Hifn

Technical White Paper – February 2005

LZS Compression Benefits of TLSComp

2/05 © 2005, Hifn, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language in any form by any means without the written permission of Hifn, Inc.

Trademarks

Hifn®, FlowThrough™, and HIPP are registered trademarks of Hifn, Inc. Hifn™ and the Hifn logo are trademarks of Hifn, Inc.

All other trademarks and trade names are the property of their respective holders.

The Desire For Data Compression

In data communications it is desirable to have faster transfer rates at lower costs. Data compression addresses these demands by reducing the amount of data that must be transferred over a medium of fixed bandwidth, thus reducing the connection time. Data compression also reduces the media bandwidth required to transfer a fixed amount of data with a fixed quality of service, thus reducing the tariff on this service.

Transport Layer Security (TLS), a standards-based version of SSL, is used extensively to secure client-server connections on the World Wide Web. Although these connections can often be characterized as short-lived and exchanging relatively small amounts of data, TLS is also being used in environments where connections can be long-lived and the amount of data exchanged extends into millions of octets. For example, TLS is now increasingly being used as an alternative Virtual Private Network (VPN) connection. Compression services have long been associated with Internet Protocol Security (IPSec) and Point to Point Tunneling Protocol (PPTP) VPN connections, so extending compression services to TLS VPN connections preserves the user experience for any type of VPN connection. Now end users can benefit from compression at layer 2 (PPP), layer 3 (IP), and layer 5 (TLS), as illustrated in Figure 1.

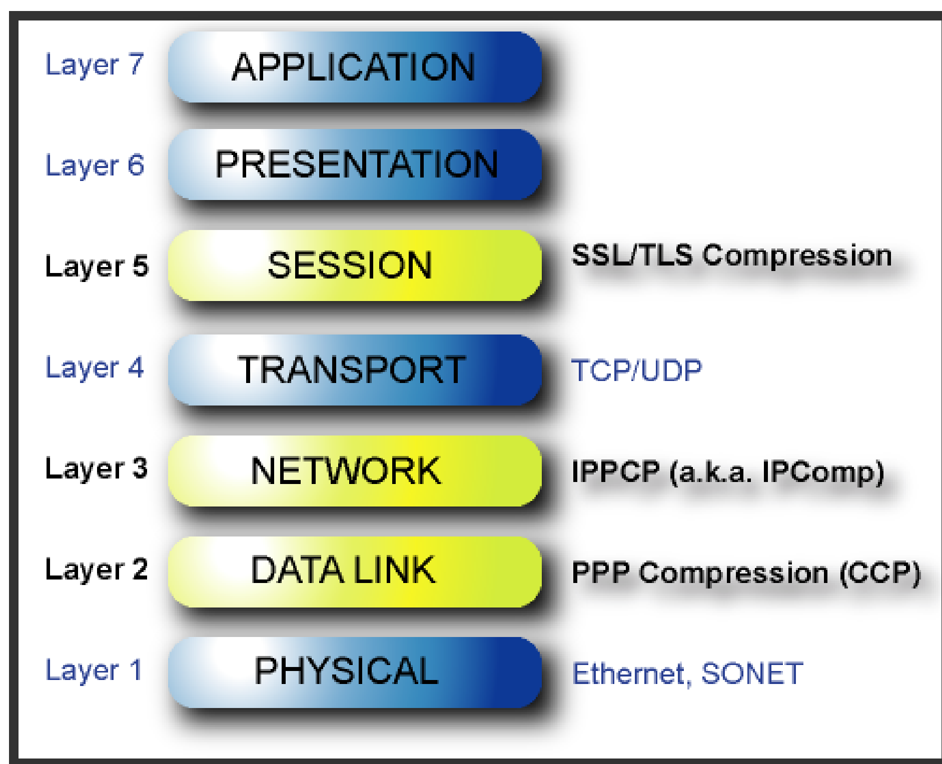

Figure 1. Comparing compression schemes in data networking

Compression within TLS is the most efficient technique to help reduce the bandwidth and latency requirements associated with exchanging large

amounts of data while preserving the security services provided by TLS. Since encrypted data cannot be compressed, TLS security defeats layer 2 and layer 3 compression, so there is a need for layer 5 compression.

The technical benefit of incorporating compression with TLS is the ability to compress all data in a TLS connection, whereas other protocols only compress certain types of browser content from the TLS server. Incorporating LZS (Lempel-Ziv Stac, an update of basic Lempel-Ziv algorithm) compression into security devices better enhances the end user experience along with protecting their data.

For example, Microsoft IIS v6.0 by default only supports HTTP compression for .HTM, .HTML, and .TXT file types on a web server. The web manager must manually configure other file types on every server. Furthermore, enabling HTTP compression uses more CPU resources, server memory, and disk storage, as the server must now compress the cached content and store it somewhere. In fact, the web manager is cautioned not to enable HTTP compression if the server's CPU load is already too high.

Conversely, TLS security using LZS compression is embedded in the hardware security accelerator that encrypts the client-server connection, which guarantees line speed performance, and requires no server CPU cycles, disk resources, nor memory resources.

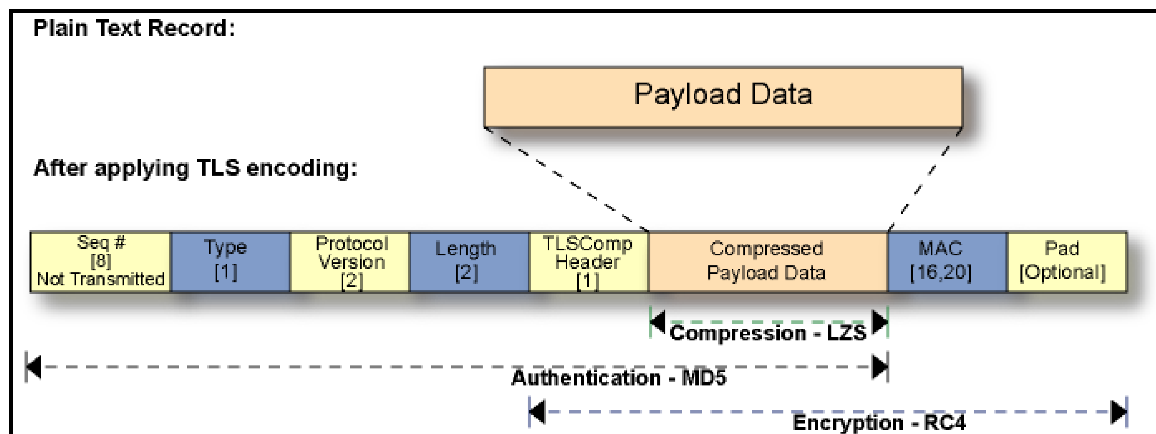

Figure 2. LZS compression with TLS security

Highly efficient compression software embedded in the client side utilizes minimal client CPU cycles to perform decompression operations. Compression software agents can be dynamically downloaded from TLS VPN servers to authenticated TLS clients.

All data that is secured by TLS in the client-server connection is also compressed, not just certain statically pre-defined file types. Furthermore, TLS compression using LZS is automatic and requires no interaction by users, IS or web managers to enable its use.

Provisions for data that does not compress is handled automatically by the TLS compression hardware on the server and software on the client.

Eliminating data expansion also preserves line speed with no server CPU cycles and minimal client CPU cycles.

LZS compression plays a vital role for data communication equipment manufacturers. Most security processors today contain high-speed compression engines that enable service providers the following benefits, which in turn, are passed on to the VPN user:

• Compressed packets consume less network equipment bandwidth

• Compression reduces fragmentation of records due to additional security headers, since payload length is decreased

• Line rate performance is significantly enhanced

The rest of this article details how to integrate LZS data compression in TLS VPN applications.

The Magic of LZS Compression

Data compression works by having a compressor at one end of the data link and a decompressor at the other end. For full-duplex communication there would be a compressor for the forward channel and a decompressor for the reverse channel at each end of the communication link. For TLS VPN applications, the server would have LZS compression built in the encryption accelerator and the client would have a downloadable software agent.

The LZS compression algorithm works by searching for redundant data strings in the raw input data stream and replacing these strings with encoded tokens of shorter length in the compressed output data stream. When the LZS algorithm finds a string in the raw input data that matches a previous string, it creates an encoded token that consists of a pointer to the previous data matched from the input stream. The encoded tokens then replace the redundant strings in compressed data stream. In this way future data is compressed based on previous data.

The compression engine maintains a “sliding window” compression history, which contains the last 2Kbytes of raw input data, as well as other data structures to accelerate the compression operation. As illustrated in Figure 3, the compression engine searches this history looking for string matches to create the tokens. Similarly, the decompression engine at the other end of the data link maintains an identical decompression history, which contains the last 2Kbytes of output data to which encoded tokens point.

Thus, the more redundant the data in the input stream the higher the compression ratio will be. Conversely, the more random the data in the stream the lower the compression ratio will be. This is why encrypted data does not compress. Compression ratio is defined as the number of input bytes divided by the number of compressed output bytes. The less output bytes the higher the compression ratio.

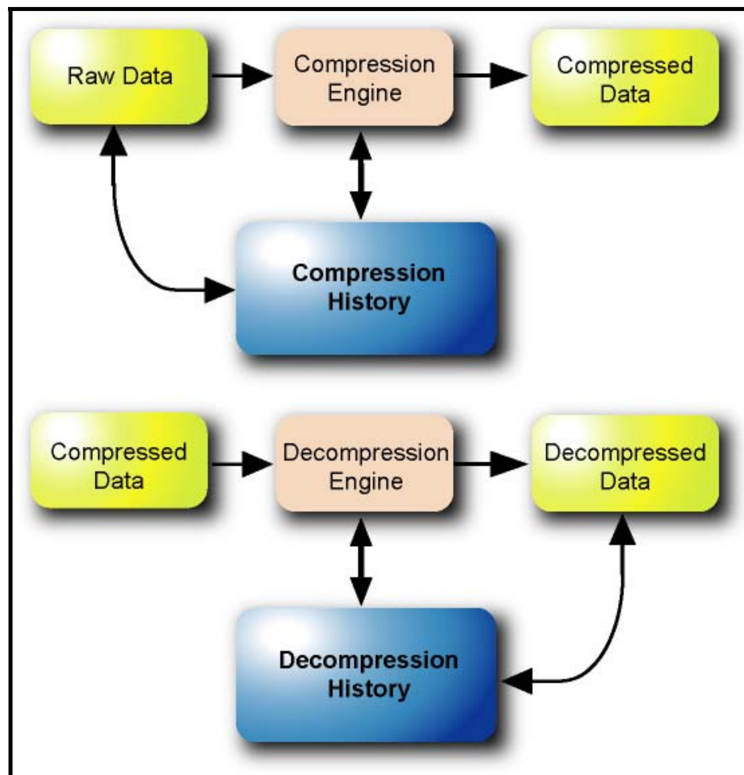

Figure 3. LZS compression & decompression process

The decompression engine uses its own 2Kbyte history to recreate the raw data pointed to by the encoded tokens created by the compression engine. The decompressor writes the decompressed output bytes back to the history, and this is how the decompression history is created & maintained at the receiver. Thus, the compression history is created at each end of the communication link, but never has to be transmitted. The compression and decompression histories at each end of the data link must always match, otherwise the decompressor may output garbage pointed to by the token.

The format of the compressed data stream consists of literal data and compressed tokens. Literal data are input data strings that could not be matched (compressed). Compressed tokens consist of an offset to a location in the compression history that contains the string match, and a length of number of bytes that match this string. An example would be "go back 150 bytes and output the subsequent 10 bytes". Thus, the tokens can be viewed as pointers into the 2 Kbyte compression history. At the beginning of a compressed record most encoded tokens are literals, until enough data is processed so that the input data stream starts matching the history content.

At the end of each compression operation, all compressed data is output and an LZS end marker is appended to the compressed data stream. The LZS end marker is a unique token used by the decompressor to find the end of the compressed data.

The LZS decompression algorithm works by taking the compressed data and performing one look-up for each compressed token. The decompressor reads

the offset, jumps to that location in decompression history and outputs the length of bytes starting at that offset.

Compression History Maintenance

In some applications, such as IP, packets are communicated over an "unreliable" media, which means packets can arrive out of order or possibly not at all. In this case the LZS history must be cleared for every packet. This means that the compressor and decompressor must start building a compression history from scratch for every packet.

However, TLS sessions are communicated over a reliable media, which guarantees in-order transport off every single byte between client and server. In this case, the LZS data compression algorithm allows the history information to be preserved between multiple compression operations, resulting in a higher compression ratio for each client-server connection.

A property of the LZS algorithm is that maintaining or clearing the history is a function entirely of the compressor. Under normal operating conditions the decompressor is unaware of whether the compressor maintained or cleared the history between successive TLS records. That is, the compressed data is "self-extracting".

Referring to Figure 4, if you consider the LZS encoded format of offsets and lengths, maintaining the history implies that there may be offsets pointing into previous records that have been compressed in this history (up to 2Kbytes back). Clearing the history between records implies that there will be no offsets pointing prior to the beginning of the current packet. Either way, the decompressor still reads the offset and outputs the number of bytes specified by the length.

```
<Compressed Stream> := [<Compressed String>] <End Marker>
   <Compressed String> := 0 <Raw Byte> | 1 <Compressed Bytes>
   <Raw Byte> := <b><b><b><b><b><b><b><b>          (8-bit byte)
   <Compressed Bytes> := <Offset> <Length>

   <Offset> := 1 <b><b><b><b><b><b><b> |           (7-bit offset)
               0 <b><b><b><b><b><b><b><b><b><b><b> (11-bit offset)
   <End Marker> := 110000000

<b> := 1 | 0

   <Length> :=
   00        = 2     1111 0110      = 14
   01        = 3     1111 0111      = 15
   10        = 4     1111 1000      = 16
   1100      = 5     1111 1001      = 17
   1101      = 6     1111 1010      = 18
   1110      = 7     1111 1011      = 19
   1111 0000 = 8     1111 1100      = 20
   1111 0001 = 9     1111 1101      = 21
   1111 0010 = 10    1111 1110      = 22
   1111 0011 = 11    1111 1111 0000 = 23
   1111 0100 = 12    1111 1111 0001 = 24
   1111 0101 = 13      ...
```

Figure 4. LZS data compression format

Because the decompressor may have references in its history prior to the beginning of the current record, it is important that when keeping history that the decompressor must decompress records in the same order they were compressed in. That is why a reliable media is required for keeping history. For, if successive packets were decompressed out of order, then an offset may point to the wrong data string and the resultant output will appear to be garbage.

Conversely, for lossy media such as IP, when clearing the history after each packet it does not matter which order packets are decompressed in a given history.

Multiple History Support

For TLS VPN servers that process multiple streams of data concurrently between many clients, it is important to note that each TLS VPN connection requires its own compression history in order to provide optimal compression ratio performance.

Alternatively, if the implementation were to utilize only one LZS history for all TLS sessions, then it must clear the history information after compressing each TLS record, degrading compression ratio for all compressed records.

To achieve maximum redundancy over multiple data streams, and therefore higher compression ratios in each data stream, the TLS VPN needs to associate separate histories in each client-server TLS context.

The cost of providing this efficiency is the amount of memory in the compressor and decompressor, which varies by implementation but can be as low as 4KB (8KB for software) for each full-duplex TLS session. In a

server, compression histories are stored in the security/compression device's local memory.

Handling Data Expansion

There are some kinds of data (e.g. random data) that will not compress. That is, under some conditions it is possible for the compressed data to be larger in size than the raw data. This is called data expansion. When maintaining histories, there are 3 possible solutions to resolve data expansion.

The first option is to simply transmit the expanded data and suffer the lost bandwidth (due to larger record size) when transmitting the current record. This preserves the potential compression ratio benefit for transmitting future record as the history is maintained. Fortunately, LZS can only expand to a maximum of 12.5% under worst-case conditions, if absolutely no raw input data could be compressed. In this case the transmission speed would be 12.5% slower than sending the raw data. However, this scenario simplifies the transmitter and receiver implementations. This option requires no communication between transmitter and receiver.

However, in some systems this lost bandwidth is unacceptable. And it is also possible that subsequent packets may not compress. The second option to deal with data expansion is to send the raw data and reset the compression history. This technique preserves the bandwidth of the current record transmission, but causes future record transmissions to suffer because the compression history will need to be rebuilt.

This second option needs special handling by the transmitter. The record processing layer of the transmitter must notify the receiver that this packet of data is not compressed, and the receiver must not decompress this packet of data. The TLS compression using LZS standard has the "compressed/uncompressed" (C/U) flag bit in the TLSComp header that communicates whether the record payload is compressed or not.

Since the compressor already compressed the data for this packet even though it did not transmit it, the decompressor's history would no longer be identical to the compressor's history, so the compressor's history must be cleared. The TLS compression using LZS standard has the "reset compression history" (RST) flag bit in the TLSComp header that communicates when the decompressor should clear its history.

The third option is to send the raw data and update the decompressor's history with the raw data, which is the most optimal of the three scenarios. This option preserves the bandwidth of both the current transmitted data (by sending the smaller of the data) and future transmitted data (by preserving the compression history).

In this case the transmitting record processing layer informs the receiver that the payload is uncompressed, but the history has not been cleared, using the C/U and RST header flags. The receiver will not decompress this record

payload. The LZS algorithm has provisions for updating the decompression history with the uncompressed TLS record payload.

This synchronizes the compressor's and decompressor's histories while maintaining the highest compression ratio. This feature is referred to as "anti-expansion".

It can be seen how the LZS compression algorithm is optimized not only for compressing data, but also for handling sub-optimal data that does not compress well.

Enhancing The User Experience

Data compression is now a vital ubiquitous technology, because it maintains the high-value user experience as remote users migrate from dial-up connections to PPTP to IPSec to TLS VPNs. LZS compression plays a vital role for data communication equipment manufacturers at communication protocol layers 2, 3, and now 5.

Hifn's high-performance security processors take advantage of this by integrating high-speed pipelined compression engines that provide compression along with security in one pass through the device. This enables service providers to deploy OEM security equipment that improves the remote user's VPN experience with performance and bandwidth benefits, such as, minimized network equipment bandwidth, reduced record fragmentation, and enhanced line rate performance.

It is important that the system implementer understand the basic principles of how compression technology works so it can be applied most effectively in a TLS VPN system. Maintaining multiple compression histories, associating compression histories with TLS sessions, and implementing anti-expansion mechanisms are vital to successfully implementing an optimal compression system in a TLS VPN solution.

The addition of LZS compression to the TLS protocol reflects the growing use of compression in all networking applications to improve data rates over standard protocols, including TLS, IPSec, PPP, Frame Relay, and IP storage networks, as well as non-standard compression applications, such as SONET and Fiber Channel networks. LZS data compression algorithm is an essential ingredient for manufacturers of VPN/firewall appliances, multi-protocol routers, remote access concentrators, web servers, server load balancers, and all other data communication devices. This why the LZS compression algorithm has been standardized by so many organizations, including the IETF (RFC3943, RFC 2395, RFC 1974, RFC 1967) Frame Relay Forum (FRF.9), ANSI (X3.241), and QIC (122).