password seeded public key authentication

Philipp Lay

15.8.2015

Disclaimer

- This is a new protocol (first presentation)
- Everyone is invited to find a flaw
- I pay a club-mate/beer/pizza for someone finding a flaw in the scheme, that needs correction

Problems with passwords

- easy to forget
- low entropy
- companies lose them
- servers do potentially know them!

=> real solution: use 2-factor authorization, but this may take a while

classic solution: use KDF to encrypt passwords

- idea: hash password together with a salt
- use a special hash, that is very expensive in both cpu and memory usage
- Problem 1: server performance
- Problem 2: server potentially knows the password
- Problem 3: classic challenge-response protocols are not possible
- We need a good KDF too, but will use it differently

Features of PSPKA

- Heavy KDF computation goes to the client
- Server learns no information about the password
- Secure password verification over unsecure channel
- Very simple with easy implementation: around 100 lines of code

What do we need?

We need 2 cryptographic building blocks:

- a good modern KDF, like Argon2i
- signature scheme with unstructured secret key space, like ed25519

The ed25519 signature scheme

- Very popular: secure design, trusted creators, fast
- genpub, sign, verify
- implementations: NaCl, libsodium and libeddsa

combine KDF with ed25519

- use KDF to generate 32byte we call sk from password and identity

$$\text{sk} := \text{KDF}(\text{identity} \mid \text{password}, \text{salt})$$

- use genpub to derive pk from sk

$$\text{pk} := \text{genpub}(\text{sk})$$

- our password hash is a base64 encoding of (salt, iter, pk)
- example: With identity = `foo` and password = `bar` we get

```
M+Dos9X7jT47e8GNlWednQD0AQAAAAAA9ft6nZYT66ZsYEa7jGQ663K0Q6gWSf6iZuCqwhkwaak
```

Authorization with PSPKA

- server generates 16 byte random data Rs and sends (salt, iter, Rs)
- client: sk := KDF(identity | password, salt)
- client generates a user random Ru and calculates

$$\text{sig} := \text{sign}(\text{sk},\ \ \text{Rs} \mid \text{Ru} \mid \text{context})$$

- send (Ru, sig) to server
- server: verify(pk, Rs | Ru | context)

Security details

- Why do we need a 'context' description?
- Why do we hash the identity together with the password?
- Why do we include Ru?

- Question: Should Rs and Ru be 32 byte instead of 16?

DH channel binding

- use Diffie-Hellman to find a shared secret ssec
- Run PSPKA and use ssec as context

Attacks

- Online attack: Spy on user's salt and mount an online dictionary or brute force attack.
- Offline attack: With the password hash an offline attack could be performed
- Protocol Attack: Try to trick the protocol.
- Attack on EC: Factor the secret key from public key (very unlikely).

May the source be with you

- demo implementation: `https://github.com/phlay/pspka`
- eddsa implementation: `https://github.com/phlay/libeddsa`
- next demo will use Argon2i
- browser implementation?

Questions?