Using Client Puzzles to Protect TLS

Drew Dean*
Xerox PARC
ddean@parc.xerox.com
Adam Stubblefield†
Rice University
astubble@rice.edu

Abstract

Client puzzles are commonly proposed as a solution to denial-of-service attacks. However, very few implementations of the idea actually exist, and there are a number of subtle details in the implementation. In this paper, we describe our implementation of a simple and backwards compatible client puzzle extension to TLS. We also present measurements of CPU load and latency when our modified library is used to protect a secure webserver. These measurements show that client puzzles are a viable method for protecting SSL servers from SSL based denial-of-service attacks.

1 Introduction

Denial-of-service attacks have become a major problem on the Internet. Major web sites have been taken down for several hours at a time by distributed denial-of-service (DDoS). The attackers have shown an interesting combination of skill and ignorance. They are able to break into tens or hundreds of machines and install their tool of choice. They then use these "zombie" machines to actually launch the DDoS attack. Some of the tools even use encrypted communications between the attacker and zombie machines. The tools forge the source IP address on the traffic they generate in order to make determining the zombie machine somewhat harder. They will pick IP addresses that are on the same subnet, in order to overcome egress filtering.

However, the tools work via brute force: they just generate random traffic (perhaps with a political message) aimed at a particular machine. While generating a gigabyte per second of traffic aimed at a single machine will bring most websites down to their knees, the sheer volume traffic stands out for anyone doing network monitoring. For ecommerce sites, the attacker could easily arrange an attack such that the website remained available, but web surfers are unable to complete any purchases. Such an attack is based on going after the secure server that processes credit card payments[1]. The SSL/TLS protocol, as it stands, allows the client to request the server to perform an RSA decryption without first having done any work. RSA decryption is an expensive operation; the largest secure site we are aware of can process 4000 RSA decryptions per second. If we assume that a partial SSL handshake takes 200 bytes, then 800 KB/s is sufficient to paralyze an ecommerce site. Such a small amount of traffic is much easier to hide.

The rest of this paper describes our design and implementation of a modification to the TLS protocol to overcome this attack. We use the idea of client puzzles to slow down the attacker sufficiently that a denial-of-service is no longer possible. The rest of the paper is organized as follows: Section 2 discusses related work, Section 3 presents our approach, Section 4 contains an analysis of our proposed scheme, Section 5 discusses future work, and Section 6 concludes.

2 Related Work

The original idea of cryptographic puzzles is due to Merkle [8]. However, Merkle used puzzles for key agreement, rather than access control. Client puzzles have been applied to TCP SYN flooding by Juels and

*This work was supported in part by DARPA grant N66001-00-1-8921.

†This work was completed while the author was an intern at Xerox PARC.

[1]Most ecommerce sites only use a secure server for transmitting payment information.

Brainard [7], who mention that SSL has a similar problem. Aura, Nikander, and Leiwo apply client puzzles to authentication protocols in general [2]; we share a number of techniques with them. However, we concentrate specifically on TLS. Dwork and Naor presented client puzzles as a general solution to controlling resource usage, and specifically for regulating junk email [3]. Their schemes develop along a different axis, primarily motivated by the desire for the puzzles to have shortcuts if a piece of secret information is known. Franklin and Malkhi use iterated application of a strong hash function as a forgery-resistant timer for web usage monitoring [6]. Again, this is slightly different, as we are not interested in counting time, rather in providing a rate limiting step for new TLS connections. Rivest, Shamir, and Wagner posed the related problem of time-lock cryptography in their 1996 paper [9]. Our goal is much much more limited than theirs; we seek only to prevent a denial of service attack on TLS. This implies that we need not concern ourselves with making our puzzles inherently sequential; an attacker capable of solving puzzles in parallel could just as easily launch multiple attacks in parallel.

3 Design and Implementation

It is always critical to be explicit about the goals and assumptions of security work. Our goals are as follows:

1. To prevent denial of service attacks on secure web servers.
2. To remain as backwards compatible as possible with TLS.
3. To minimize additional server load added by implementing these measures.

These goals are listed in order of importance. Clearly, solving the problem we are considering is the first priority. Remaining compatible with existing TLS implementations is much more important than minimizing impact on server performance, because purchasing small additional amounts of CPU power is generally inexpensive.

We make the following assumptions about the environment:

1. That the server being protected has ample network bandwidth. We cannot prevent the brute force DDoS attack; we only aim to prevent an attack against TLS.
2. That legitimate clients, seeking access to a heavily loaded server, are willing to perform a computation that takes no more than a few seconds, and often much less.

These assumptions acknowledge fundamental tradeoffs in availability. We are working at the TLS layer, on top of TCP. If there is insufficient bandwidth for TCP to operate, there is nothing we can do about it here. We require each client to make a small sacrifice in peak performance to make average performance better. This is almost always a good tradeoff to make.

3.1 Design

The TLS protocol breaks up the underlying TCP stream into a record oriented protocol. The unshaded portions of Figure 1 diagram the opening TLS handshake. The TLS specification specifies that unknown (to a particular implementation) record type shall be ignored. Therefore, we use a new record type for the puzzle messages. This allows us to we remain backwards compatible with old TLS implementations that do not support puzzles. Though such implementations may time out a connection if they do not reply to a puzzle, they will not notice any protocol violations. This technique is only applicable to TLS and does not work for SSLv3 as SSLv3 does not discard unknown record types. When the server is not under attack, no changes in the TLS protocol are required.

In order to prevent the denial-of-service attack against TLS, we need to add a new message after the Server Hello message and before the Server Done message. See the shaded portions of Figure 1. This message contains a cryptographic puzzle and is only sent when the server is under load. The server will then wait on a response message before continuing with the handshake protocol.

3.1.1 The Client Puzzles

To be useful as a client puzzle, a puzzle needs to be solvable in a predictable amount of time. The puzzle generally should not take too long to solve (*e.g.,* no more than a second or so on a relatively slow machine), but at the same time, there should be no known shortcut to solving the puzzle. In addition, the server needs to be able

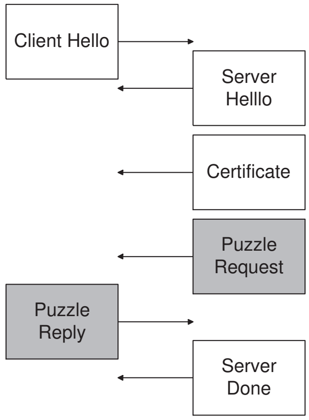

Figure 1: The TLS handshake protocol. The shaded portions are our additions.

to generate puzzles while doing much less work than the client solving them. Of course, the server also needs an efficient method of verifying the correctness of a proposed solution.

For $h(x)$, a preimage resistant hash function, a client puzzle is the triple $(n,x',h(x))$, where x' is x with its n lowest bits set to 0. Both MD5 and SHA-1 are conjectured to be preimage resistant. The solution to the puzzle is the full value of x. Because $h(x)$ is preimage resistant, the best way for a client to generate x, is to try values in the domain bounded by 0 and x' until a match is found. This should take, on average, 2^{n-1} calculations of $h(x)$. The server, on the other hand, needs to generate a random block (for MD5 and SHA-1, 512 bits) of data, and evaluate the hash function twice. Note that we generate an entire random block rather than just the unknown bits to prevent an attacker from effectively precomputing all possible puzzles.

3.1.2 The Message Format

The client puzzle message format we use is described using the TLS presentation language in Figure 2. The value *PuzzleLength* is the number of unknown bits in *PuzzlePreimage*. *PuzzleHash* is the target value and is computed by taking the MD5 hash of the correct puzzle solution. In the reply message, *PuzzleSolution* is the client's solution to the puzzle. The server checks to make sure that $MD5(PuzzleSolution) = PuzzleHash$.

```
struct {
 uint8 PuzzleLength;
 uint8 PuzzlePreimage[64];
 uint8 PuzzleHash[16];
} PuzzleChallenge;

struct {
 uint8 PuzzleSolution[64];
} PuzzleResponse;
```

Figure 2: The client puzzle messages

3.1.3 Determining Server Load

We desire only to send puzzles when the server is overloaded for two reasons:

1. While the server is not overloaded, we would like all TLS clients, not just those with our modifications, to be able to communicate with the server.
2. There is no point to adding more latency to TLS connection setup when the server is not overloaded.

Determining when a server is overloaded due to incoming TLS connections is one of the interesting problems we faced. There is no obviously correct metric: the machine's load is due to many factors, and the rate of new TLS connections may be quite high, but still manageable, *e.g.,* if the machine has a hardware cryptographic accelerator. Note that the computationally intensive section of the TLS handshake protocol occurs after the client sends the *ClientKeyExchange* message[2]. At that point, the server must perform a public key operation. In the most common case, this operation is a RSA private key decryption. This is the operation that we seek to protect the server from having to complete for malicious users.

The metric we choose to use counts the number of RSA decryptions that we have committed to performing. We increment this count when either we decide not to send a client puzzle or when the correct solution to a client

[2] There are other computationally intensive sections of the protocol if the server agrees to an anonymous or export cipher suite. These are beyond the scope of this paper.

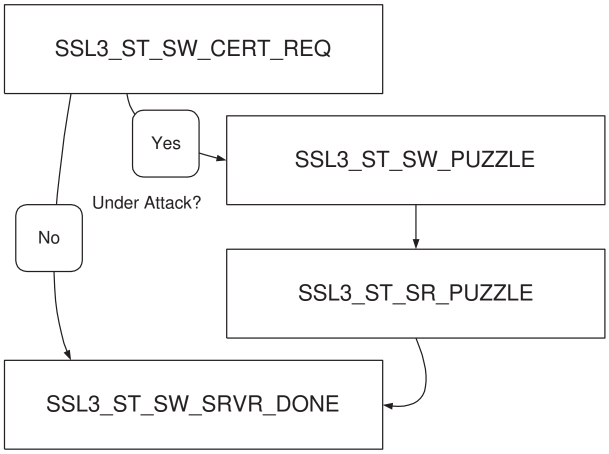

Figure 3: TLS states in OpenSSL

puzzle is submitted. The count is decremented after the corresponding RSA decryption has completed or if the connection is closed before the RSA operation is begun. This measures exactly what we care about: whether there is a backlog of public key operations to be performed in the near future.

By detecting whether this value is above a specified limit, the server can determine whether or not to send a client puzzle. More complicated schemes based on this metric such as a state machine with distinct entrance into and exit from client puzzle mode levels or statistical regressions are also possible.

3.2 Implementation

In order to measure the effectiveness of our solution, we modified the OpenSSL library to support servers and clients that understood our puzzle protocol. On the server side, hooks were added to the `mod_ssl` Apache module and as a client the TLS enhanced version of lynx was used.

3.2.1 The OpenSSL Library

OpenSSL is an open-source library that includes support for the SSL and TLS protocols as well as the underlying cryptographic operations needed by SSL and TLS [5]. OpenSSL handles connections on a per-socket basis and does not keep any global process state. This prevents a clean separation between our modified OpenSSL library and server applications because we need to measure the

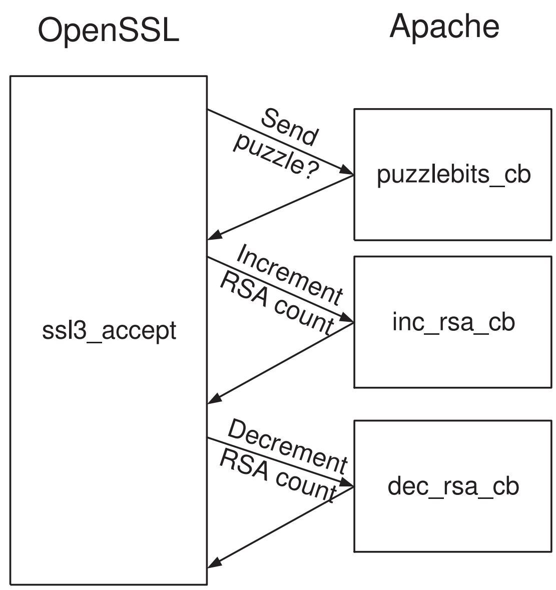

Figure 4: Control flow in OpenSSL with client puzzles

global server load. On the client side however, the application never needs to be aware whether the puzzle protocol took place. Clients can trivially support the protocol just by relinking with the modified library.

In OpenSSL, the TLS handshake is implemented as a state machine representing the current location in the protocol. To add support for puzzles on the server, a new state was added after the server certificate request state. In this state the server either sends a puzzle request and switches to a state waiting to receive the puzzle reply or immediately switches to the server done state. The puzzle reply state will wait to receive a puzzle solution before switching to the server done state. If a puzzle solution is never received the connection will time out. This is diagrammed in Figure 3.

On the client side, we treat the reception of a puzzle as an “unexpected event”, in the incoming message handler because the client is expecting a handshake record. The puzzle solution is then computed and returned to the server before the handshake processing continues.

The biggest challenge is deciding whether a server should send a client puzzle. Because OpenSSL has no notion of application or system wide state, it has no way to count the number of RSA operations a server has committed to. To remedy this problem, we provide callbacks to alert the application whenever we commit to or finish an RSA private decryption. We also add a callback that

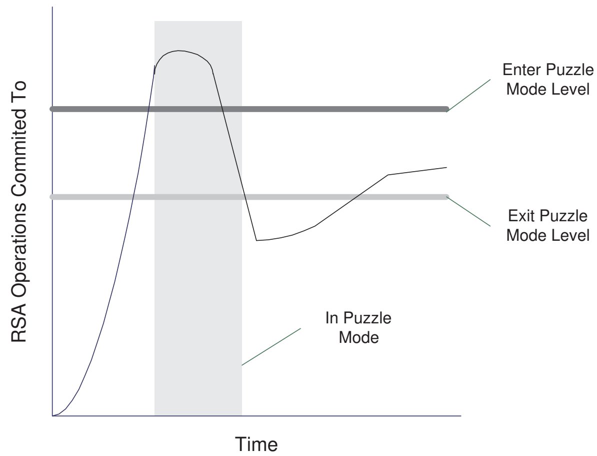

Figure 5: Puzzle states

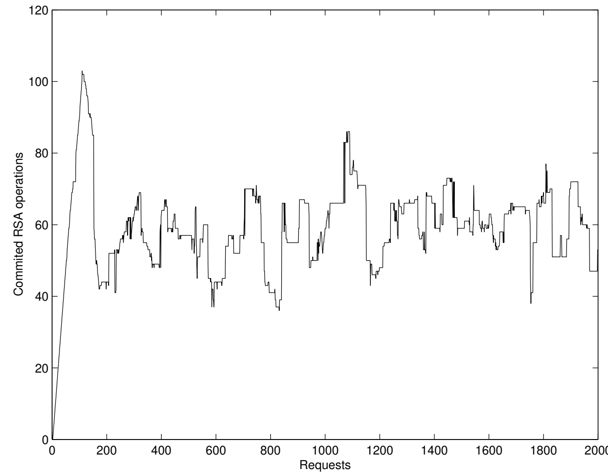

Figure 6: Committed RSA operations at each request without puzzles (during attack)

allows the server to decide whether to send a client puzzle on the current connection, and if so, how many bits the puzzle should be. This control flow is shown in Figure 4.

3.2.2 The Server

Our server implementation was based on the Apache webserver [1] with the `mod_ssl` module [4]. In Apache, every connection is handled by a different UNIX process. This makes sharing information between connections rather difficult. We needed to count the total number of RSA private key operations that all of the Apache processes had committed to in order to correctly determine when to send client puzzles. We used a page of shared memory, protected by file-based mutexes, to store the count of committed operations, and also whether a puzzle was sent on the last connection. This provided us with a reasonably simple and efficient implementation.

The server uses two user specified values to tell OpenSSL whether to send a client puzzle. One value is the maximum number of private key operations to commit to before sending puzzles. Once it starts sending puzzles it continues until the number of committed operations drops below the other specified value. We use separate thresholds for turning on and off puzzles to provide some hysteresis so that the server does not oscillate in and out of puzzle mode too rapidly. This process is outlined in Figure 5. The selection of these values is described in Section 4.

3.2.3 The Client

Integrating this scheme on the client side was surprisingly simple. It can be added to any client that supports TLS through OpenSSL simply by relinking with the new library. Initially we had planned to add a status message to alert the user when a puzzle was being completed, but the time needed to complete a puzzle is so short that this did not end up being necessary.

4 Analysis

In this section we analyze the effectiveness of our client puzzle protocol in protecting a webserver against a TLS based denial of service attack. To benchmark the server, we used a modified version of Dan Boneh's multi-threaded TLS benchmark. Our test server was an 750 MHz Athlon with 256 MB of RAM running FreeBSD 4.0. Using OpenSSL's RSA implementation and benchmarks, the server was able to complete 148 1024-bit private key operations a second.

4.1 Performance Without Client Puzzles

We first ran our benchmark on an copy of Apache version 1.3.12 with `mod_ssl` version 2.6.5 without support for the puzzle protocol. We did add some minimal profiling support to this build. Using one client and

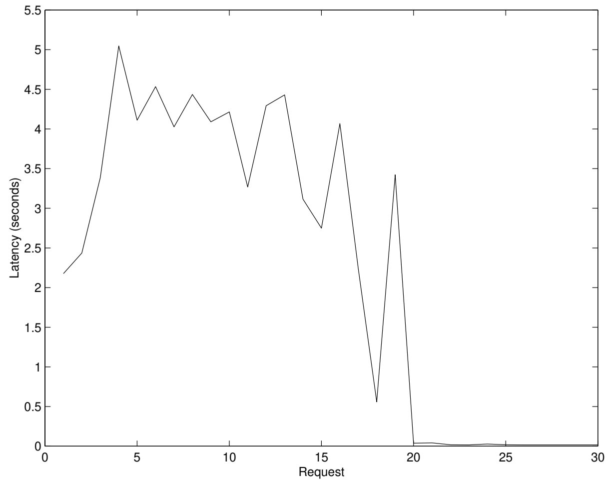

Figure 7: Latency for a legitimate client without puzzles (during and after attack).

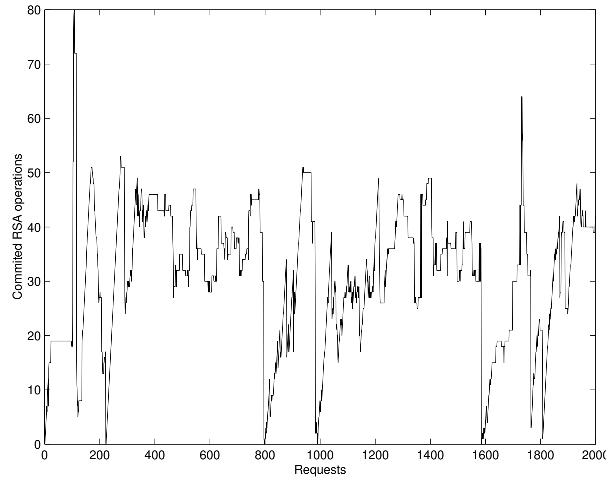

Figure 8: Committed RSA operations at each request with puzzles (during attack). Note the different scale from figure 6.

less than 550Kbps of traffic, we were able to completely load the server. As shown in Figure 6, the number of pending RSA operations was continually increasing for the first 100 TLS connections made to the server during a simulated attack. At this point, there were no more Apache processes available to handle additional requests, so the number of pending requests falls as RSA operations complete with no new operations being committed to, as clients are unable to make new connections to the server. Figure 7 shows the latency experienced by a legitimate user trying to connect to the server during this period during a representative benchmarked run. By using two attacking computers, we were able to double the latency experienced by the legitimate user. These simulated attacks can be continued indefinitely by the attacking computers. These results show that an unprotected TLS server is indeed vulnerable to these attacks.

4.2 Performance With Client Puzzles

The situation when using the client puzzle enabled version of Apache is much better. During the non-client puzzle tests, we observed that the loaded server was able to complete approximately 60 requests per second. We therefore decided to set the upper bounds on committed RSA operations to 40. That way, a client would have to wait no more that a second to connect. After that bound was reached, we decided to leave the system in puzzle mode until the number of committed RSA operations dropped below 10. This prevented the server from switching into and out of puzzle mode too often during a continuous attack. After testing different values for the length of the client puzzles to send, we settled on 20-bit puzzles, as a value that stopped even multiple attackers but did not disrupt client operations. A possible improvement to this scheme would be to increase the puzzle length depending on the length of time that an attack has been taking place. This would slow down attackers even further, but would also cause legitimate client wait times to increase.

Figure 9 shows the latency for a legitimate client during a denial of service attack waged against the modified server using multiple attackers. The number of pending RSA operations is shown in Figure 8. The server console remained usable throughout the attacks and its processor was never completely loaded. The extremely low, and neat constant, latency is the key metric. These results indicate that the puzzle protocol was effective in preventing the TLS based denial of service attack.

4.3 Security Considerations

Because the whole point of using TLS is to provide a secure connection between hosts, we look now to the security implications of our client puzzle protocol. We begin by noting that there are no shared keys between the client puzzle protocol and the handshake protocol. The client puzzle protocol merely "stops" the TLS handshake protocol until it completes. Because the handshake protocol is not time dependent (with the exception of a times-

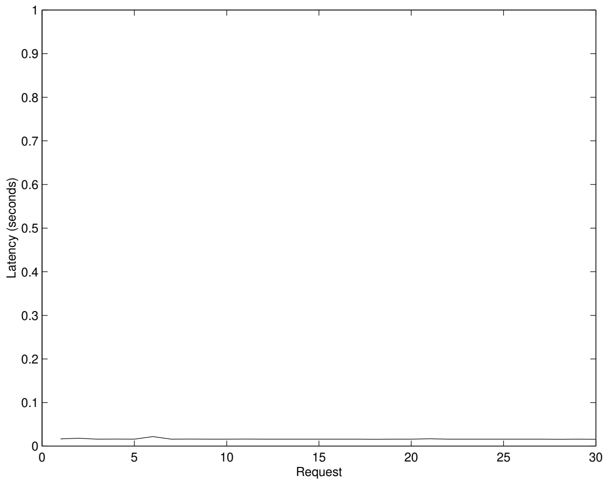

Figure 9: Latency for a legitimate client with puzzles (during and after attack). Note the different scale as compared to Figure 7.

tamp sent at the beginning of the protocol that does not even need to be correct), this does not seem to negatively affect security. The only possible problem is that the puzzle protocol is generating random data from the same pool as the handshake protocol. However, as long as the random number generator used in the TLS implementation is not poly-time distinguishable from true randomness, the client puzzle protocol should have no effect on the security of TLS. Of course, a TLS implementation that uses an insecure random number generator has much more serious security problems that are beyond the scope of this work.

5 Future Work

We have built a functional prototype and examined its behavior. While the system behaves well, further improvements could be made with more sophisticated strategies for determining when to start and stop the puzzle requests. A further degree of control is available by dynamically adjusting puzzle length depending on conditions at that moment on the server. Of course, a security proof for this scheme would be desirable.

6 Conclusions

Client puzzles are an effective means of countering a denial-of-service attack against TLS servers. We have presented an implementation that remains fully compatible with existing TLS clients when the server is not under attack. This is the best possible compatibility. We have shown that puzzle sizes can be chosen that keep the server available, even under duress, while adding latency below the humanly perceptible threshold for interactive response.

Acknowledgments

We thank Matt Franklin and Dan Boneh for useful discussion about this work. We thank Diana Smetters for helpful comments that improved the presentation of this paper.

References

[1] The Apache HTTP server project. `http://www.apache.org/httpd.html`.

[2] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. Dos-resistant authentication with client puzzles. In *Proceedings of the Cambridge Security Protocols Workshop 2000*, LNCS, Cambridge, UK, April 2000. Springer-Verlag.

[3] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Proc. CRYPTO 92*, pages 139–147. Springer-Verlag, 1992. Lecture Notes in Computer Science No. 740.

[4] Ralf S. Engelschall. mod_ssl: The Apache interface to OpenSSL. `http://www.modssl.org/`.

[5] Ralf S. Engelschall. Openssl: The open source toolkit for SSL/TLS. `http://www.openssl.org/`.

[6] Matthew K. Franklin and Dahlia Malkhi. Auditable metering with lightweight security. *Journal of Computer Security*, 6(4):237–255, 1998.

[7] Ari Juels and John Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In S. Kent, editor, *Proceedings of NDSS '99*, pages 151–165, 1999.

[8] R. C. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21:294–299, April 1978.

[9] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release cryptography. (Preliminary version posted on the web by Rivest.).