# SCRAM

## and other interesting things about authentication

Marko Mrdjenovič, Klevio

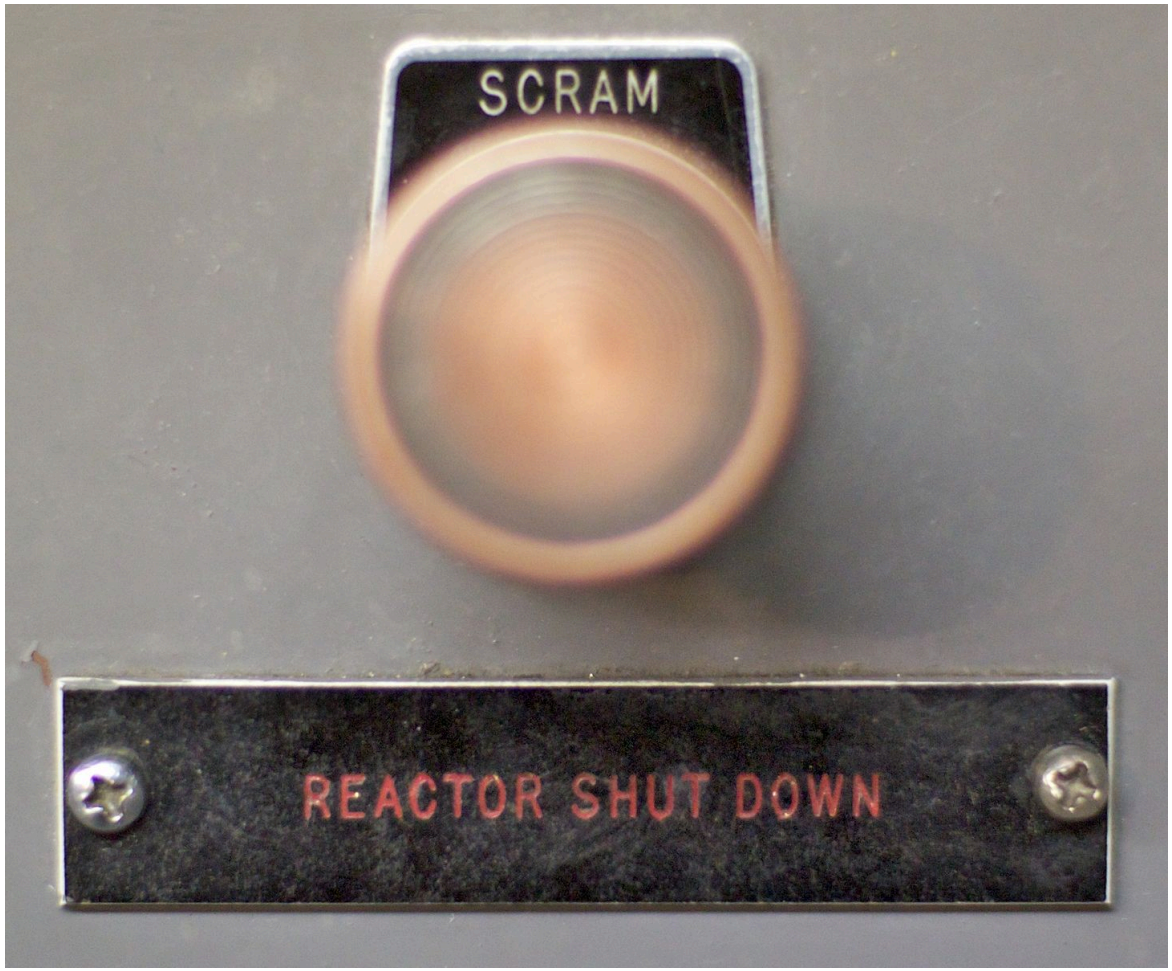
LjPyMeetup February 2019

# Marko Mrdjenovič

Klevio

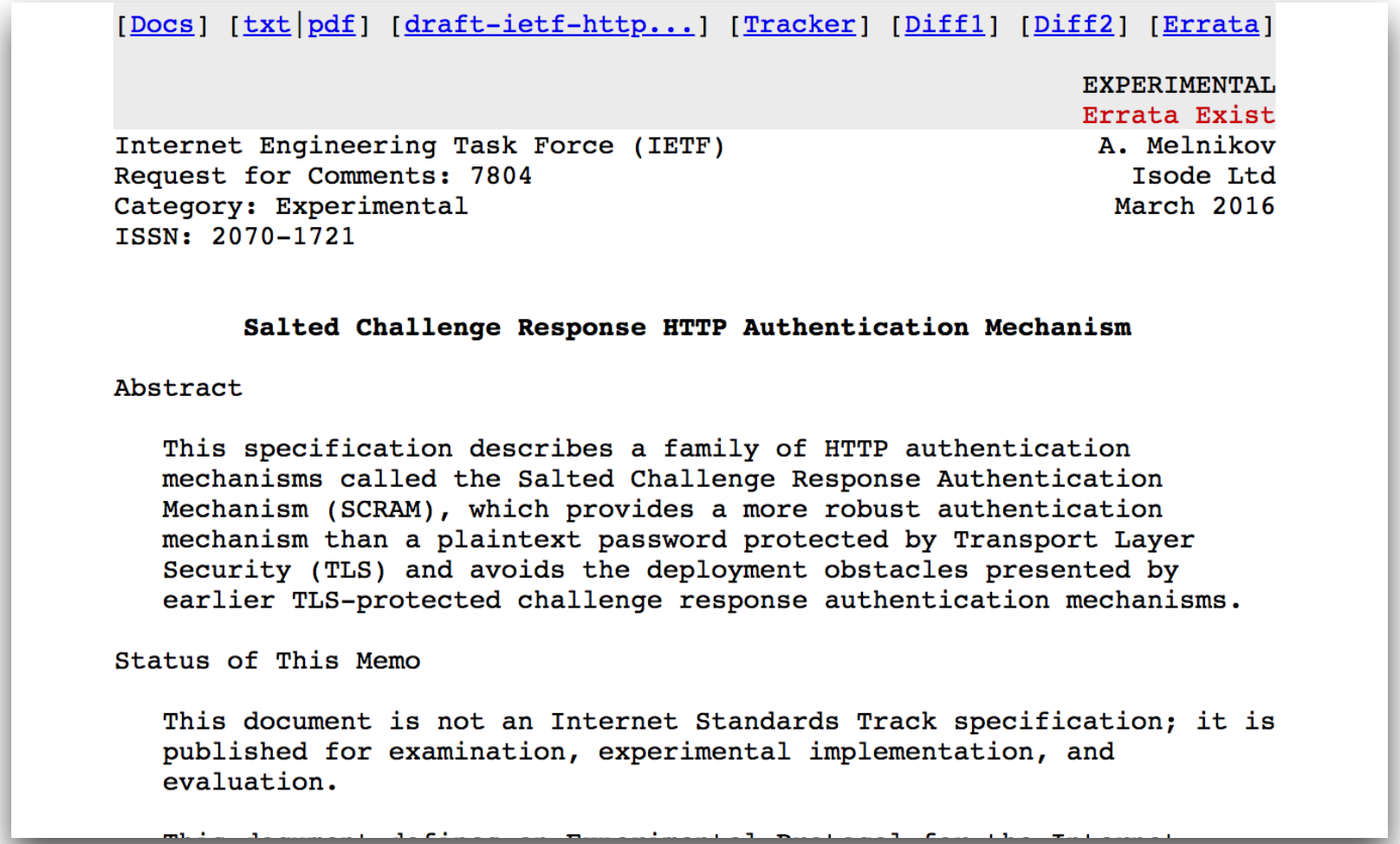
# About me

- Basic, Pascal
- HTML, php, JavaScript, Macromedia Flash 4, VBScript
- Java
- IA & UX
- JavaScript, Python




# SCRAM

Safety Control Rods Activation Mechanism



[Docs] [txt|pdf] [draft-ietf-http...] [Tracker] [Diff1] [Diff2] [Errata]

EXPERIMENTAL
Errata Exist

Internet Engineering Task Force (IETF) — A. Melnikov
Request for Comments: 7804 — Isode Ltd
Category: Experimental — March 2016
ISSN: 2070-1721

**Salted Challenge Response HTTP Authentication Mechanism**

Abstract

This specification describes a family of HTTP authentication mechanisms called the Salted Challenge Response Authentication Mechanism (SCRAM), which provides a more robust authentication mechanism than a plaintext password protected by Transport Layer Security (TLS) and avoids the deployment obstacles presented by earlier TLS-protected challenge response authentication mechanisms.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for examination, experimental implementation, and evaluation.

# SCRAM

## Salted Challenge Response Authentication Mechanism

RFC 5802, RFC 7677, RFC 7804

# password-based challenge-response authentication mechanism

providing authentication of a user to a server

**Authentication**
is the act of confirming the truth of an attribute of a single piece of data claimed true by an entity.

from Greek: αὐθεντικός *authentikos*, "real, genuine", from αὐθέντης *authentes*, "author"

**Authorisation**
is the process of verifying that "you are permitted to do what you are trying to do”

**= Not what we’re talking about.**

# Currently

# UIs

- Password
- 2FA
  - OTP
  - Out of band
- Third-party (github, facebook, twitter, …)

# Challenges

- Making it more secure usually means worse UX
  - Complex passwords hard to remember
  - 2FA usually means longer time to log in
  - Shorter sessions
- Do you trust 3rd party providers?

# Exchange

| Alice | Eve | Bob |
| --- | --- | --- |
| “Hey Bob, it’s Alice, the password we agreed on is PASSWORD” | PASSWORD for Alice | |
| | SESS_ID | “Hi Alice! Use SESS_ID so we don’t need to do this every time” |

# Exchange

| Alice | Eve | Bob |
| --- | --- | --- |
| “Hey Bob, it’s Alice!” | Alice | |
| | | “Hey, what’s the password we agreed on?” |
| “PASSWORD” | PASSWORD for Alice | |
| | SESS_ID | “Hi Alice! Use SESS_ID so we don’t need to do this every time” |

# Vectors

- Steal the password
  - From user
  - From client
  - From network
- Steal the session
  - From network
  - From client

# Protections

- Make users use strong passwords
- OTP
- OOB
- Make users change password often
- Shorter sessions
- Additional auth for important actions

# APIs

- Password-like:
  - API key
  - OAuth 2.0
  - JWT
- Sign every request:
  - OAuth 1.0a

# Phones

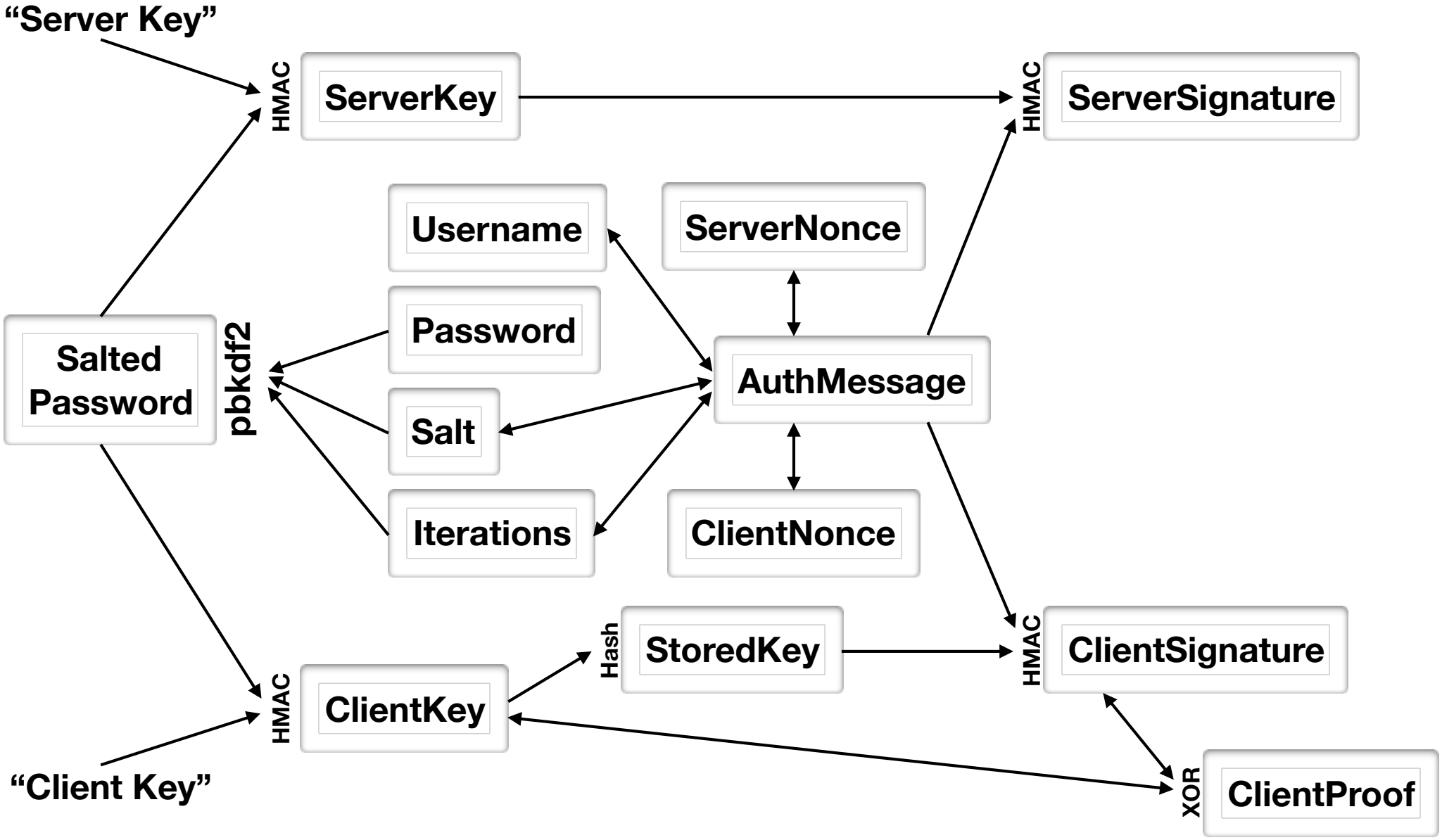
- OTP still works, but is quite a bit more annoying
- 2FA options limited
- longer password useful only through managers
- users don’t expect to log in every time

# SCRAM

# Why?

- Stealing server DB is not enough to impersonate a client
- Server cannot impersonate a client
- Supports server authorised proxy
- Mutual authentication is supported, only client is named

# Components


"Server Key"
HMAC
ServerKey
HMAC
ServerSignature
Username
ServerNonce
Password
Salted Password
pbkdf2
AuthMessage
Salt
Iterations
ClientNonce
Hash
StoredKey
HMAC
ClientSignature
HMAC
ClientKey
"Client Key"
XOR
ClientProof

# Exchange

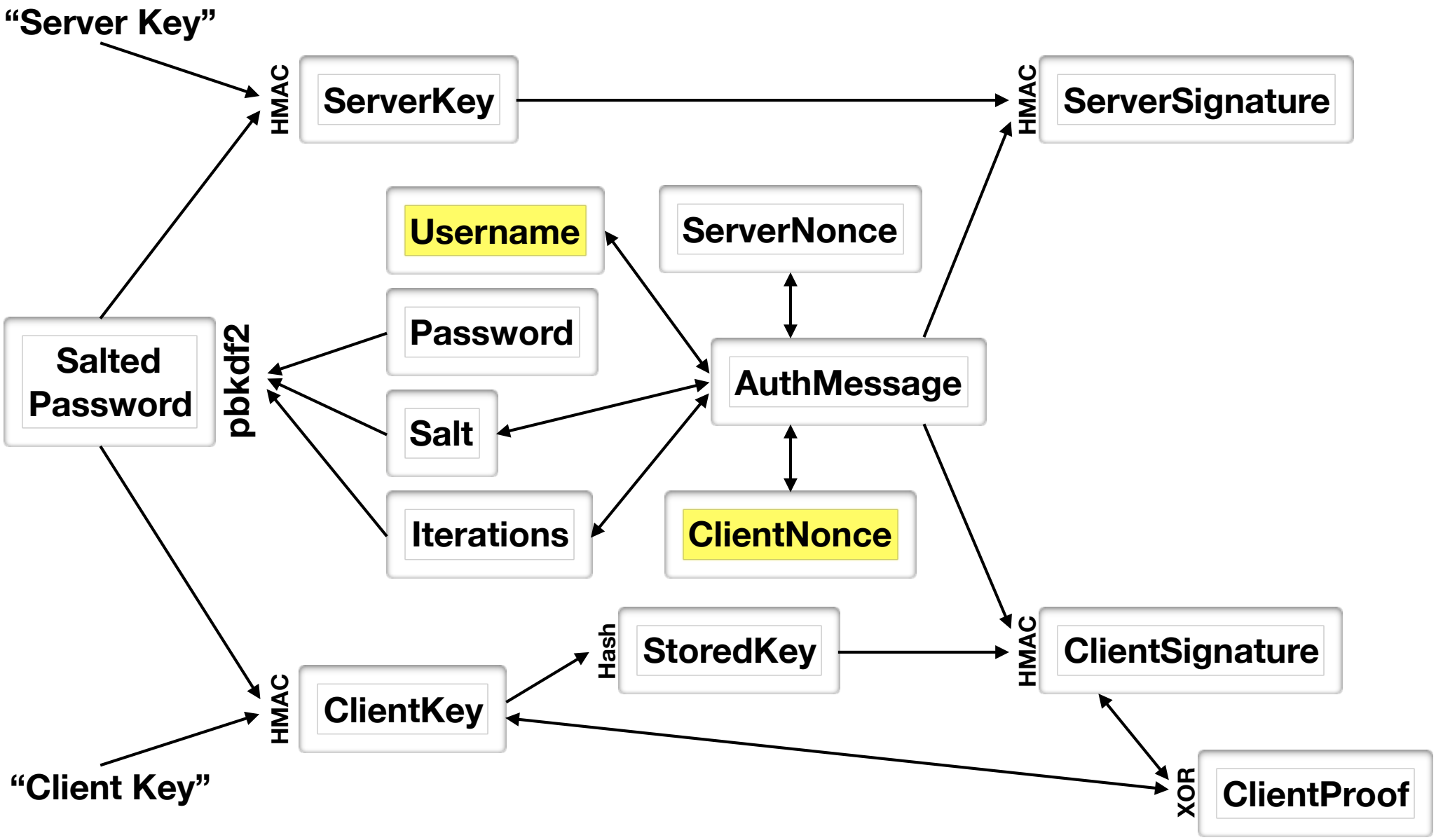
| Alice | Eve | Bob |
| --- | --- | --- |
| “Hey Bob, it’s Alice, here’s some random data” | Alice, client nonce | |
| | server nonce, salt, iteration count | “Hey, I want password proof based on this salt and iteration count with added random data.” |
| “Sure Bob, here’s that random data and the proof.” | client proof | |
| | server signature | “Alice, here’s my proof.” |
| Great, Bob. | | |

# 1. client-first msg

```
n,,n=user,r=rOprNGfwEbeRWgbNEkqO
```

- gs2_header: `n,,`
- username: `n=user`
- client nonce: `r=rOprNGfwEbeRWgbNEkqO`

# client-first msg

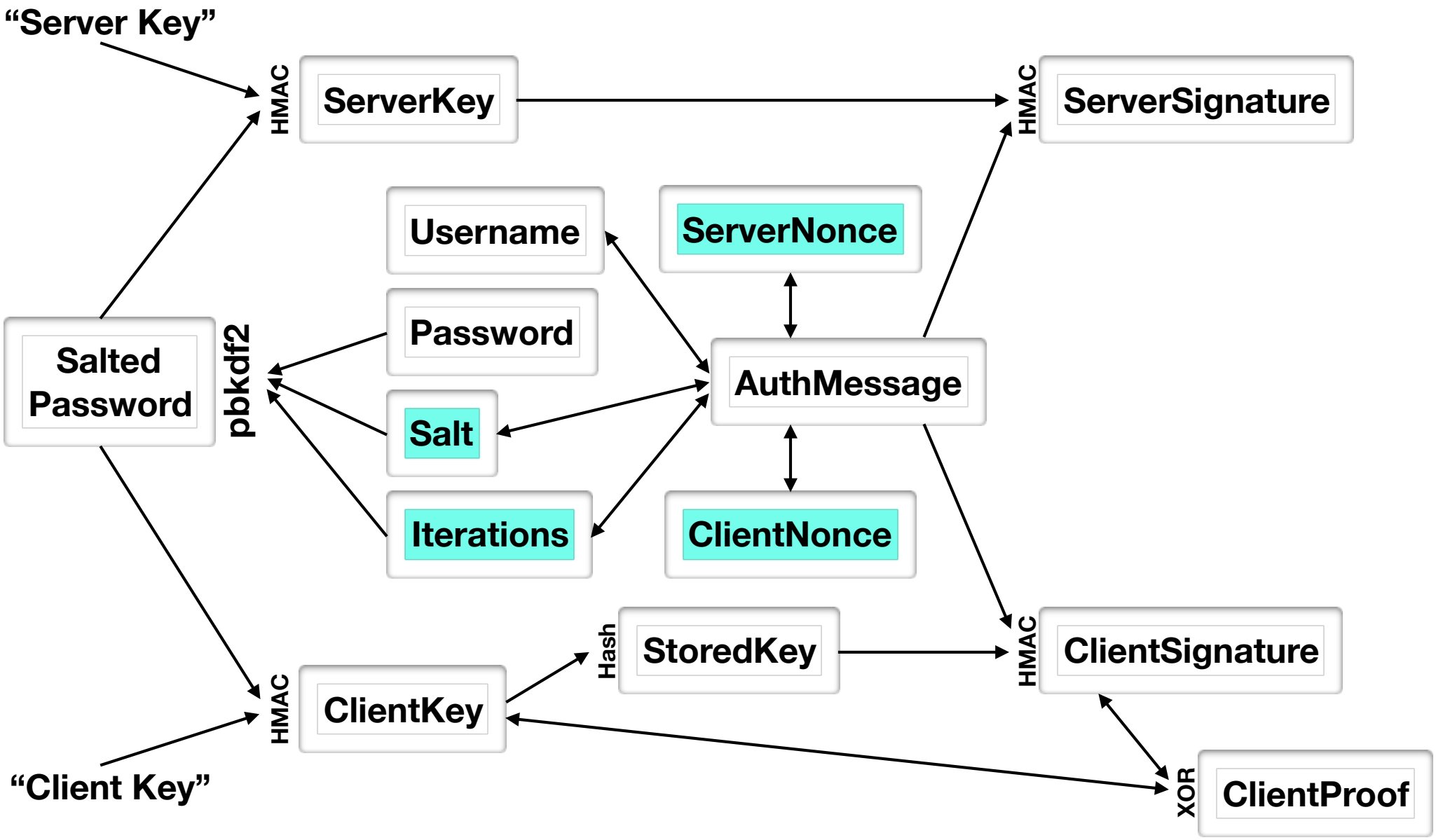


# 2. server-first msg

```
r=rOprNGfwEbeRWgbNEkqO%hvYDpWUa2RaTCAfuxFIlj)
hNlF$k0,s=W22ZaJ0SNY7soEsUEjb6gQ==,i=4096
```

- client+server nonce:
  `r=rOprNGfwEbeRWgbNEkqO%hvYDpWUa2RaTCAfuxFIlj)hNlF$k0`
- b64(salt): `s=W22ZaJ0SNY7soEsUEjb6gQ==`
- iterations: `i=4096`

# server-first msg

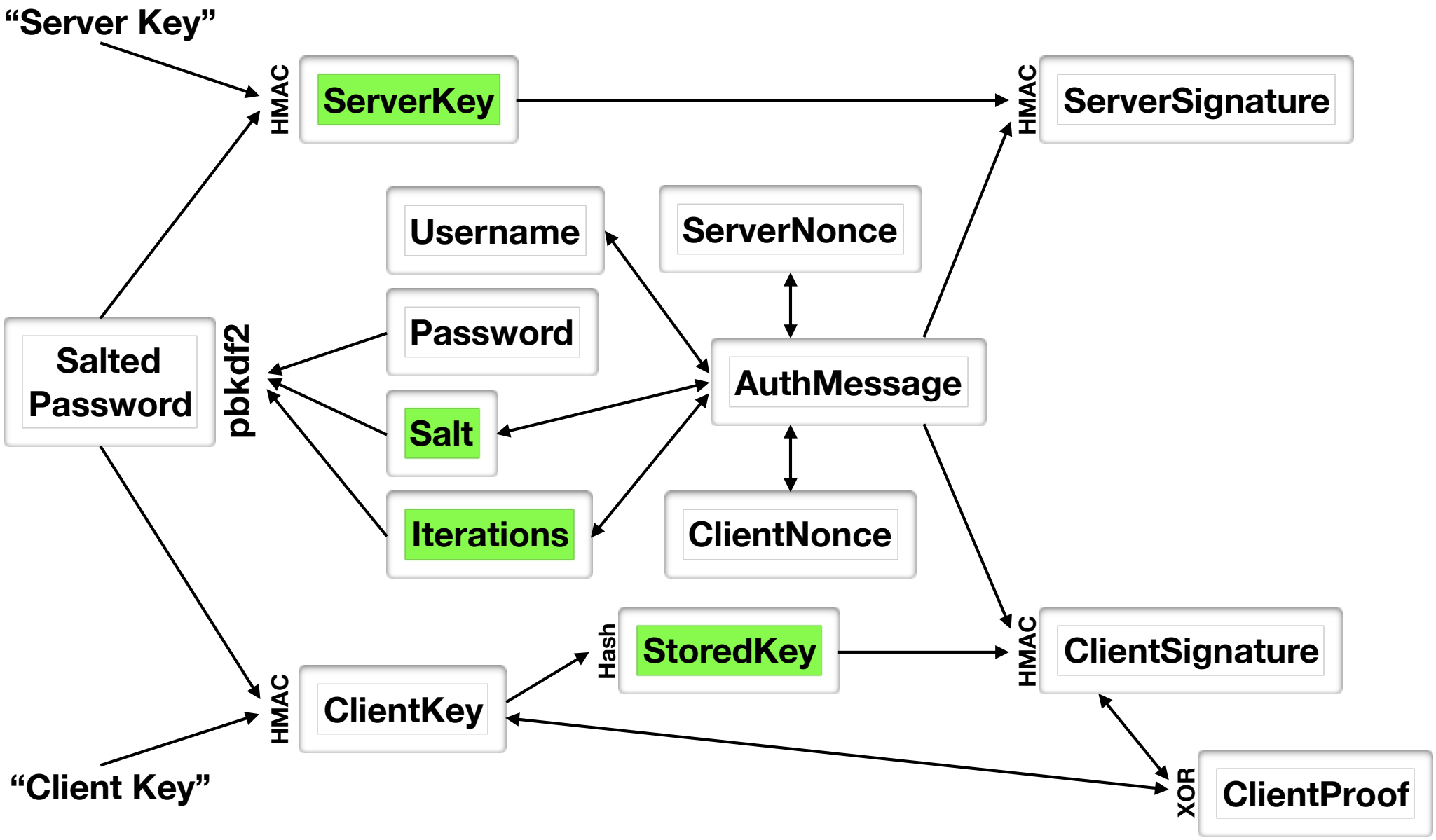


# server storage

```
W22ZaJ0SNY7soEsUEjb6gQ==$4096$sha256:WG5d8oPm
3OtcPnkdi4Uo7BkeZkBFzpcXkuLmtbsT4qY=:wfPLwcE6
nTWhTAmQ7tl2KeoiWGPlZqQxSrmfPwDl2dU=
```

- based on user in client-first, server fetches from storage:
  - salt
  - iterations
  - stored key: `hash(hmac(pbkdf2(pwd, salt, i), “Client Key”)))`
  - server key: `hmac(pbkdf2(pwd, salt, i), “Server Key”)`

# server storage

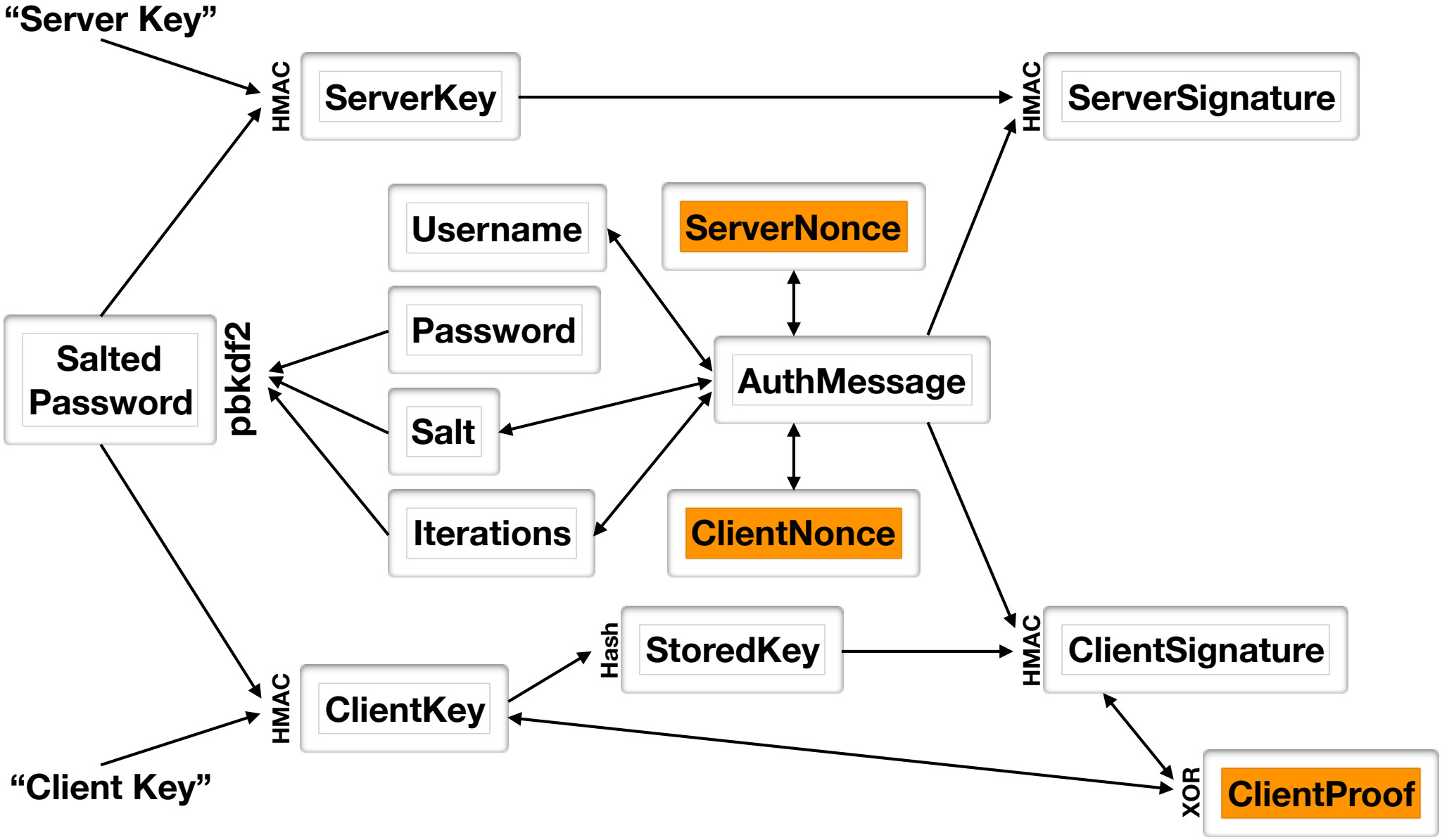


# 3. client-final

```
c=biws,r=rOprNGfwEbeRWgbNEkqO%hvYDpWUa2RaTCAf
uxFIlj)hNlF$k0,p=dHzbZapWIk4jUhN+Ute9ytag9zjf
MHgsqmmiz7AndVQ=
```

- b64(gs2 header): `c=biws`
- client+server nonce:
  `r=rOprNGfwEbeRWgbNEkqO%hvYDpWUa2RaTCAfuxFIlj)hNlF$k0`
- b64(client proof):
  `p=dHzbZapWIk4jUhN+Ute9ytag9zjfMHgsqmmiz7AndVQ=`

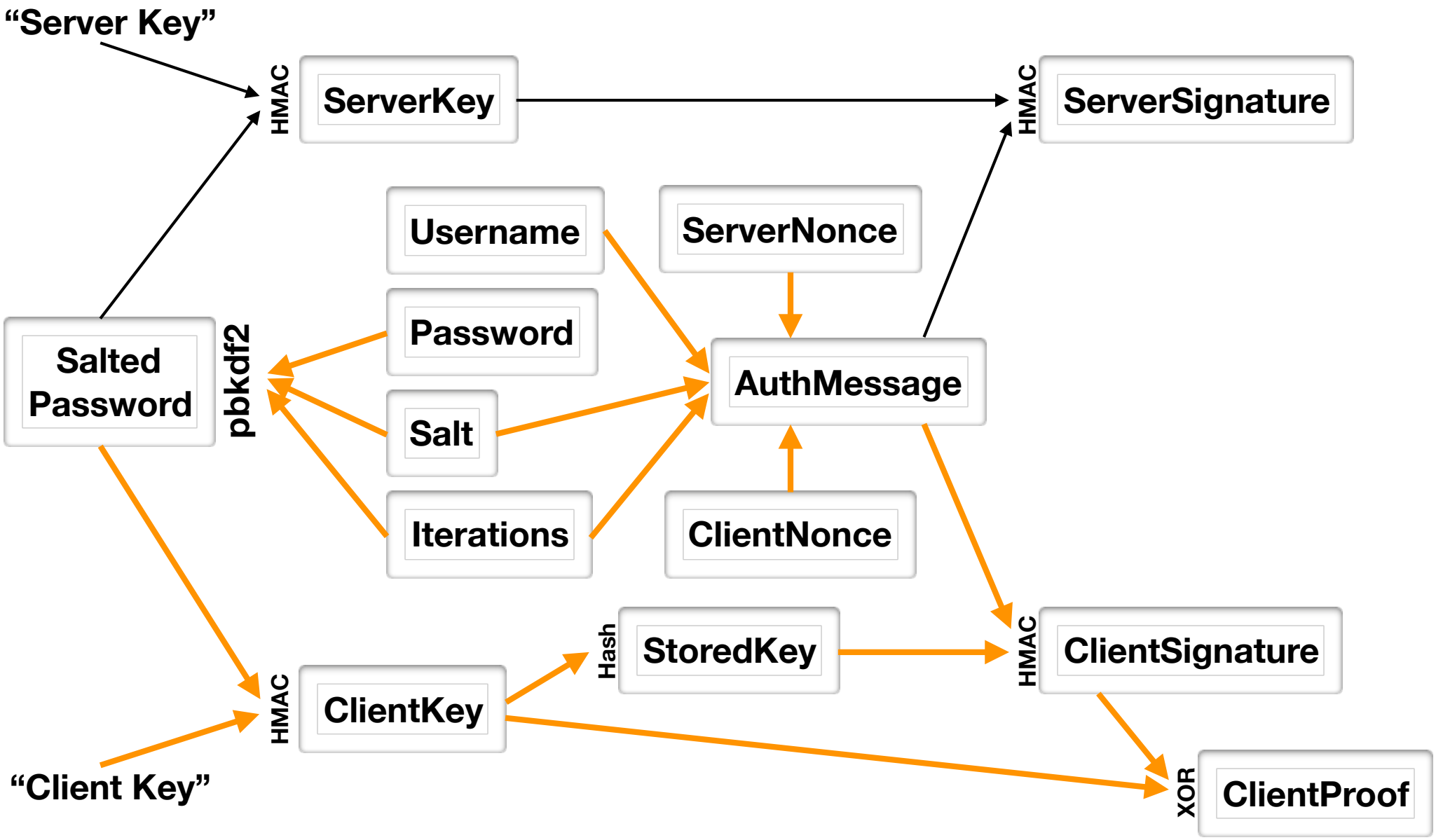
# client-final



# calculating client proof

- calculate SaltedPassword with pbkdf2 from:
  - entered password
  - salt and iterations count from server-first
- calculate ClientKey with HMAC from SaltedPassword and “Client Key”
- calculate StoredKey with Hash from ClientKey
- prepare AuthMessage from first 3 messages*
- calculate ClientSignature with HMAC from StoredKey and AuthMessage
- calculate ClientProof with XOR from ClientKey and ClientSignature

# client path

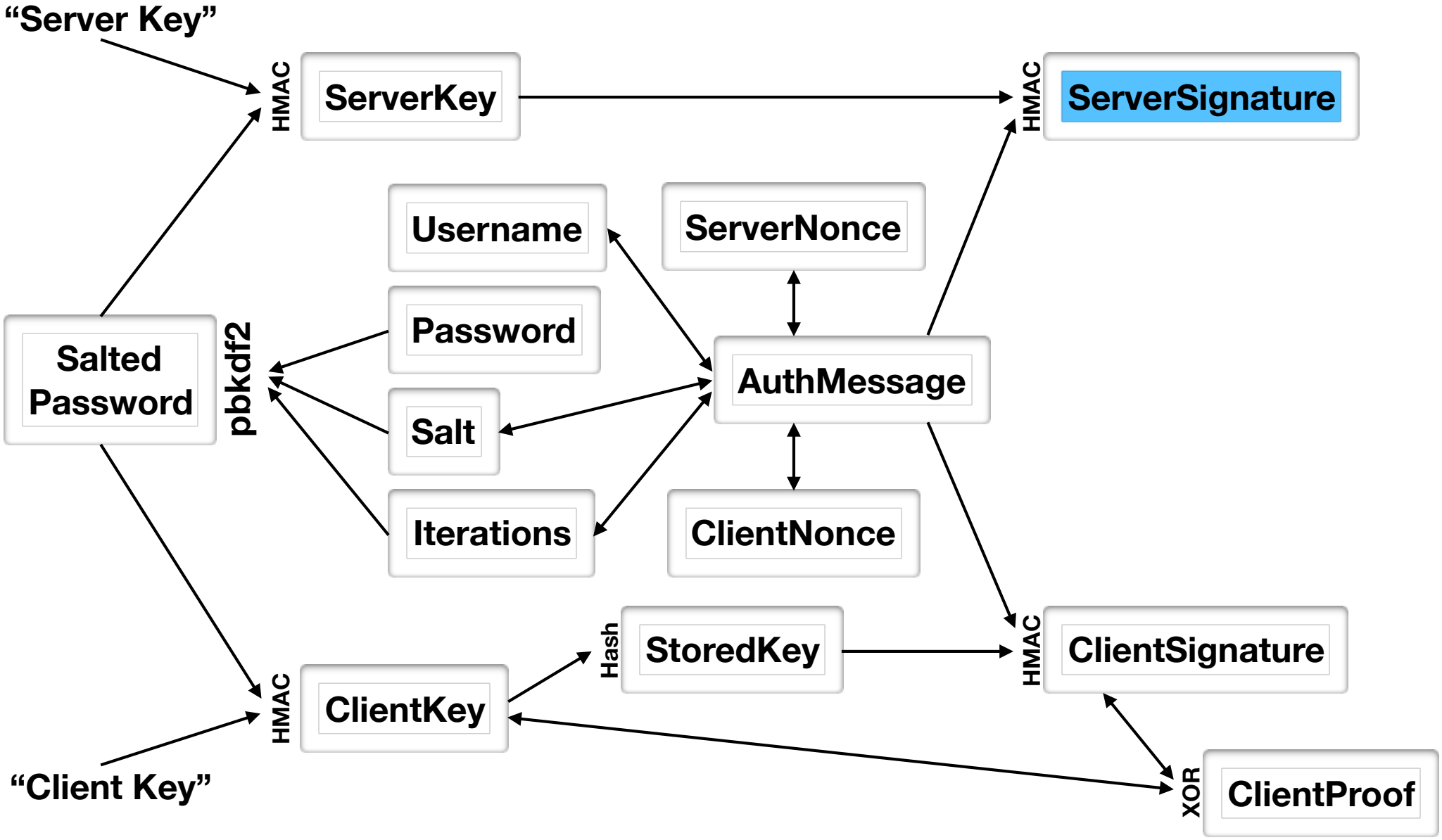

"Server Key"
HMAC
ServerKey
HMAC
ServerSignature
Username
ServerNonce
Password
Salted Password
pbkdf2
AuthMessage
Salt
Iterations
ClientNonce
Hash
StoredKey
HMAC
ClientSignature
HMAC
ClientKey
"Client Key"
XOR
ClientProof

# 4. server-final

```
v=6rriTRBi23WpRR/wtup+mMhUZUn/
dB5nLTJRsjl95G4=
```

- b64(server signature):
  `v=6rriTRBi23WpRR/wtup+mMhUZUn/dB5nLTJRsjl95G4=`

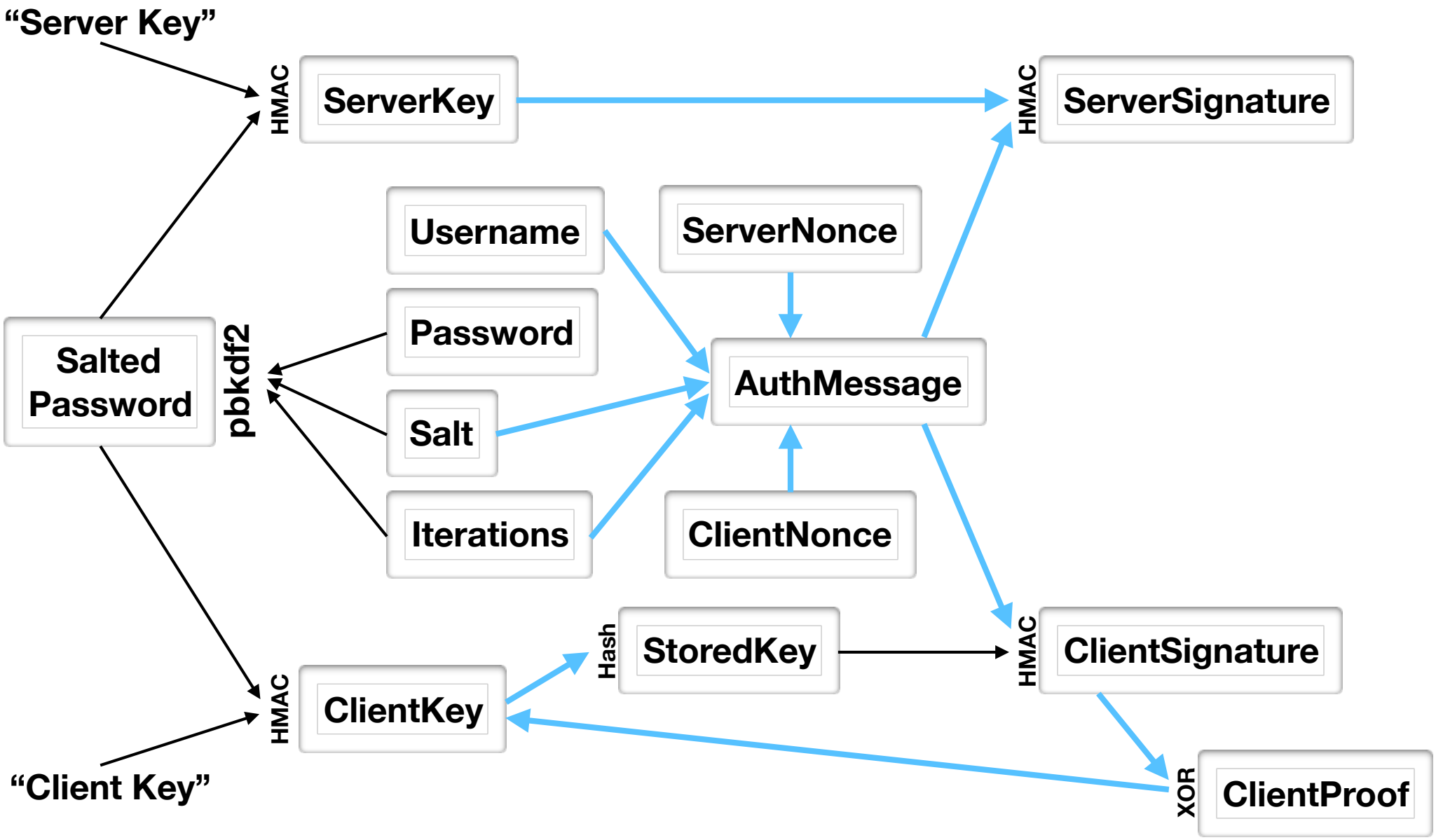
# server-final



# confirming client proof

- prepare AuthMessage from first 3 messages*
- calculate ClientSignature with HMAC from StoredKey and AuthMessage
- calculate ClientKey with XOR from ClientSignature and ClientProof from client-final
- calculate StoredKey with Hash from ClientKey
- calculate ServerSignature with HMAC from ServerKey and AuthMessage
- if calculated StoredKey and stored StoredKey are same, client knows correct password; return ServerSignature

# server path


"Server Key"
HMAC
ServerKey
HMAC
ServerSignature
Username
ServerNonce
Password
Salted Password
pbkdf2
AuthMessage
Salt
Iterations
ClientNonce
HMAC
ClientKey
Hash
StoredKey
HMAC
ClientSignature
"Client Key"
XOR
ClientProof

# 5. client-check

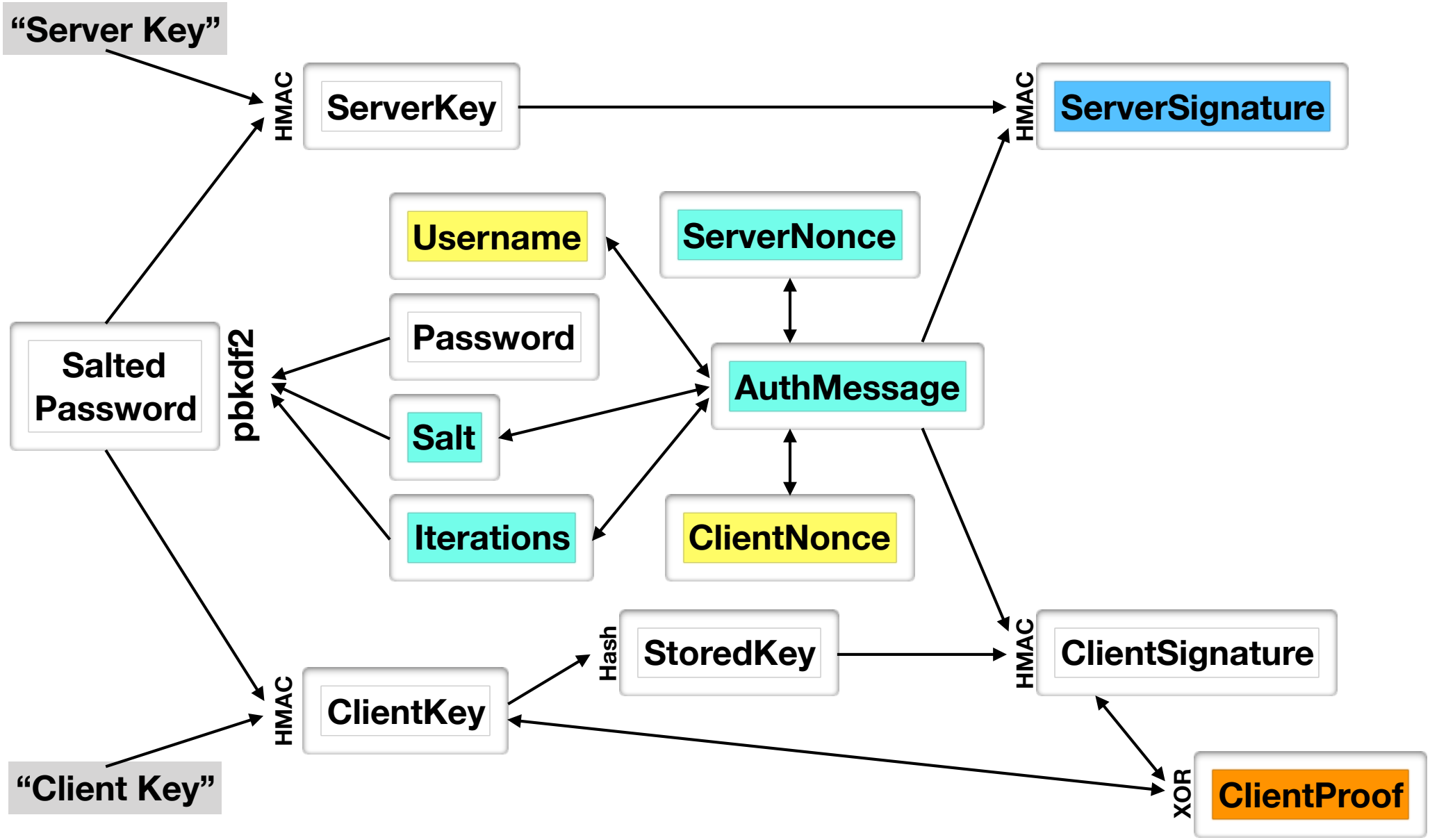
- calculate ServerKey with HMAC from SaltedPassword and “Server Key”
- calculate ServerSignature with HMAC from ServerKey and AuthMessage
- if calculated ServerSignature and ServerSignature from server-final are the same, server knows correct password

# 5. client-check

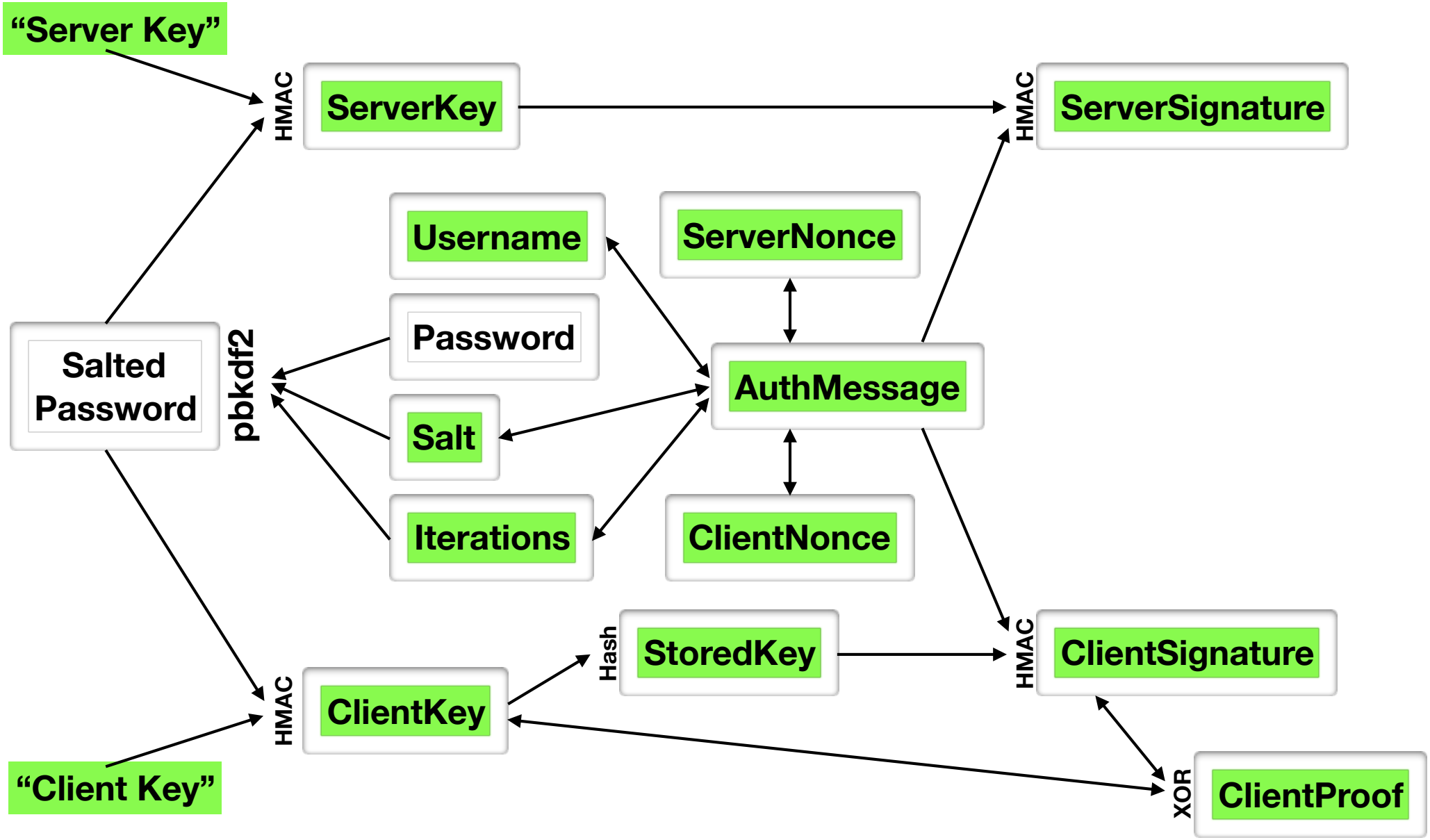
- calculate ServerKey with HMAC from SaltedPassword and “Server Key”
- calculate ServerSignature with HMAC from ServerKey and AuthMessage
- if calculated ServerSignature and ServerSignature from server-final are the same, server knows correct password

# Components

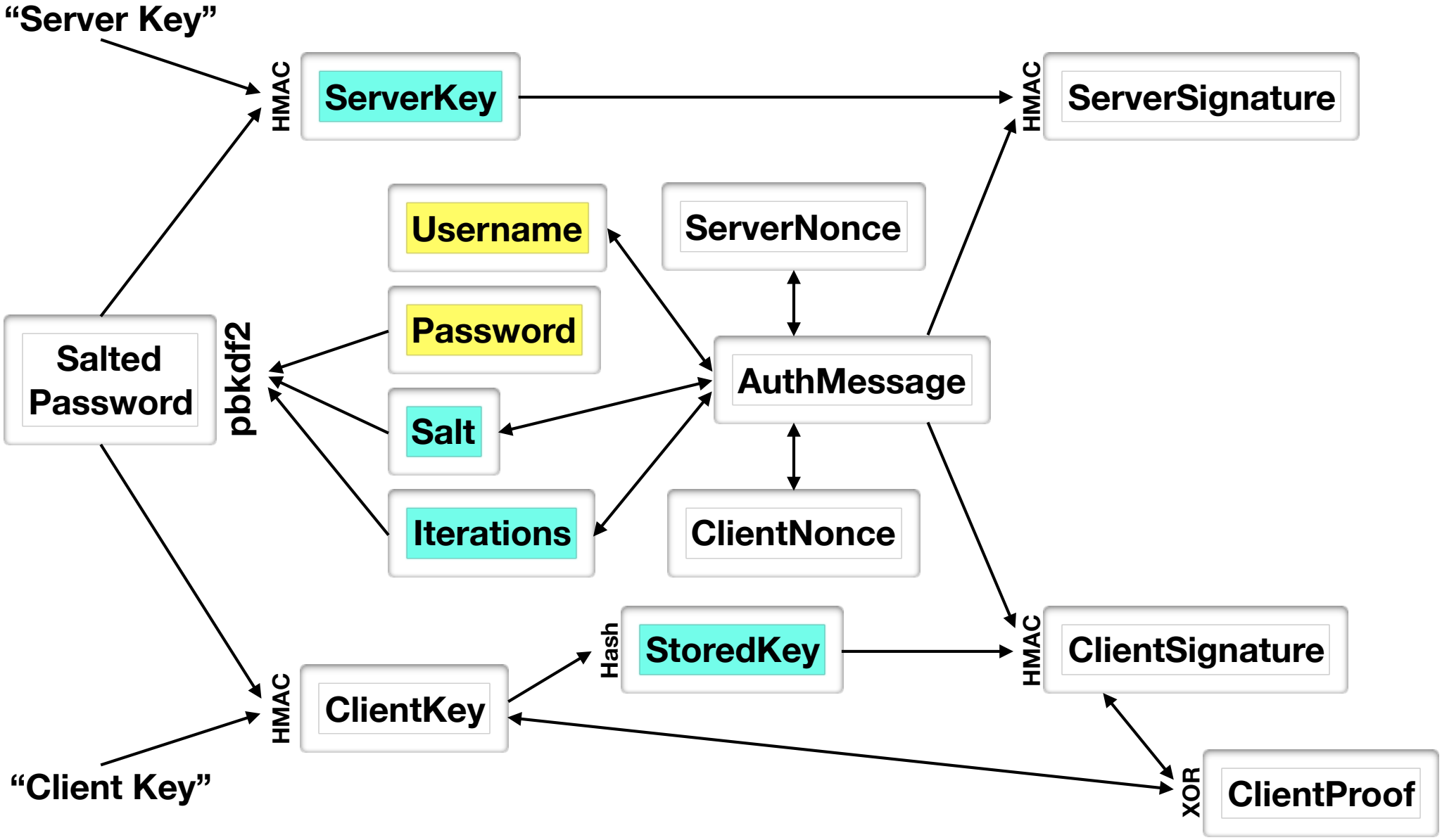
# public


"Server Key"
HMAC
ServerKey
HMAC
ServerSignature
Username
ServerNonce
Password
Salted Password
pbkdf2
AuthMessage
Salt
Iterations
ClientNonce
Hash
StoredKey
HMAC
ClientSignature
HMAC
ClientKey
"Client Key"
XOR
ClientProof

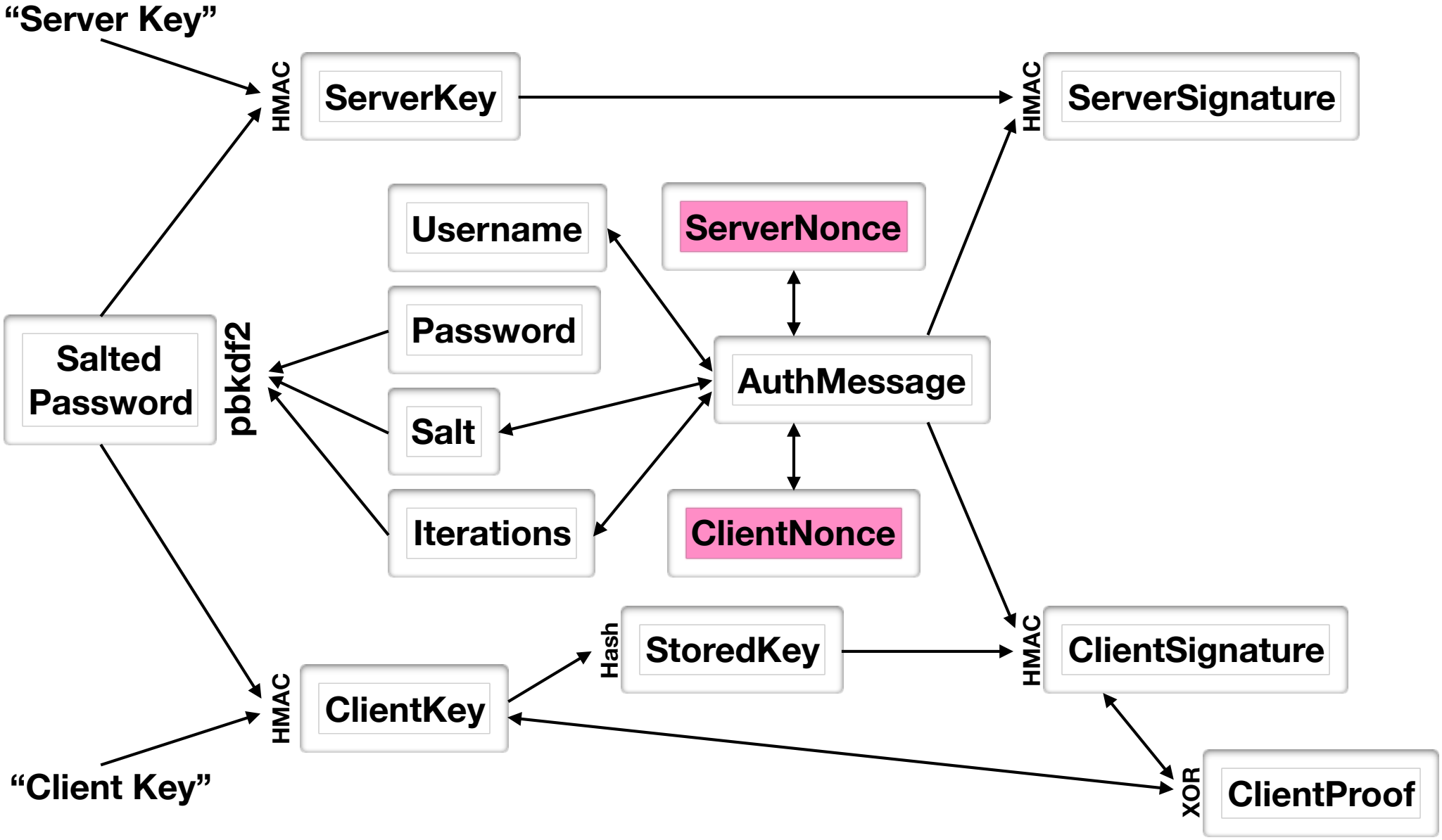
# server/client session



# server/client storage



# forward secrecy



# What now?

- Both server and client now know ClientSignature (non-public, “ephemeral”)
  - You could use it to derive an encryption key
  - You could use it to HMAC messages

# Forgot password?

- Server does not need to see the password
- When server gets Password or SaltedPassword it should calculate StoredKey and ServerKey, store them and forget the inputs

# Iteration management

- Iteration count should always be a number that guarantees processing of as much as you can spare (at least 0.1s)
- Legacy login gives you password every time, so you can change storage every time
- With SCRAM you really can’t

# Deployment

- With SCRAM having pbkdf2(pwd, salt, i) can mean you can impersonate the client
- Even if you store pbkdf2(pwd, salt, i), you can roll out SCRAM without users changing password - when a user logs in, change the salt and store the SCRAM string instead

# Where can I use it?

- Server - server with HTTP Auth
- Client - server

# Did you know about…

```
var cryptoSubtle = window.crypto.subtle;
return cryptoSubtle.importKey(
    "raw", password, {name: "PBKDF2"},
    false, ["deriveBits"]
).then(function (key) {
    return cryptoSubtle.deriveBits({
            name: "PBKDF2",
            salt: salt,
            iterations: iterations,
            hash: "SHA-256"
        }, key, 256);
}).then(function (res) {
    var ab = new Uint8Array(res);
    return btoa(String.fromCharCode.apply(null, ab));
});
```

# or...

```
var cryptoSubtle = window.crypto.subtle;
return cryptoSubtle.importKey(
   "raw", key, {name: "HMAC", hash: "SHA-256"},
   false, [“sign"]
).then(function (key) {
   return cryptoSubtle.sign(
      {name: "HMAC", hash: "SHA-256"},
      key, msg);
}).then(function (res) {
   var ab = new Uint8Array(res);
   return btoa(String.fromCharCode.apply(null, ab));
});
```

# or...

```
var cryptoSubtle = window.crypto.subtle;
return cryptoSubtle.digest(
    "SHA-256",
    data
).then(function (res) {
    var ab = new Uint8Array(res);
    return btoa(String.fromCharCode.apply(null, ab));
});
```

# window.crypto

- https://caniuse.com/#feat=cryptography
- all users:
  90.91% support
  2.49% partial
- tracked:
  95.92% support (97.25% desktop, 94.85% mobile)
  2.62% partial (5.62% desktop, 0.21% mobile)
- Live table: https://diafygi.github.io/webcrypto-examples/

# Python

- Couldn’t find a package, so I wrote one: https://github.com/friedcell/python-scramauth
- Published today, supports python3.5+
- Has 100% coverage
- Does not YET implement SCRAM HTTPAuth
- Likely still missing some checks on incoming data
- Not battle tested

# Exchange

```
import base64
import scramauth

client = scramauth.ScramClientSession(“user”, "pencil")
stored_data = scramauth.ScramSession.get_storage_string(
    "password", scramauth.ScramSession.get_nonce(16), 4096)
server = scramauth.ScramServerSession(stored_data)

c1 = client.create_client_first()  # send c1 to server

s1 = server.create_server_first(c1)  # respond with s1

c2 = client.create_client_final(s1)  # send c2 to server

s2 = server.create_server_final(c2)  # respond with s2
# if s2, client is validated on server

check = client.check_server(s2)
# if check, server is validated on client
```

# Practically

- Server will need to store data so that second client request can be handled
- If you don’t already have a session, it makes sense to store data based on full nonce (or maybe hash(nonce)) in a key-value store
- What you do after client is authenticated is up to you

# Thank you.

Questions?

# One last thing…

- At Klevio we are looking for backend developers
- On backend we use:
  - Python 3.5
  - Tornado
  - Redis
  - postgreSQL
  - JWT with EC256
  - websockets