu-crypto/PSIPK-ssh)
concrete performance (in OpenSSH):

<table>
<thead>
<tr>
<th># of keys</th>
<th>RSA keys only</th>
<th>{EC,Ed}DSA keys only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(worst case for us)</td>
<td>(best case for us)</td>
</tr>
<tr>
<td>client</td>
<td>server</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>time: 60 ms</td>
<td>time: 9 ms</td>
</tr>
<tr>
<td></td>
<td>comm: 12 kB</td>
<td>comm: 8 kB</td>
</tr>
</tbody>
</table>

github.com/osu-crypto/PSIPK-ssh

2 commodity desktop computers on LAN
**concrete performance (in OpenSSH):**

<table>
<thead>
<tr>
<th># of keys</th>
<th>RSA keys only</th>
<th>{EC,Ed}DSA keys only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(worst case for us)</td>
<td>(best case for us)</td>
</tr>
<tr>
<td>client</td>
<td>server</td>
<td>time</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>60 ms</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>320 ms</td>
</tr>
</tbody>
</table>

github.com/osu-crypto/PSIPK-ssh

2 commodity desktop computers on LAN
**concrete performance (in OpenSSH):**

<table>
<thead>
<tr>
<th># of keys</th>
<th>RSA keys only (worst case for us)</th>
<th>{EC,Ed}DSA keys only (best case for us)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>comm</td>
</tr>
<tr>
<td>client</td>
<td>server</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>60 ms</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>320 ms</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>1200 ms</td>
</tr>
</tbody>
</table>

github.com/osu-crypto/PSIPK-ssh

2 commodity desktop computers on LAN
our protocol

client
set of secret keys
# of server keys;
identity of authorized keys

server
set of “authorized” public keys
# of client keys;
were any of them authorized?

✓ efficient, practical
✓ mixture of existing RSA & EC keys
✓ safe without special per-site configuration

github.com/osu-crypto/PSIPK-ssh
ia.cr/2022/740
client

set of secret keys

# of server keys;

identity of authorized keys

server

set of “authorized” public keys

# of client keys;

were any of them authorized?

☑ efficient, practical
☑ mixture of existing RSA & EC keys
☑ safe without special per-site configuration

github.com/osu-crypto/PSIPK-ssh
ia.cr/2022/740
(backup slides)
github over SSH:

Client → github.com

authenticate server

username = git

authenticate

negotiate choice of pk

server must decide set of authorized keys before running our protocol!

server does not know repository name yet!

use repository name as username

commit to repository name
github over SSH:

client

authenticate server

authenticate

server must decide set of authorized keys before running our protocol!

I server does not know repository name yet!

I use repository name as username

negotiate choice of pk

commit to repositoryname
github over SSH:

client → github.com
authenticate server

username = git

negotiate choice of pk

authenticate

commit to repositoryname

server must decide set of authorized keys before running our protocol!
github over SSH:

- Client
- authenticate server
  - username = git
  - negotiate choice of pk
- authenticate
- commit to repositoryname

- Server must decide **set of authorized keys** before running our protocol!
- Server does not know repository name yet!
github over SSH:

client, new.github.com

authenticate server

username = repositoryname

our protocol

server must decide set of authorized keys before running our protocol!

server does not know repository name yet!

use repository name as username

authenticate

commit