Proof of Work

CHRISTIAN HELLMAN
and FELIX LEOPOLDO RIOS

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2010

Proof of Work

CHRISTIAN HELLMAN
and FELIX LEOPOLDO RIOS

Bachelor's Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2010
Supervisor at CSC was Mikael Goldmann
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/
hellman_christian_OCH_rios_felix_leopoldo_K10039.pdf

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

This essay covers the basics of a technique called proof-of-work which is an attempt to have clients show their dedication to a remote service before gaining access to it as an attempt to prevent attacks. Drawbacks of the concept and issues that might occur are discussed and solutions are evaluated. In the puzzle section we cover aspects of different puzzles that the client could be forced to solve in order to provide a proof of work to the server. The essay discusses which style of proof-of-work that might be suitable for web services and the result of a simple implementation is included.

Referat

Proof Of Work

Denna essä tar upp grunderna i en teknik som heter proof-of-woork. Det är ett försök att få klienter att visa sin dedikation till en tjänst innan den blir tillgänglig i ett försök att motverka attacker. Nackdelar med konceptet och problem som kan uppstå diskuteras och lösningar utvärderas. I sektionen Puzzles undersöker vi två olika pussel som en klient kan bli tvingad att lösa för att kunna visa ett bevis på arbete till servern. Essän tar upp vilken stil av proof-of-work som kan vara lämplig för webtjänster och resultatet av en lätt implementation är inkluderad.

Contents

1 Introduction **1**
1.1 Introduction 1
1.2 Background 1
1.3 Purposes 2

2 Proof of Work **3**
2.1 Puzzles 3
2.2 Investigated Puzzles 4
2.2.1 Reverse Computing a Hash 4
2.2.2 Discrete Logarithm Problem 5

3 Implementation **9**
3.1 Possible Drawbacks and Issues 9
3.2 Web Browser Limitations 9
3.3 Implementation 10
3.4 Test of Calculation Time 12
3.5 Discussion 12

Bibliography **15**

Chapter 1

Introduction

1.1 Introduction

Denial-of-Service[1]-attacks are a common problem when providing services over a network. An attacker intensively makes multiple requests to a service which causes overload and failure at server side. This is often seen among web based services such as forums where spammers are abusing the search function or rapidly submitting other types of forms that requires effort from the server to complete the request.

A method for preventing DoS-attacks is to have the client show its dedication towards the service before gaining access to it. As a proof of dedication, the client is requested to compute an answer to an algorithmic puzzle. The puzzle should be hard to solve but the solution should be easy to verify. When the computation is done, the answer is sent to the server which verifies the solution. The puzzle puts heavy load on the client if several requests are made in a short time span; this prevents the client from abusing the service. This method of authentication for using a service is called a proof-of-work-protocol or simply proof-of-work. Since this only causes load on a single spamming client, proof-of-work cannot prevent distributed DoS-attacks since several clients then share the workload. Distributed attacks are discussed in a paper by Laurie and Clayton[8] and will not be covered in this essay.

1.2 Background

Proof-of-work was first explored in the 1990's when problems such as spam and DoS-attacks became more common. Ari Juels and John Brainard discusses a way to implement client puzzles in order to resist TCP SYN attacks [5] which is an attack where an attacker opens a huge number of TCP socket connections and leave them opened.

Ed Kaiser and Wu-chang Feng describes a way to use proof-of-work as an alternative to CAPTCHAs which is a very common method for preventing spam on

[1] The abbreviation DoS will be used from here.

the Internet. Their implementation mod_kaPoW is a module to the Apache web server [6, 2].

The on-demand music service Spotify uses proof-of-work for decreasing the load on their authentication system and to prevent brute force password attacks [3]. Of course, there could be more unknown implementations of proof-of-work in closed source projects.

1.3 Purposes

Proof-of-work should only be used whenever a server needs to process a request and the processing demands more server resources than sending out a puzzle. If responding to a request is as demanding as sending out a puzzle, proof-of-work is rendered useless since the server side processing time for multiple requests are equal in both cases. This will not ease up the server workload during an attack.

Proof-of-work should be used when clients can accept a small delay for requested data. Examples of this are login forms and search forms. The easiest way to calibrate the difficulty of the puzzle is when you have information about the client platforms and specifications; a homogeneous client base is excellent.

It is clear that proof-of-work should not be used when fast request processing is required.

In this essay we shall investigate how the proof-of-work works to prevent overload of a web server.

Chapter 2

Proof of Work

2.1 Puzzles

A puzzle should not require any user interaction to be solved, and for every puzzle there should be a good lower time complexity bound for the best known algorithm that solves the puzzle. As mentioned earlier the puzzle that the client will have to solve should be quite hard to compute for the client and easy to verify by the server. A common property of the puzzles is that they have to be non pre-computable. If a puzzle is pre-computable, an attacker could spend some time calculating solutions to puzzles and store them in a table before an attack. When attacking, the solution for the puzzle is looked up in the table and provided without any significant processing time, rendering proof-of-work pointless. It will be necessary for the server to identify the clients so that one client cannot solve a puzzle for another client.

The difficulty of the puzzle is an important decision – will devices with different performances be able to compute the same puzzle within the same time limit using the same difficulty? Furthermore, since the clock frequency of computers is stagnating while the number of cores increases, it is a good idea to have a puzzle that is not scalable i.e. not able to be computed faster by computing on several cores simultaneously.

We have examined two different puzzles and we have discussed their properties in 2.1.

Necessary Properties for a Puzzle

A summary of necessary properties for every puzzle is given below.

- The solution should be hard to compute for the client, and at the same time easy to verify for the server.
- A client shall only be able to compute its own puzzle.
- Pre-calculation of a puzzle should not be possible.

- There should be a good lower time complexity bound for the best known algorithm that solves the puzzle.
- The difficulty of the problem should be configurable.

2.2 Investigated Puzzles

2.2.1 Reverse Computing a Hash

The puzzle presented below is based on reverse computation of a hashed string from *DOS-resistant Authentication with Client Puzzles*[1]. Normally, it is almost impossible to reverse a hash to its original string but it is easier to find a string that hashes to a similar hash. This is the key idea for this puzzle.

$$h(X, s) = \underbrace{000\ldots000}_{m\ zeros} + Y \tag{2.1}$$

where h is a hash function, $m \in \mathbb{Z}$ and s is a string. As mentioned above, hash values from strings are very hard to reverse. A known puzzle for use in proof-of-work is to provide the client with a string s used as a seed and a level of difficulty m. The difficulty m specifies how many leading zeros the hash h should contain. The client then attempts to find a string X which has a hash value with the specified number of of leading zeroes and which also contains the seed sent from the server i.e. it computes $h(X, s)$ where X and s are concatenated. The string s used as seed is re-generated on every new request and is needed to prevent the client from using pre-calculated tables for the solution.

Protocol

The difficulty m is pre-assigned. The protocol is described below.

Client Sends a request for a service to the server.

Server Generates a random s and sends difficulty m and seed s.

Client Tries to concatenate s with a guessed string X and computes $h(X, s)$ where X and s is concatenated. This process is repeated until the calculated hash starts with m leading zeros. When X is found the client provides the server with it.

Server Verifies that $h(X, s)$ has m leading zeros and provides the client with the requested service.

Figure 2.1 may help to visualize the situation.

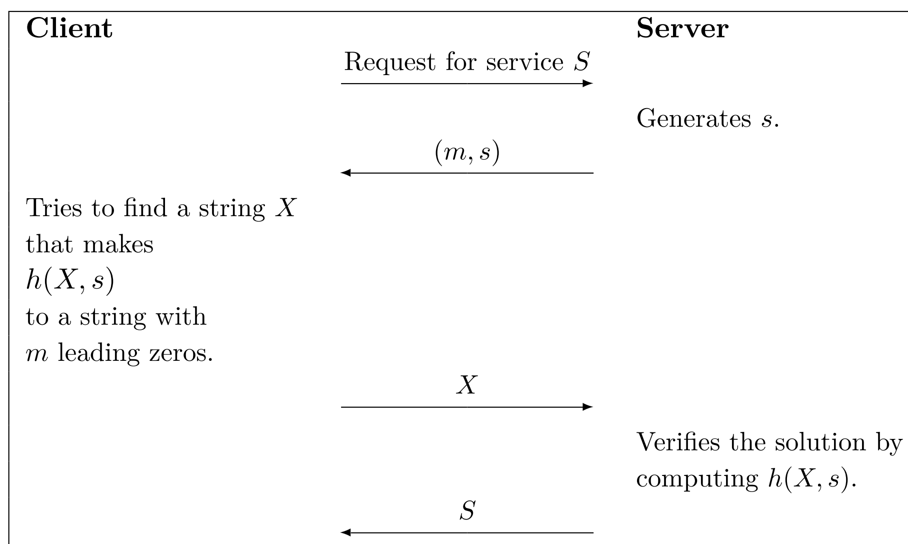

Figure 2.1. Flow chart for the *Reverse computing a hash* puzzle.

Properties

The puzzle solutions are not pre-computable since the server generates a random s at each request. The puzzle is also easy to implement since most programming languages provides some hash function that can be used. To calibrate the difficulty of the puzzle one increases the value of m. This leads to a puzzle that is exponentially harder to compute which is inefficient when trying to find an appropriate difficulty. This can easily be solved by limiting the leading hash numbers in the solution to a range of allowed numbers instead of just zeros.

2.2.2 Discrete Logarithm Problem

The puzzle presented below is a special case of the Discrete Logarithm Problem [9].

$$y = g^x \mod p \tag{2.2}$$

where p is a prime number, $x \in \mathbb{Z}_p$, $y \in \mathbb{Z}_p \setminus \{0\}$ and $g \in \mathbb{Z}_p \setminus \{0, 1\}$. This puzzle forces the client to calculate x_{client} [1] that makes (2.2) true where y, g and p is given by the server.

The server chooses a sufficiently large p and generates at random x_{server} [2] and g and calculates $y = g^{x_{server}} \mod p$ and stores x_{server}. The server then sends y, g

[1] x_{client} serves as x in (2.2) at client side.
[2] x_{server} serves as x in (2.2) at server side.

and p to the client. The client searches for x_{client} such that $y = g^{x_{client}} \mod p$ and sends x_{client} back to the server. The server then simply makes the comparison $x_{server} = x_{client}$ to verify that the correct x has been sent.

The space needed to store all values of all combination of g and x increases in $O(p^2)$. Hence a big value on p will protect a against such pre-computed tables. On the other hand, when increasing the value of p the difficulty of solving the puzzle also increases. One solution to solve that problem is to at the server side when generating x_{server} do not generate over the whole $\mathbb{Z}_p$ but only over some range specified by a range start, r_{start} and a range size, r_{size}. The difficulty of the puzzle will then be growing due to r_{size}. Moreover to prevent a malicious client from pre-calculate the combinations from r_{start} to $(r_{start} + r_{size})$, r_{start} should also be chosen randomly.

Protocol

The prime number p and the range size r_{size} are pre-assigned. g and r_{start} are randomly generated and should be re-generated every nth request to prevent the client from using pre-calculated tables, where n is an integer specified at server side. To generate new values of g and x_{start} at every request would also cause too much load on the server. The protocol is described below.

Client Requests for a service S.

Server Makes the following steps:

1. Generates a random $r_{start} \in [1 \ldots ((p-1) - r_{size})]$ every nth request.
2. Generates a random $x_{server} \in [r_{start} \ldots (r_{start} + r_{size})]$.
3. Generates a random g where every nth request.
4. Computes $y = g^{x_{server}} \mod p$ and stores x_{server}.
5. Sends y, g, p, r_{start} and r_{size} to the client.

Client Tries to find an $x_{client} \in [r_{start} \ldots (r_{start} + r_{size})]$ that fulfils the equation $y = g^{x_{client}}$ and sends x_{client} back to the server.

Server If the comparison $x_{client} = x_{server}$ is true, server sends the requested service to the client.

Figure 2.2 may help to visualize the situation.

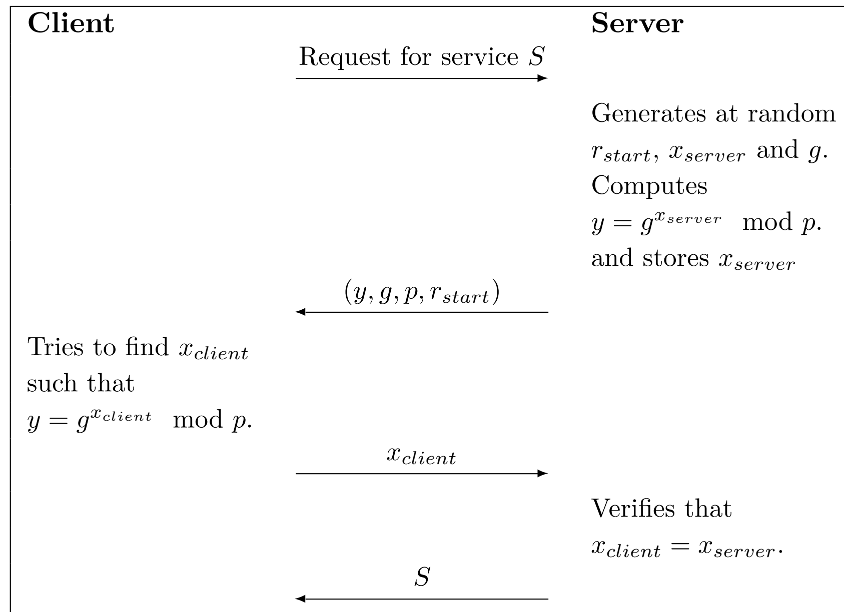

Figure 2.2. Flow chart for the *Discrete logarithm problem* puzzle.

Properties

As mentioned above a table with pre-computed values of x for all combination of y and g will increase in $O(p^2)$. Remember, this is for only one specific prime. For every newly decided prime a new table would have to be made. As an example, by letting $p = 99999989$ and assuming that every integer is represented by 4 bytes, a table with pre-computed solutions would be of approximately $4 \cdot 9.5 \cdot 10^{15}$ bytes.

Chapter 3

Implementation

3.1 Possible Drawbacks and Issues

When implementing proof-of-work as a solution against search abuse in web services, the first question raised is probably whether or not the calculation time ruins the experience for the user. The time it takes to solve the puzzle could be an issue for the client since the wait might be strenuous. This can be solved entirely or partially in numerous ways. A possible solution could be to have the puzzle solution calculated as the search query is entered.

Assume that the average search query consists of 2.4 search terms, as suggested in [12], and that a person composes the search query at a rate of 19 words per minute [7]; this makes the time for entering the query 7.58 seconds. Now assume that the person searching does not enter any advanced filter rules [12] and that the query is executed instantly; we have 7.58 seconds to asynchronously calculate the solution to the puzzle without affecting the client's experience. The fast-typing users who already knows their query in beforehand would have to wait a few seconds for their search to be executed but it might still be faster than the more common CAPTCHA-solution. To ease up the process even more, proof-of-work could be turned off for logged in users since the registration process should handle the event of a user being a non-human spammer.

Another issue is low performance clients where the work takes too much time. Different platforms have different speeds that may affect the solving time.

3.2 Web Browser Limitations

When implementing proof-of-work to protect a web search function, the client side programming language will probably be JavaScript. JavaScript is not multi threaded, henceforth a puzzle cannot be solved while the user fills in a form without the web browser acting unresponsive. There are a few solutions to this problem. We could do the puzzle calculations after the user submits the query but, as mentioned earlier, this might ruin the experience for the user. Another solution is to

use Web Workers [13], an API for running scripts in the background. Web Workers are currently only implemented in a few browsers.

Different browsers may have different implementations of JavaScript of various speed. We will investigate this in Section 3.4.

The web browsers Mozilla Firefox and Google Chrome have time limits that limits the computation time for a JavaScript. When the limit is exceeded the browser prompts the user with a question whether to continue running the script or not. It is obvious that this would ruin the experience for a user.

3.3 Implementation

Ajax is a programming style for the web using JavaScript to asynchronously send a request to a server. Some server side language such as PHP [10] is often used to perform actions on the server and XML is used to pass data between the client and the server. Ajax could for example be a used in a search field that makes a database query each time a new character is typed. Here it might be a good idea to let the client solve a puzzle before getting access to fulfill the request.

Our implementation of a simple proof-of-work system consists of a small web page with an input form that represents a search box or a log in form. Upon load, the page queries the server silently using Ajax for a puzzle. The puzzle is then solved in JavaScript at client side. Since JavaScript is synchronously executed and single threaded one might in the future want to use Web Workers to make the GUI available while performing the calculations. When the form is submitted the server verifies that the puzzle is correctly solved before providing the client with the requested information. PHP was chosen for the server side but the technique should be similar in other languages. The reverse hash calculation puzzle using md5 as hashing function was used in this implementation. The JavaScript API jQuery [4] was used because of its simple syntax. To ensure that the client does not change identity and solves another clients puzzle, PHP sessions was used. Sessions works both on client and server side and has a unique identifier [11]. The default max lifetime for a session is 1440 seconds [10]. An attacker could use this and cause a DoS-attack by requesting a service protected by proof-of-work several times and ignore the puzzles. This could lead to an unacceptable number of open sessions. To prevent this one should add a time limit within which each puzzle should be solved. After the limit has exceeded, the session will be destroyed. The length of this limit should be configured according to the difficulty of the puzzle.

The implementation is available for testing and downloading at `http://wproj.nada.kth.se/~chellman/pow/`.

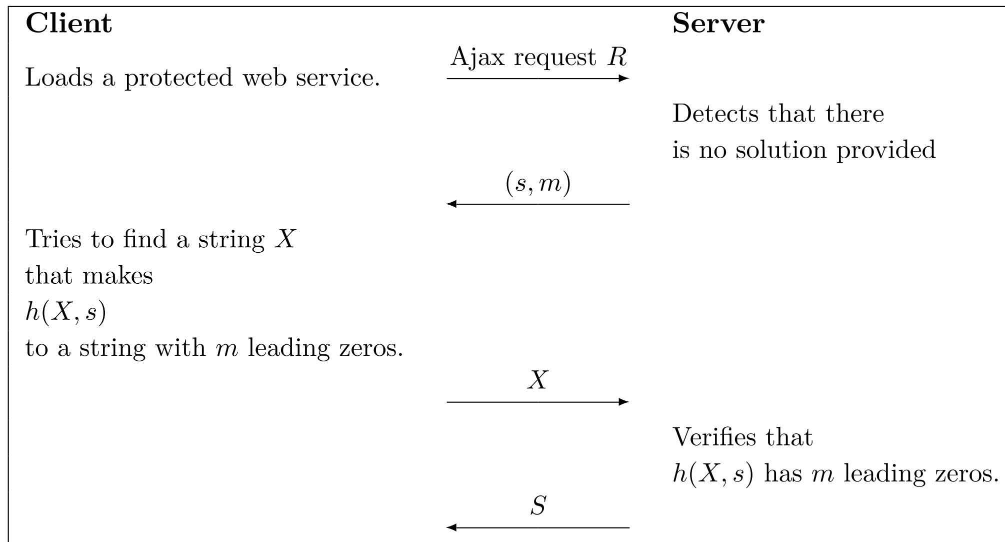

Figure 3.1. Flow chart for the protocol of the implementation.

Protocol

The protocol for the implementation is described below.

Client Loads a JavaScript enabled page which makes an Ajax request for information from a PHP script.

Server The PHP script detects that no solution has been specified in the request and provides the client with a seed and a difficulty.

Client The JavaScript receives the variables in the Ajax call back function and starts calculating a solution by generating strings for concatenation and hashes the results. When a correct amount of leading zeros according to the difficulty shows up in the start of the hash, another Ajax call is made to the PHP script; this time with a solution as an argument.

Server The PHP script detects that a solution is provided and concatenated the solution with the seed and hashes it to make sure that the first part are zeros. The seed and difficulty are remembered through a PHP session, unique for this client. The PHP script returns the requested information to the JavaScript.

Figure 3.1 may help to visualize the situation.

m	Time (milliseconds)		
	Firefox 3.5.2	Firefox 3.6.3	Google chrome 4.0.207.0
1	98.5	8.4	3.8
2	2575.6	110.5	55.5

Figure 3.2. The results are mean times for solving 100 puzzles. m is the difficulty of the puzzle.

3.4 Test of Calculation Time

The implementation was tested by sending 100 requests in sequence from two different web browsers. In this test, Google Chrome and Mozilla Firefox was used. This was to examine if two web browsers using different JavaScript engines would have a noticeable difference in the puzzle solution calculation time. Different difficulties was tested to see if the two browsers would both fall into an acceptable time range at the same difficulty.

The computer running on the client side was a MacBook Pro with an Intel Core 2 Duo 2.53 GHz CPU. Firefox was tested using both version 3.5.2 and version 3.6.3. The version of Chrome was 4.0.207.0. Results of the test are given in Figure 3.2.

3.5 Discussion

Proof-of-work is simple to implement. Choosing a good proof-of-work puzzle is trivial compared to the issues facing different client platforms as we conclude from Figure 3.2. If the difficulty of the puzzle differs from client to client depending on what platform they have then prevention of spoofing is hard.

The JavaScript implementations efficiency differ much between the browsers. Chrome solves the proof-of-work implementation much faster than Firefox. This makes it hard do have the puzzles solved equally fast. If the puzzle difficulty is tweaked for each specific browser, an attacker would simply identify itself as the slowest browser and get the easiest puzzle.

A possible workaround for the particular case with Firefox and Chrome different speeds is to have Firefox using Web Workers. This would make the JavaScript execute in the background and that might compensate for the slow execution speed. Although this helps for now, Chrome and other browsers can implement Web Workers in the future and gain the same advantage. There is also a future possibility that Firefox improves its JavaScript engine or that another puzzle might be more suitable for its current engine.

Possible future implementation improvements

When Web Workers are more commonly available, proof-of-work should use it to not lock up the user interface while doing the calculations. If a script executes for a

long time without Web Workers, browsers sends a warning and tells the user about an unresponsive script.

A time limit for the computation of a puzzle is necessary to prevent attackers from open several unused sessions, as mentioned in Section 3.3.

Bibliography

[1] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. *DOS-resistant Authentication with Client Puzzles*. Springer Berlin / Heidelberg, 2001.

[2] Captcha. `http://captcha.net`, May 2010.

[3] Despotify. `http://despotify.com`, May 2010.

[4] Jquery. `http://jquery.com`, May 2010.

[5] Ari Juels and John Brainard. *Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks*, pages 151–165. Proceedings of NDSS '99 (Networks and Distributed Security Systems), San Diego, CA, February 1999.

[6] Ed Kaiser and Wu cheng Feng. mod_kaPoW Protecting the Web with Transparent Proof-of-Work. 2007.

[7] Clare-Marie Karat, Christine Halverson, Daniel Horn, and John Karat. Patterns of entry and correction in large vocabulary continuous speech recognition systems. *ACM*, 1999.

[8] Ben Laurie and Richard Clayton. 'Proof-of-work' Proves Not To Work. 2004.

[9] Mathworld wolfram. `http://mathworld.wolfram.com/DiscreteLogarithm.html`, May 2010.

[10] Php. `http://php.net`, May 2010.

[11] Php sessions. `http://php.net/manual/en/intro.session.php`, May 2010.

[12] Amanda Spink, Dietmar Wolfram, Major B. J. Jansen, and Tefko Saracevic. Searching the Web The Public and their Queries. *Journal of the american society for information science and technology*, 2001.

[13] Web workers. `http://whatwg.org/specs/web-workers/current-work`, May 2010.

www.kth.se