Design Notes for an Efficient Password-Authenticated Key Exchange Implementation Using Human-Memorable Passwords

Author:
Paul Seymer
CMSC498a

Contents

1 Background

1.1 *HTTP 1.0/1.1*

The HTTP protocol is considered a request/response protocol since the client application initiates the protocol by sending a request to a web server, and receives a response containing status information, and the object requested (if applicable). More precisely, the client sends (in clear text) a request method (GET or POST), the HTTP protocol version, and URI. The Client then sends a stream of information containing any POST content it may need to send to the webserver, in the case of a POST request (i.e. form submission). The Server then responds with a status line that includes a success/error code and other Meta data, immediately followed by the object requested (if applicable).

The HTTP protocol is spoken over TCP/IP, typically on TCP port 80. In HTTP/1.0, a new connection is made for each request/response. In a later revision of the protocol (HTTP/1.1) the connection can stay open for more than one request/response.

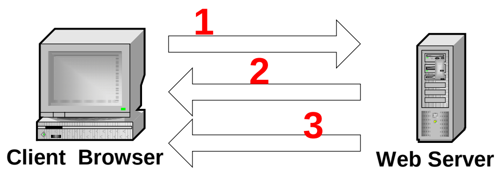
In this implementation, it is assumed that HTTP 1.0 will be used by the browser. All test results and performance data is based on this assumption.

1.2 *Password Authenticated Key Exchange*

This system is an implementation of an Efficient Password-Authenticated Key Exchange found in a paper by Jonathan Katz and Rafail Ostrovsky and Moti Yung [3]. Low-level key exchange protocol specifications can be found in that paper.

2 High Level Design

2.1 *Normal Clear Text HTTP*

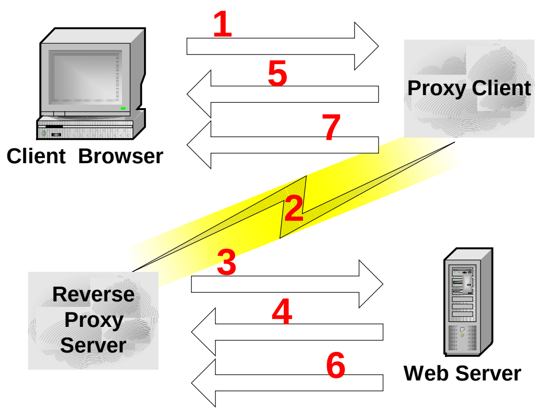


(Figure 2.1.1)

Normal HTTP 1.0 communication sequence

Normal HTTP 1.0 requests occur over a single socket connection in two main phases. As shown in Figure 2.1.1, a client browser sends out a request to a web server (1), and the web server replies with the appropriate response metadata (2) and the object requested (3) if appropriate. For the purposes of this document, the response chain has been split into two separate phases, one for the clear text response (2), and the other for the object data (3). It is useful to keep in mind, however, that all transactions are preformed during the same socket connection, and the object data immediately follows the response.

2.2 HTTP with PAKE Client / Server Model



(Figure 2.2.1)

Secure HTTP 1.0 communication sequence with PAKE

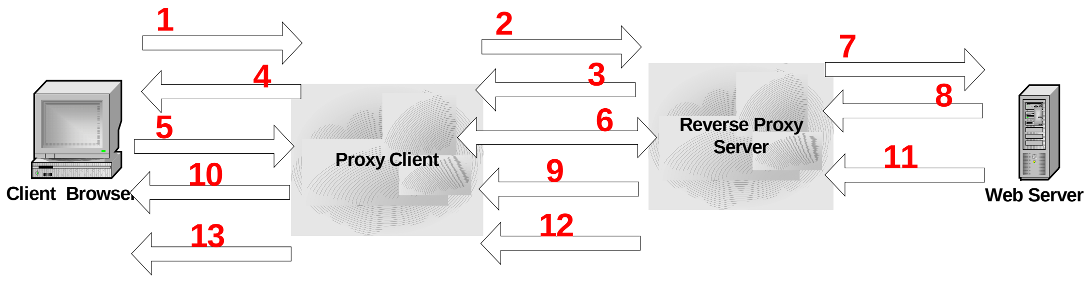
When we try to solve the problem of securely communicating in a way that is transparent to an end user, we are immediately presented with a few challenged. The obvious is how to implement some middleware in between the client and web server that ensures secure communication.

In this implementation, the solution is to create a PAKE-Aware Proxy Client for end users to point their browsers to, and PAKE-Aware Reverse Proxy Server for web servers to listen for incoming requests from the PAKE-Aware client. In Figure 2.2.1, we see this PAKE Proxy client/server architecture in action.

A client browser makes its usual HTTP 1.0 request (1), but instead of talking directly to a web server, it talks to a proxy. This proxy then encrypts the request and sends it to the Pake-Aware reverse proxy sitting on the web server's network (2). The reverse proxy then decrypts the request, and sends the decrypted request to the web server (3) in the same fashion as a regular browser would. The web server then sends back the response to the reverse proxy server (4), the reverse proxy encrypts it, and sends it back to the client's proxy client for decryption and tunneling to the browser (5). The same path is followed for the object's data (6 and 7). At no point during this transaction is the request or response transmitted over the Internet in clear text.

One challenge with using this sort of model is that both the client and server proxies need to have a symmetric key. There also needs to be some sort of validation process that the end user is who the server proxy thinks they are. This is typically done with passwords, but most human memorable passwords are weak against things such as dictionary attacks and other forms of password cracking. This motivates the need to use the Efficient Password-Authenticated Key Exchange [3] protocol, which uses (among other things) the Cramer-Shoupe cryptosystem and a one time signature scheme in order to support the notion of provable security and one-time use keys.

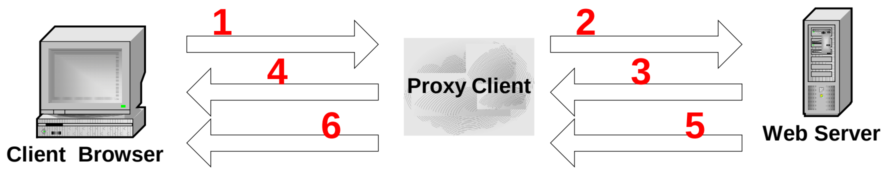
The Password-Authenticated Key Exchange (PAKE) protocol is initiated as shown in figure 2.2.2. This process is initiated whenever a client contacts a PAKE-aware server for the first time (1 and 2). The reverse proxy indicates to the client proxy that it is PAKE-aware (3), and that the client should ask the user for authentication credentials, and encryption methods (optional). The proxy client, instead of returning an object to the browser, returns a form asking for authorization credentials (4). The user then enters their username and password, and submits the form (5). The proxy client then performs the PAKE protocol with the reverse proxy (6), sends the original request (7), and communication continues following the basic HTTP 1.0 protocol procedures (8-13). The keys are stored for the client/server pair, and used in the future for secure communication, without the need to perform the key exchange again.



(Figure 2.2.2)

PAKE Initialization

When we solve the problem of securing communication to certain servers, its required that we not damage the way in which the browser communicates with non-secure servers. In cases where the client is not talking to a PAKE aware server (see Figure 2.2.3), normal HTTP 1.0 is performed (1 and 2), and communication is simply tunneled through the proxy (3, 4, 5, and 6). This is accomplished because the proxy client is expecting a specific notification from the proxy server in the secure case that differentiates it from a normal HTTP 1.0 response. If this notification text is not present in the response, the proxy client assumes it is communicating with a non-secure non-PAKE server, and defaults to the clear text HTTP 1.0 protocol.



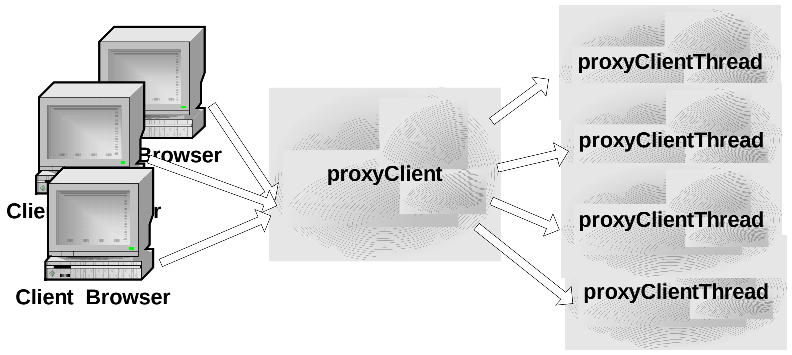
(Figure 2.2.3)

Normal HTTP 1.0 communication sequence with PAKE

3 Low Level Design

3.1 *Client Proxy*

The Client Proxy works in such a way that multiple browsers can connect to it at once (like normal proxies do). In order to accomplish this, a single executable runs (*proxyClient object)*, listening for incoming connections from the browsers in a port specified at the command line. When a connection is made, a thread (*proxyClientThread* object) is spawned to serve that particular browser connection. This new object then handles all connection and communication activities for the browser request for the life of the connection, and the *proxyClient* executable goes back to listening for new connections. Figure 3.1.1 illustrates this relationship.



(Figure 3.1.1)

Client Proxy Executable Object Model

The proxyClientThread object reads the request from the browser and parses it for remote host, port, and URL information. It then makes a connection with the intended remote server. A couple of things can happen at this point depending on which of the following scenarios exists:

- The remote server could be a non-PAKE-aware server.
- The remote server could be a PAKE-aware server that you have a key for.
- The remote server could be a PAKE-aware server that you have no key for.
- The request could be a POST from the browser providing PAKE authentication credentials (Continuation of the communication process with no shared key).

3.1.1 *Proxy to Non-PAKE web server*

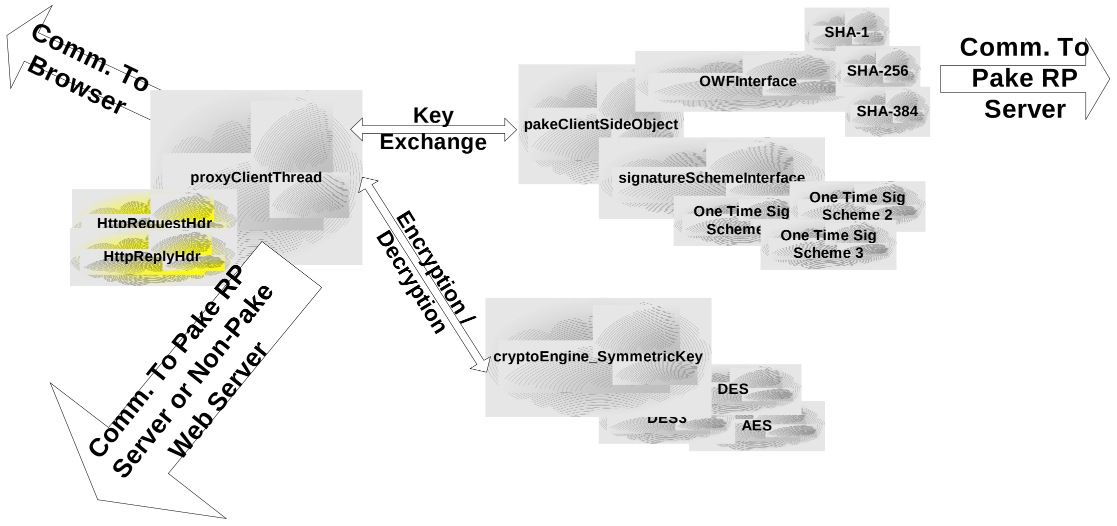
This type of communication is the simplest to perform. The *proxyClientThread* simply uses its internal functions and classes to parse the request, and then initiates communication to the remote web server. A check is made on the response header to make sure that the server is Non-PAKE (PAKE server reverse proxies add a unique response line to their response header), and then the data is tunneled between the browser and server through its connection.

3.1.2 *Proxy to PAKE-aware web server with shared Key*

This type of communication is occurs when the PAKE protocol has already been performed. It is different from the last subsection's object model in that now a *cryptoEngine_SymmetricKey* object is created using a shared key it has stored in a list of keys it keeps during run-time. The proxy then makes a connection to the remote server indicated in the request, and sends over a special string indicating to the reverse proxy that it will be conducting a secure transfer. The browser request header is then encrypted using the *cryptoEngine_SymmetricKey* object, and sent over the connection to the remote server where the proxy's PAKE counterpart receives it, decrypts it and sends back an encrypted response and object payload (see next section for Server-Side implementation details). Figure 3.1.2 illustrates this relationship.

3.1.3 Proxy to PAKE-aware web server without shared Key

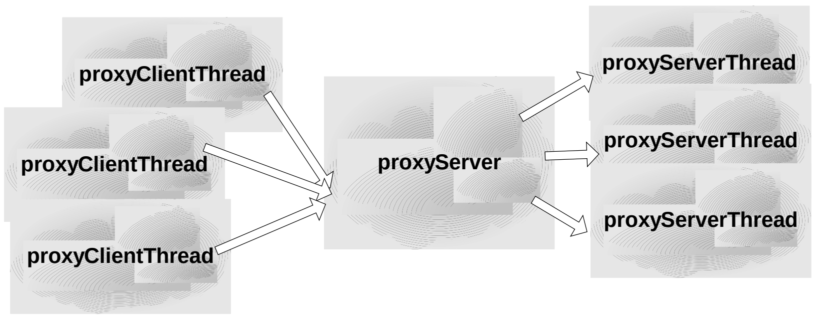
When the proxy client connects to a PAKE reverse proxy server, and the two do not share a symmetric key, the PAKE reverse proxy server sends back a special line in its response header letting the client side proxy know it needs to initiate the PAKE protocol. However, in order to initiate the PAKE protocol, authentication credentials must first be obtained from the end user. This is accomplished by sending back a generic response to the client's browser, followed by an HTML form asking for a username and password. The form is preset to post directed back to the PAKE client proxy when the user submits it. Since this process terminates the original connection with the end user, original request is stored in the proxy client in a HashMap keyed on a serial number generated when the HTML form is sent to the client, and parsed in the submitted form. When a user submits the, form data for username and password are parsed and the proxy client looks up the end users original request in the HashMap and connects to the server listed in the original request. Then, a *pakeClientSideObject* object is created, and the PAKE protocol is conducted with the remote server. This object is described more in detail in a later section, but upon receipt (and storage) of the shared symmetric key, the original request is encrypted and sent to the reverse proxy. Communication then continues as it would in the case where each server had a shared key, and the original requested object is sent back to the end user in reply to the form submission making the user's internet activities interrupted only once each time they need to authenticate.



(Figure 3.1.2)
Client Proxy Thread Object Model

3.2 *Server Reverse Proxy*

The reverse proxy server is also multithreaded and handles multiple connections to it in the same way that the proxy client does (see figure 3.2.1) by creating a new thread each time a connection is made.

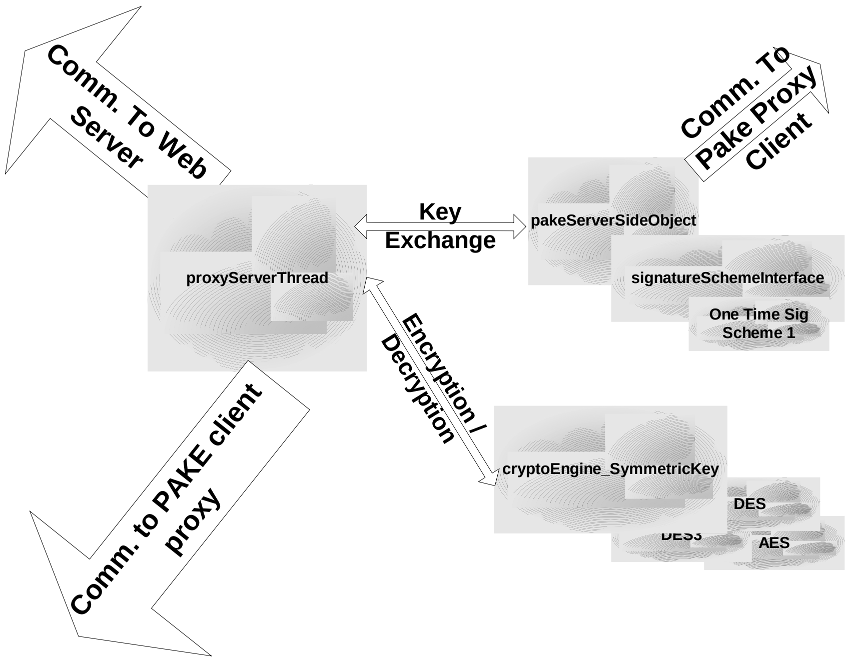


(Figure 3.2.1)

Client Proxy Thread Object Model

But the server-side elements of this system are much simpler then its client-side counterpart. There will only be three server-side types of communication:

- A PAKE Proxy Client requests communication with no shared Key.
- A PAKE Proxy Client requests communication with a shared Key.
- A PAKE Proxy Client requesting initiation of the PAKE protocol.



(Figure 3.2.2)

Server Reverse Proxy Thread Object Model

3.2.1 Communication with a shared Key

In the event communication is initiated with a shared Key present, the *proxyServerThread* handles all server-side communication. It first receives a special clear text header from its proxy counterpart indicating it should use a secure channel. It then creates a *cryptoEngine_SymmetricKey* object initialized with the shared Key, decrypts the request header, and sends the request to the local web server it is reverse proxy-ing for. It then encrypts and tunnels the response back to the proxy client.

3.2.2 Communication without a shared Key

When a shared key is not present, the server element simply sends back a header indicating to the requester that it is communicating with a PAKE-aware web server, and should initiate the PAKE protocol. The request never reaches the local web server in this case.

3.2.3 PAKE initialization request

When the proxy client initializes the PAKE protocol, a *pakeServerSideObject* is created to handle the server-side protocol tasks. After the key is exchanged, it receives an encrypted request from the PAKE client, and continues secure communication as described in subsection 3.2.1.

3.3 pakeClientSideObject/pakeServerSideObject Object Model

3.3.1 One Way Functions

Currently, the hash functions this system can use are SHA-1, SHA-256, SHA-384, and SHA-512 and are selected globally at the command line for the proxy client and reverse proxy server.

3.3.2 One-Time Signature Schemes

This particular build uses the Identity function, and 1024-bit RSA function, to build its One-Time signature scheme, however with slight modification the protocol objects can use different signature schemes. These modules can be coded independently and “plugged” into the pakeClientSideObject and pakeServerSideObject objects by passing in additional command line arguments.

3.3.3 Prime Numbers in Java

Prime numbers in Java have been created using the BigInteger class with a prime number certainty of 100. This certainty means that there is a probability of 1 – ½^100 or 0.9999999999999999999999999999921 that the number selected is truly prime.

3.3.4 Password Management

Usernames are stored on the server’s filesystem in clear text, and passwords are stored as an output from java.lang.String.hashCode(). In the future, these will be stored with a more robust hash function.

3.3.5 Credentials Management

Credentials are passed through the proxy client via a form on a web page, which is returned from the client when a PAKE-aware webserver is contacted. The username is entered into the form as a textfield, and the password is entered as a password. The proxy client then takes these tokens and attempts to negotiate a shared symmetric key according to the PAKE protocol. In the event that the password is valid, the original request is fulfilled, but in the event that the password is not valid, the form is sent back to the client’s browser so that new tokens can be collected from the user.

3.3.6 PAKE Public Info Generation

The public information used in the PAKE protocol is dynamically generated with the following model:

p =90903287098125491501408197193663424816333077417203499627188848
122021251095383

q =45451643549062745750704098596831712408166538708601749813594424
061010625547691

Generators are created by taking the squaring (mod p) first length(p) – 1 numbers of the SHA-1 hash of the following strings (respectively for each generator) :

- g1 = WWW.UMD.EDU
- g2 = WWW.SUN.EDU
- h = JAVA.SUN.EDU
- c = WWW.TESTUDO.UMD.EDU
- d = WWW.WASHINGTONPOST.COM

In the event that the hash is less than length(p) – 1 the hash is concatenated with itself until it is longer than length(p) – 1

3.4 Key Management

3.4.1 Client Side/Server Side

When the PAKE protocol is performed, both the Client and Server components have a shared Key. In order to prevent the client proxy from requesting a new key for each request, the Keys are stored in memory in each module. The Client Proxy stores the key in a HashMap indexed by the remote host's host and domain name. The Server portion stores the key in a HashMap indexed by the IP Address of the requesting Client.

At runtime, the server is given a command line argument specifying how many times the shared key can be used before it expires. Each time the key is used, a value unique to each key is decremented. When this value becomes zero, the server drops the key resulting in the proxy client requesting new credentials from the user, and renegotiating another key.

3.4.2 Synchronization/Pre-mature Key Expiration Issues

In the event a key is lost in one of the PAKE modules, either by application restarts, key expiration, or a critical exception is thrown, a problem arises with synchronization (one side may expect the other to have a key it no longer has). Since there is no longer a shared key the PAKE client or server re-initiates the process of getting credentials from the end user, and conducting the PAKE protocol to create a new symmetric key.

4 Usage Guidelines

4.1 Initial Request - non-secured Side Channel

It is important to keep in mind that there is one special case where client data is sent outside of the secure channel. When a PAKE exchange has not been conducted prior to a given request there are no shared keys. In this case the Proxy Client must first send its request to the PAKE-Aware web server in order to identify the remote server as PAKE-Aware. If this request header contains form data that is secure, it will be transferred as clear text and not encrypted. Most implementation of forms would never create an instance of this, whereas in order for the browser to contain the form, it would be necessary to have already received it from the web server, requiring a prior connection resulting in a prior instance of the PAKE protocol.

The case where this problem would come up would be when a form is sent to the client browser, but POST-ed to a different server. If that different server is PAKE-Aware, the form submission would be sent in the clear to that server.

This occurs because normal HTTP communication needs to be maintained on client computers that use the proxy. Since there is no initial identification of server types before you communicate with them, and non-PAKE web servers do not send any information back to the client except for a request response, this implementation detail was necessary.

5 Installation and Runtime

5.1 Client Proxy Installation

Simply unzip the pakeClient.zip and open a command line in whatever directory you unzipped the file to and type:

```
java proxyClient <<port>>  <<One Way Function >>  <<Signature Scheme >>
```

If you omit the portnumber, the default port of 2003 is used. If you omit the One Way function name, SHA-1 is used. The RSA function is used by default to generate one-time signing/verifying keys for the signature scheme.

5.2 Browser Configuration

The end user's browser must be pointed to the proxy prior to use. Simply configure your browser to point all HTTP traffic to 'localhost' on whatever port you are running the proxy on (port 2003 by default)

5.3 Server Reverse Proxy Installation

Simply unzip the pakeServer.zip and open a command line in whatever directory you unzipped the file to and type:

```
java proxyServer <<port>> <<One Way function >>  <<Signature Scheme >> <<# key uses>>
```

The port number used here is the port what the web server is listening on. If you omit the portnumber, the default port of 2005 is used. If you omit the One Way function name, SHA-1 is used, and RSA is used by default to generate one-time signing/verifying keys for the signature scheme.

5.4 Web Server Configuration (Optional)

The web server should be restricted to accept connections only from the reverse proxy. In this case, the reverse proxy should be run locally on the web server, and the web server should only accept localhost connections, and on whatever port you decide to run on (other than 80, since the reverse proxy listens to incoming connections on that port)

6 Known Limitations

6.1 Existing Defects

- Due to differences in browser and webserver requests, certain requested objects do not properly tunnel through the reverse proxy. This will be corrected in a future release, in the future, more robust request/response handling will be implemented to deal with these differences.
- 304 Responses are non-existent, due to a bad fix that modifies all objects to have a last-modified time of "Mon, 20 Jul 1987 01:01:00 GMT";
- Binary files (like images) have problems being sent over the encrypted channel, and do not decrypt correctly.

7 Future Development

7.1 *Defect Correction*

- Increased error handling
- More robust reverse proxy that handles all types of web server objects.
- Secure POST'ing to a pake-aware server.

7.2 *Functionality Enhancements*

- Better handling of non-PAKE communication with reverse proxy (non-PAKE users attempting to connect to the proxy should get a more user-friendly response)
- User Administration functions
- More than just DES as a crypto engine for the secure channel.

References

1. *Hypertext Transfer Protocol -- HTTP/1.1* RFC - http://www.w3.org/Protocols/rfc2616/rfc2616-sec1.html#sec1.4

2. Bouncy Castle DES Example - http://wireless.java.sun.com/midp/ttips/dataencryp/

3. Efficient Password-Authenticated Key Exchange Using Human-Memorable Passwords. Jonathan Katz and Rafail Ostrovsky and Moti Yung - http://eprint.iacr.org/2001/031/

Acknowledgements

The cryptographic engines, hash functions, and binary-hex-binary converters used in this implementation were found in the latest java cryptographic APIs released from www.bouncycastle.org.

The HttpReplyHdr and HttpResponseHdr and a few utility functions in the proxyClientThread classes were taken from or based on an example posted on comp.lang.java on May 14, 1996 by Sherman Janes (sherm@ibeam.jf.intel.com)