MASSIMO BERNASCHI, FRANCESCO CASADEI, AND SAMUELE RUCO

migration of secure connections using SockMi

Massimo Bernaschi joined the IBM European Center for Scientific and Engineering Computing (ECSEC) in 1987 in Rome, where he spent ten years working in the field of parallel and distributed computing. Currently he is Chief Technology Officer of the Institute for Computing Applications, which is part of the Italian National Research Council (CNR). He is also an adjunct professor of Computer Science at "La Sapienza" University in Rome.

massimo@iac.cnr.it

Francesco Casadei has a master's degree in Computer Science from Rome University "La Sapienza." Since October 2005 he has been with Quadrics Ltd as a software engineer. Before then, he spent six months at the IBM T.J. Watson Research Center (Yorktown Heights, NY) as a research scholar.

francesco.casadei@quadrics.it

Samuele Ruco graduated in Computer Science in 2006 from Rome University "La Sapienza." Since then he has been with Quadrics Ltd, a Finmeccanica Company, working on embedded systems and parallel programming.

samuele.ruco@quadrics.it

***SOCKMI* IS A SOLUTION FOR MIGRATING** SSL/TLS secure connections between Linux systems that extends recent work on TCP/IP migration [1]. Secure Socket Layer (SSL) and Transport Layer Security (TLS) [2] add security to any protocol that uses reliable connections, such as TCP, to establish a "virtual circuit" from a client to a server. We chose to provide support for SSL migration because it is one of the leading technologies used today to secure connections, especially those to applications hosted by Web servers. It can be used for encapsulation of various higher-level protocols such as http (to form https), ftp, smtp, and nntp. It can also be used to tunnel an entire network stack to create a Virtual Private Network (e.g., in OpenVPN [3]).

The migration mechanism involves the following levels:

- Network: IP packets must be redirected to the importing host (the target system).
- Transport: All TCP information must be transferred to the host that imports the connection.
- Application: Session keys and other sensitive data needed to ensure the integrity of the secure connection are likewise migrated to the target system.

There are a number of situations in which the migration of secure connections can be useful: for instance, when there are requirements of load balancing, quality of service, and fault tolerance and it is not possible (or is undesirable) to restart the connection. With respect to other solutions, (i) it is able to migrate both ends of a connection; (ii) it does not require cooperation on both ends; (iii) it can be activated in any phase of the connection; and (iv) it does not require changes to existing Linux kernel data structures and algorithms. Moreover, the mechanism used for sending specific application-level information can be used for any application protocol.

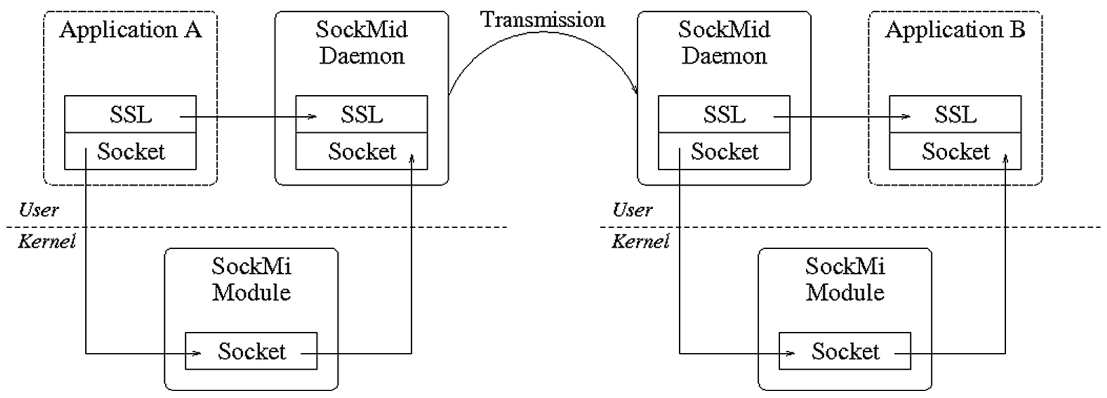

FIGURE 1: THE DESIGN OF THE MIGRATION MECHANISM IMPLEMENTED IN SOCKMI

The design of SockMi (see Figure 1) aims at achieving the following goals:

- Transparency: The connection endpoint that does not migrate should not be affected by the migration mechanism in any way with the exception of possible (but limited) delays owing to the "triangulation" mechanism described in Ref. 1. This implies that no information should be exchanged between the peers to accomplish the migration.
- Portability: The migration mechanism should have minimum impact on the underlying operating system, meaning that: (i) no patches to the kernel must be required; (ii) no new system calls must be introduced. To fulfill these requirements, we implemented the migration mechanism as a loadable kernel module (LKM) and defined an API (application programming interface) that hides all implementation details.
- Symmetry: It should be possible to migrate both connection endpoints.
- Versatility: It should provide a general mechanism that supports the migration of other application protocols.

The main components of SockMi are the module, the daemon, the API, and the IP redirection mechanism.

The module is the core of the TCP/IP socket migration mechanism. In particular, the module is responsible for: (i) saving (restoring) the state of migrating sockets during the export (import) phase; (ii) exchanging information about migrating sockets with the SockMi daemon; (iii) providing low-level primitives to activate and control the socket migration facility.

Besides the state of the socket, migrates also affect the corresponding "in-flight data." These data are found in the receive and transmit queues of the socket, which contain, respectively, packets received by the system but not read by the application and packets to be sent, or packets already sent but not yet ACKed.

The module holds sockets ready to be imported in three different import lists, corresponding to the TCP hash tables managed by the Linux kernel: (i) the bound socket list; (ii) the listening socket list; (iii) the connected socket list. These lists change their length dynamically when a new socket is received from the daemon or an application imports a socket. However, to avoid potential memory problems, we set a maximum length for each list. In case a list reaches its maximum length, no more sockets can be queued and the import fails with an error. The module is SMP-safe and supports the preemption mechanism available in the 2.6 kernel. Further information is reported in Ref. 1.

The daemon (SockMid) works in combination with the module to support the socket migration mechanism and in combination with the libappsockmi library to support the application migration mechanism. The daemon carries out different tasks depending on the situation. During the *export phase*, it receives the application's data from the exporting process and the associated socket state from the module; during the *negotiation phase*, it communicates with other daemons running on other hosts in order to choose where to migrate the connection; finally, during the *import phase*, it writes the state of importing sockets to the module internal buffers and it handles the imported application protocols. Moreover, the daemon receives importing requests from local processes and checks whether a request matches an imported connection.

During the import and the export phases, the module and the daemon components need to exchange information about the state of migrating sockets. Since the module lives in the kernel address space whereas the daemon is a normal user process, it is not possible to resort to standard Inter Process Communication (IPC) primitives to exchange data between them. To overcome this difficulty we implemented a buffer sharing system via the mmap() primitive. The module is seen by the daemon as a character device that, through its mmap() file operation, makes its internal buffers available (i.e., it acts as a memory device). In this way, kernel buffers can be read and written by the daemon as if they were in user space. In addition, during these phases, the daemon and the libappsockmi library components need to exchange information about the application protocol data. To this end, the library sends http message requests to the daemon, which replies with a status code. When required, the reply message contains appropriate application data.

The socket migration entails the search for a host willing to "import" the connection. To be eligible to the import of a connection, a host must run an instance of SockMid, which defines and supports a communication protocol among daemons that run on different hosts. This negotiation protocol follows a plain request-response-confirm scheme that can be summarized as follows:

- When host A exports a connection, it sends a request in multicast to the daemons that run on other hosts.
- When a request arrives, a host—say, T—replies, provided that either the socket is explicitly exported to that host or no specific target host is defined in the request.
- If host A does not receive a valid response within a predefined timeout period, the migration fails.
- The first valid response triggers a confirmation mechanism by which host A notifies the daemon running on host T that it has been selected as the importing host.

Choosing the first valid response is a very natural yet simple policy. Other more sophisticated policies based on rules or heuristics could be used. For example, it could be useful to maintain statistics on previous migrations and select the target host in such a way as to achieve load balancing among importing hosts. Collective communication among the daemons relies on multicast. This means that all instances of SockMid have to join the same multicast group and bind() to the same UDP port.

SockMi provides a simple Application Programming Interface (API) to activate the connection migration mechanism. The API is implemented by two user space C libraries: libappsockmi and libsockmi, which are part of the distribution.

The libappsockmi library consists of two functions which provide applications with an easy-to-use method for importing and exporting secure connections: import_ssl() and export_ssl().

To import one or more secure connections, an application calls the library function import_ssl(). This function is designed to poll the availability of "exported" SSL sessions matching the import criteria specified by the application. If one or more matching sessions are available, then the function imports them immediately, by rebuilding the SSL session from the application data and by replacing the local socket associated with the connection. Otherwise, if no matching connection is available, the function waits until either a timeout occurs or one or more "exported" sessions becomes available for import. The prototype of the import_ssl() function is defined as follows:

```
int import_ssl(struct import_ssl_req *irqs, unsigned int nirqs, int timeout);
```

The arguments to import_ssl() are (in order) an array of import requests, the number of such requests, and the maximum waiting time until a successful import occurs (with a negative value blocking import_ssl indefinitely). The information required to formulate an import request are the following: (i) a pointer to the main OpenSSL structure to be replaced with the imported session; (ii) the preferred state the imported connection should have; (iii) the set of criteria a connection must match in order to be imported.

The import criteria let the application define the "properties" of the connection to be imported. Such criteria are the set of allowed secure connection states, the local and remote IP addresses, and the local and remote TCP ports. The import_ssl() function tries to fulfill all requests according to a best-effort policy. Upon completion import_ssl() returns one of the following values:

- 0, if the function timed out before any secure connection could be imported
- –1, if an error occurs
- the (positive) number of connections that have been successfully imported

Note that, even if successful, the function does not guarantee that *all* requests have been satisfied. Furthermore, even if the function returned an error, *some* secure connections may have been imported. Thus, upon return, the application should scan the array of requests and check the output field ssl_state, which either reports the state of the (possibly) imported session or is set equal to 0 to indicate that the request could not be satisfied.

Exporting SSL sessions is much simpler than importing, because there is neither need to specify criteria nor a wait time. To export secure connections an application calls the function:

```
int export_ssl(SSL *ssl, int state, int af, const void *to)
```

The first argument is a pointer to the main OpenSSL structure; it contains all the references to the information that needs to be transferred. The second argument is the state of the connection (e.g., connected client but SSL handshake not performed, connected server with SSL handshake performed, listening server). The last two arguments allow the network address of the importing host to be defined. The to argument may be a null pointer if there is no need to specify a target system. In this case the function export_ssl() lets the migration mechanism automatically select a target system, according to the internal policy of the SockMid daemon. This function returns 0 on success or –1 if an error occurs.

The libsockmi library consists of similar functions for sockets not associated to SSL sessions [1].

In the migration of secure connections, such as those provided by SSL, the difficulties arise primarily from the need of exporting and importing a number of keys and the information required to maintain consistency in the cipher subsystem. We tested our solution with the OpenSSL [4] implementation of the SSL and TLS protocols. OpenSSL is composed of two layers: (i) the SSL Record protocol, which is layered on top of TCP and allows the encapsulation of various higher-level protocols; (ii) the SSL Handshake protocol, which allows server and client to authenticate each other and to negotiate all security-related parameters (e.g., encryption algorithm, cryptographic keys) before the application protocol begins to transmit or receive data. The SSL Record protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a Message Authentication Code (MAC), and encrypts and transmits the result. Received data is decrypted, verified, decompressed, and reassembled, then delivered to the application. This protocol specifies four connection states: current read and write states and pending read and write states. Each state specifies a compression algorithm, an encryption algorithm, and a MAC algorithm. Thus the protocol must migrate all four connection states. In addition, it must migrate the parameters for the following algorithms: the MAC secret, the bulk encryption keys, the Initialization Vector (IV), and the sequence number for the connection in both read and write directions. The sequence number must be set equal to zero whenever a connection state becomes active, and it is incremented after each record. Moreover, it migrates other information, such as certificates and public and private keys. Currently, all these data are exchanged in the clear between the exporting daemon and the importing one. This assumes that the migration of OpenSSL connections happens in a controlled environment where there is no danger of key data being sniffed or malicious hosts offering to import connections just to grab connection-related data.

The target system uses this information to open new SSL sessions with the same SSL context. The data structures involved in defining the state of a secure connection can be determined by inspecting the SSL structure. These data structures have cross-references implemented as C pointers to memory locations. As a consequence, a simple approach based on data copy is not going to work, because pointers would make no sense in a different address space both for a migration to another host and for a migration to the same host. Thus a primary requirement of the migration mechanism is to preserve the referential integrity among the data structures that define the state of a connection.

To save the SSL information to a local memory buffer, an application calls the `save_ssl()` function defined as follows:

```
int save_ssl(void **pbuf, SSL *ssl, int ssl_state)
```

This function allocates a local buffer and saves to it the state of SSL connection. The arguments are (in order) a pointer to the output buffer, the pointer to the SSL structure that defines the connection, and the state of the connection. On success, this function returns the size (in bytes) of the allocated buffer. On error, –1 is returned.

To restore the SSL information from a local memory buffer, an application calls the `load_ssl()` function, defined as follows:

```
int load_ssl(void *buf, size_t buf_len, SSL *ssl, int ssl_state)
```

This function rebuilds session keys and other sensitive data needed to ensure the integrity of the secure connection. The arguments are (in order) the local memory buffer, the size (in bytes) of the buffer, the pointer to the SSL structure (whose contents are updated by the function), and the state of the secure connection. On success, the function returns 0; otherwise –1 is returned.

When a socket migrates to a different host it is necessary to redirect packets coming from the peer toward the host that imports the socket. To this purpose, we resort to a special combination of Network Address Translation (NAT) operations. In particular, we employ a Destination NAT (DNAT) such that packets received by the exporting host for the migrated socket are redirected to the importing host. For this redirection the standard NAT capabilities offered by the netfilter module of the Linux kernel are adequate [5]. The DNAT is triggered by the daemon running on the exporting host.

In addition, packets sent to the peer must have the same IP source address as the original host (for otherwise the peer would reply with an RST packet). In this case, we employ a Source NAT (SNAT) on the importing host such that the source address of packets sent by the imported socket is translated into the address of the exporting host. The SNAT required a modification to the standard NAT mechanism since the latter has a side-effect: The reply tuple is changed according to the applied address translation. The problem is that netfilter expects to receive packets having a destination address equal to the translated source address whereas, in our case, the DNAT sets the destination address equal to the real address of the importing host. To solve the problem, we resorted to the NAT helper mechanism available in the netfilter architecture. Basically, it allows us to invoke a custom procedure we wrote that performs the address translation but does not alter the reply tuple. Note that, in case the host that exports the socket can (or must) give up its IP address in favor of the host that imports the socket, it is much easier to add an "alias" IP address to the importing host. Further information about IP packet redirection is available in Ref. 1.

As shown in Figure 2, the migration consists of three phases: (i) export, (ii) negotiation, and (iii) import.

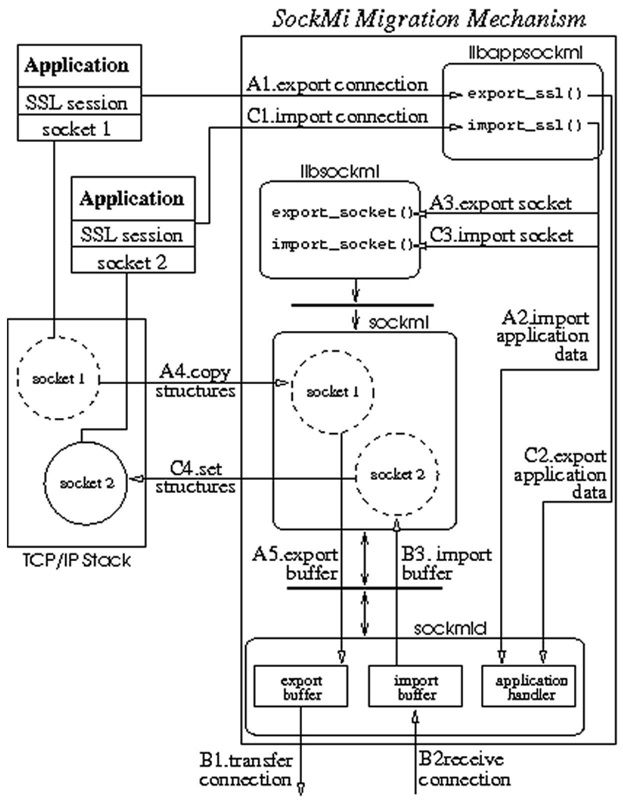

FIGURE 2: THE MECHANISM FOR THE MIGRATION OF AN SSL CONNECTION

The first phase is activated by the application that wants to export the secure connection. This phase can be summarized as follows:

A1. The exporting application calls the export_ssl() function.
A2. The libappsockmi library saves all required SSL information into a memory buffer. For this purpose, it uses the save_ssl() function. Then the library transfers the buffer to the daemon through an http message and the daemon saves the buffer in a local file that is transferred to the importing daemon during the negotiation phase (see B1 and B2).
A3. The library exports the associated socket using the export_socket() function.
A4. The module saves the state of the exporting socket into a memory buffer.
A5. Finally, the module exchanges information about the exporting socket with the daemon.

The negotiation phase can be summarized as follows:

B1/B2. The exporting daemon communicates with daemons running on other hosts in order to choose where to migrate the connection; the selected daemon receives all data about the exported secure connection.
B3. Then the importing daemon writes the state of the importing socket to the module internal buffers.

The last phase can be summarized as follows:

C1. The importing application calls the import_ssl() function specifying an array of requests.
C2. The libappsockmi library asks the daemon whether one matching connection is available. On success, the daemon's reply contains the data of the matching connection and the criteria used for importing the associated socket.
C3. The library imports the associated socket using the import_socket() function specifying the received criteria.
C4. Finally, the module restores the state of the importing socket into the kernel data structures.

SockMi is a mechanism, based on the cooperation of a kernel module and a daemon, that allows one to migrate an end of a secure connection to another Linux system running the same software. It is a complete solution for the migration of the network, transport, and application layers. It is compatible with OpenSSL version 0.9.7i and with Linux versions 2.4 and 2.6. The source code is available from http://sockmi.sourceforge.net.

Recently, we started to test the migration mechanism with real-world protocols and applications that use a secure connection (e.g., https, sftp, ssh). Another possible future activity is to add support for the authentication of the daemons that, in order to work in a potentially hostile environment, need to use secure channels for their communications. As to its porting to other UNIX-like operating systems, this depends mainly on the availability of the kernel source code. From this point of view, the porting looks possible, for instance, to systems belonging to the BSD family. OpenSSL is available on Windows systems, but at this time we have not yet analyzed if or how these systems can support the migration of TCP connections. This appears to be, in any case, a major effort since the implementation of sockets in Windows is significantly different with respect to UNIX-like operating systems.

REFERENCES

[1] M. Bernaschi, F. Casadei, and P. Tassotti, *SockMi: A solution for migrating TCP/IP connections*, 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP '07), pp. 221–228, 2007. Available from http://sockmi.sourceforge.net/public.html.

[2] T. Dierks and C. Allen, *The TLS Protocol*, RFC 2246, January 1999.

[3] http://openvpn.net.

[4] http://www.openssl.org.

[5] http://www.netfilter.org.