Password Hashing Delegation: How to Get Clients to Work for You

Thomas Pornin

CGI

Passwords14 Las Vegas

Outline

http://www.bolet.org/makwa/

Password Hashing and Delegation

The Battlefield

Passwords are weak
because human users choose and remember them.

Offline dictionary attack: attacker tries passwords "at home" and can check his guesses against password-dependent values.

- *Password-based encryption:* data is encrypted with a key deterministically derived from the password.
- *Client authentication:* a *server* stores elements which are enough to decide whether a given user password is correct or not (hashed passwords).

The Battlefield

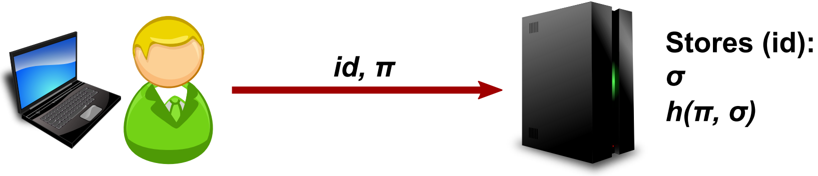
Attacker's weapons:

- *Patience:* the attacker may afford to spend several days on a hashed password; the user wants to log in within one second.
- *Parallelism:* the attacker has many passwords to try.
- *Specialized power:* the attacker can use dedicated harware and does not have a business to run.
- *Moore's law:* computers get faster over time; human brains do not.

Defender's weapons:

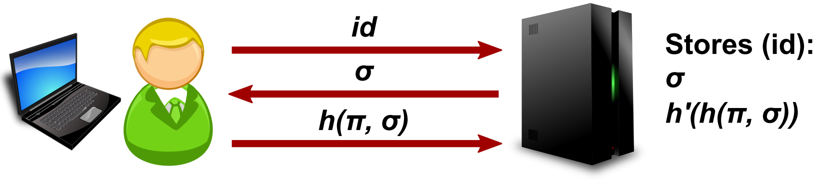
- *Salts:* prevent cost-sharing (if the attacker wants to break N hashed passwords, he must pay N times the cost).
- *Slow hashing:* the hashing function can be made arbitrarily slow so that each attacker's guess is expensive – but so is each user password verification.

Client Authentication: Classic



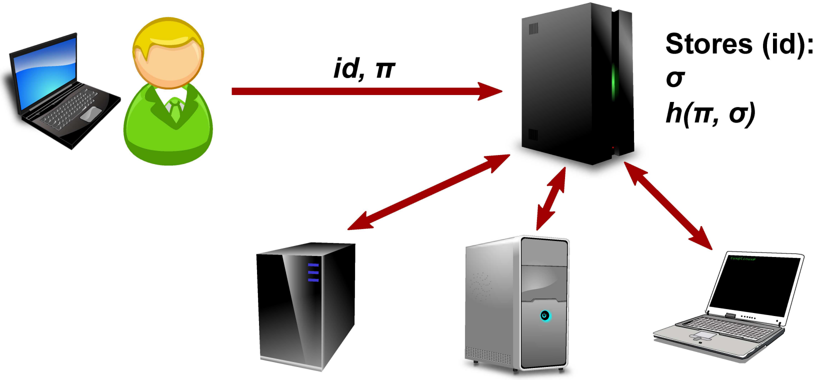
- Server stores for each user the *salt* (σ) and the hashed password ($h(\pi, \sigma)$).
- Server recomputes the hash from the password sent by the user.

Client Authentication: Server Relief



- Server stores for each user the *salt* (σ) and the hash of the hashed password ($\mathrm{h}'(\mathrm{h}(\pi, \sigma))$): hash function h' is fast (e.g. SHA-256).
- *Client* computes the slow part of the hash.

Client Authentication: Delegation



- The slow hash is computed by *untrusted* 3rd-party systems.

Password Hashing Delegation

Password Hashing Delegation is about enlisting extra computers into the defender's army.

- **Delegation systems cannot run offline dictionary attacks.**
- Hashing cost can be delegated to rented muscle (cloud...).
- Hashing cost can be delegated to *other connected clients*.
- Parallel delegation: using several delegation systems for a single password verification.

Delegation requires mathematics; it cannot be applied to just any password hashing function.

Makwa

Makwa

Makwa is a candidate to the Password Hashing Competition.

Main characteristics:

- based on modular arithmetics
- CPU-only cost (*not* memory-hard)
- algebraic structure enables advanced features: offline work factor increase, fast path, escrow
- **can be delegated**
- named after the Ojibwe name for the American black bear

Makwa Core

Let n be a *Blum integer*:

- $n = pq$ for two prime integers p and q.
- $p = 3 \pmod 4$ and $q = 3 \pmod 4$.
- p and q have similar sizes.
- n is large (at least 1280 bits, 2048 recommended).

Let $QR(n)$ the set of *quadratic residues* modulo n:

$$QR(n) = \left\{ x^2 \middle| x \in \mathbf{Z}_n \right\}$$

Properties

- Squaring is a permutation on $QR(n)$.
- It is (mostly) one-way if p and q are unknown.

Makwa Core

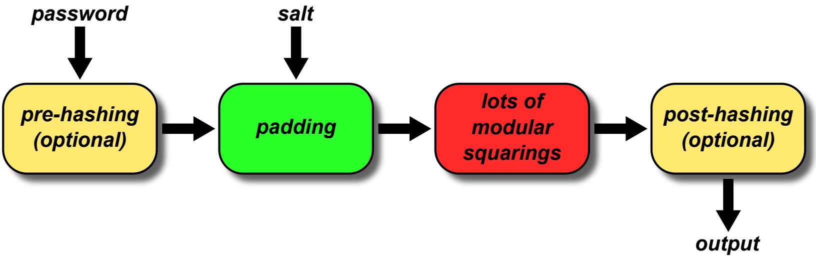
Main Idea

"Hash" the password by repeatedly squaring it modulo n.

- When p and q are unknown, no shortcut is known to speed up the computation.
- Proposed for "time-lock puzzles" since 1996[1].
- Knowledge of p and q can be used as a shortcut.
- Algebraic structure amenable to delegation.

[1] *Time-lock puzzles and timed-release Crypto*, R. L. Rivest, A. Shamir and D. A. Wagner, Massachusetts Institute of Technology, 1996.

Makwa Structure



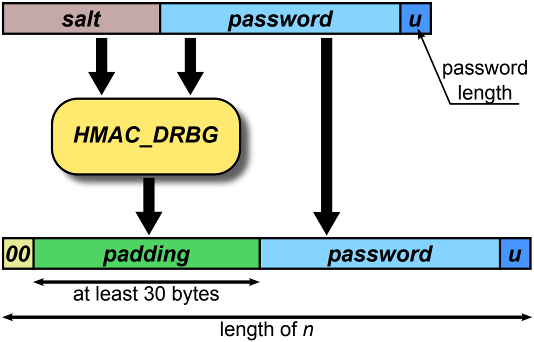
- Pre-hashing allows for passwords of arbitrary length.
- Post-hashing yields unbiased bytes (KDF usage).
- Hashing and padding use HMAC_DRBG.

The Makwa H KDF: HMAC_DRBG

- Proposed as a PRNG since *ca* 2004 by NIST (published as part of SP 800-90A since 2006).
- Security "proven" in 2008[1].
- Uses HMAC internally (recommended underlying hash function: SHA-256).
- Used in Makwa for all hashing-like steps (pre-hashing, padding and post-hashing).
- Performance of H is **not** relevant to Makwa.

[1] *Security Analysis of DRBG Using HMAC in NIST SP 800-90*, S. Hirose, Information Security Applications (WISA 2008), LNCS 5379, 2008.

Padding



- deterministic
- reversible
- depends on salt *and* password
- pseudorandom bytes are most signifiant (big-endian convention)

Squarings

Modulus n

- The modulus is a parameter to Makwa.
- Modulus generation: similar to RSA private key generation.
- Factorization needs not be known to anybody for proper operation.

- Work factor: $w \geqslant 0$
- $w + 1$ squarings: equivalent to raising to power 2^{w+1} (there is always at least one squaring)
- With $w = 0$: equivalent to Rabin encryption.
- CPU cost: proportional to w.

Features: Fast Path

If p and q are known, a "fast path" computation is feasible:

- Compute modulo p and q separately.
- Modulo p: raising to power 2^{w+1} is equivalent to raising to power e_p where:

$$e_p = 2^{w+1} \pmod{p-1}$$

- Results modulo p and q are recombined with the *Chinese Remainder Theorem*.
- Randomized masking can be applied to thwart timing attacks.

Total cost is similar to RSA private key operation.

Features: Fast Path

Usage scenario for fast path:

- Hashed passwords are stored in a database.
- Database is shared between several front-ends.
- *Some* front-end servers can be entrusted with knowledge of p and q (extra shielding, HSM, no PHP...).

Important Consequence

p and q are a *private key*: keep them safe !
If the "fast path" is not needed, p and q can be discarded after generation of n.

Features: Escrow

If p and q are known, the password can actually be recovered:

- Again, compute modulo p and modulo q.
- Modulo p: revert $w+1$ squarings with exponent e'_p:

$$e'_p = \left(\frac{p+1}{4}\right)^{w+1} \pmod{p-1}$$

- Two candidates are obtained modulo p, and two modulo q, for a total of four candidates modulo n.
- Recompute padding to identify the right candidate.

Total cost is similar to RSA private key operation.

Features: Fast Path and Escrow

Password escrow may be useful in the following situations:

- Allowing for recovery of forgotten passwords (useful for password-based encryption).
- Support for authentication protocols which need the cleartext password (e.g. APOP).
- Regular detection of weak passwords by the sysadmin.

All these features can be achieved generically by hashing the password *and also* encrypting it asymmetrically with an escrow public key. Makwa allows merging the hashed password and escrowed password into a single value.

Features: Offline Work Factor Increase

Work factor w should be regularly increased to keep track of technological advances: when a new server is deployed, it computes faster, and thus calls for a higher w.

Generic method: wait for the user to come by again; when the password is known, rehash it on the fly with the new work factor.

With Makwa: take the stored value (work factor w) and square it $w' - w$ times to compute the new value for work factor w'.

Features: Offline Work Factor Increase

Advantages of Makwa-powered work factor increase:

- No need to deploy the verify-and-rehash logic in the front-end servers.
- Upgrade to the new work factor is completed within a single administrative procedure.
- Upgrade can be done at a convenient time (e.g. at night).
- If p and q are known, the fast path is applicable (useful to upgrade 1 million passwords in one go, and without pushing the p and q values to the front-end servers).
- If p and q are known, a work factor *decrease* can be done.

Feature Matrix

Availability of features depends on options:

Variant	Unlimited input	Short output	Offline WF increase	Escrow
core Makwa	no	no	yes	yes
pre-hashing	yes	no	yes	no
post-hashing	no	yes	no	no
pre- and post-	yes	yes	no	no

Delegation is always possible.

Delegation: Parameter Generation

For $i = 1$ to 300:

- Generate a random r_i modulo n
- Compute: $\alpha_i = r_i^2 \pmod{n}$
- Compute: $\beta_i = \left(\alpha_i^{2^w}\right)^{-1} \pmod{n}$

The (α_i, β_i) pairs are the *delegation parameters*.

- need not be secret
- are computed only once, in advance
- are specific to a given value of w
- can be generated with n alone (the “fast path” helps but is not necessary)

Delegation

To delegate computation of $y = x^{2^{w+1}} \pmod{n}$ from system A to system B:

- A generates 300 random bits (b_i).
- A computes:

$$z = (x^2) \prod_{b_i=1} \alpha_i \pmod{n}$$

- A sends z (and n, w) to B.
- B computes and sends back z' to A:

$$z' = z^{2^w} \pmod{n}$$

- A computes:

$$y = z' \prod_{b_i=1} \beta_i \pmod{n}$$

Delegation

Delegation Properties

- The delegation system cannot learn x or y.
- The delegation system cannot even recognize whether two delegation requests are for the same value x or not.
- Security relies on intractability of the *multiplicative knapsack problem*.

Costs:

- CPU cost on the source system: about 300 multiplications (half of cost of RSA); it can be optimized further with tables.
- CPU cost on the delegation system: w squarings.
- Network costs: only one request and one answer; messages have the size of n.

Parallel Hashing

The Need For Parallelism

Password hashing should be amenable to parallelism:

- Most computing hardware (from smartphones to servers) is multi-core.
  - Several cores can be used to process several distinct requests simultaneously.
  - In some usage contexts, requests don't occur simultaneously (e.g. hard disk encryption) and using several cores for a single password would offer a significant gain.
- When delegating, the delegation systems may be slower than the server.
  - In particular in a Web context, where client code relies on Javascript.

Parallel Password Hashing (Simple Case)

Let f be a password hashing function, with inputs:

- Password: π
- Salt: σ
- Work factor: w

Let h be a hash function (a "random oracle").

Parallel password hashing function pf_m (spreads computation over m computing units):

$$pf_m(\pi, \sigma, w) = \bigoplus_{i=0}^{m-1} h\left(f\left(\pi, \sigma + i, \frac{w}{m}\right)\right)$$

Parallel Password Hashing (Simple Case)

- The space of salt values must be large enough to accommodate the increased usage without collisions (m salt values per hashing).
- The role of h is subtle but important.
- The h function may already be included in the password hashing function itself (with Makwa, the post-hashing step can play the role of h).
- If the function f has several costs (e.g. CPU *and* RAM) then the consequences of parallelism can be complex.

Parallel Password Hashing (General Case)

Scenario: a server must authenticate clients; the server stores password hashes. Computations are delegated to already connected clients. The clients are *slow* (Javascript...) and *unreliable*.

- At least m clients must collaborate to reach the required security level.
- The server must send delegation requests to more than m clients to cope with failing clients.
- The connecting user is waiting and is *not patient*.

Parallel Password Hashing (General Case)

The h function outputs elements of a finite field $\mathbb{K}$:

- When using distinct passwords and random salts, the values $\mathrm{h}(\mathrm{f}(\pi, \sigma, w))$ must be indistinguishable from a random *uniform* selection of values in $\mathbb{K}$.
- We assume that there exists a bijective mapping from integers (in the 0 to $\#\mathrm{K} - 1$ range) to elements of $\mathbb{K}$.

Practical Case

Method also works for when the output of h is a *sequence* of elements of $\mathbb{K}$. So we can use *bytes* and do bytewise computations in $\mathrm{GF}(2^8)$.

Parallel Password Hashing (General Case)

Interpolated Polynomial

Let (ϕ_i) $(1 \leqslant i \leqslant t)$ be a sequence of t *distinct* elements of $\mathbf{K}$.
Let (v_i) $(1 \leqslant i \leqslant t)$ be a sequence of t elements of $\mathbf{K}$ (not necessarily distinct from each other).
Then there exists a *unique* polynomial $\Lambda \in \mathbf{K}[X]$ of degree at most $t-1$ such that:

$$\Lambda(\phi_i) = v_i$$

for all i from 1 to t.

- The coefficients of $\Lambda = \sum_{j=0}^{t-1} \lambda_j X^j$ can easily be recomputed with Lagrange polynomials (see Shamir's Secret Sharing).

Parallel Password Hashing (General Case)

Parameters:

- m: minimum number of delegated work units that must be necessary to recompute the password hash.
- t: number of delegation requests that will be issued ($t \geqslant m$).
- π: the input password.
- σ: the salt.
- w: the total work factor.

Parallel Password Hashing (General Case)

Password Registration:

- For $i = 1$ to t, compute:

$$h_i = h\left(f\left(\pi, \sigma + i, \frac{w}{m}\right)\right)$$

- Compute the polynomial Λ such that, for all $i = 1$ to t:

$$\Lambda(i) = h_i$$

- Store $\Lambda(0)$ and all $\Lambda(k)$ for $k = t + 1$ to $2t - m$ (total storage: $t - m + 1$ elements of $\mathbf{K}$).

Registration cost: t parallel invocations of f with work factor w/m.

Parallel Password Hashing (General Case)

Password Verification:

- Compute (delegate) for h_i $(1 \leqslant i \leqslant t)$.
- Using m of the answers *and* the stored values $\Lambda(k)$ for $k = t + 1$ to $2t - m$, rebuild the Λ polynomial.
- Verify that the value $\Lambda(0)$ matches that which was stored.
- If less than m answers are obtained, then it is not feasible to know whether the password is correct or not (even probalistically).

Verification cost: t parallel invocations of f with work factor w/m (at least m must succeed).

Parallel Password Hashing (General Case)

Summary:

- At registration time, we derive the password into t sub-hash values.
- The t values define a polynomial of degree at most t.
- We save $t - m + 1$ *other* polynomial outputs.
- At verification time, we recompute at least m sub-hash values.
- Combined with the saved $t - m + 1$ values, the m values are more than enough to rebuild the polynomial: t values define the polynomial, the $t + 1$-th is used to check proper reconstruction.

The process can be done byte by byte; computations in $\mathrm{GF}(2^8)$ are easy and fast.

Performance Measures

Model for Estimations

- Makwa's core is a sequence of modular squarings.
- 80% (at least) of a RSA private key operation consist in modular squarings.

Therefore:

- We can implement Makwa using the same library as optimized RSA implementations (e.g. OpenSSL's "BN" library).
- We can use RSA performance as an estimate for Makwa performance.

Modular Squarings

- Rely on native code optimized library (OpenSSL, GMP...).
- Use Montgomery's multiplication (`BN_mod_mul_montgomery()`).
- "Fast path": better than straightforward squarings when the number of squarings w exceeds 34% of the modulus length (about 700 for a 2048-bit modulus).
- **Java:** use `BigInteger.modPow()` (it is backed up by native code in some JVM, especially Android).

Modular Squarings in Javascript

Javascript's numbers are IEEE 754 floating-point values (53-bit mantissa).

- Store 26 bits per word.
- Scale words down: 26-bit word x ($0 \leqslant x < 2^{26}$) is represented by floating point value $x \cdot 2^{-13}$.
- After multiplication, extract high word from 52-bit result by using the *floor()* function (faster than right-shifting).
- Use the `~~z` expression instead of `Math.floor()`.

Modular Squarings in Javascript

```
for (var i = 0; i < size; i ++) {
    // ...
    for (var j = 1; j < size; j ++) {
        z = u * x[j] + cm * m[j] + y[j] + r;
        zh = ~~z;
        y[j - 1] = z - zh;
        r = zh * IBASE2;
    }
    // ...
}
```

- Operand is `x[]` (words scaled by 2^{-13}).
- Result is accumulated in `y[]` (words scaled by 2^{-26}).
- Modulus is `m[]`.
- `IBASE2` is equal to 2^{-26}.

Software Performance

Measures in squarings per second on an Intel Core i7-2620M (2.70 GHz):

Platform	squarings/s	ratio
C + OpenSSL 1.0.1f	571000	1.0
Java (32-bit)	20400	28.0
Java (64-bit)	94300	6.0
Javascript (Chrome 36.0)	31200	18.3
Javascript (Safari 7.0.5)	20700	27.6
Javascript (Firefox 31.0)	28000	20.4
C + FPU (IEEE 754)	42400	13.5

Makwa and GPU

A 2011 study[1] compares RSA performance between general-purpose CPU (AMD Phenom II 1090T) and GPU (NVIDIA).

CPU and GPU offer similar performance for RSA, both per dollar and per Watt.

- "Per dollar" is about buying the hardware.
- "Per Watt" is about running the hardware.

[1] *On the Performance of GPU Public-Key Cryptography*, S. Neves and F. Araujo, 22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2011, pp. 133–140.

Makwa and FPGA / ASIC

Existing ASIC for RSA are used in *Hardware Security Modules*.

- Very expensive (cost of FIPS 140-2 / EAL certifications).
- Old designs (because of certifications).
- Not competitive with CPU.

Some FPGA include many DSP (e.g. Xilinx XC7VX690T) which can *theoretically* be used for many modular squarings, but the hardware cost is still prohibitive (cost factor at least 3).

Makwa on FPGA / ASIC

Though Makwa is structurally ASIC-friendly, integer multiplications is one of the most optimized tasks in CPU, and existing FPGA and ASIC hardware are not *economically* up to it.

Conclusion

Makwa and Delegation

- Delegation can *potentially* tilt the game in favour of the defender.
- Apart from delegation, Makwa is a “decent” password-hashing function with features (fast path, offline work factor increase...).
- Software implementations can build up on existing big integer and RSA libraries.
- Surprisingly, existing GPU and FPGA don’t seem too good for fast Makwa implementations.

Work Still Needed

- Formal security proofs (knapsack problem, equivalent to factorization...).
- FPGA and ASIC implementations.
- Statistics on browser performance in the field.
- Full-scale experiments for delegation + parallelism.

Volunteers are welcome.