KTH Computer Science
and Communication

A Proof-of-Work Approach Using the IP Header to Protect Against DoS Attacks

JONATAN SVENSSON, JOHAN ZEECK
jonats@kth.se, jzeeck@kth.se

Bachelor's Thesis at NADA
Supervisor: Mikael Goldmann
Examiner: Mårten Björkman

Abstract

Denial of Service attacks are a serious threat to server robustness and stability. This thesis seeks to examine the possibility of implementing a proof-of-work protocol which effectively prevents this type of attacks. Its main focus is to investigate a general concept that includes sending puzzles over the Internet layer, creating a cost as a puzzle per request which is easily adjusted depending on current server load. The cost validates that the request is in fact legitimate by forcing the client to process a solution to the given puzzle before granting resources on the server. An implementation of the protocol is explained and the results of the testing confirm that proof-of-work could be an effective method to counter standard DoS attacks.

Referat

Denial of Service-attacker är allvarliga hot mot servrars robusthet och stabilitet. Den här kandidatuppsatsen ämnar att undersöka möjligheten att implementera ett proof-of-work-protokoll som effektivt motverkar den här typen av attacker. Fokus ligger på att utreda ett generellt koncept som skickar pussel över internetlagret. Detta görs genom att införa en kostnad per efterfrågan som enkelt regleras beroende på serverns belastning. Kostnaden är ett pussel som verifierar att en efterfrågan är legitim och inte en attack genom att tvinga klienten att lösa pusslet innan den kan fortsätta. En implementation av protokollet beskrivs och resultaten av körningarna bekräftar att proof-of-work kan vara en bra metod för att hindra vanliga DoS-attacker.

Contents

Statement of Collaboration

The thesis was formed working together using collaborative software. Texts were revised and commented on by both parts and discussed in meetings and virtually. Due to the cooperative effort it is difficult to divide the sections without giving merit to both parties. In spite of this, the following paragraph describes which sections the members have focused on the most during the course.

Jonatan has written the majority of: *Problem Statement*, *Method*, *Research*, *Implementation/Protocol*, *Performance* and *Challenges*. Johan has written most of the following sections: *Background*, *Purpose* and *Implementation/Puzzle*. Remaining sections were written in complete collaboration.

The implementation code in Java was to a large extent written by Johan with some contributions provided by Jonatan.

Chapter 1

Introduction

One of the most common and critical security threats to web servers are DoS (Denial of Service) attacks. DoS attacks are targeted efforts to deny a service from functioning properly on a computer system. In the case where multiple clients focus on overwhelming one system the attack is referred to as DDoS (Distributed DoS attack).[1] These types of attacks exploit bugs or features in the operating system or vulnerabilities in the TCP/IP implementation. Unlike many other targeted attacks, a DoS attack is not aimed towards stealing or compromising data, but to keep authorized users from accessing resources on a computer system.

The typical attack is when clients send tremendous amounts of requests to a web server, in hope for a host malfunction, due to the limit of requests that can be handled by the system hardware. Its purpose is often to express ideological criticism, political discontent or personal harassment.[2] The goal of the authors of this thesis is to create and implement a security protocol which may effectively deny some of these requests with malicious intent.

1.1 Background

There is a lot of research in the field of DoS attacks. Different countermeasures and methodologies have been discussed extensively by academia. The general idea is to force clients requesting a resource to prove that the request is in fact legitimate. One popular tool that is common for this type of verification is reCAPTCHA (CAPTCHA stands for Completely Automated Public Turing test to tell Computers and Humans Apart). It generates distorted pictures of words that the user has to interpret and send as a part of a request to the server. The server only has to compare two strings to verify a valid request.[3]

There are several other services that work almost in the same way. Some examples are Gimpy, Bongo, Pix and Eco. Not all use words like reCAPTCHA. Bongo, for example, uses geometric shapes and lets the user solve a visual pattern recognition problem.[4]

The downside of using services like reCAPTCHA is that it is implemented in the application layer. The Internet Protocol Suite consists of several layers. Depending on the model used the layers are labeled differently but we chose to use application layer and Internet layer according to RFC122.[5] If an attacker decides to mount an attack on a lower layer than the application, the protection is futile. Another criticism to CAPTCHA systems is that they are not very good in terms of usability. Users with impaired sight that mainly rely on the content of the web page to be read to them cannot complete the puzzle.[6] The reCAPTCHA implementation also requires the user to do an active response. An optimal solution should work without the user noticing any part of the protection process.

Another type of defense is to apply a cost for every request sent to the server that clients must pay before accessing resources on the server. The cost is represented as a puzzle the client must solve before the host can process the request. Cryptographic puzzles are a widely-known concept in proof-of-work. The idea is that the puzzle being used by the protocol is the proof that some work has been done to solve the puzzle before substantial resources are allocated by the server. Ideally the puzzle should be hard to solve and comparatively easy to verify its solution, it should be easy to regulate its hardness, and be small in size. Puzzle fairness is the property that all clients should have the same delay when processing a puzzle. Another important property is non-parallelizability, meaning the problem should not easily be solved by multiple machines.[7] This last property is not something this thesis will focus on due to lack of knowledge.

1.2 Purpose

This thesis is part of the degree project for computer science bachelor students at KTH. The goal of this study is to investigate an effective way of denying DoS attacks without affecting normal users. In order to fortify the defense mechanisms, a different approach than the one offered by applications such as CAPTCHA is needed. The best solution would be a fully automated system that carries out a protocol in which the user needs to verify its legitimacy, implemented globally in a network so that an attacker cannot avoid it. Describing an optimal scheme is difficult because of the great amount and diversity of different attacks available, and moreover the various hardware configurations in a network.

This thesis seeks to remedy some of these problems by analysing previous research and determining which is most favorable. It describes the possibilities of such a protocol and presents an implementation of it - evaluating its strengths and weaknesses and overall performance.

Chapter 2

Problem Statement

The main problem is to identify a distinguish request from a DoS attack. Only accepting requests from trusted sources has its downsides, such as false positives or the non-inclusive behaviour towards unknown but legitimate clients.

It is imperative that the server can process the requests regardless of source. If the server can force clients to compute a solution of a puzzle for each request, some DOS attacks could be prevented and the threat mitigated. Placing a cost in the form of a puzzle on each request makes clients pay with computing power for each request done. As a side effect the average amount of requests sent is decreased and if the solution to the puzzle is invalid or nonexistent the request will be detected as an attack. Furthermore, the puzzle needs to be implemented in a way so that it can verified much quicker than it can be solved, to prevent the server from being overwhelmed with verification requests. The simplicity of verification is also important so that the server cannot be attacked with huge amounts of incorrect solutions.

Another area that needs to be addressed is the distribution of the puzzles to clients. There are several approaches: host server distribution, the use of bastions as an intermediary, or even a distributed network of these.

Additionally, puzzle design needs to be of a very simplistic nature without affecting the hardness. This is because the proof-of-work is meant to be implemented as far down in the Internet protocol suite layers as possible. Using a complex problem such as trees structures to describe puzzles might cause problems during implementation since one can only work with a limited set of bits set by industry standards in communication protocols.[8] This could cause it to become unfeasible to implement in the desired Internet layer.

Lastly, having a puzzle that is easily adjustable with minor changes to the protocol gives the advantage of being able to set thresholds depending on current server load. For instance, if it is known that a server is under attack, the hardness of the puzzle should be elevated so that the client sending requests takes longer to solve these, giving prolonged time for the server to clear enough requests to stabilize.

In conclusion the central subjects to be answered are: determining an effective and balanced puzzle that is easily generated and verified, how to distribute these to clients and how these should be implemented as a proof-of-work protocol.

Chapter 3

Method

In the scope of this thesis there is not enough time to implement a full-scale protocol in the network layer. The focus lies therefore in researching the possibilities of such a protocol and evaluating its features for future research. First the subject is explored, with an overview of puzzles and distribution of these. Naturally these findings are not complete but are a starting point for the upcoming section *Implementation.* Here the protocol proposal is introduced and described in detail, along with the extension possibilities.

In conjunction with the report, an application is programmed in Java according to the outline of conclusions in *Research* and the approach described in *Implementation*, which includes a server and client and a simulation of handling IP headers. Instead of implementing the protocol in the network layer the environment is simulated in the application layer due to time constraints. The application is tested on various systems and its performance efficiency evaluated. With this theoretical research and empirical prototype several conclusions can be drawn that may be used for further research and proper implementation.

Chapter 4

Research

4.1 Puzzles

Typically, the method used for a proof-of-work is to transmit the puzzle in the application layer of the network since most attacks originate from designated software. However, it restricts the protocol to work in a more general environment and must be applied to all applications in need of security. It is therefore desirable to find a solution applicable to all systems. The optimal implementation needs to be located on the Internet layer, also known as the network layer according to the OSI (Open Systems Interconnection) model. In order to be able to send it in the Internet layer, such as with a packet, the size of the problem must be minimal. Currently the IPv4 implementation allows at most 255 bytes in the options field, thus being the upper bound for maximum puzzle size. However since the packet already could have other options defined there might not always be enough space. In the future, concerning the upcoming IPv6 standard, this protocol could be implemented in an Extension Header and is therefore not limited by other options.[9]

According to previous statements, a puzzle in a proof-of-work must be relatively hard to solve and it must be easy to verify its solution. Due to these two requirements one can directly associate the puzzle problem with different problems in the area of cryptography.

For instance, when protecting passwords, hashes and salts are used to protect the integrity of the password. It is widely known that a SHA-1 (Secure Hash Algorithm) hash function is easy to compute but very difficult to cryptanalyze and most computer languages already implement hash functions as part of their default library.[10]

Another subject in computer security using computationally hard problems is encryption of information. It is known that one can crack a RSA key with enough computer power and time, however as the key length becomes longer it becomes exponentially harder to break.[11] As a part of cracking RSA keys one must calculate integer factorisation or try to brute-force a value for d in $m = c^d (mod\, n)$, also known as the discrete logarithm problem.[12]

These three are identified as apt problems to be reviewed: reverse hash, integer factorisation and the discrete logarithm problem, and will be studied as potential puzzles.

4.1.1 Integer Factorization

According to the fundamental theorem of arithmetic, every positive integer can be represented as a product of one or more prime numbers.[9] Prime factorization is unique and gives a representation of an integer as a product of primes.[9] Not all numbers are equally hard to factorize, semiprimes (the product of two prime numbers) being the hardest of these instances. By virtue of their supreme difficulty, semiprimes are a core part of the RSA algorithm, used when calculating the totient function in the algorithm.[9]

There are several advantages with using integer factorization as a puzzle. The concept is extremely simple, the server sends a single, large integer to the client and expects a correct factorization as a result. The puzzle is easily adjustable by the server, being able to send larger integers, or even semiprimes, to the client if under great pressure. However, there are some difficult implications. The representation of the result is a set of integers, in an arbitrary order depending on the algorithm used, and according to our requirements must be sent all at once. Setting and getting the integers may be hard to do effectively for all integer sizes. Because of the IPv4 option size limitation, the maximum amount of integer factors that can fit are 63 if every integer is 32 bits (total size 255 bytes). Since the integers are usually represented with 32 bits, one can assume this is the best practise when retrieving the integers. Verification of these numbers could prove challenging for a server handling many different integer factorization puzzles simultaneously.

Owing to the hardness of the problem there are currently no efficient general-purpose integer factorization algorithms for large integers. Knowing additional information about an integer, such as any of its factors, greatly simplifies the problem.[9] Using a defined type of numbers in a very specified size range and applying a good algorithm for these predefined integers is a feasible solution as a proof-of-work puzzle. Unfortunately this approach slightly constrains the freedom of adjusting the problem according to server load.

4.1.2 Discrete logarithm

Similar to integer factorization in its properties and solving techniques, the discrete logarithm problem is yet another hard to solve mathematical problem. The problem is used in the Diffie-Hellman Key Exchange where two parties compute an algorithm to acquire a shared secret value for encrypted communication.[13] Given the prime number n and an arbitrary number a, relatively prime to g and n, there exists exactly one number μ among the numbers $0, 1, 2 ... \phi(n) - 1$ that satisfies the relation a $\equiv g^{\mu} (mod\ n)$.[9] Relatively prime implies the greatest common divisor$(g, n) = 1$

and $gcd(a, n) = 1$ must be fulfilled for g and n, and is the crux of achieving a hard to solve relation.[5]

Provided the client solves μ the verification for the server is incredibly quick and easy - at its best it is constant O(1) in terms of complexity. It is because it merely uses a simple hash table lookup to identify the puzzle and verify the solution. The easiest and most intuitive naive algorithm is to augment μ to higher powers of g and test for each value of μ if the relation is satisfied - worst-case solving complexity is O(ϕ(n)). Several other more efficient and sophisticated algorithms such as the Baby-Step-Giant-Step method, or Pohlig-Hellman method exist that ought to be considered when implementing discrete logarithm as a proof-of-concept puzzle.[14]

The naive algorithm is easy to implement and should be hard to compute for all standard desktop computers for large input integers. Naturally, devices with more computational power will solve these quicker. Assuming adversaries are not using systems with extremely potent hardware the puzzle is solved with little time difference between computers. These things in mind, CPU power disparity or the existence of slightly more effective algorithms do not impair discrete logarithm as a possible puzzle.

The major feature of this problem is the fact that the communication between server and client is merely three integers one way and one integer back to verify a puzzle. The simplicity of the problem and constant size makes it easy to implement in an IPv4 option header.

It is important to study how the time complexity changes for the algorithm for different integer sizes. Using the naive algorithm one needs to be aware of its average computation time for a desktop computer and send an appropriately hard problem to solve without affecting network service to a large extent. This average data is also used to set the time limit for a client to respond to its request for solution. An implementation suggestion and performance analysis is explained later in the thesis.

4.1.3 Hash inversion

In several papers the reverse hash problem is used as a proof-of-work puzzle and it is also partially used by Hashcash, which tries to compute partial match to a hash. It requires the email sender to generate a header to attach with the email that can easily be verified by the server. The sender tries to compute a hash that includes at least 20 (adjustable) zeroes as the beginning of the hash and keeps trying until successful. The recipient must only verify that the hash actually includes these zeroes to verify that work has been done by the client. It has many advantages: it is easy to implement and no central server is required, its postage system does not affect email sending speed noticeably and has been tested extensively by the community.[15]

Hash inversion of difficulty d takes 2^{d-1} hash operations to compute its solution. The drawback is that depending on the setting of d the difficulty of the problem varies a lot, and depends heavily on the resources available client-side. An attacker with a designated ASIC (Application Specific Integrated Circuit) can solve these inversions incredibly quick and will overwhelm a server if no precautions are taken.[7]

4.2 Distribution

In several puzzle-centered proof-of-work the communication between server and client is described as "challenge-response protocol". The client requests a service from a server, which responds with a puzzle and demands the client to solve it before admitting the client to any resources. When the response is given, the server verifies the solution and either grants access to the system for a time period or denies the request. Depending on protocol the server may respond in different ways to wrong solutions. It may ban the IP from any additional requests or send a lower difficulty puzzle if the server is not as loaded as when the original request arrived. [8]

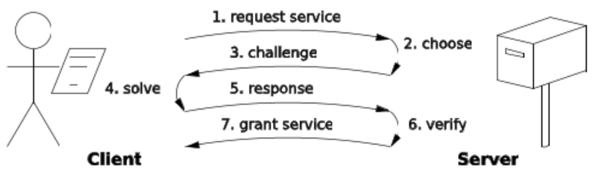

Figure 4.1. Challenge-response protocol.[16]

4.2.1 Server

A simple solution for puzzle distribution is to simply let the server handling requests issue the puzzles. The server holds a database of a set of puzzles. One of these puzzles is sent to a client when it tries to connect, unless that client is already trusted or banned from the server. It is important that the server also keeps track of which client got what puzzle otherwise there is no way of validating that the client solved a valid puzzle. Since the server needs to keep track of all these things, a banlist, a list of accepted connections and a list of connecting clients, it will create some additional server load. To avoid a fast overload the server should also remove stale connections that are not responding fast enough.

4.2.2 Bastion

Previous research at Princeton University and RSA Laboratories, shows that use of Bastions as an intermediate puzzle distributor is effective.[17] A bastion is described as a secure external entity, that can be a robust independent server or a network of these that serves with the purpose of distributing puzzles. The bastion does not need to be a specific server, it could be a publicly accessible data source. The importance lies in being able to eliminate puzzle distribution as a point of compromise. The bastion can hold puzzles of an arbitrary number of servers without knowledge of these. It is one level above of the servers using the bastion and can function for many types of puzzles. The flow for a bastion can be implemented in a number of ways as long as high importance is given to the anonymity and security of distribution. However, little attention has been given to the possibility of mounting DoS or other attacks on the bastion itself, compromising the protocol and putting the protected server at risk.

Letting a bastion act as an intermediate when distributing the puzzles makes it harder to overwhelm the actual server. But since the server does not keep track of the puzzles or its original solver, the bastion is a target nevertheless. If an attacker decides to mount an attack on the bastion itself this would lead to complications since the server cannot verify incoming requests correctly.

Chapter 5

Implementation

5.1 Protocol

Proof-of-work at the network level implies that a puzzle must be sent as a header in an IP packet. Servers cannot know which clients want to access resources and which of these are legitimate, therefore the client needs to request the puzzle from the server before a solution algorithm is run. The challenge-response protocol adheres to this methodology and is the core principle of the proposal. As previously described, the client first requests the puzzle and the server responds with a puzzle of appropriate difficulty - depending on the current server load. The server then either gives the client a time slice in which it may operate on the server or a temporary IP ban depending on the solution provided. This section provides a description of how to implement the proposed protocol and an actual implementation of it in Java can be provided by the authors via e-mail.

The protocol is initiated when a request arrives to the server. The server does a contain check on the three hashed tables for the client IP. The client can then be contained in one of three states: banned, accepted and connecting or be missing altogether.

Banned: the client has previously solved a puzzle incorrectly or failed to answer in time and is then placed in the banned table for a period of time specified by the administrator.

Accepted: the client has solved a puzzle successfully and may run the current request on the server.

Connecting: the client has been issued a puzzle and is now responding to the query with a solution. The header is read and the client changes state into either accepted or banned.

Requests not contained in any of these are regarded as new and are handled by merely sending a puzzle to them and adding their IP to connecting.

If the client IP already is in the ban list the packet is immediately dropped. The variable setting the time a client should be banned is set depending on type of server. On the other hand, if the client is in the accepted list, the client may run its request on the server with no further processing. In addition to the free request, the client is admitted for a short period of time in which it may request data from the server before another puzzle is submitted to be solved by the client. It should be allowed enough time so that the user can get some work done on the server before proving more work. But it should also have a reasonable limit so the server cannot be overwhelmed during the time slice. Testing showed that a simple Java TCP server that accepts a connection and then immediately drops it, was almost instantly depleted of its resources when bombarded through a socket by another netbook. As soon as the requests started to enter the system resources were utilized at maximum. Although it was done in the TCP layer, it is reasonable to believe it would be similar at a lower level. This concludes that the time slice given for every valid puzzle solution should be very short.

The time given is a parameter that should be configured for each setup because it greatly depends on the resources available on the server and the current level of efficiency of DoS-attacking software. As a rule of thumb, to avoid the risk of having a client that has solved a puzzle crashing the server, the grace period given on the server should probably be scaled to seconds. Because of the insecurity of the measurements no exact values are proposed and are up to the implemented to set the appropriate values. Our own testing and values set are shown in the section *Results*.

5.2 Puzzle

Working with the Internet layer puts a few restrictions on practicalities regarding puzzle design. When implemented in the IPv4 header, the option header cannot exceed 255 bytes. However in the IPv6 the option header is replaced with extension headers. An IPv6 packet can have several extension headers so it is plausible to have larger numbers than allowed by the IPv4 Option header. Even though the standard is slowly shifting towards IPv6 our suggested protocol will be backwards compatible. The puzzle has to be of minimal processing power to put together and to verify. The discrete logarithm problem is very easy to implement and is notorious in academia for its comparatively hard computation. This is why it is used as part of the DH key exchange. The integers used as a part of the exchange are much larger because these puzzles should not be feasible to solve. These attributes confirm the discrete logarithm problem as optimal for our approach.

To understand how the numbers are placed in the header it is important to understand how an option header is composed. The first eight bits consist of the header to the option, the following eight express the total size of the option that is to be transmitted and the rest belong to the actual data.

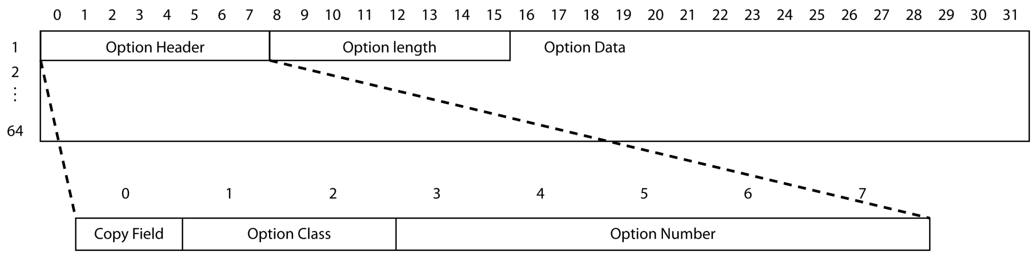

Figure 5.1. IPv4 Option Header layout.

Distributing the puzzle server to client, the discrete logarithm problem requires three integers. We require that all the integers are represented as the same number of bits, in practical terms the size will be 32 bit, 64 bits or 128 bits - although this may be extended up to the maximum capacity if necessary.

We designed a copy of the IPv4 option header in Java. The header is represented with a byte array of variable size, depending on the input to the header constructor. The first byte has a valid value like the first byte of a proper IPv4 header with the Copy field set to one, option class number set to 2 and the option number set to 31. The second byte, representing the length of the header, is also set to a value at creation somewhere between 0 and 255 depending on how large the option header needs to be. What follows is the actual Option Data, whose size is based on the length set in byte number 2. To mimic the behaviour and the calculations a normal package needs to do we also ensured that the byte length of the header is always divisible by four with the help of padding bytes all set to the value zero.

It is up to the server and the client to ensure that the numbers of the puzzles are written with the right length and to a correct position within the header class. When issuing a puzzle to a client, the server writes the three integers in the data field, with no spacing. The sender will be responsible to write the numbers in the right place into the Option Data field and the receiver will have to check so that it is a proper option header. If any errors occur when reading the option header the receiver will drop all the packets from that client for a set time.

In line with previous description of the puzzle structure, it will be sent as three integers. It will be expected by clients that the numbers arrive in the following order inside the header: a, g, n.

The integer in the Option Length field also has the condition to be divisible by three, otherwise it will be assumed to be erroneous. It is also assumed that there is no padding between the three numbers, hence every integer will be expected to take up one third of the Option Data field.

The solution will be a single integer μ. This integer is expected to be fill the entire size of the Option Data field of the option header. The rest of the puzzle does not need to be sent since the original parameters are saved server-side with the hashed IP. In our implementation the client solves the problem with the help of the naive algorithm described in the section *Research*.

5.3 Distribution

Puzzles are held by the main server. In order to be able to use different difficulties they are assigned to different lists. In our implementation they are split into four lists depending on their difficulty to solve, each one corresponding to a specific server load. Puzzles are fetched by the protocol from the corresponding list by first determining the total amount of connections currently interacting with the server. The very easy list consists of puzzles who are solved on average within a second, easy about 1-2 seconds, medium 3 seconds and hard 3-12 seconds. The difficulty of the puzzles was specified by the size of the prime number. Instead of wasting time generating the prime numbers runtime, these are read from several precomputed text files. The total time for generating 500 puzzles for each list is barely noticeable using this method and extending this size will have minimal consequences.

The lists are generated on the server start and refilled at the administrator's request. These should be renewed when a majority of the puzzles have been used to avoid the possibility of attackers saving the puzzles and using these to exploit the system. A server can keep the same list of primes and generate new a, and g for each prime with minimum computational power. Puzzles are chosen from the appropriate list at random and the same puzzle can be sent several times. The distribution to clients is seemingly random and the probability of receiving the same puzzle again is in practice low.

The distribution algorithm is simple. An new client requests a puzzle and the server does a server load check before choosing a random puzzle from the correct list. The puzzle is then written to a response packet and sent back to the client. A timer is set which gives the client double the expected time for solving a puzzle to make up for slow responding clients. If the client does not respond within the given time the thread on the server that is handling the request is stopped.

5.4 Post-Validation Behaviour

Following a successful validation of puzzle, the client is given a short period of time where it may request additional resources. We call this time the grace period of a request. The exact time is not specified due to the aforementioned tests and the difficulty of measuring these. The protocol should be tested and stress tested on the hardware to achieve an optimal value of the grace period. For the purpose of testing, the time slice was set to 10 seconds. If the response is incorrect or faulty in any way, the server will drop any additional requests by the client for a period of time. This is done by adding the client to the hashed IP ban list. In our particular application we implemented a 15 minute ban but this value could of course be adjusted appropriately to match the servers resources.

Chapter 6

Results

In order to study the performance of the implementation an application was developed. Its goal was to ensure that the proposal would defend against a live DoS attack. This testing was also the source of several conclusions discussed later in the thesis. Most importantly it displayed the advantages and flaws of our approach and the possibilities of future research on the matter.

6.1 Setup

The environment is a simple server/client environment with our protocol implemented in the application layer. The stress test involves several clients trying to overwhelm the server simultaneously while the server responds with proof-of-work requests and adjusts the difficulty of the puzzles according to the load. Since we do not have an unlimited amount of clients distributed on the Internet, we completely disregard the IP and simulate it by sending multiple requests from one client and creating a thread for each on ther server. If we can successfully hold off the attacks and keep the server stabilized the protocol can be deemed successful. In addition we look at how a normal client user perceives the protocol, both in a normal server condition and under attack. The users should only notice minor delays in the communication. The puzzle threshold below is the amount of connections held when the server switches to a harder puzzle list. The attack is mounted by a thread starting new clients continuously which open active connections to the server.

Experiment 1
Grace period time slice: 10 seconds
Ban time: 15 minutes
Puzzle thresholds: 0-10, 11-20, 21-30, 31-∞
Time to solve puzzle type: 0.5, 1, 2, 1-8 seconds
Clients: 1 (same as server)

Experiment 2
Grace period time slice: 10 seconds
Ban time: 15 minutes
Puzzle thresholds: 0-10, 11-20, 21-30, 31-∞
Clients: 3 computers (4 threads per computer). 12 threads solving puzzles in total.

Experiment 3
Grace period time slice: 10 seconds
Ban time: 15 minutes
Puzzle thresholds: 0-100, 101-200, 201-300 , 301-∞
Amount of clients: 4 computers (4 threads per computer). 16 threads solving puzzles in total.

Experiment 4
Grace period time slice: 10 seconds
Ban time: 15 minutes
Puzzle thresholds: 0-2500, 2501-5000, 5001-7500 , 7501-∞
Amount of clients: 3 computers (8 threads per computer). 24 threads solving puzzles in total.

6.2 Performance

As stated earlier, before starting work on the proof-of-work application we set up a simple test server and application to see how quick we could overload the server with a single client sending a long string. The results were daunting - the server could not handle more than a few seconds of traffic.

Our first experiment was a single computer running the client/server and protocol in Eclipse IDE. The system stabilized at around 40-50% CPU power and 15-30 connections. The very easy puzzles were solved within 1 second, and after the connections passed the threshold of 20 connections, the puzzles took 1-2 seconds to solve and send back. The next threshold (medium) was rarely reached and often took between 1-5 seconds. Testing of hard puzzles showed the time was between 1-8 seconds. This experiment allowed us to set the range of difficulty for puzzles. The very easy puzzle should not be noticed when being solved and the easy ones should be around a second. Medium and hard ones should note a slight delay since the server is already at high capacity.

In experiment two, the amount of clients connecting was drastically increased with the same configuration. The results were better than expected. All clients were quad core 2.83Ghz client with 4Gb of ram and could not keep up with more than four processes solving hard puzzles that take on average 4.5 seconds. Since the threshold was set low, the clients reached hard puzzles easily and struggled with these. The server (quad core 2.4Ghz 6Gb ram) on the other hand, had no trouble verifying puzzles from 12 different threads simultaneously. The system resources were barely affected by the attacks. The verification stabilized at around 50 connections accessing resources. The server was still not particularly affected by the attacks and we decided to increase the amount of threads and raise the threshold for experiment three. Clients still could not solve more than 4 puzzles simultaneously with 100% CPU power. The server on the other hand still worked on 10-15% CPU power. The connections stabilized around 150-200 connections.

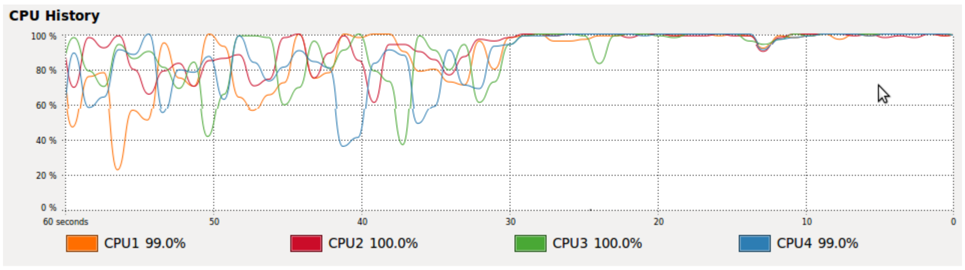

Figure 6.1. CPU load on a client going from 3 to 4 threads solving medium puzzles simultaneously.

For experiment four the configuration was altered to increase the load on the server. Since the server had not been very overworked yet we decided to lower the threshold a great amount so that clients could solve easier puzzles. The clients experienced almost no delay when requesting - which is the desired property if the server is not overloaded. The threshold for connections was 2500 before it would start issuing easy puzzles. The 24 threads could not send solutions fast enough to actually reach this limit and stabilized at around 1000 connections. The server was only using 20% of its resources and the clients used approximately 20% of CPU power to solve 8 very easy puzzles simultaneously. These findings showed that the protocol can handle 24 verification threads at the same time without being particularly affected. Also, the very easy puzzle hardness significantly changed the load on the clients. Instead of being able to run 4 threads it could now run 8 (and probably many more). The delay on the clients was for the most part very small. In an actual attack by more clients the delay would progressively grow longer as the hardness was increased.

Our results were conclusive. There are three core parameters that change the configuration of the protocol performance. The prime numbers of the puzzles can be switched to get a higher/lower puzzle difficulty client-side. Furthermore, the puzzle threshold can be set so that more connections are available at a given level - either increasing or decreasing the server load. Lastly, the grace period may be changed to specific needs. Although increasing the time slice is not recommended due to the increased risk, is also allows for more connections. The final settings should be assessed according to server hardware and use.

Chapter 7

Discussion

7.1 Challenges

The diversity of different computer setups is huge. There are users that still operate machines from the late nineties and others acquire the latest technology as soon as it is available. This disparity is an issue when trying to develop a general protocol that is dependant on the user's hardware. The goal was that as many users as possible should feel the same delay when using a proof-of-work as a countermeasure to DoS attacks, but it is very hard to achieve. Scientists and engineers around the world are increasing the power of computers continually. According to Moore's law the amount of transistors in a computer doubles approximately every two years. Although this law might not be completely true for upcoming years, the progress has shifted towards parallel computing and the efficiency of the software linked to this. Instead of increasing clock power, researchers believe the next performance growth drivers are: hyperthreading, multicore and caching.[18] It is most probable the ability to solve hard problems efficiently will increase in the future and is one of the reasons why we left the size of our integers variable.

We also faced the challenge of simulating a real DoS attack. To test the server performance under intense pressure there are two ways to proceed. Either the amount of clients is drastically increased or the time slice given for an accepted client is longer. The former is hard to simulate without great resources, and as students we were restricted to using a few machines in our university computer labs. The later leaves a huge vulnerability since clients in the grace period do not need to use the protocol and have time to mount attacks in between the puzzles.

Some research argues for memory-bound puzzles that do not depend on CPU power whatsoever, however these are still open questions. These puzzles assume an upper-bound on the memory cache and may not be applicable to future technology. These puzzles are also highly dependant of certain parameters of the memory configuration. Clients without large amounts of memory such as smartphones or PDAs may not be able to utilize these puzzles at all.[7]

There are several other interesting factors to have in mind. For instance, the advances within the domain of algorithms. Several other faster algorithms exist and there is also the possibility of finding an efficient algorithm if P = NP. However, this famous uncertainty is still one of the unsolved problems of computer science. Implementing one of these more efficient algorithms on a client and running it on our protocol would solve the problem faster than the protocol expects and could prove to be a vulnerability. Another risk is the danger of attackers developing specific software to circumvent the protocol or mount attacks on the protocol itself; this could be injecting code on the headers or exploiting the way the puzzles are generated. These security risks need to be studied carefully when implementing proof-of-work on a live system.

An aspect that also should be researched when implementing on a live system is what happens to the original payload of the inital packet to the server. There are several ways to handle the data for this type of protocol. For instance, copy the data to the new packet as a response or drop it and ask the client to resend it with the puzzle solution packet.

7.2 Possibilties

In the previous section a single bastion was shown to be a good method of distributing ephemeral puzzles to servers and works well without being especially susceptible to targeted attacks. We studied the use of bastions but did not implement this method to distribute puzzles. It was mainly because we had not allocated time to study this extra complexity. In addition, investigating all security aspects was not essential for a working protocol prototype. The use of bastions increases the difficulty to mount a DoS attack on the server but is not fail-proof. We believe that a system with several bastions would work even better. Decentralized networks have various advantages. There is not a single focal point to focus attacks on and usually the different systems are backups for each other if one were to fail. It could also work with some bastions merely generating puzzles while others are in charge of sending puzzles. We thought about introducing our own idea called distributed bastion system. This system would work as a tree, unknown to both client and servers and would serve puzzles as needed depending on the servers requesting. When it comes to distributed systems distributing puzzles, no research could be determined on the subject and the idea is left for others to discover its potential.

The discrete logarithm puzzle we used could be modified to another problem to counter disparity between clients. The tour puzzle described by Abliz and Znati using hash computations is a good alternative because it was specifically developed to thwart this problem.[7] A proper evaluation between their puzzle, ours and others', is recommended before settling for one. In order to fit the puzzle in the IP header, the size of the chosen puzzle must be relatively small and is an important factor to be mindful of. There is the also the possibility of researching a memory-dependant puzzle that may work better than a heavy CPU using.

Compared to many other protocols we chose to study the implementation as part of an IP packet. This idea was simulated with a Java application reading and writing option headers. The time restriction and our own knowledge of the subject prevented the implementation in the actual network layer. For someone knowledgeable in the subject it should not be very hard to apply the ideas we have presented to the option header in IP packets. It is our hope there will be more research conducted for this type of proof-of-work.

7.3 Conclusions

We believe that proof-of-work is a good method to counter DoS and DDoS attacks. Our simulation showed that if every network device would implement an additional protocol that stores some form of puzzle in the IP header, the impact of these types of attacks are partially mitigated. Mounting attacks on the actual protocol would in most cases include the need to reimplement how the IP header is handled at your specific computer which is very hard for the average user. Attacking with a DDoS attack implies the reconfigurations needs to be performed on every client the attacker controls because otherwise the protocol would defend itself. To say proof-of-work will solve DoS issues is an overstatement. Given the results, the study suggests it greatly obstructs attackers by forcing clients to show proof of work for each request.

The findings from the results showed that the server was only affected by how many connections it had active and not the difficulty of the incoming puzzles. The client-side on the other hand was greatly affected by the difficulty of the puzzles. When the difficulty increased to a level above very easy we noticed that the clients' resources were utilized to a greater extent. Four concurrent easy puzzles were enough to fully load a client. The quick verification and low server load in all experiments provides evidence of the discrete logarithm being acceptable according to our puzzle criterias.

There are some issues that need to be studied in more detail. The protocol would be improved if it was less dependent on the clients hardware. Another source of weakness is the lack of research of actual DoS attacks and the software used. Further research might prove the need for a more secure protocol capable of withstanding malicious conduct.

The experiments showed that this type of protocol can be implemented and provide a layer of defense against attackers. Its efficiency in a real environment is questionable depending on the nature of the attacks and the amount of clients connected. It is also difficult to say to which degree a user is prepared to wait for a server response without testing on actual users. Therefore many parameters are set as variables to be configured. More extensive analysis of distributing puzzles over actual IP packets and an authentic implementation would strengthen our conclusions in the matter.

Bibliography

[1] Symantec-Norton.com, "The 11 most common computer security threats... And what you can do to protect yourself from them.." `http://www.symantec-norton.com/11-most-common-computer-security-threats_k13.aspx`, April 2012.

[2] L. Zeltser, "8 Reasons for Denial-of-Service (DoS) Attacks." `http://blog.zeltser.com/post/10775687288/reasons-for-denial-of-service-attacks`, April 2012.

[3] Google, "What is reCAPTCHA." `http://www.google.com/recaptcha/learnmore`, April 2012.

[4] J. L. Luis von Ahn, Manuel Blum, "Telling humans and computers apart (automatically)." `http://www.cs.cmu.edu/~biglou/captcha.pdf`, April 2012.

[5] R. B. Internet Engineering Task Force, "Requirements for internet hosts – communication layers." `http://tools.ietf.org/html/rfc1122`, April 2012.

[6] J. L. Harry Hochheiser, "HCI and societal issues: a framework for engagement," tech. rep., Towson University, 2007.

[7] T. Z. Mehmud Abliz, "A guided tour puzzle for denial of service prevention," tech. rep., University of Pittsburgh, 2009.

[8] F. Coelho, "An (almost) constant-effort solution-verification proof-of-work protocol based on merkle trees." `http://eprint.iacr.org/2007/433.pdf`, April 2012.

[9] E. W. Weisstein, "Mathworld–a wolfram web resource.." `http://mathworld.wolfram.com/`, April 2012. Search: FundamentalTheoremofArithmetic, Semiprime, PrimeFactorization, PrimeFactorizationAlgorithms, DiscreteLogarithm, RelativelyPrime.

[10] S. Manuel, "Classification and generation of disturbance vectors for collision attacks against sha-1." `http://eprint.iacr.org/2008/469.pdf/`, April 2012.

[11] T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, J. W. Bos, P. Gaudry, A. Kruppa, P. L. Montgomery, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann, "Factorization of a 768-bit RSA modulus." `http://eprint.iacr.org/2010/006.pdf`, April 2012.

[12] D. Ireland, "RSA Algorithm." `http://www.di-mgt.com.au/rsa_alg.html`, April 2012.

[13] K. Palmgren, "Diffie-Hellman Key Exchange - A Non-Mathematician's Explanation." `www.recursosvoip.com/docs/english/WP_Palmgren_DH.pdf`, April 2012.

[14] M. Nilsson, "Discrete logarithms in Cryptography." `http://w3.msi.vxu.se/users/mni/math_crypt/100428.pdf`, April 2012. Published 28 April, 2010.

[15] Hashcash, "Hashcash FAQ." `http://hashcash.org/faq/`, April 2012.

[16] Wikipedia, "Proof-of-work system." `http://upload.wikimedia.org/wikipedia/en/5/55/Proof_of_Work_challenge_response.svg`, May 2012.

[17] J. A. H. Brent Waters, Ari Juels and E. W. Felten, "New Client Puzzle Outsourcing Techniques for DoS Resistance." `http://www.cs.utexas.edu/~bwaters/publications/papers/outsource_paper.pdf`, April 2012.

[18] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software." `http://webcache.googleusercontent.com/search?q=cache:http://www.gotw.ca/publications/concurrency-ddj.htm`, April 2012.