Passwords^14

Proof of work as an authentication factor

Philippe Paquet - Jason Nehrboss

Housekeeping

We have 5 minutes reserved at the end for Q&A

Presentation and sample code are available on github:

https://github.com/jaegerindustries/passwords14

Running sample code is online:

http://passwords14.jaegerindustries.com/

Introduction

The most popular method of authentication is passwords.

While passwords are flawed, they are here to stay, at least for the near / medium future.

Passwords are attacked in two ways: offline (password cracking) and online (password guessing).

Offline attacks

In an offline attack, the adversary obtain the cryptographic hashes of all the password.

The adversary then ‘crack’ the cryptographic hashes using either dictionaries or simple brute force.

Counter-measures

Best practice is to use a key stretching algorithm such as PBKDF2, bcrypt or scrypt.

Key stretching is a computationally expensive transformation, on purpose, to defend against offline attacks.

However, key stretching offers adversaries an opportunity to perform online denial of service attacks.

Online attacks

Opportunistic adversaries scan the whole internet for weak passwords or default credentials. The effort required for that kind of attack is very low.

For advanced adversaries, tools, dictionaries created from previous data breaches and intelligence work on social media make the job easy.

Counter-measures

Account lockout can lead to denial of service attacks.

IP lockout can be problematic and can be bypassed using open proxies.

Delayed responses can be masked by attacking multiple targets at once.

Counter-measures

Better counter-measures are not widely adopted.

Tokens and side channel verifications are effective but usually expensive or difficult to implement.

CAPTCHAs are not user friendly and there are services to go around them anyway.

Computational imbalance

Attacking is computationally cheap.

Defending is computationally expensive.

Even more so when you use passwords stretching.

Proof of work

We propose adding a proof of work as an additional factor of authentication.

The client has to solve a computationally expensive problem and send the solution, the proof, to the server.

It is moderately hard on the client side while it is easy to check on the server side.

Proof of work

The technique is similar to hashcash “digital stamp” by Adam Back which is used to fight email spam.

That technique is also used to combat comment spam.

We also propose improvements to make the technique more effective.

Benefits

This is transparent to the end user. No behavior change.

This is easy to implement.

It restore the balance: login requests are computationally more expensive to the client than to the server.

Brute force attacks are still possible, at a cost.

Generic implementation

The server sends the client a seed and details of the work to be performed.

The client does the work and submits the proof along with the credentials.

The server verifies the proof before considering the credentials.

Our POC Implementation

We use partial hash collision for the proof of work.

“Find a number such as the hash of the provided nonce plus that number will start by x bits that are zero”.

hash (nonce + number) = 00000…

Our POC Implementation

Server side written in PHP.

Client side written in JavaScript.

Hash algorithms: MD5, SHA1, SHA256, SHA512 and RIPMD160.

We use OpenSSL to generate nonces.

Server side

Current response to a failed login is usually passive.
We propose an active approach.

Scale difficulty according:

- IP address.
- Number of failed attempts.
- Profile (combination of metrics).

Or even better, scale difficulty according to load.

Active approach

Escalate the amount of work, in response to perceived abuse. Linear or exponential increase?

Can we shun bad clients by giving them a 15 years problem?

Consider additional risk factors.

Can we increase trust or confidence with a third factor of authentication (token, etc...)?

Server performance

Nonce generation may be a bottleneck.

Proof of work verification is fast.

Check the proof of work before considering other authentication factors.

Scaling the proof of work difficulty according to load alleviate flash mob problems.

Client side

This technique slows down login speed.

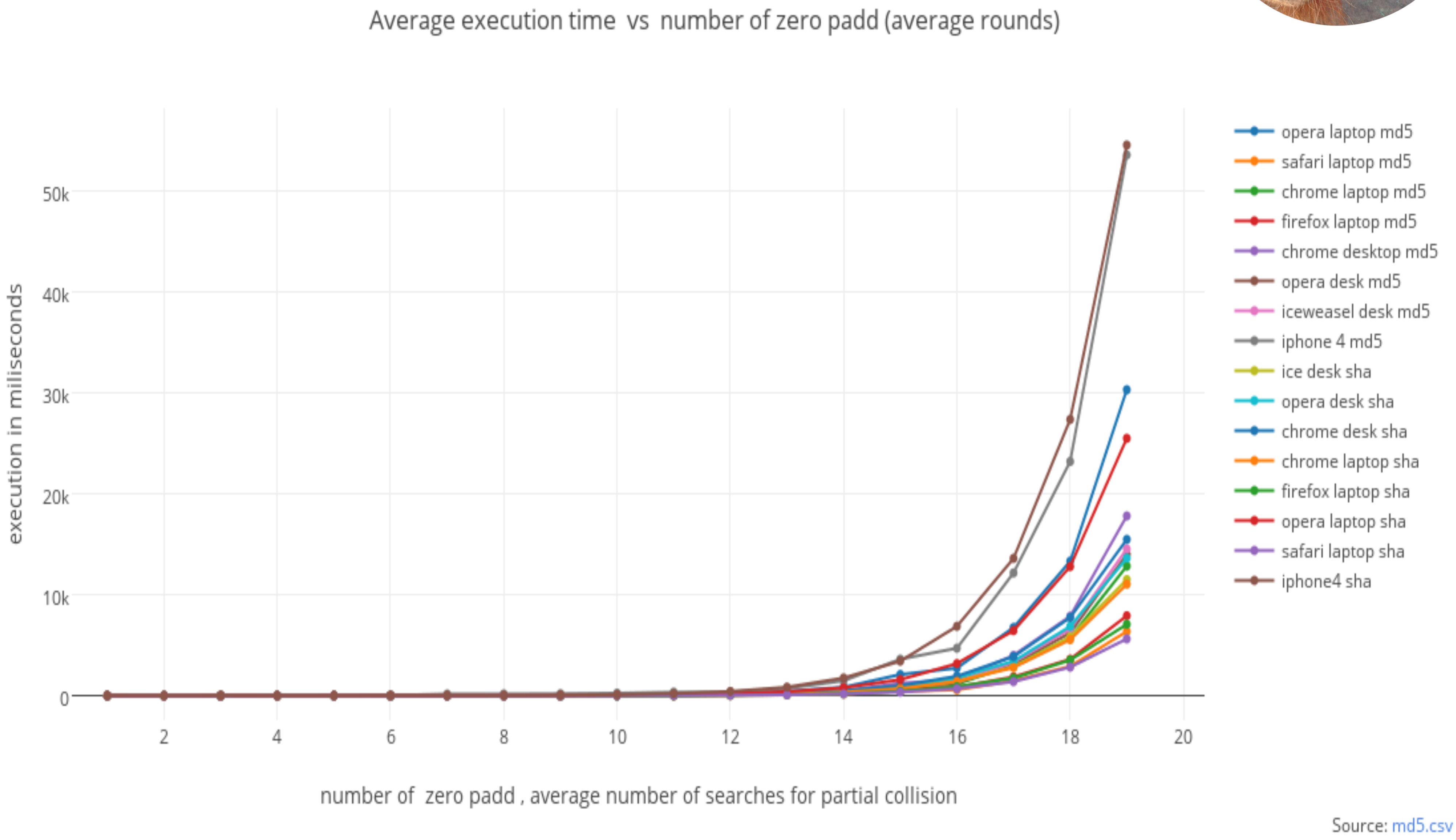
Too slow and users notice (and complain).

Too fast and it wont be enough “work”.

Average compute time is ok, outliers are a problems.

With a JavaScript implementation, the JavaScript engine can makes a big difference.

Execution times



Source: md5.csv

MD5 / SHA1

Start with requesting 11 zeros. Average execution is approximately 50 milliseconds.

End with requesting 18 zeros. Average execution on phones begin to exceed 24 seconds.

These are averages, it will be faster and slower.

Attacks

Adversaries are in a difficult position. They can either:

Pre-compute all possible proofs of work.

Run the JavaScript as found (CPU).

Write specific attack code (CPU or GPU).

Attacks

Would specific GPU attack code be a free pass?

# GPU cores >= median iterations (searching). Is a 50% chance of calculating a correct proof in one execution.

Core count of the latest GPU (5632), is greater than median search rounds of 12 bits of partial collisions.

Can cycling algorithms, fix GPU speedups?

Counter-measures

At the end of the day, there is no free lunch. You have to spend the cycles, whether on the CPU or on the GPU.

Cycling hash algorithms.

Metahash: randomly chaining hash algorithm.

Escalation will eventually catch up.

Conclusion

More a deterrent factor than authentication factor.

Computational cost of attack is now exponentially greater than the computational cost of defense.

Questions ?

Contact us

Philippe Paquet

philippe@jaegerindustries.com

Jason Nehrboss

jbossvi@gmail.com