Stealthy Network Transfer of Data

N Veerasamy and C J Cheyne

Council for Scientific and Industrial Research

Defence, Peace, Safety and Security

Pretoria, South Africa

nveerasamy@csir.co.za

***Abstract*—** Users of computer systems may often require the private transfer of messages/communications between parties across a network. However, unwanted interception/sniffing of such communications is also a possibility. An elementary stealthy transfer scheme is therefore proposed by the author. This scheme makes use of encoding, splitting of a message and the use of a hashing algorithm to verify the correctness of the reconstructed message. For this proof-of-concept purpose, the authors have experimented with the random sending of encoded parts of a message and the construction thereof to demonstrate how data can stealthily be transferred across a network so as to prevent the obvious retrieval of data.

***Keywords*—** construction, encode, interception, stealthy.

I. Introduction

NETWORKS are used to send communications between parties. A computer network is a set of computers using common protocols to communicate over connecting transmission media [1]. Often the need arises to send a message that should not be exposed to unauthorized parties. The confidentiality of the message/data needs to be maintained. Confidentiality is the prevention of unauthorized disclosure of information [2]. Sniffing and the interception of traffic would expose the contents of the message if the data is transported in the clear. Plaintext packets on a network can be captured which would reveal the message to the unauthorized party. Confidentiality, the protection of unauthorized disclosure of information, plays an important role in information security [3].

A communications message itself can be based on a number of languages, symbols, punctuation and other characters commonly used in day-to-day writing. Encoding systems were introduced as a means of processing, storing and transporting data irrespective of the language, software, application or protocols. Encoding is used to support network communications by providing a uniform way of representing data. Unicode is a character encoding system, like ASCII, designed to help developers who want to create software applications that work in any language in the world [4].

The rationale and functionality of encoding standards were therefore applied to the formulation of the stealthy network data transfer scheme. Developers use encoding to ensure that their code never needs adaptation, in that it is represented in a universally interpretable format. A classified message too should be represented in a universally understandable means without being easily observable.

Encoded characters are represented in a byte format and in this way the casual packet capturing of encoded data will appear like garbled letters to the untrained eye. Some skill would be required to firstly identify the need to decode the data and thereafter actually perform the operation. Simply detecting which bytes form a character is a complex contextually-dependant process [4]. The principle of encoding was therefore utilized to prevent the obvious reading of a secret message, were it to be intercepted. The plaintext version of the message would be initially converted into its encoded version before being transported (where it would again be encoded for transportation purposes). In addition, the message would be split to prevent easy observation if captured. The data would have to be combined in the correct order to be properly interpreted.

Therefore, further encoding was utilized for the original purpose that it was intended for- transportation and translation requirements. In essence, the data was therefore encoded twice, firstly to hide the contents of the message and thereafter to properly transmit the data.

In this proof-of-concept exercise, the authors wished to demonstrate the capability of using encoding, splitting, random sending of the fragments and reconstruction based on a hash algorithm verification so as to prevent the obvious reading of the secret message, were it to be intercepted.

II. Network Transfer Scheme

At a high-level, the scheme of stealthily transferring data across a network consisted of the sending of a split encoded message between two parties. For the sake of clarity the sending party is referred to as the Client and the receiving party the Server. The basic premise is to split a message into a number of parts, encode and randomly send them to a server and thereafter reconstruct the original message after having received the split portions. As the number of fragments increases, so does the time and complexity of the reconstruction process.

After having received the various split portions, the task involves decoding and placing the fragments in the correct order so as to rebuild the message. The solution provided to this problem involved the use of a hashing algorithm. A hash function converts plaintext into a message digest (MD) with fixed length [5]. Hashing is thus the application of a function that produces a unique key. Hashing is used in computer security to check for the integrity of data. Changes in files can be detected through the generation of different hash values. If the hashing function is initially calculated on a file, small changes carried out on the file, and the hashing function applied once again on the file, a different key will be generated. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest [6]. The implemented approach was therefore to calculate the hash on the original message and thereafter compute the hash on the various combinations of the received fragments until a match was found. This would confirm that the string combined in that particular order was the correct message.

The next two sections explain the implementation techniques on the two communicating parties' ends: the Client and the Server.

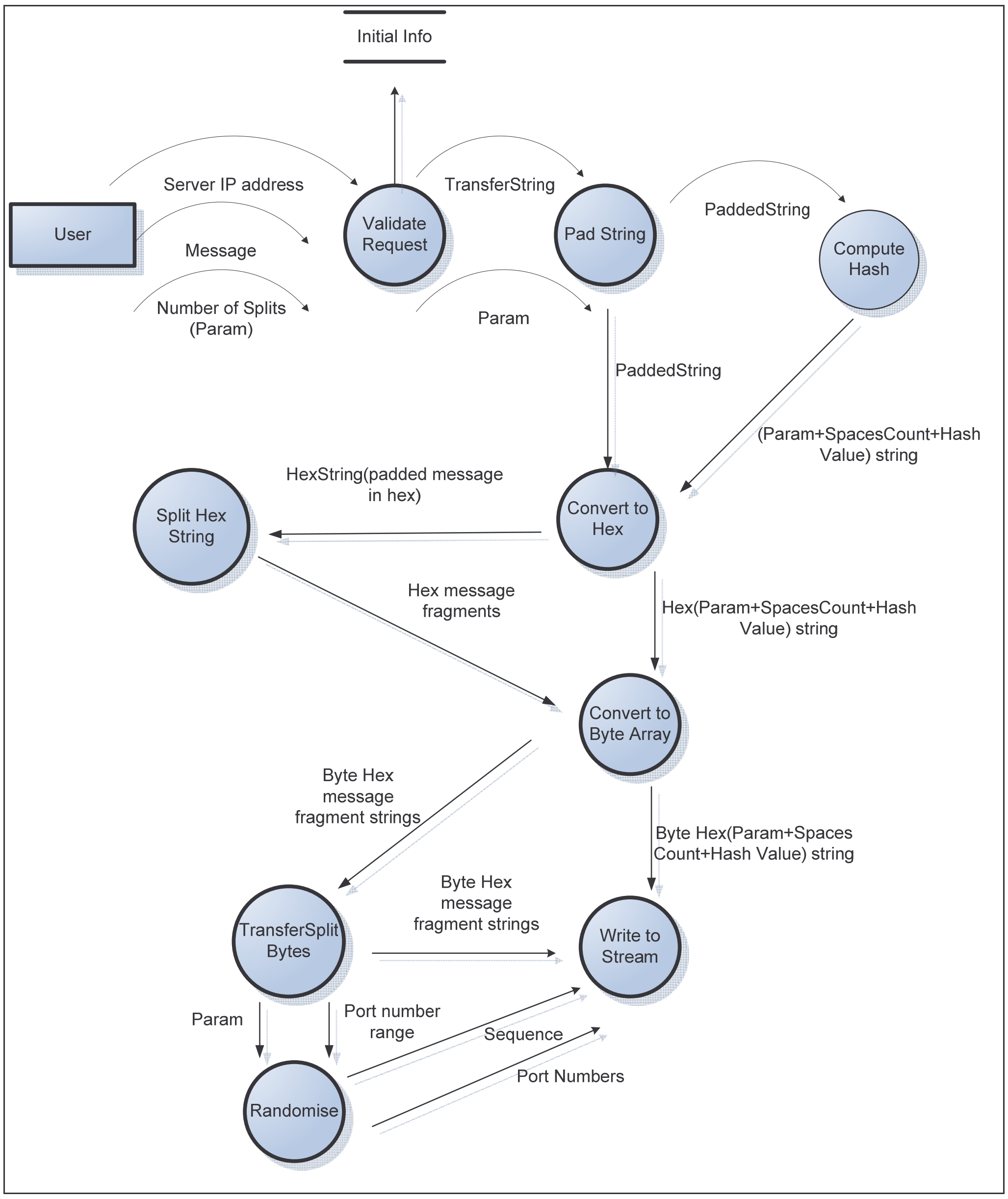

Figure 1: Process Model of Client

III. CLIENT

In this Section, a description of the user input, encoding and splitting processes follow.

A. Validation and Padding

Figure 1 shows the process diagram (data flow) on the client side whereby the initial message is specified and encoded. After the user specifies the message, server IP address and number of splits and the request is deemed to be valid (applicable server and legitimate integer value for number of splits), the next step involved padding the string, if necessary.

The motivation behind padding the string stemmed from the implications that if the string length was not a multiple of number of splits, different size fragments would be generated (with the possibility that the last fragment being the shortest). As with password cracking, if a shorter segment is identified, it can be used to crack the password, so too, if different length message fragments were sniffed, the shorter fragments could be analysed to identify the encoding scheme. All it takes is adding a few characters to the length of a (lowercase) password to make it just as effective as a password that uses a mix of characters [7].

Similarly, the same principle can be applied to secret message, and just as a password's effectiveness is heightened with increased length a secret message's when split into uniform fragments will too have better effectiveness. Therefore in an effort to ensure that all the message fragments were the same length, the original message was padded with the required number of spaces so as ensure the padded length of the message was a multiple of the number of splits. (Spaces were used as a convenient character to lengthen the message in this proof-of-concept exercise).

B. Compute Hash

Thereafter the (padded) string was written to a file on which the hash function is computed. Hash algorithms have crucial functions in security systems [5]. One such function is to verify the integrity of data. This property would be utilized in the scheme by initially computing the hash value of the original string and thereafter comparing the value to the hash value generated for each combination of fragments.

The hashing function md5deep was utilised. Md5deep was executed by running a batch file which started the md5deep executable with the required parameters (source and destination file.) After running the batch file, the md5deep hashing function also saves its output to a file. The output file contained the hash value and the source of the file on which the hashing function was calculated.

The hash value was obtained from the file by splitting the file into tokens and extracting the first token (which contains the hash value) into a string to be transported to the server at a later stage. (The computation of the hash value is carried out in a separate class so that both the Client and Server can utilise its functionality. Server will use the functionality to compute the hash value of the different combinations of joining the message fragments).

The values corresponding to the number of splits, number of padded spaces and hash value of the message, were combined into a single string that would also be encoded and sent to the server. In this way, all the message splits as well as the Param+SpacesCount+HashValue string would be transferred.

C. Convert to Hex

With the development of symbol technology and the requirement of information exchange, people wanted a unified character system to represent all of these characters—hence, Unicode [7].Therefore, the data would be converted into its hexadecimal Unicode equivalent (to hide its content) which would later be encoded once again when the data was transported.

The encoding process (of both the message and the Param+SpacesCount+HashValue string) consisted of a conversion to hexadecimal. (Built in C# functionality was used for this process – function name unicodeEncoding with "X2 " as a parameter to carry out the hexadecimal encoding). The result of the initial encoding was an array list containing the hexadecimal values of each letter.

D. Convert to Byte Array

As the aim of this research was to implement a basic stealthy network transfer scheme, the approach of simplifying transfer requirements was adopted. A socket transfer method which required the data to be contained in a byte array was utilized.

Because of the socket transfer requirements some additional conversions needed to take place. The hexadecimal representation of a string was contained in an array list. The data would be transferred to the server in a byte array. The array list with hexadecimal values was therefore converted to a byte array (initially read into a string and each value of the string converted into its byte representation and stored in a byte array.) In this way, the data was encoded into its hexadecimal equivalent and thereafter into the byte representation of the hexadecimal.

E. Split Hex String

The encoded message was thereafter split into the specified number of fragments. The message fragments would be sent through a set of determined ports in a random order (so as to ensure that the message fragments were not merely sent in their logical order for simple reconstruction). After formulating a random order and random set of ports that would be utilized from a predetermined pool (10 ports between 8000 and 8009 in this implementation), the data could be written to the stream and sent through to the server.

F. Write to Stream

The individual encoded strings constituting the message were thereafter sent through a TCP socket on the randomly calculated ports, where the order was based on the value assigned in the randomise computation. In addition, the hash value of the original message, the number of fragments and number of spaces the original message were padded with, was also encoded and sent to the server. Therefore the fragments as well as the hash and split information was transferred and awaited receipt on the Server side. The next section deals with the receipt and reconstruction of the secret message on the Server end.

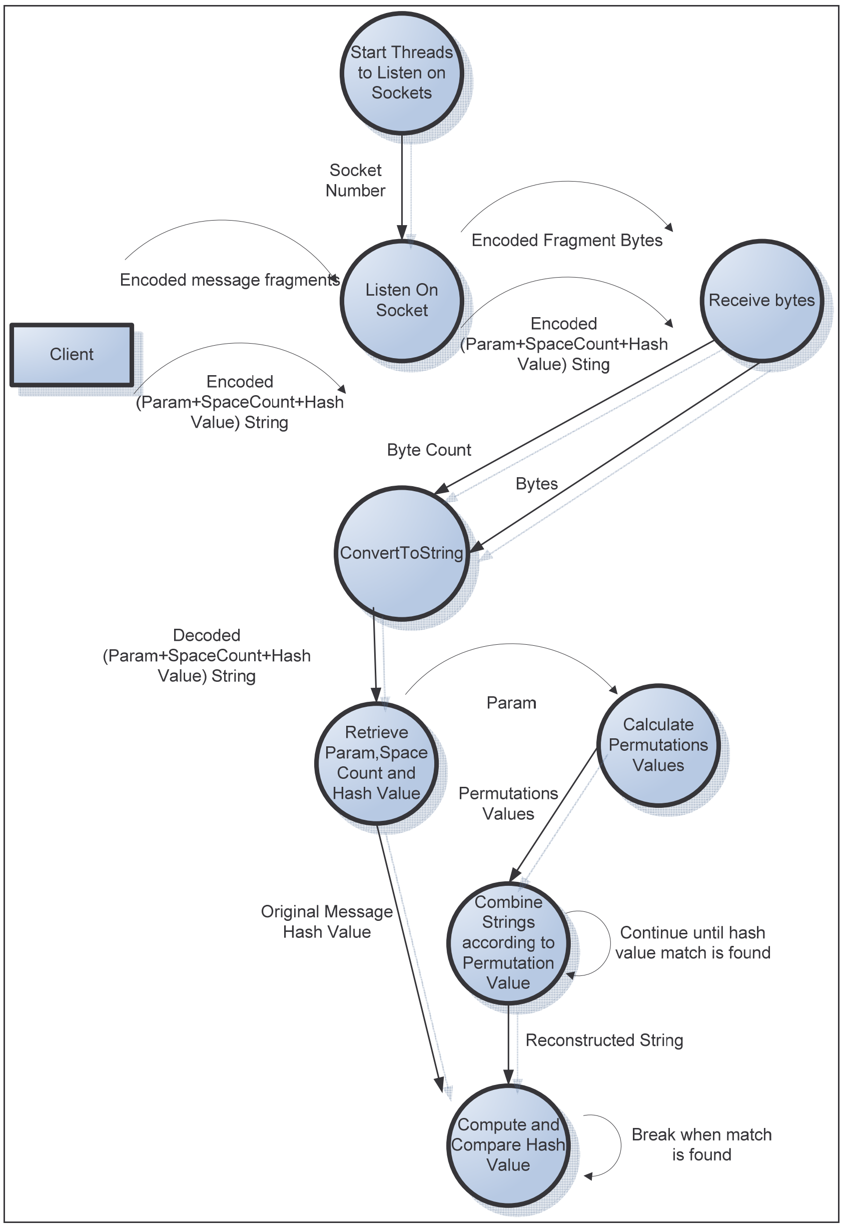
Start Threads to Listen on Sockets
Socket Number
Encoded Fragment Bytes
Encoded message fragments
Listen On Socket
Receive bytes
Client
Encoded (Param+SpaceCount+Hash Value) Sting
Encoded (Param+SpaceCount+Hash Value) String
Byte Count
Bytes
ConvertToString
Decoded (Param+SpaceCount+Hash Value) String
Param
Retrieve Param,Space Count and Hash Value
Calculate Permutations Values
Permutations Values
Original Message Hash Value
Combine Strings according to Permutation Value
Continue until hash value match is found
Reconstructed String
Compute and Compare Hash Value
Break when match is found

Figure 2: Process Model of Server

IV. SERVER

This section describes the processes for receipt and reconstruction on the Server side after the client has sent through the multiple fragments and other required information (number of spaces and parameter and hash values) The process diagram depicting the Server end functionality is given.

A. Start Threads and Listening on Socket

Figure 2 shows the process diagram (data flow) on the Server side. Initially, when the server is started up ten threads are initiated and listening for data on the set of predetermined ports commences. The client randomly sends through the encoded message fragments and the Param+SpaceCount+HashValue string using random ports in the predetermined set.

B. Retrieve Param, Space Count and Hash Value

After receiving the Param+SpacCount+HashValue string, the string is decoded. (ConvertToString process in Figure 2) This involves the conversion from the byte array to the hexadecimal representation of the data and thereafter its plaintext equivalent. After carrying out the convert to string process, the param, space count and hash values are extracted. The param value is (number of fragments) is used to determine when to stop listening on the socket. As data is received on each random port, the data is immediately decoded and stored. A Boolean value (running) is initially initiated as true and is used to continued listening on the socket (switch threads when necessary). In addition, a count is kept of the number of strings received. When this count reached the parameter value (number of fragments sent by client), the boolean value "running' is changed to false. This indicates to the Server to stop listening on the sockets (as all the fragments have been received) and the reconstruction can begin.

C. Calculate Permutations

The reconstruction process makes use of a Permute class. The open source logic from C# Crawler was implemented to compute the different permutations (all possible combinations of the fragments). The class makes use of binary functions (performs swapping) and produces a list of permutations depending on the number of elements in a string (number of permutations of elements in string corresponds to the number of splits). The Function takes a string of characters, and writes down every possible permutation of that exact string, so for example, if "ABC" has been supplied, should spill out:

ABC, ACB, BAC, BCA, CAB, CBA. [9]

The Permutation class was therefore used to calculate all the permutations from 0 to a defined maximum value. The maximum value was determined by the number of fragments received. The output of an input sting "012" was for example (012,021,102,120,210,201). This is equivalent to six permutations on three splits.

D. Combine Fragments According to Permutation Value

Thereafter, the individual stings were combined in the order stated in each permutation. For example if the original message was "Adam eats an apple" and strings received from the client (and stored in an array) were:

String 1:"ts an a" String 2:"Adam ea" String 3: "pple. "

The fragments would be combined in the order 012 which is equivalent to "ts an aAdam eapple. " (Make use of array indexing where 0 corresponds to string one and index 1 corresponds to string 2, etc.)

E. Compute and Compare Hashes

The Server, like the Client, also made use of the written Hash class. Using the string "ts an aAdam eapple. " as combined in the previous step, the Calculate function of the Hash class would write the string to a file and compute the hash value. The newly computed hash value would be extracted and compared to the hash value sent from the client. If the two hash values corresponded, the string was a match and the message was printed out. In this case, the hash values would obviously not match.

If the two hash values did not correspond, the reconstruction process would continue and the combination of strings would be computed in the order of the next permutation. For example, the next permutation was 021. The individual strings were therefore combined in the order "ts an a Adam eapple". This hash of this string would not match the hash sent by the client either. This process was therefore followed recursively until a match of the hash values was found. The permutation 120 would form the original message "Adam eats an apple." and the hash generated on this combination would match the hash from the client.

It should be noted that the number of permutations increases exponentially as the number of fragments increases. The reconstruction process would therefore combine all the fragments in various orders based on the values obtained from permutation calculation process. Due to the randomness of the fragment transfer and the substantial increase in permutations as the fragments increase, the processing time of the reconstruction can differ during each run.

Finally to ensure the integrity of the data, the number of spaces appended to the end of the message was removed (if any spaces were originally appended to the string). (Spaces were added to ensure all the message splits were the same size.) The removal of the spaces was performed by using the substring function (position was determined based on the value that was originally sent from client). The original message sent from the client is therefore obtained through a process of decoding, and cyclically combining, computing and checking.

V. Conclusion

In some instances the confidentiality and integrity of data is of the utmost importance. Therefore a stealthy network transfer scheme has been devised to prevent the easy viewing of the data were it to be intercepted. The scheme aims to ensure that the transferred data is protected.

The data is initially encoded and split into a number of equal sized chunks before being randomly sent over a set of ports to the receiving end. The described process (that makes use of encoding, message splitting, randomized transfer and hash value matching) explains the method of obscuring the data so as to hide its contents to intruders, but reveal the message to the authorized party.

In this way, the scheme aims to prevent the simple sniffing and acquisition of a secret message. An intruder would have to determine that the data being transferred has been encoded twice and correctly reconstruct the fragments of the data. In this proof-of-concept exercise the authors have implemented a simple but effective stealthy network transfer scheme.

VI. References

[1] J.S. Quartermaan, and J. C. Hoskins, "Notable Computer Networks," Communications of the ACM, vol.29, pp 932-971, Oct. 1986.

[2] D. Gollman, "Computer Security," New York: John Wiley & Sons, 2002, pp 5-7.

[3] S. Gürses, J.H. Jahnke, C. Obrey, A. Onabajo, and T. Santen, "Eliciting confidentiality requirements in practice, " in *Proc. 2005 conference of the Centre for Advanced Studies on Collaborative research*, Ontario, 2005, pp. 101-116.

[4] Unicode.org, "Basic Questions", Available: http://www.unicode.org/faq/basic.html.
[5] M. Wang, C. Su, and C. Wu, "A HMAC Processor with Integrated SHA-1 and MD5 Algorithms," in *Proc. Of the 2004 conference on Asia South Pacific design automation: electronic design and solution fair ASP-DAC 2004*, pp. 456-458.
[6] R.L. Rivest, "The MD5 message digest algorithm," RFC 1321, the Internet Society, April 1992.
[7] M. Burnett, and D. Kleiman, "Perfect Passwords", Rockland: Syngress, 2006, pp. 53-59.
[8] A. Y. Fu, X. Deng, L. Wenyin, and G. Little, "The Methodology and Application to Fight Against Unicode Attacks," in *Proc. of the second symposium on Usable privacy and security*, Pennsylvania, pp 91-101.
[9] S. Hadaya, "Permutations", C# Crawler, Available: http://radio.weblogs.com/0111551/stories/2002/10/14/permutations.html .

Namosha Veerasamy has obtained a BSc: IT Computer Science degree and a BSc. Computer Science (Hons) degree with distinction from the University of Pretoria. She is currently completing her Masters in Computer Science and is employed as a researcher at the Council for Scientific and Industrial Research (CSIR) in Pretoria.

Corien Cheyne has obtained a BSc: Computer Science degree from the University of Pretoria. She is currently studying towards a BSc Computer Science (Hons) degree, and is employed as a researcher at the Council for Scientific and Industrial Research (CSIR) in Pretoria.