Buletinul Stiintific al Universitatii “Politehnica” din Timisoara, ROMANIA
Seria AUTOMATICA si CALCULATOARE
PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE
Vol.49 (63), 2004, ISSN 1224-600X

Threshold Puzzles: The Evolution of DOS-resistant Authentication

Valer BOCAN

Department of Computer Science and Engineering, Politehnica University of Timişoara, Bd. V. Pârvan, 300223 Timişoara, Romania
E-mail: vbocan@dataman.ro, WWW: http://www.dataman.ro

***Abstract* – Client puzzles have been proposed to add DOS resistance to authentication protocols. Due to the parallel design of puzzles, the technology is vulnerable to the so-called strong attacks. This paper advocates the need for time management of solved puzzle instances and introduces the “threshold puzzle” and “strong attack” concepts.**

***Keywords:* security, denial of service, attack, authentication, client puzzles, threshold puzzles**

I. INTRODUCTION

Denial of service attacks are a major problem in today’s interconnected world. There are numerous examples of websites which have been attacked and brought to their knees for hours: Yahoo!, Amazon, eBay, they have all lost hundreds of thousands of transactions during the down time which translates in losses of millions of dollars.
Attackers are known to exploit the end-user ignorance and break into hundreds of thousands of system to install their tool of choice. These “zombie” systems are capable of receiving commands from a central operations center via encrypted channels. The main reason for the very existence of such “zombies” is the generation of bogus traffic targeted to a specific website. In order to make tracking more difficult, the source IP address may be spoofed but in the same time may be chosen from the same subnet in order to avoid egress filtering [3].

Generating a massive traffic to a single destination is likely to alarm any system administrator that does a minimal monitoring. For ecommerce sites, the attack may be more subtle since the website may be available while the attack concentrates on the secure payment server so that nobody is able to make a successful purchase. The SSL and TLS protocols allow expensive operations (RSA) to be performed at the request of unauthenticated clients. If a large site can process around 4000 RSA operations per second and a partial SSL/TLS handshake consumes on average 200 bytes, then all it takes is approximately 800KB / sec. to paralyze the ecommerce site [3].

In order to add DOS-resistance to any authentication protocol, the design principle should be that the client always commits its resources before the server does and at any point during protocol execution the cost for the client should be greater than for the server. The client cost may be increased artificially by asking it to do some work whose difficulty may be effortlessly chosen by the server. At the same time, the verification for correctness should not place a burden on the server since that would defeat the very purpose of the technique.

II. RELATED WORK

In his 1978 paper [4], Merkle was the first to come up with the idea of cryptographic puzzles but he only applied puzzles for key agreement rather than authentication. Client puzzles have been applied to TCP SYN flooding by Juels and Brainard [2] who mention that SSL has the same problem and give a rigorous proof of the security characteristics. Aura, Nikander and Leiwo apply client puzzles to authentication protocols in general [1]. Client puzzles were also proposed as a regulating measure against junk mail by Dwork and Naor [5] and the related problem of time-locked cryptography was discussed by Rivest, Shamir and Wagner [6]. However the inherent sequential nature of time-locked cryptography also makes it very difficult for the server to verify the solution.

III. CLIENT PUZZLES

Before committing resources the server should ask the client to solve a problem, as seen in figure 1. Regardless of the specific implementation, a good puzzle should have the following properties (as described in [1]), the last of which being new:

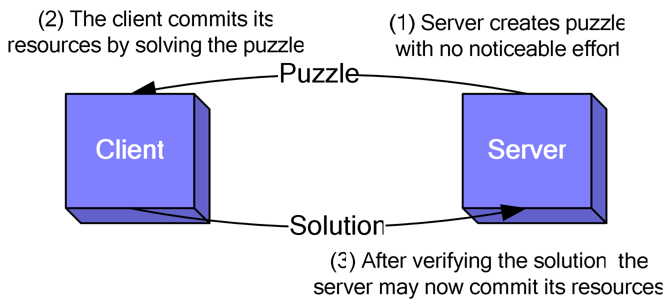

(2) The client commits its resources by solving the puzzle
(1) Server creates puzzle with no noticeable effort
Puzzle
Client
Server
Solution
(3) After verifying the solution the server may now commit its resources

Fig. 1. Principle of the client puzzle protocol

1. Creating a puzzle and verifying the solution is inexpensive for the server.
2. The cost of solving the puzzle is easy to adjust from zero to impossible.
3. The puzzle can be solved on most types of client hardware (although it may take longer with slow hardware).
4. It is not possible to precompute solutions to the puzzles.
5. While the client is solving the puzzle, the server does not need to store the solution or other client-specific data.
6. The same puzzle may be given to several clients. Knowing the solution of one or more clients does not help a new client in solving the puzzle.
7. A client can reuse a puzzle by creating several instances of it.
8. The puzzle should not be solved in less than a predetermined amount of time.

The natural choice for a client puzzle is the brute force reversal of hash functions such as MD5 or SHA1 since they have a simple structure and can run on a variety of hardware platforms. Juels and Brainard [2] have also proposed the use of a reduced round cipher instead of the hash function but that is beyond the scope of this paper.

A. *Creating a New Puzzle*

Periodically (say once every few minutes), the server generates a random value $\mathbf{N_S}$. In order to prevent attacks by guessing the nonce, the value should have 64 bits of entropy and should not be a predictable value such as a time stamp. This entropy should be enough to prevent an attacker to precompute «nonce-result» pairs and the occasional matches caused by birthday attacks would not do too much harm here. The server has to decide the difficulty level **k** of the puzzle, based on the current conditions. To sum up, the puzzle that is broadcast to clients is the pair:

$$\text{« } \mathbf{N_S, k} \text{ »}$$

- N_S – server nonce (usually 64-bit, unpredictable quantity)
- k – puzzle difficulty level

B. *Solving the Puzzle*

To solve the puzzle, the client generates a nonce N_C. The purpose of this nonce is twofold. First, if the client reuses a server nonce N_S, it can create a new instance by generating a new N_C. Second, without the client nonce an attacker could compute the puzzle and send the result back to the server before the client does. 24 bits of entropy should be enough to prevent the attacker from exhausting the values of N_C given that N_S changes frequently.

The client must repeatedly apply a hash function to a quantity and the puzzle is considered solved when the first **k** bits of Y are equal to 0.

$$\mathbf{h(C, N_S, N_C, X) = Y}$$

- h – cryptographic hash function, such as MD5 or SHA
- C – client identity
- N_S – server generated nonce
- N_C – client generated nonce
- X – solution of the puzzle

Since the server changes N_S periodically, while it considers N_S recent, it must keep a list of correctly solved instances in the form of N_S-N_C pairs so that previous solutions cannot be reused.

Since there are no known shortcuts to find out X, the only possibility is to search for it by brute-force. The difficulty level **k** (i.e. the number of zeros at the beginning of Y) dictates how long the puzzle will take to solve. If k equals 0 then no work is required, whereas if k equals 128 (for MD5) or 192 (for SHA), the client must reverse an entire one-way function which is computationally impossible.

C. *Puzzle Difficulty*

The parameter k represents the puzzle difficulty. The task of establishing it at the time of puzzle generation is rather tricky, since there is no obvious metric that one can use in a real-world implementation. According to [3], the best approach would be the number of already committed RSA operations rather than the current processor load or the number of incoming requests.Unfortunately, the puzzle difficulty follows an exponential curve and thus it is limited in practical purposes. To solve a puzzle of difficulty k, the client needs to perform on average $2^k - 1$ operations. In [1], Aura, Nikkander and Leiwo state that reasonable values for the difficulty level (k) are between 0 and 64. By experimenting, I have found out that the reasonable range is much narrower and for small difficulty levels, the time needed to solve the puzzle for level k may be greater than the time for level k+1.
As of today (beginning of 2004), the average web client is capable of approximately 4500 – 5000 MIPS which leads to 0.02 milliseconds per cryptographic operation. Thus, the puzzle difficulty curve looks as in figure 2. For difficulty levels above 20, the time needed to solve the puzzle is prohibitive, hence the limited practical applicability. A cryptographic operation is considered an attempt (not necessarily successful) to solve the puzzle and includes the time needed to build up the quantity to apply hash to and the actual computation of either an MD5 or SHA function.

In order to obtain a more accurate scale for the puzzle difficulty parameter, Jules and Brainard [2] proposed that puzzles be split into several smaller puzzles of equal difficulty that should be solved separately and the general result be the combined individual result. Aura, Nikkander and Leiwo [1] stated that the same granularity can be achieved by combining sub-puzzles of varying difficulty, at a slightly lower cost for the server, but that is yet to be confirmed by experiment.

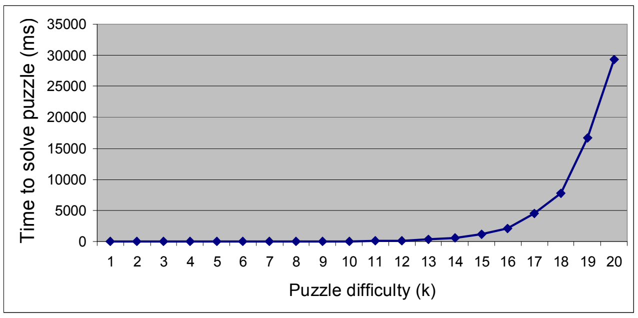


Fig. 2. Solving time for different puzzle difficulties

IV. THRESHOLD PUZZLES

Client puzzles have proved effective both in theory and in practice. They are secure and perform well in most scenarios. Regardless of the particular client being serviced, the puzzle difficulty is chosen based on a metric that refers strictly to the server resource commitment. Since puzzles may be broadcast and are generated at precise intervals, this "one size fits all" solution is not perfect since different clients have various computing powers. I have noted that client puzzles are vulnerable to a particular form of attack (called henceforth "strong attack") due to the highly parallel nature of the puzzle. A strong attack is defined as a denial of service attack mounted by an attacker with access to massive computing power. The attacker is able to solve puzzles in a time much shorter than a legitimate client. The schematic of a strong attack is shown in figure 3.

Suppose that a server authenticates a number of legitimate clients and the initial puzzle difficulty is set to zero. When a strong attack is in progress, the server has the tendency to gradually increase the puzzle difficulty up to high values in order to cope with the important amount of work required to service the attacker's requests. While puzzle difficulty may be increased up to impossible, this also means a DOS attack in its own right targeted against legitimate clients who may never solve a puzzle such difficult.

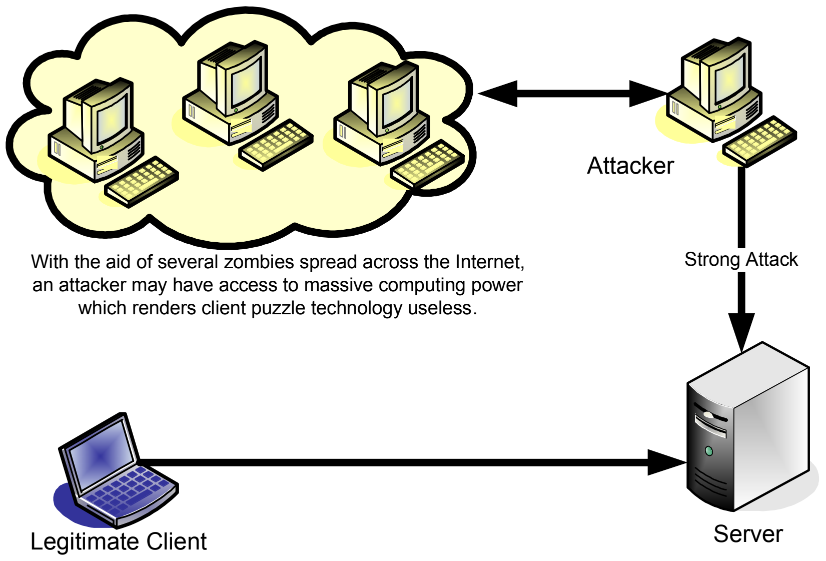
Although not very likely, a strong attack is possible. If an attacker had access to other N computers (with N being sufficiently large so we speak about massive computing power), then time needed to solve a puzzle with difficulty k would be divided by N. The SETI program [7] and the effort to break the RSA algorithms [8] are real world examples of how hundreds of thousands of computers are put to work together for a common purpose. The cumulated power of the *distributed.net* network exceeded the equivalent of 160000 PII 266MHz computers, which clearly shows that strong attacks are possible under certain circumstances.

I propose two changes in the existing client puzzle specification:

- Limiting the difficulty level so that the puzzle remains within usability margins.
- Adding a minimum response time to the puzzle definition.

A. *Limiting the Puzzle Difficulty Level*

Although the current design of the client puzzle as it is described in [1] specifies a difficulty range from 0 (no work required) to 128 or 192 (impossible, depending on the hash function used), a real-world implementation of an authentication protocol is likely to choose a reasonable range for the puzzle difficulty, say between 0 and 25, due to the exponential scale which gives a narrow usability margin. Having difficulty levels close to impossible may open a new avenue of attack against the legitimate clients themselves and this is an issue even more serious than attacking just the server.



Attacker
Strong Attack
With the aid of several zombies spread across the Internet, an attacker may have access to massive computing power which renders client puzzle technology useless.
Legitimate Client
Server

Fig. 3. Schematic of a strong attack on an authentication protocol protected by client puzzles

B. Establishing the Minimum Response Time

The basic idea is to add the timestamp at which the server nonce was generated to the list «N_S, N_C, X, k» which is kept by the server in order to prevent reusing puzzle instances. When the server receives a solution to a puzzle, it can calculate the time it took the client to solve the puzzle and that should not be less than an estimated duration. If it is, then the server is under a strong attack and should immediately cease communication with the client in question. On average it takes $2^k - 1$ operations to solve a puzzle of difficulty k, hence the formula to estimate the time needed to solve a puzzle of difficulty k is:

$$\mathbf{T_{estimated} = (2^k - 1) * T_{operation}}$$

- $T_{estimated}$ – the estimated time for solving the puzzle
- k – puzzle difficulty level
- $T_{operation}$ – minimum time for performing a cryptographic operation (currently in the range 0.01 – 0.02 milliseconds, must be determined experimentally or the Moore law should be applied at the time actual implementation is done)

The estimated time represents the acceptance threshold for the client puzzle. A client puzzle with the above mentioned changes is called a **threshold puzzle**.

V. DOS-RESISTANT AUTHENTICATION USING THRESHOLD PUZZLES

Client puzzles have been used to add DOS-resistance to authentication protocols in [1]. Using threshold puzzles does not incur important changes and the scenario is similar.

The protocol normally begins with a client requesting a connection, in the form of a **ClientHello** message. The server generates the puzzle (the N_S and k parameters) and sends the **ServerHello** message back to the client. Optionally, the message may be time stamped and signed in order to prevent attackers from forging puzzles generated by the server. If the ClientHello message is missing from the design of the authentication protocol, then the server may broadcast ServerHello messages with the same nonce. The server nonce must change periodically.

Any client willing to talk to the server has to generate a random nonce N_C and must correctly solve the puzzle and supply the C, N_C and X parameters for verification. In case it wants to initiate several connections to the same server, the client may reuse the puzzle by generating a new N_C.

Upon receipt of a solved puzzle, the server checks whether the client C has already submitted a solution with the same N_S and N_C. This check ensures that solutions are not replayed. At this point the protocol on the server side is different than the one described in [1], since the server performs an additional step. The server checks whether the puzzle was solved in a time shorter than the estimate. If that is the case, then the server is under a strong attack and drops the connection to the client in question, without committing any resources. If the time exceeds the estimate, then the server proceeds with calculating the hash, verifies the signature and continues the normal protocol execution. See figure 4 for the schematic of an authentication protocol that uses threshold puzzles.

VI. EXPERIMENTING WITH THRESHOLD PUZZLES

In order to prove the theory behind threshold puzzles, I performed a series of experiments. The most popular and the most widely deployed authentication protocol is SSL and that was a natural choice for testing the theory. An excellent C# open-source implementation of the SSL/TLS suite of protocols is the Mentalis Security Library [9]. The library is modified so that it has support for creating, solving and verifying puzzles. Message signing was omitted for brevity.

The puzzle challenge contains the time at which it was generated (so that the client knows how old the puzzle is), the requested difficulty level and the server nonce.

```
[Serializable]
public class PuzzleChallenge
{
        public DateTime TimeStamp;
        public int      Difficulty;
        public ulong    ServerNonce;

        public PuzzleChallenge(int Difficulty)
        {
        this.Difficulty = Difficulty;
        TimeStamp       = DateTime.Now;

        Random rand = new Random();
        ulong a = (ulong)rand.Next();
        ulong b = (ulong)rand.Next();
        ServerNonce     = a + 65536 * b;
        }
}
```

The puzzle solution contains the client ID, the original server nonce, a randomly generated client nonce and the solution of the puzzle. The C# class looks as follows:

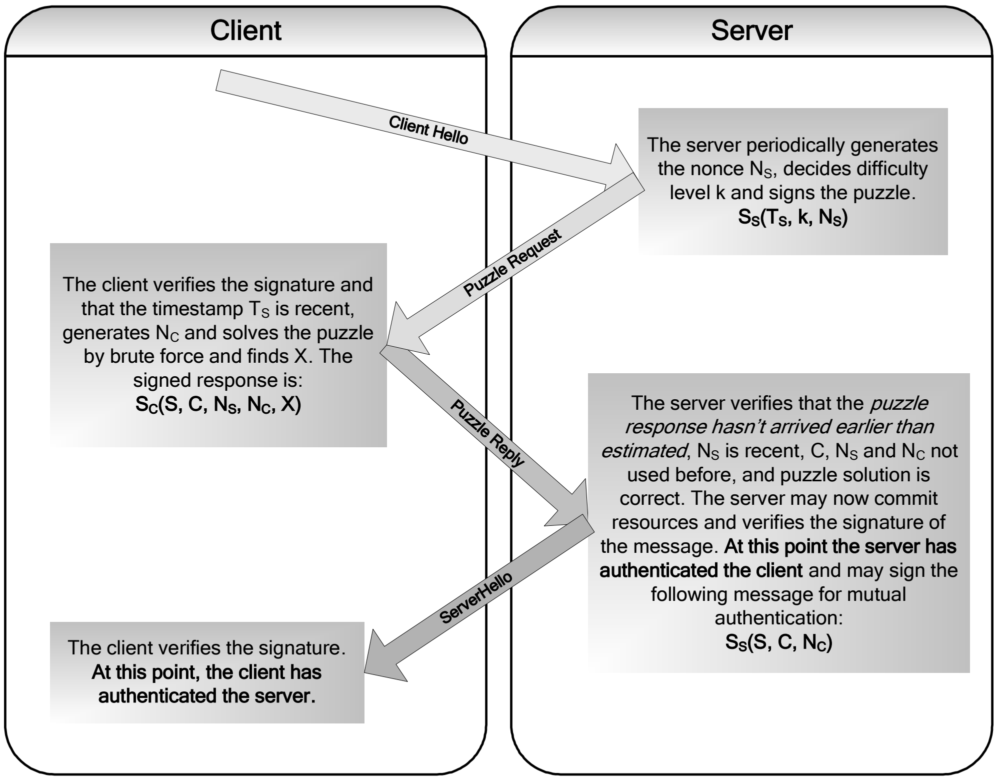

Client
Server
Client Hello
The server periodically generates the nonce N_S, decides difficulty level k and signs the puzzle. $S_S(T_S, k, N_S)$
Puzzle Request
The client verifies the signature and that the timestamp T_S is recent, generates N_C and solves the puzzle by brute force and finds X. The signed response is: $S_C(S, C, N_S, N_C, X)$
Puzzle Reply
The server verifies that the *puzzle response hasn't arrived earlier than estimated*, N_S is recent, C, N_S and N_C not used before, and puzzle solution is correct. The server may now commit resources and verifies the signature of the message. **At this point the server has authenticated the client** and may sign the following message for mutual authentication: $S_S(S, C, N_C)$
ServerHello
The client verifies the signature. **At this point, the client has authenticated the server.**

Fig. 4. Schematic of an authentication protocol protected by threshold puzzles

```
[Serializable]
public class PuzzleSolution
{
        public string  ClientID;
        public ulong   ServerNonce;
        public uint    ClientNonce;
        public ulong   Solution;
        public PuzzleSolution(string ClientID)
        {
        this.ClientID  = ClientID;
        ClientNonce = (uint)(new
Random()).Next();
        }
}
```

Both classes are marked as serializable to allow integration in the protocol binary stream. The server nonce is a 64-bit quantity and the client nonce is a 32-bit quantity.

The hash function (MD5 in this particular case) is calculated by transforming the parameters C, N_S, N_C and X into a stream of bytes and concatenated into a larger buffer. The MD5 function calculates the hash of the buffer, as follows:

```
ulong ComputeHash(string ClientID, uint
ClientNonce, ulong ServerNonce, ulong X)
{
// Build MD5 cryptographic provider
MD5CryptoServiceProvider md5 = new
MD5CryptoServiceProvider();
byte []buff1 =
BitConverter.GetBytes(ClientID.GetHashCode());
byte []buff2 =
BitConverter.GetBytes(ClientNonce);
byte []buff3 =
BitConverter.GetBytes(ServerNonce);
byte []buff4 = BitConverter.GetBytes(X);
int pos = 0;
byte []buffer = new byte[buff1.Length +
buff2.Length + buff3.Length + buff4.Length];
Array.Copy(buff1, 0, buffer, pos,
buff1.Length);
pos += buff1.Length;
Array.Copy(buff2, 0, buffer, pos,
buff2.Length);
pos += buff2.Length;
Array.Copy(buff3, 0, buffer, pos,
buff3.Length);
pos += buff3.Length;
Array.Copy(buff4, 0, buffer, pos,
buff4.Length);
pos += buff4.Length;
// Compute hash
ulong longhash =
BitConverter.ToUInt64(md5.ComputeHash(buffer),
0);
return longhash;
}
```

In order to find the puzzle solution (X), brute-force must be used and the simplest approach is to cycle through all possible values of a 64-bit quantity. When the puzzle solution is correct (the first k bits are all 0s), the cycle is stopped.

```
PuzzleSolution Solve(PuzzleChallenge Puzzle)
{
PuzzleSolution ps = new
PuzzleSolution("ClientID");
// Copy over the server nonce
ps.ServerNonce = Puzzle.ServerNonce;
// Verify whether the timestamp is newer than 1
minute
if(Puzzle.TimeStamp.AddSeconds(60) <
DateTime.Now) throw new Exception("Puzzle is
older than the configured amount of time.");

for(ulong x = System.UInt64.MinValue; x <
System.UInt64.MaxValue; x++)
{
ulong longhash = ComputeHash(ps.ClientID,
ps.ClientNonce, Puzzle.ServerNonce, x);
if(BitCounter(longhash) == Puzzle.Difficulty)
{
        ps.Solution = x;
        break;
}
}
return ps;
}
```

The BitCounter function has been omitted for brevity. Its purpose is to count the number of 0 consecutive most-significant bits from the supplied quantity.

The SSL library was modified so to include two additional messages (PuzzleChallenge and PuzzleReply) before the ServerHelloDone message which ends the SSL handshake protocol. Based on the modified SSL library, the following modules have been created:

- **Legitimate Client** – a normal client which follows the normal SSL execution path as it is supposed to.
- **Malicious Client** – a client who has access to massive computing power (this is simulated by not solving the puzzle at all and the server not checking for solution correctness) and may yield several connection requests in a short time.
- **Normal Server** – a SSL server protected by the client puzzle technology.
- **Threshold Server** – a SSL server protected by the threshold puzzle technology

Using the mentioned modules I performed several tests. Each tests involved two Pentium IV - class computers connected through a 100 Mbps Ethernet link, the “client” computer running Windows 2000 Professional and the “server” running Windows 2003 Server Standard, respectively. For each individual test, see the average time for a request issued by a legitimate client trying to connect to the server.

A. Legitimate Clients Connecting to a Normal Server

Under normal circumstances, the clients experienced minimal delays, due solely to the SSL handshake protocol and transfer time throughout the network. The puzzle mechanism was not used since the server was perfectly capable of servicing all requests in time. The average request time is 4545 ms.

B. Malicious Client Attacking a Regular Server

When at least one of the clients is malicious, the server load increases dramatically and so does the puzzle difficulty. The legitimate clients are forced to solve difficult puzzles and at a certain point, the delay experienced by them is prohibitively long, causing a DOS attack targeted against legitimate clients themselves. The average request time is 10339 ms.

C. Malicious Client Attacking a Threshold Server

If the server uses the threshold puzzle technology, the malicious clients are spotted immediately (since they tend to solve puzzles a lot sooner than they normally should) and the server does not commit resources for them and therefore the overall puzzle difficulty is not increased significantly. The average request time is 4553 ms.

VII. CONCLUSIONS

I have shown that client puzzles are vulnerable to attacks mounted by malicious clients that have access to massive computational power and that the lack of upper bound for the puzzle difficulty parameter may result in denial of service attacks against legitimate users themselves rather than only the server. As a corrective measure, I introduced the concept of threshold puzzle which has the benefit of keeping track of the client solving times and protects both the server and its legitimate clients from DOS-attacks.

REFERENCES

[1] Tuomas Aura, Pekka Nikander, Jussipekka Leiwo – “DOS-resistant Authentication with Client Puzzles”, 2001, http://research.microsoft.com/users/tuomaura/Publications/aura-nikander-leiwo-protocols00.pdf

[2] Ari Juels, John Brainard – “Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks”, Proceedings of NDSS, 1999

[3] Drew Dean, Adam Stubblefield – “Using Client Puzzles to Protect TLS”, http://www.csl.sri.com/users/ddean/papers/usenix01b.pdf

[4] R. C. Merkle – “Secure Communications Over Insecure Channels”, Communications of the ACM, April 1978

[5] Cynthia Dwork and Moni Naor – “Pricing via Processing or Combating Junk Mail”, Proceedings of CRYPTO ’92, Springer Verlag 1992.

[6] Ronald R. Rivest, Adi Shamir, David A. Wagner – “Time-lock Puzzles and Timed-release Cryptography”, 1996, http://lcs.mit.edu/~rivest/RivestShamirWagner-timelock.pdf

[7] SETI @home Program, http://setiathome.ssl.berkeley.edu/

[8] The Distributed.net Organization, http://www.distributed.net

[9] Mentalis C# Security Library, http://www.mentalis.org