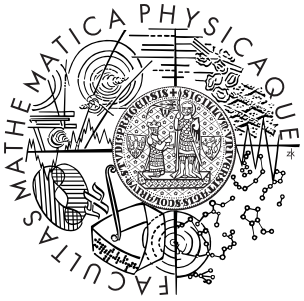
Charles University in Prague, Czech Republic
Faculty of Mathematics and Physics

BACHELOR THESIS



Szabolcs Gróf

STalk – zabezpečená aplikace pro komunikaci
STalk – Secure Communication Application

Network and Labs Management Center – SISAL

Supervisor: RNDr. Libor Forst

Study Program: Computer Science
Field of Study: Programming

2008

I would like to give thanks to my supervisor, RNDr. Libor Forst, for his time and support, and also to my mother for her patience and support.

I hereby certify that I wrote the thesis by myself, using only the referenced sources. I agree with lending the thesis.

In Prague, May 30, 2008 Szabolcs Gróf

Contents

1 Introduction **6**

2 TCP/IP protocol suite, the weaknesses of TCP and the need for cryptography **8**
2.1 Introduction 8
2.2 Layering 9
2.3 TCP: Transmission Control Protocol 10

3 Cryptographic algorithms **17**
3.1 Introduction 17
3.2 DES – Data Encryption Standard 17
3.3 RSA – Rivest-Shamir-Adleman public-key encryption 18
3.4 Probabilistic primality test 18

4 Architecture **20**
4.1 Connection and communication architecture - Talk vs. STalk 20
4.2 Security architecture of STalk 22

5 STalk – implementation **24**
5.1 `stalkd` – STalk server 24
5.2 `stalk` – STalk client 27
5.3 `log_users` – Tell the server, which users to log 29
5.4 `miller_rabin` – The Miller-Rabin probabilistic primality test implementation 29
5.5 `rsa_generate_keys` – Generating the RSA key pair 29

6 User's guide **31**
6.1 `stalkd` 31
6.2 `stalk` 32

6.3 `log_users` 34
6.4 `rsa_generate_keys` 34

7 Conclusion **35**

Bibliography **37**

Název práce: STalk – zabezpečená aplikace pro komunikaci
Autor: Szabolcs Gróf
Katedra: Středisko informatické sítě a laboratoří – SISAL
Vedoucí bakalářské práce: RNDr. Libor Forst
e-mail vedoucího: `forst@sisal.mff.cuni.cz`

Abstrakt: V předložené práci studujeme metody použití kryptografických algoritmů pro zajistění bezpečnosti komunikace v počítačových sítích a přinést tak nový pohled na klasický UNIXovský konzolový program `talk`. Práce na začátku popisuje základní přehled rodiny protokolů TCP/IP a představuje stručný kryptografický přehled. Další část popisuje architekturu `stalku`. Na konec je popsaná implementace a použití programu.

Klíčová slova: síťové programování, kryptografie, TCP/IP, RSA, DES, talk

Title: STalk – Secure Communication Application
Author: Szabolcs Gróf
Department: Network and Labs Management Center – SISAL
Supervisor: RNDr. Libor Forst
Supervisor's e-mail address: `forst@sisal.mff.cuni.cz`

Abstract: In the present work we study the methods of using cryptographic algorithms to ensure the security of communication over computer networks and give a new perspective on the classic UNIX console chat program called `talk`. The thesis starts by describing the various protocols of the TCP/IP protocol suite, and a brief cryptographic overview. The following part describes the architecture of `stalk`. Finally, the program's implementation and usage is described.

Keywords: network programming, cryptography, TCP/IP, RSA, DES, talk

Chapter 1

Introduction

Communication is a very important aspect of our everyday life. As information technologies and programming technologies improve, newer and newer forms of communications are developed. One of the most common ways of communication is through a computer network. Under the UNIX operating system, one of the oldest programs that ensure communication between users that are logged on to the computer from a terminal is `talk`.

This thesis will discuss a new approach of designing an application for UNIX console, based on the classical `talk` utility. Although numerous chat programs exist around the world with more and more beautiful and user-friendly interfaces, there are many people who feel comfortable in front of the good, old console. This thesis' goal is to help these people.

`Stalk` will also process characters immediately, and will use a split screen console window for communication, just like the classical `talk`. However, there are many things that differ in `stalk` and the classical `talk`.

`Stalk` is based on the client-server infrastructure. There is a server, which accepts connections from clients, and then sends the list of connected users to each client. The client then may choose, which other user he wants to talk to. The transport protocol, which it uses, is TCP over IP network protocol.

Since today security gets greater and greater attention - because of hackers and crackers who never stop trying to lay their hands on information and property that does not belong to them, - `stalk` uses cryptographic protocols to minimize the chance of any unwanted interception of the communication. It makes effort to implement the RSA algorithm, which is used to exchange secret keys between the client and the server, and also the DES algorithm for communication itself.

Chapter 2

TCP/IP protocol suite, the weaknesses of TCP and the need for cryptography

2.1 Introduction

The TCP/IP protocol suite was created in order to have a common protocol, which could be used in computer networks for communication. It uses the IP protocol as network protocol, which has the advantage of running over any kind of link protocol (ethernet, ATM, etc.). Nowadays, TCP/IP is used widely across the world, for ordinary, small local area networks (LANs), as well as for large, wide area networks (WANs), and the largest network on the world, the Internet, also uses this protocol suite to communicate.

One of the advantages of TCP/IP is that its specifications come in the form of RFCs (request for comments), which are widely accessible to anyone, and anyone can write his/her own RFC. If the TCP/IP community finds an RFC's content useful and implementable, it will be accepted, and incorporated into the TCP/IP protocol suite. This way is ensured that even though the specifications are open for anyone, the protocol suite itself will only use those technologies that are proven to work well.

A brief overview of some of the protocol suite's elements will follow (based on [3]), for complete overview, see [3] or the RFCs.

2.2 Layering

Networking protocols are usually built up from layers. This way it is ensured that the tasks that are needed to be accomplished in order for the information to be transmitted are divided up among the layers, which makes it easier to implement them separately, and makes the whole system more effective. However, while the ISO/OSI networking model (which is rarely used, but is a very good example to understand the layering techniques) has 7 layers, the TCP/IP protocol suite has only 4 layers. They are shown in the table below.

Layer	**Example**
Application	Telnet, FTP, e-mail, etc.
Transport	TCP, UDP
Network	IP, ICMP, IGMP
Link	device driver and interface card

Each layer has a unique responsibility.

1. The link layer, sometimes called the data-link layer or network interface layer, is responsible for transferring information between two computers that are directly connected. TCP/IP doesn't specify this layer's protocols in any way, instead it attempts to design the rest of the layers (mainly the network layer and IP) to run over any kind of link layer protocol as mentioned in the introduction. This feature of the IP network protocol is often called *IP over all*.

2. The network layer (sometimes called the internet layer) is the one that ensures that data is transmitted between two computers that can be anywhere in the computer network. It has to identify the computers, and find the best path to deliver the information. In TCP/IP, the IP protocol does most of this job. It is an unconnected, unreliable protocol whose purpose is to transmit data as fast as possible even if some of it will be lost. This method is called "best effort." Reliability and connectivity, in case of need, has to be ensured by the layers above. IP uses packets to transmit data. The ICMP and IGMP protocols help the IP's work.

3. The transport layer provides a flow of data between two hosts, for the application layer above. In the TCP/IP protocol suite there are two vastly different transport protocols: TCP (Transmission Control Protocol) and UDP (User Datagram Protocol).
   TCP is a reliable and connected transport protocol. It ensures that a connection is established between two hosts, and used for transmitting data, and ensures that the data arrives in the correct order and without any damage. It is a rather robust protocol that incorporates many features to achieve reliability and connectivity. It will be described a little more in detail later on since this is the transport protocol that is used by `stalk`. TCP uses segments to transmit data.
   UDP, on the other hand, is a much simpler transport protocol. It uses datagrams to transmit data, it is unreliable, unconnected and can pretty much be considered as a simple superstructure over IP.
   Each of the applications that use TCP/IP may choose which transport protocol is suitable for him. Some may need the services of TCP, but for some it might unnecessarily complicate the communication.

4. The application layer handles the details of the particular application.

2.3 TCP: Transmission Control Protocol

It is very important to understand the concepts of TCP to be able to see the weaknesses behind this mechanism, which result in the need of applying cryptography in case of `stalk`. This is the reason why a bit more in-detail description of this protocol will follow here. Concrete weaknesses and their misuses are also discussed below.

TCP (just like every other protocol in the TCP/IP protocol suite) uses the IP network protocol. However, IP packets can be dropped on their way to the receiver so, in order for TCP to be reliable, it has to provide a way to monitor data flow, and resend information that has been lost.

To achieve connection orientedness, TCP has to ensure that a connection is established, communication is realized with information flowing in the correct order through the connected channel and that the connection is terminated at the end since IP does not provide any of these features.

TCP connections are one-to-one, which means that there have to be ex-

actly two hosts that connect to each other. One TCP connection cannot be made between more hosts.

TCP uses a mechanism that is called *byte stream*. It means that data is sent through a connection without any markers where the border between possible data sections are. If the sender sends 20 bytes then another 20 bytes and finally 10 bytes of data, the receiver may read it as 30 plus 20 bytes, or in any other sized pieces. That is why the two connected hosts have to be careful while reading data from a TCP connection because although the order of the bytes received is ensured to be correct, whether the bytes were sent all at once or separately (and where the borders are) are not.

Byte stream also means that bytes sent through the TCP connection are not interpreted in any way. Anything can be encoded in them, the only rule that TCP has is that it sends the encoded data as 8 bit atoms, which is not further examined. The interpretation of each byte is left to the processes that are part of the given connection.

Now, let us look at the TCP segment format a bit closer. Figure 1 shows its format. It contains a lot of information about the source host, the destination host and the connection itself.

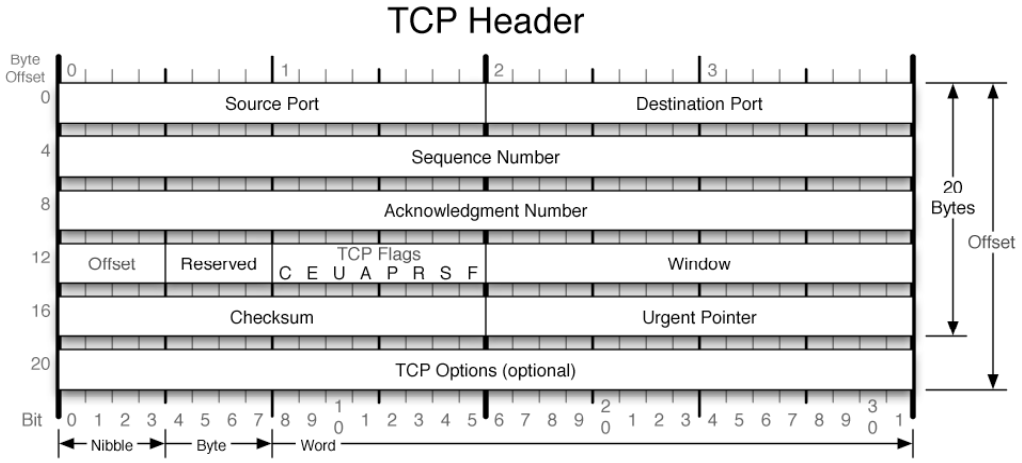


Figure 1.: TCP header format

1. 16-bit source port and 16-bit destination port numbers – they are very important because together with the source and destination IP address, they identify a TCP connection. These ports are bound to so called *sockets*, which are implemented as a programming interface for communicating over TCP/IP. The concept of these socket interfaces is that the programmer can write to and read from them, as he/she

would from/to a file, and the data (instead of being written to the hard disk) will be sent to the second communicating host.

2. 32-bit sequence number – it is important to identify the order of the segments in order to ensure connection orientedness since IP does not provide this. Sequence numbers help the receiver to identify in what order the segments were sent, and reorder the segments before passing them to the requesting process (to ensure the byte stream techniques), no matter what the original arrival order was.

3. 32-bit acknowledgement number – it is important to tell the other side which segments have arrived so that it would know which ones to resend to ensure reliability. This is done by sending an acknowledge number, which contains the number of the segment that has arrived last (while all the others preceding this segment had also arrived) + 1. Acknowledgement number may also be understood as the sequence number of the next expected segment.

4. 4-bit header length.

5. 6 bits reserved.

6. TCP flags – they are important because they give information to the other communicating host about the TCP connection. These flags are:

   (a) *URG* – "the urgent pointer is valid, the content of the Urgent header field points to the data that the receiver would like to expedite." [5]

   (b) *SYN* – synchronize sequence numbers to initiate a connection.

   (c) *ACK* – the acknowledgement number is valid.

   (d) *PSH* – the segment is to be passed to the process that it is addressed to immediately after arrival.

   (e) *FIN* – transmission complete, no more data will be sent in this direction through this connection.

   (f) *RST* – cancel/reset connection.

7. Window size – it is a 16-bit number that specifies the number of bytes that the receiver is willing to accept. Its main goal is to ensure that a fast host would not send too much information to a slow receiver. If

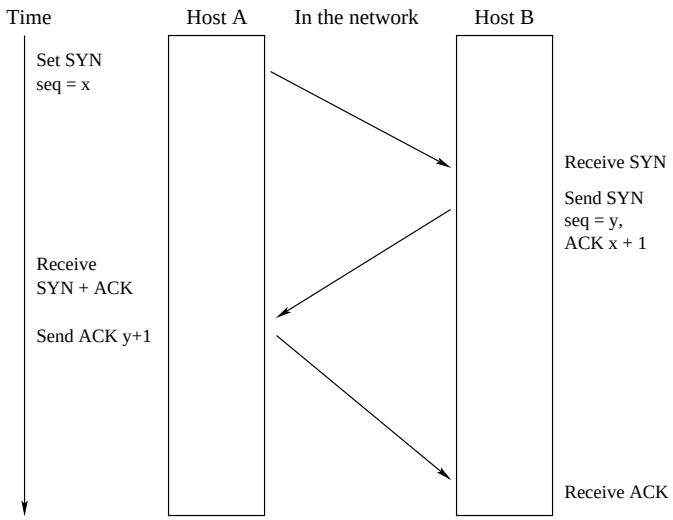
the receiver processes the segments slower then they arrive, after the overflow of the receiver's buffer many of the segments would have to be dropped and resent, causing larger traffic in the network.

8. 16-bit TCP checksum – it covers the whole TCP segments, and ensures that it is not damaged. If the checksum is incorrect, the segment is dropped so that the sender would time out and resend it.

9. Urgent pointer – "it is valid only if the URG flag is set. This pointer is a positive offset that must be added to the sequence number field of the segment to yield the sequence number of the last byte of urgent data. TCP's urgent mode is a way for the sender to transmit emergency data to the other end." [3]

10. Options – for example the MSS, the maximum segment size, etc.

11. The data itself



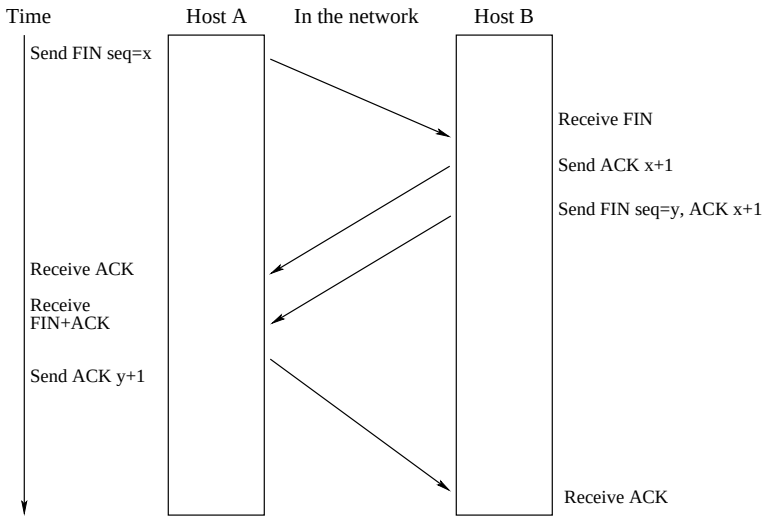


Figure 2.: TCP connection establishment and termination

TCP connection establishment, communication, and connection termination

TCP connections are established using the so called *three-way handshake*. Its steps are as follows:

1. The client sends a segment to the server with the SYN flag turned on and the client's initial sequence number set to a random number between 0 and $2^{32} - 1$.

2. The server sends a segment to the client with the ACK flag turned on to acknowledge the previous received segment, acknowledgement number set to the client's initial sequence number + 1, the SYN flag turned on and the server's initial sequence number set to a random number between 0 and $2^{32} - 1$.

3. The client sends a segment to the server with the ACK flag turned on to acknowledge the previous received segment, and the acknowledgement number set to the server's initial sequence number + 1.

This way both the client and the server know that a connection has been set up, and they can start sending data to each other. Each time a new segment is sent, the sequence number of the sender is increased by one, and included in the segment so that the receiver could identify the order of the segments even if they don't come in the right order. The acknowledge number is always set to the number of the next expected segment.

A window is also used, which means that only certain amount of segments are sent without acknowledgements. If acknowledgements are received, the window is slided over one by one to the next segments, until there is no unacknowledged segment in it. If a segment arrives that has a sequence number that is not inside the "window" the segment is dropped, and a correction acknowledgement segment is generated with the expected segment number.

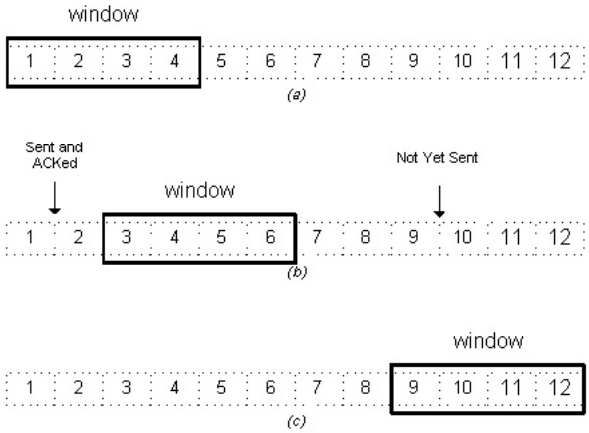


Figure 3.: TCP window mechanism

Closing a TCP connection takes four steps (two steps to close each direction):

1. The closing side sends the next segment with the FIN flag turned on and the sequence number and acknowledge number set as usual.

2. The other side sends back a segment with the ACK flag set and the acknowledge number set as usual.

3. The other side sends a segment with the FIN flag turned on and the sequence number and acknowledge number set as usual.

4. The closing side sends back a segment with the ACK flag set and the acknowledge number set usual.

A connection can be closed (or resetted) by setting the RST flag as well. This way there won't be any acknowledgement or any other segments sent back.

A scheme of all these steps is shown by figure 2.

Weaknesses of the TCP protocol

There are several weaknesses of the TCP protocol that can be exploited by a hacker. There is a lot more detail of them in [5].

1. After a session is authenticated, TCP segments come and go without any further examination.

2. The larger the window size is, the easier for someone to guess a sequence number that fits into the window and send a RST segment with that sequence number.

3. If someone knows the next sequence number to come, it can use it to send segments to the other side, which causes the other side to expect segments with sequence numbers from a different range than the sender is about to use. This is called *desynchronization*.

4. If someone desynchronizes both of the communicating sides, they start sending each other *resynchronizing segments*, which contain the expected sequence number of the packets coming from the other side, and the usual sequence number. The other side sees that the sequence number is not what he expected so he sends a resynchronizing segment to the sending side, and this goes on and on until one of these segments is lost. It is called the *TCP ACK storm*, which can take up a lot of network capacity and bandwidth.

These weaknesses call for security mechanisms that ensure that the correct data went from the correct source to the correct destination. The best solution is applying cryptographic algorithms; that is what is implemented in `stalk` as well.

Chapter 3

Cryptographic algorithms

3.1 Introduction

"Cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication" (definition from [2]).

Cryptographic protocols include a huge number of definitions and algorithms, and since `stalk` only implements three of them, only these (DES, RSA, Miller-Rabin) will be briefly introduced. For further detail on these or on other cryptographic algorithms, see [2].

3.2 DES – Data Encryption Standard

For encrypting communication, `stalk` uses the DES symmetric-key algorithm, which is the most well-known symmetric-key block cipher. "Recognized world-wide, it set a precedent in the mid 1970s as the first commercial-grade modern algorithm with openly and fully specified implementation details. It is defined by the American standard FIPS 46-2." [2]

Since it is a simmetric-key algorithm, both communicating sides use the same encryption and decryption keys, which can be easily derived from each other. The key length is 64 bits, of which 16 48 bit round keys are extracted for encryption and decryption.

Being a block cipher, DES encrypts 64 bit plain text at once, and maps it to 64 bit cipher text.

A DES has four weak keys (which means that by encrypting the plain text twice using the encryption key, we get back the plain text itself) and six pairs of semi-weak keys (which means that by encrypting the text first with the first key and then with the second, we get back the plain text). `Stalk` checks the generated DES keys so that none of these weak or semi-weak keys are generated.

3.3 RSA – Rivest-Shamir-Adleman public-key encryption

The RSA cryptosystem, named after its inventors R. Rivest, A. Shamir, and L. Adleman, is the most widely used public-key cryptosystem. Since it is an asymetric key system, each receiving side has to have a public key, that is distributed to everyone and a private key that is known only to the given side.

The RSA is a fairly simple algorithm, but in order to get the desired security out of it, we need long keys ($768 <$ bits), and in order to generate these keys, we need half as long prime numbers. Often it is not trivial to decide whether a number of such length is a prime or not. This is why probabilistic primality tests have been developed.

3.4 Probabilistic primality test

Since the RSA algorithm needs prime numbers, and these prime numbers need to be very large ($768 <$ bits), we cannot just choose a number and check all the numbers smaller than its square root whether their $gcd = 1$ or not because it would take too long. Instead, in practice, probabilistic tests are used, which will answer the question "Is prime?" correctly with probability p. If a test like this says that a number is not a prime, we can be certain that the number is not prime. On the other hand, if the test announces "yes," we have a $1 - p$ probability that it is wrong. However, if we can maximize this probability, we can minimize the chance of false answer.

There are three well-known probabilistic primality test algorithms: Fermat test, Solovay-Strassen test, Miller-Rabin test. Since Miller-Rabin test

is the strongest of these three (which means that if Miller-Rabin declares a number to be prime then the other two would do so as well), the Miller-Rabin test is implemented in `stalk`. (For further detail on probabilistic primality tests, see [2])

Chapter 4

Architecture

4.1 Connection and communication architecture - Talk vs. STalk

Although `stalk` is very similar to `talk` as far as the visualization of the communication is concerned, the communication and the invitation itself use completely different approach then the one used in `talk`.

First, and foremost, `talk` uses UDP for communication, while TCP was selected for `stalk`. The reason for this is that `stalk`'s protocol is strongly state oriented, which means that there cannot be sent any type of data at any time. While this applies to `talk` as well, it is not as fundamental as in `stalk`, and it is solved using different ports for invitation and for the communication itself.

The reason why `talk` uses UDP is that the way it sets up communication between clients is based on messages. The caller sends a message to the callee, telling him about the callers request to talk. To receive the request, there must be someone listening on the other side on the given port. It is solved by using an inet daemon that listens on the given port, and starts the talk daemon if there is a request coming on that port. The talk daemon then notifies the callee, and he can decide whether to accept the invitation or not. After that, inet daemon and talk daemon are no longer required for the communication itself (Figure 4.).

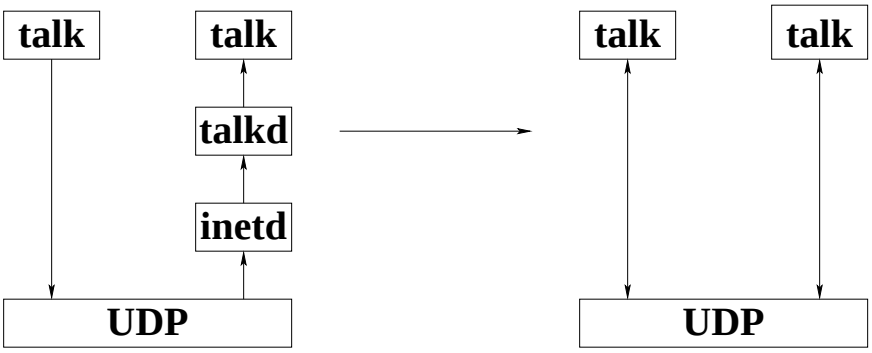


Figure 4.: `Talk` architecture: invitation and communication

`Stalk`, however, uses a completely different approach to the setup of the communication as well as to communication itself. Since it is not always obvious, where the user that we are looking for is situated - besides he can sit in front of a computer which is behind NAT - it can be much more effective to have a central server that accepts clients and sends the connected clients' list back. The given client then can select from the list of other clients, whom to talk to. Everything is going through the server, so not only the selection of the party, but also the communication itself (Figure 5.). This approach has several reasons:

1. One of them is the above mentioned NAT problem - the clients may be behind NAT and not be able to connect to each other directly.

2. Further, there is an option of logging the communication not only on the side of the clients, but also on the server's side.

3. With classic `talk`, it is also a problem to configure the firewall correctly, so that `talk` could operate. Because of the multiple port usage, a special connection tracker is needed, which is generally not part of the kernel, and requires a lot of trouble to get working. `Stalk` does not need anything like this.

This, of course, has a great disadvantage as well: it gives a lot of work to the server because it will have not only the responsibility of managing clients, but forwarding the messages of the communicating sides as well.

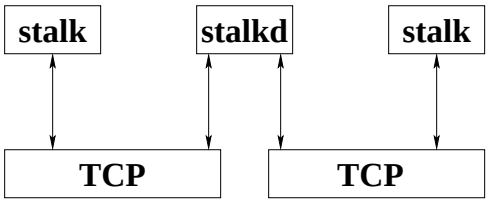


Figure 5.: `STalk` architecture: client-server communication throughout the whole process

4.2 Security architecture of STalk

`Stalk` uses hybrid ciphering in order to achieve secure communication. The simplest and most effective way to achieve this would have been to use, for example, the `OpenSSL` library, but among the thesis' goals was also to achive practical understanding and to experiment with the implementation of these algorithms.

Within hybrid ciphering, `stalk` uses two algorithms, the asymmetric key system RSA for exchange of symmetric keys, and than the symmetric key system DES for communication itself. Here is a brief description of how the `stalk` security protocol works:

1. The client connects to the server.

2. The server sends its RSA public key to the client.

3. The client generates a DES key, ensures that it is not a weak key, encrypts it using the server's RSA public key and sends it to the server.

4. The server decrypts the DES key and stores it for communication with the client.

5. The server uses the DES key of the client to encrypt the connected users' list, which is sent to the client by the server.

6. As long as the client does not choose a partner, changes in the connected users' list is sent to him to keep him up-to-date.

7. After the client selects a partner, his selection is encrypted and sent to the server.

8. The server notifies the partner, the callee's DES key will be used for further communication between the two clients, and communication begins.

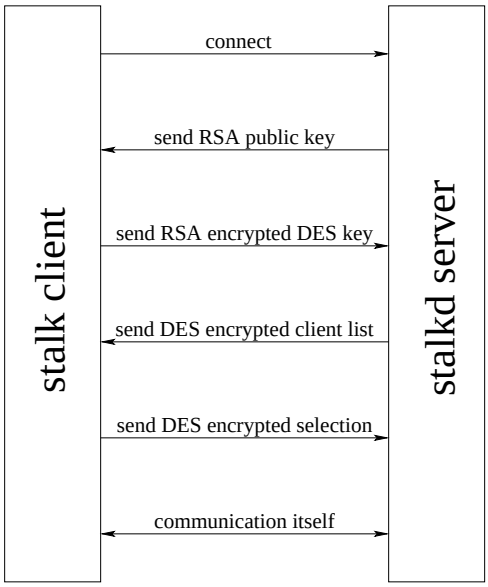


Figure 6.: `STalk` protocol

Chapter 5

STalk – implementation

5.1 `stalkd` – STalk server

The server part of the application handles connections from clients and serves them. This is a daemon, which runs in the background.

In the beginning, right after the variables are declared, command-line arguments are checked. The `-l` argument indicates that the server has to log the global traffic (who connects, from where, who disconnects, etc.). The `-u` argument indicates that the users' input, who are listed after this argument, are to be logged. The log files will have the name {`username`}`.srv.log` and the global log file will have the name `.main.srv.log`. The `-k` argument means that the user wants the server to look for the key file at a different location than the default, which is `.stalk_rsa_keys`.

Right after this, it reads the file, called `.stalk_rsa_keys`, which has to be located in the working directory (or the one we specified after the `-k` argument). Right after the execution of the program, it tries to open this file to read the RSA public and private keys. It is very important because although the communication itself with the clients is encrypted by the symmetric-key algorithm DES (since it is much faster then the public-key algorithms), the secret key is transferred to the server using the public-key system, RSA. The RSA keys are stored in a data structure called "long_num," which stores the length of the value and value itself as a char array.

The next step, the server does, is get rid of the terminal. It simply `fork()`s to a child process, the parent dies, and then closes all the standard streams (input, output, error).

After this is done, the server initializes the data structures, used by the

TCP stream sockets, clears the file descriptor set that will check all the connections, sets up a listening socket, binds it to the computer's IP address and sets it up to listen.

The final preparation before the main loop of serving the clients could begin consist of opening and preparing the main log file if it is necessary and setting the way signals are handled. The signal handler function is the `endit()` function, which simply closes all the log files that are open at the time of the signal, and then shuts down the program. Right before the main loop, the program also initializes a message queue for turning user logging on and off.

Then, the endless loop serves the clients. At the beginning of each loop, the `select()` function selects those file descriptors that are active for reading (which means that they have something that the server could read).

This is followed by the checking of the message queue, to see if logging for someone needs to be turned on or off. Another program, `log_users` – which will be explained later on – is used to send the names of the users whose logging status needs to be changed, which is done through the above mentioned message queue. If there is a name in the message queue, then the program checks, whether the user is among the logged users or not. If he/she is, then he/she will be removed, and his/her log file will be closed. If he/she is not, then he/she will be added to the users to be logged, and his/her log file will be opened (or created if necessary).

Right after this is done, the program goes through all the descriptors that are registered and checks which was marked as active by the `select()` function. If an active descriptor (socket) is found, the server will choose how to serve it:

1. If the active socket is the listener socket, then the program accepts the new connection. It adds a new client element to the linked list of on-line clients, gets its socket descriptor with the `accept()` function, and sets all the other attributes to 0 (except for the client's logfile's descriptor, which is set to -1). Then, the RSA public key is sent to the client so that it could generate a symmetric key and send it back.

2. If the active socket is not the listener socket, and after checking the client structure of the socket, the server finds that no DES key has been set for the client yet, the raw DES key is read from the socket, and decrypted, using the RSA private key. Since it takes some time to do that, the program starts a new thread to commit this operation to

let other clients communicate while the calculation proceeds. After the raw key is decrypted, the DES key fetching algorithm is used to get the 16 DES round keys, which will be used for encryption/decryption.

3. If the DES key has been set, but the user's name has not, then the program first checks, whether the thread has already calculated the DES round keys or not. If not, the socket is left active, the loop continued. If it has, then it is assumed that the client is sending his/her username to the server. So the server reads from the socket and decrypts the username and password using the DES symmetric-key. The server then checks the password (using the password file `.stalk_passwords`) and checks whether there is already a user with the same name logged on and idle (meaning that he/she is not talking to anyone). If the password is incorrect, a message is sent indicating this to the client, and the connection is closed, and if a user is already logged on and idle, another message is sent to the client, and then the server closes the connection. If none of these situations happen, the username is stored among the client information, the clients username and hostname are sent to all the other on-line and idle clients, the other clients' information is sent to the given client, the log file is opened if it is requested and the connection is logged in the main log if logging is turned on.

4. If the server has every information about the given client, and the client is idle, it is assumed that a connection request is sent. The server then creates a request message and sends it to the target client. It also sends the symmetric key that is used with the requesting client, so the two clients would use the same key so that they could communicate. The connection will also be stored in a linked list. If the target client is not found, then an error must have occured, and an error message is sent to the requesting client.

5. Finally, if none of the above situations occur, then it is assumed, that the received data is part of a communication. The other client, to whom the sender talks to, is searched, and the data is passed on to him/her. If logging of the sender is requested, the data is also decrypted and stored in the log file.

The program exits, when it gets one of the terminating signals.

5.2 `stalk` – STalk client

The client part of the application connects to the server, visualizes the other clients who are connected and, after a talking partner is selected, it reads every key that has been pressed, and immediately sends it to the screen and to the talking partner as well.

At first, the client checks if the command-line arguments are correct. It needs at least two arguments: the username and the hostname to connect to. A `-l` parameter may also be given indicating that logging of the communication is requested. The logfiles will have the names {`username`}`.log`.

The next step is to observe the target hostname, get the IP address, fill up the data structure required to connect to the server, and attempt to connect to the server.

After the client is connected, the RSA public key of the server is obtained, a DES symmetric key is generated, encrypted using the RSA public key and sent to the server to use for the communication with the client. The server needs to acknowledge that it has calculated the DES key and is ready to receive further information.

After the server's answer, the user is prompted for a password. The terminal is preconfigured not to echo the characters as they are entered so that nobody could read the password from behind the user. The username and the password is then concatenated, and a '@' is inserted between them. Then the whole text is encrypted and sent to the server. If the server does not indicate that the username/password is incorrect, nor that the user is already connected and idle, the clients hostname is received.

The next step is to get the list of all the connected users and their hostnames, where they are connecting from. There is a function, called `recv_items`, which reads 8 characters at a time, decrypts them, checks if it is a hostname, a username or a connection request, and handles it accordingly – which means that if it is a hostname/username, it is added to the list of connected users (or if it is already added, then it is removed because it means that the user has disconnected or started talking to someone), and if it is a connections request, than the partner's name and hostname are set, and the menu part of the program finishes.

While the program reads users and connection requests from the network, it also sets up a menu, which visualizes the users that the client may select from to talk to. The program always updates the menu, writes out the names of the users and their hostnames and draws little arrows next to the

username and hostname of the client that is presently selected, using the *up* and *down* keys. If *enter* is hit, the selected user is returned.

After the talking partner's name is returned, the `who_connects` variable is checked to see if the talk request came from us or from the other client.

If the request came from us, then the username is encrypted and sent to the server to notify our partner about the talk request.

On the other hand, if the request came from another client, then the DES key of the other client is read, and the original key is replaced by this new one, so that the two communicating clients would use the same key for the communication.

After the partner is selected and the two clients have the same key, communication itself can begin. First, signals are handled, `SIGWINCH`, which is sent when the user's terminal size is changed is handled by resetting the local and the remote user's screen sizes and redrawing the window, while all the terminating signals are handled by the `screen_end` function, which closes the network connections, stops the application and waits for the user to hit *Ctrl* + *X* to exit.

The program then checks if logging is requested, and if so, the log files of both talking clients are opened for logging.

An endless loop realizes the talking itself. It has two tasks:

1. It checks the non-blocking socket that has been set up earlier. If there is something to read, it means that the other client sent something, so the program reads it from the network, decrypts it, and puts it on the screen of the partner. The printable characters are also stored in a data structure for scrolling back and forth the screen. If logging of the partner is requested, the charapters are written into the logfile as well.

2. The second task is to get any character from the local user as well. The character is read from the screen; if it is writeable, then it is written out to the clients screen, encrypted, sent to the partner and, if it is requested, logged as well. The character, again, is stored in a data structure for further scrolling. If the character is some special character, then the screen is adjusted accordingly.

The program exits, when it gets one of the terminating signals or when `Ctrl + X` is pushed.

5.3 `log_users` – Tell the server, which users to log

This program of the package is very simple. It requires one command-line argument, a username. Its purpose is to change the status of logging of the given user. It opens the IPC message queue that is read by the STalk server and writes the username into the message queue. The server then reads this, and handles this accordingly. The program returns immediately after writing the username into the message queue.

5.4 `miller_rabin` – The Miller-Rabin probabilistic primality test implementation

It is the implementation of the algorithm that has been explained above. It gets a command line argument, a number, and it tests the number for primality, and returns 1 (true) or 0 (false). The number of testing cycles can be set using the `MILLER_RABIN_RELIABILITY` macro. This program is used by the RSA key generator.

5.5 `rsa_generate_keys` – Generating the RSA key pair

The RSA key generator simply generates numbers, which will have the decimal length of `RSA_KEY_LENGTH_HALF`. This macro can be changed, but the longer the keys, the longer it will take to find the two primes. After finding the two primes, they are multiplied, and the public key is received. The Extended Euclidian algorithm is implemented to find the private key. After the key is found, it is saved to the file `.stalk_rsa_keys`. The program also uses special functions to deal with long numbers – to choose the greater one of them, to add them, to subtract them, to multiply them, to do integer division and to do modular exponentiation. These function follow the normal mathematical algorithms to get the result, they only use special a data structure for long numbers.

If we want to store our new key in a different file than the default, the

`-k` option handles these requests.

Chapter 6

User's guide

The usage of this program is pretty simple. Gcc compiler is needed to compile, and program `make` to do it in the correct order. After unpacking, just simply run 'make' and 'make clean' to remove unnecessary temporary object files. The use of each part of the program is described below. 'make deinstall' will delete the binaries. There is an example password file (`.stalk_passwords`) and an example RSA key file (`.stalk_rsa_keys`) included.

6.1 `stalkd`

The server part of the package is a daemon. Once it is started, there is nothing we have to care about because it does the job. It takes three arguments:

1. `-l` argument means that the global traffic is to be logged, which means who is connected, from where, etc. The main log will be written to the file called `.main.srv.log`.

2. `-u user1,user2,...` argument sets the users' communication who are listed to be logged. The files where the log will be written will be called {`username`}`.srv.log`.

3. `-k keyfile` a different key file can be specified than the default.

The server can be shut down by sending it a terminating signal.

6.2 `stalk`

`stalk` needs two command line arguments: a username and a host to connect. A little time is needed, and the user has to type his/her password for authentication. If the authentication is successful (and nobody else with the same name is logged on and idle!), the menu will appear, where the client can select who he/she wants to talk to.

Figure 7.: The `stalk` user selection menu

With the *up* and *down* keys, the user can move the selection arrow up and down among the connected users, or by pressing *Ctrl + X*, the user can quit. By pressing enter, talk with the selected user will be started.

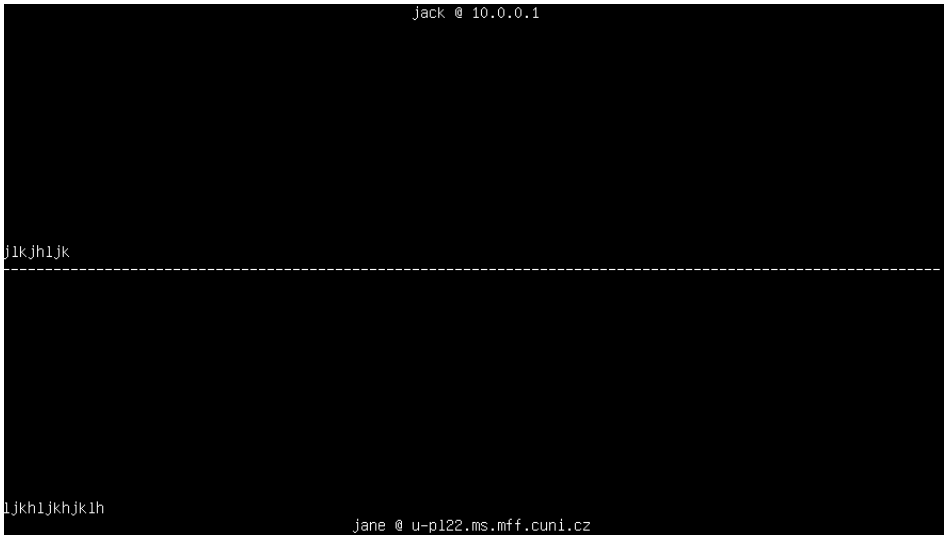


Figure 8.: The `stalk` communication screen

Once we start chatting, the following keys may be used:

1. Normal letters, numbers and other printable characters – they will be displayed and sent to the other client without modification.
2. *Ctrl + H* or *Backspace* – the last typed character is deleted.
3. *Ctrl + J* or *Enter* – new line inserted.
4. *Ctrl + P* – the screen is redrawn.
5. *Ctrl + X* – the communication is cut.
6. *Ctrl + R* – the user's screen is scrolled up.
7. *Ctrl + F* – the user's screen is scrolled down.
8. *Ctrl + T* – the partner's screen is scrolled up.
9. *Ctrl + G* – the partner's screen is scrolled down.
10. *Ctrl + K* – the logging of the user is toggled.
11. *Ctrl + L* – the logging of the partner is toggled.

Once the communication is cut, the screen will be held until *Ctrl + X* is pressed, and then the program exits.

6.3 log_users

This part only notifies the server to toggle the logging status of a certain user. Just give the username as the command-line parameter, the program will do the rest.

6.4 rsa_generate_keys

This program does not need any arguments, it simply needs to be run, and it will generate the RSA keypair, which it will save to the file called `.stalk_rsa_keys`. However, if we specify the `-k` parameter, we may request a different location for our new key file. The generation may take a longer time depending on the key length.

Chapter 7

Conclusion

While TCP/IP is a very effective communication protocol, as it is described, it has a lot of weaknesses that can be exploited by hacking methods like TCP Hijacking. By using cryptographic methods for `stalk`, the chances of these kinds of attacks can be minimized, and the interception of communication can also be prevented. The only problem with this is that there is no mathematical proof that these algorithms cannot be broken, which means that there is always a little chance that someone finds an algorithm that breaks these ciphers at the worst moment. On the other hand, however, 100% security is not possible.

There are some improvements that could be made to this application. One weakness is that the passwords of the users are held in a plain text file, which poses the threat of stealing the password file. A very useful improvement would be to implement one of the SHA or MD hash algorithms, and hash the passwords instead. This way, even if the password files are stolen, provided that the algorithms are correct, the thief would not be able to recover the passwords.

Another improvement could be for clients to store the public keys of stalk servers in separate files so that it would not be necessary to receive them through the network. This way a man-in-the-middle attack could be prevented.

As far as the implementation process is concerned, there had been some difficulties that had to be overcome. Since TCP uses byte stream (as it was mentioned above), reading from the stream needed great care so that the correct bytes would have been read. Since DES uses blocks to encrypt and decrypt data, if only one of the received bytes is not read at the correct

moment or at the correct state, the rest of the communication will be completely desynchronized, which would produce invalid data.

Another hard task was to implement the RSA algorithm correctly. It is important to generate a correct prime number and also to implement long number operations (like long number modular exponentiation) correctly. There is no certain proof that the implementation attempt was successful, but during the testing period of the program, there has not been any invalid key generation nor any invalid encryption/decryption. The speed of these processes, however, could be improved further, since it takes for a while to generate a key and to encrypt/decrypt as well.

Even though there are a lot of things that could be improved in this application, while examining and implementing these methods, the author has achieved new experiences on the fields of UNIX network programming and cryptography as well.

Bibliography

[1] Hall Brian E.: *Guide to Network Programming Using Internet Sockets*, On-line: http://beej.us/guide/bgnet/

[2] Menezes Alfred J., van Oorschot Paul C., Vanstone Scott A.: *Handbook of Applied Cryptography*, CRC Press, 2001.

[3] Stevens Richard W.: *TCP/IP Illustrated*, Addison-Wesley, 1994.

[4] Stevens Richard W.: *UNIX Network Programming*, Prentice Hall, 1998.

[5] Wegener Christoph, Dolle Wilhelm *Hijack Prevention*, On-line: http://www.linux-magazine.com/issue/58/TCP_Hijacking.pdf