Greyhound
Solaris Kernel SSL Proxy

(Amendment of PSARC 2002/557 - SSL support for NL7C)

Overview

This document describes the architecture of an in-kernel SSL proxy module, interfacing the Solaris TCP/IP stack.
The proxy implements the SSL protocol in a bump-in-the-stack way. A non-SSL ready server program will be able to handle client requests over SSL. The SSL processing is done in the kernel, so that the server program communicates with the stack in "cleartext" (non SSL), and remote clients communicate with the stack over an encrypted/SSL connection.

Background, Motivation, and Changes from PSARC/2002/557

The case above introduced extensions to the Network Layer 7 Cache (NL7C, PSARC/2005/038) to enable it to serve HTTPS packets from the kernel web content cache, in addition to HTTP packets. The three main components of the project were:

1. a kernel module, kssl, that implements the server side of the SSL protocol, and maintains a table for the in kernel proxy server (keys, certificate, supported cipher suites, SSL port and proxy port)
2. an SMF based administrative interface for the kernel SSL proxy
3. changes to the NL7C code to call the kssl module entry points for establishing SSL sessions and processing HTTPS (SSL protected) packets.

The first two components are not changed by this amendment.

The last one had to be changed to offer in-kernel SSL for any TCP socket, that is **not** configured for NL7C.
The following reasons warranted the change:

- The limited usefulness of the kernel (NL7C) caching in cleartext some content that the customer wants to secure with SSL. The security concern was brought during the case's commitment review and and resulted in a TCR for putting strong language in the documentation warning about the exposure.
- NL7C doesn't support HTTP Chunking nor HTTP version 1.1, which are both required for webservers to publish SpecWEB benchmark results. The Ontario platform (first CMT Niagara based machine from Sun) has set aggressive performance goals, and must beat the competition SpecWEB2005 numbers of an Intel Xeon 3GHz by a factor 6x. The use of the the kernel SSL proxy boosts the performance of the SunONE Webserver by 20+ % over the pure userland SSL implementation, and allows us to meet that goal.

- The followup project, (code named Chihuahua, http://solsec.sfbay/chihuahua, PSARC/2005/596) is planning to deliver support for client side SSL in kernel, and user-level changes that leverage the kernel SSL proxy, for any (SSL) socket, not limited to HTTP.

The reminder of this document describes the new interaction between the TCP/IP stack and the kernel SSL proxy.

Requirements and Design Considerations

Performance impact of the handshake messages.

The SSL handshake involves exchanging between six and nine messages to setup the connection. Only *application_data* messages carry a payload useful for the application. Handshake messages should be processed internally by the SSL proxy. An application blocked in a read(), recv() or poll() shouldn't be woken up when a handshake, an alert, or partial data message arrives, just to get an EAGAIN then go back to sleep.

Cost of cryptographic operation – accountability and resource control

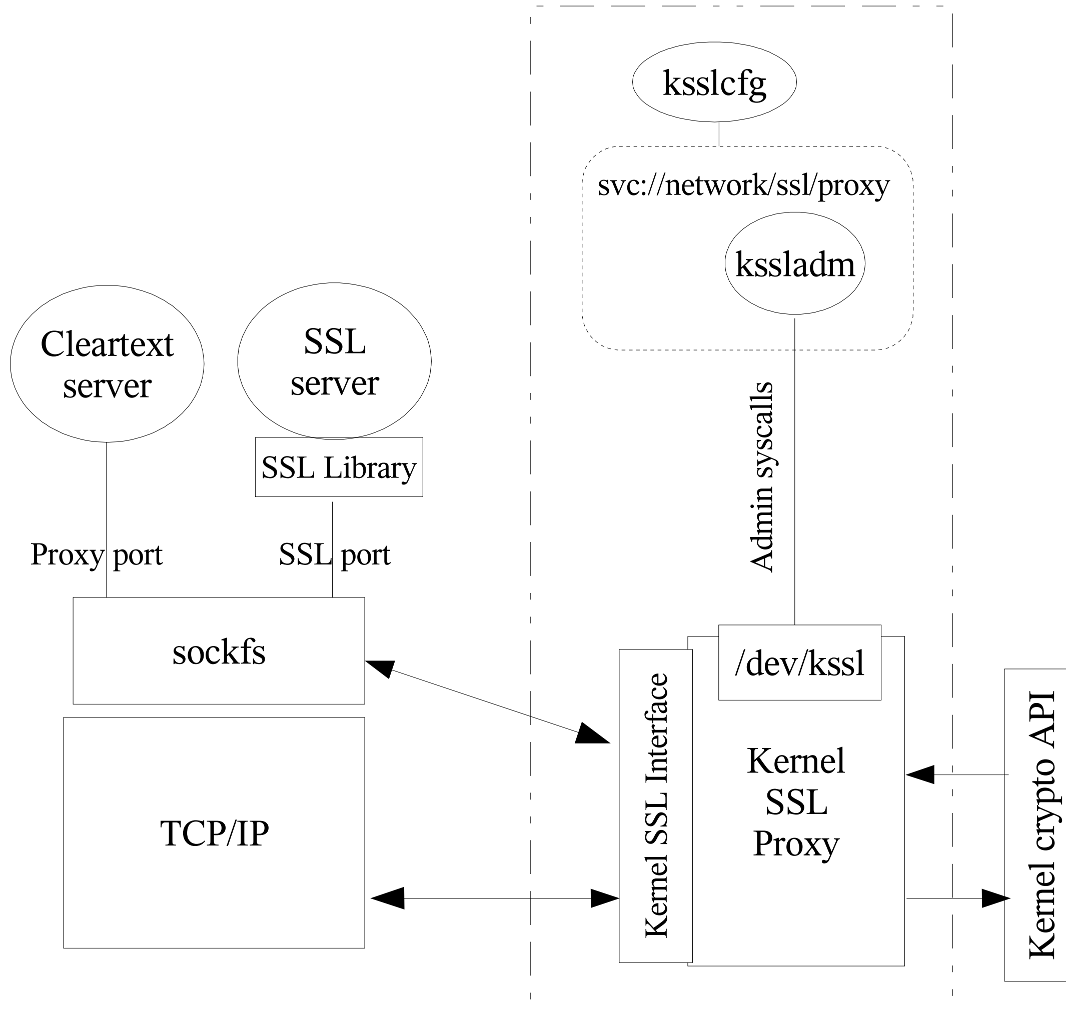
Around 15% of the overall CPU consumption during a web server run can be spent just doing RC4 and MD5 (see Apendix A). When SSL is implemented in a userland library, the process' microstate is accrued the time spent in cryptographic operations. When SSL is offloaded to a Kernel proxy, we need to preserve the same level of observability. The time spent in crypto should be reflected in prstat -m for example. Encryption and Message Digest functions need to be called from the context of the process they are being performed for, and not see their cost diluted as a non attributable overall system overhead.

Fall-back to user land

Some upper layer (above TCP) protocols have defined dedicated ports for transporting their SSL version, which are different from the cleartext (non SSL) version (e.g. TCP port 443 is reserved by IANA for HTTP over SSL, whereas the same protocol is transported over a different dedicated port; TCP 80, when not protected with SSL.)
The project should assure that the stack communicates with the remote SSL clients over the SSL port defined for the protocol being protected, and over a different port, called proxy port, with the local server.
The kernel SSL proxy doesn't implement the full set of ciphersuites defined by SSL/TLS, It is possible to have two instances of the server program for the upper layer protocol, one listening on the regular port and one listening on the SSL encapsulated version, and implementing the full set of ciphersuites in a user-level SSL library. In such configuration client connections requesting a ciphersuite not supported by the kernel proxy are forwarded to the user-level SSL server, as a fallback mechanism.

High Level Picture

This is a simplified picture of the new greyhound stack. As stated above, the components inside the dashed lined rectangle come from PSARC/2002/557, and are not affected by this amendment.



SSL processing (reminder from PSARC/2002/557)

The proxy maintains a table of the following entries:
{IP address, SSL port, proxy port, <SSL parameters>},
where the SSL parameters include the private key, server certificate, acceptable cipher suites, resumed sessions cache size and timeout.
A userland command, kssladm, configures the SSL proxy by adding/removing entries from that table (***kssl_entry_tab***), using ioctls to /dev/kssl. See ksslcfg(1M) and kssladm(1M) man pages.
Each ***kssl_entry_t*** in ***kssl_entry_tab*** also has a chained list of opaque handles to endpoints bound to the {IP adress, SSL port}. The sockfs code will use these endpoints later to fall back to userland. (see Setup & bind and Fallback path paragraphs below).

The state of an SSL connection is represented by a context structure, ***ssl_t***. It has a pointer to the ***kssl_entry_t*** parameters structure, as well as the chosen cipher and MAC algorithms, the key materials, and the evolving cryptographic context for the incoming and outgoing streams of SSL records.

The stack (TCP and sockfs) calls two main routines of the KSSL API:
- **kssl_handle_record()** which takes an SSL record and returns a decrypted, and MAC verified message carrying the payload, without the SSL header and tail.
- **kssl_build_record()** which takes a cleartext payload in an mblk and returns the SSL record.

Setup and bind

The socket structure (***sonode_t***) is extended to carry the following SSL related information:
- ***so_endpt_type***, which is an enumeration:
    - **KSSL_HAS_PROXY**: This endpoint is there only as a conduit to a userland fallback to handle SSL ciphers that the kernel proxy cannot. Listeners on this socket expect to receive SSL records.
    - **KSSL_IS_PROXY**: the kernel proxy communicates with the non-SSL server in cleartext over this socket.
    - **KSSL_NO_PROXY**: the kernel SSL proxy isn not configured to use this endpoint neither as a cleartext server nor as an SSL fallback.
- ***so_kssl_ent***: An opaque pointer to a ***kssl_entry_t*** structure, an entry in the proxy table.
- ***so_kssl_ctx***: An opaque pointer to the context of an SSL connection. It is present only for sockets marked **KSSL_IS_PROXY**.

The tcp structure (***tcp_t***) is also extended to carry a ***tcp_kssl_ent*** and a ***tcp_kssl_ctx*** akin to the socket's.

During a normal ***bind()***, the socket's bind routine ***sotpi_bindlisten***() builds a **T_bind_req** TPI message with the IP address and port passed to the system call, and sends it down to bind with to the transport layer. ***sotpi_bindlisten***() then waits for the replied **T_bind_ ack.** ***sotpi_bindlisten***() is changed to first call ***kssl_check_proxy***() which looks up the ***kssl_entry_tab*** for an entry that matches the requested bind IP address. There's one of three

case, depending on the requested bind port:

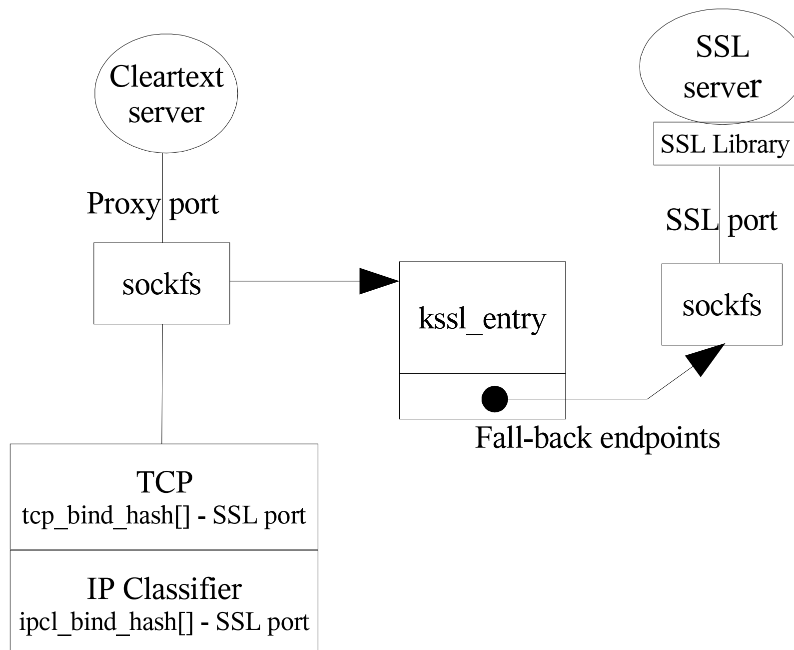
- it matches the proxy port of that ***kssl_entry*** - this binding end point will be acting as a proxy for the SSL port in that ***kssl_entty***. In that case three changes are made to the T_bind_req:
    - the SSL port is substituted to the bind (proxy) port, so that TCP and IP insert the corresponding tcp_t and conn_t as bound listeners to receive traffic destined to the SSL port.
    - the T_bind_req's primitive type is changed to **T_SSL_PROXY_BIND_REQ**
    - an mblk carrying the ***kssl_entry*** handle is apended to the **T_bind_req** mblk.

  kssl_check_proxy() then returns **KSSL_IS_PROXY**.

- it matches the SSL port of the ***kssl_entry*** - the process ***bind()***'ing will be sending/expecting SSL packets through this socket, and can act as a userland fallback for a proxy one. The endpoint handle (the socket's address, passed as an opaque pointer) is chained to the fallback endpoints of the ***kssl_entry***. The **T_bind_req** is changed into a **T_bind_ack** as if it came back from a successful bind with the transport layer, and ***kssl_check_proxy***() returns **KSSL_HAS_PROXY**.
  sotpi_bindlisten() then jumps to past its waking up after receiving the **T_bind_ack.**

- it matches none of the two ports - ***kssl_check_proxy***() returns **KSSL_NO_PROXY** and the rest of the bind operation continues as usual.

At the end of the bind() of the SSL and proxy listeners, the situation looks as follows:



The TCP bind routine will save the ***kssl_entry*** coming with the **T_SSL_PROXY_BIND_REQ** in the listener's tcp_t.
Consequently, later connection requests coming to the SSL port, and that can be handled by the kernel SSL proxy, will get an eager ***tcp_t*** already set up to communicate with the proxy listener process.
The substitution of the proxy port before binding with TCP/IP has the desirable security effect of rejecting any external connection requests directly targeting the proxy port, and attempting to bypass SSL and access the cleartext instance of the server.

Establishing SSL connections:

Connections to the SSL port come on the IP bound client attached to the TCP listener on the proxy port. The new eager has inherited the same ***kssl_entry*** of the parent listener so it is marked as "pending".
Normally, TCP sends a **T_conn_ind** message up-streams to listener socket at the end of the 3-way TCP handshake, to notify the coming of a new connection, and awaits the **T_conn_res** reply coming on the new accepted stream before starting to deliver packets to it.
For a pending eager, however, no **T_conn_ind** is sent up since we don't know yet whether this connection will be handled by the process bound with the TCP listener (the proxy one), or it will fallback to the userland SSL port listener.
tcp_rput_data() calls a new routine in TCP, ***tcp_kssl_input()*** which will handle every incoming packet. The first call for a pending ***tcp*** structure will allocate an embryonic SSL context (***ssl_t***) and attach it to ***tcp->tcp_ssl_ctx.*** The ***ssl_***t is initialized with the SSL parameters from the ***kssl_entry*** and will be filled in as the SSL handshake steps advance.
The first SSL handshake record is **ClientHello** offering a list of cipher suites the client is willing to negotiate.
If at least one the ciphersuites can be handled by the kernel SSL code, a **T_SSL_PROXY_CONN_IND** (a T_conn_ind extended with another mblk carrying the ***tcp_kssl_ctx***) is sent up to notify sockfs of the incoming connection. sockfs creates a new socket and attaches the ***tcp_kssl_ctx*** to its ***sonode_t.*** It is this ***kssl_ctx*** that will passed later to ***kssl_handle_record()*** and ***kssl_build_record()*** for data packets.

All subsequent incoming packet on this TCP connection will be re-routed to kssl_input() which will accumulate full SSL records and process the cleartext handshake ones.

If no ciphersuite offered in the **ClientHello** is supported by the kernel level SSL, ***tcp_kssl_input***() will enqueue that **ClientHello**, and send a **T_SSL_PROXY_CONN_IND** **without** a trailing mblk with the *kssl_ctx*_t. sockfs (strsock_proto()) will receive that T_SSL_PROXY_CONN_IND, recognize that it needs a userland endpoint bound to the SSL port to fallback to, so it calls ***kssl_find_fallback***(***kssl_ent***) and gets then the ***sonode*** for the fallback socket.
The **T_SSL_PROXY_CONN_IND** is turned back into a regular **T_CONN_IND,** and forwarded to the listening fallback socket, to complete a normal ***accept***(). From that moment on, kernel SSL is out of the picture for that stream. The remote client will be exchanging SSL records with the userland SSL server.

Data path

A new stream head hook is introduced, a function pointer (a ***msgfunc_t)*** called ***sd_wputdatafunc*** . ***strmakedata()*** (the routine that transfers data from user buffers into newly allocated message blocks) calls (*stp->sd_wputdatafunc)(stp->vp, ... , bp) if it is not NULL
When sockfs recognizes the case the endpoint will be acting as an SSL proxy, it sets the new hook to a function that calls ***kssl_build_record***() to encrypt and compute the MAC of the outgoing message first.
At the end of the TCP ***accept***() code, TCP sends an **M_SETOPTS** to convey the maximum block size, the write offset, etc ... to the stream head. The **M_SETOPTS**'s content will be changed to take into account the extra offset for the SSL header, the space to leave at the tail, and the maximum SSL record size. The ***stroptions*** structure is extended with a new member, ***so_tail***, that allows modules to tell the stream head how many bytes to set aside at the end of messages it allocates for outbound packets. A new ***so_flag***, called ***SO_TAIL***, is added to indicate the presence of an ***so_tail*** in the ***stroptions*** structure.

On the way in, TCP accumulates a full SSL record before sending it up. The reason for this is to avoid delivering partial rsoCecords causing blocked recv()/read()/poll() to wake up prematurely in case there isn't at least a full SSL record that can be decrypted.
Another new stream head hook is introduced, a ***msgfunc_t sd_rputdatafunct***
kstrgetmsg() calls that hook for any packet it retrieves from the stream head's read queue, and before copying out its content to user buffers.
Again, sockfs sets this hook to a function that calls ***kssl_handle_record***() which verifies the MAC, decrypts the payload, and strips out the SSL header and tail of the incoming record. The work is done in the context of the reading thread waking up from the ***strwaitq***().
kssl_handle_record() decrypts an entire record at a time. The application may ask for less than the full record, so there will be leftover, decrypted, to be put back in the stream head's read queue. ***kssl_handle_record***() marks the message with a new flag **MSGCOOKED** to avoid decrypting the leftover again.

Appendix A

The following is the break up of the cost of the top 30 functions in the Solaris kernel called during a SPECweb2005 Banking Workload with NCP and Greyhound 0809 sync'ed with snv21

- 18000 SW2005-SESSIONs
- Ontario (version 2.0 chip) 8x1.2Ghz 16GB RAM
- Sun One 7.0 MS5 Build as of 07/10

```
Functions sorted by metric: Exclusive KCPU Cycles

Excl. KCPU          Incl. KCPU          Name
Cycles              Cycles
     sec.       %        sec.       %
51167.692 100.00   51167.692 100.00   Total
15890.225  31.06   15890.546  31.06   USER_MODE
 9183.314  17.95    9183.534  17.95   IDLE
 4141.617   8.09    4141.617   8.09   arcfour_crypt
 3277.122   6.40    4183.897   8.18   MD5Transform
 2513.028   4.91    2583.407   5.05   mutex_vector_enter
 1646.272   3.22    1647.022   3.22   bcopy
  568.528   1.11     568.528   1.11   mutex_enter
  525.217   1.03    3994.104   7.81   ipge_start
  506.104   0.99     506.104   0.99   copyin
  445.301   0.87     445.301   0.87   hv_ncs_request
  358.501   0.70    2063.533   4.03   tcp_rput_data
  323.136   0.63     909.146   1.78   ipge_intr
  310.998   0.61    4866.304   9.51   tcp_send_data
  306.655   0.60    7606.501  14.87   putnext
  268.418   0.52     271.340   0.53   hat_getpfnum
  262.434   0.51     413.049   0.81   kmem_cache_alloc
  227.069   0.44     230.471   0.45   SHA1Transform
  205.143   0.40     205.143   0.40   pcacheset_cmp
  202.352   0.40     236.435   0.46   kmem_cache_free
  191.144   0.37     191.144   0.37   copyout
  190.213   0.37     190.213   0.37   mutex_exit
  161.203   0.32    4481.965   8.76   MD5Update
  143.771   0.28     201.591   0.39   utl0
  131.112   0.26    1508.095   2.95   kstrgetmsg
  129.210   0.25     259.091   0.51   allocb
  118.883   0.23     139.898   0.27   strpoll
  117.632   0.23     117.632   0.23   atomic_add_int_nv
  114.570   0.22    2608.074   5.10   tcp_send
  107.845   0.21    3406.193   6.66   ip_input
  106.915   0.21     224.497   0.44   tcp_output
  106.114   0.21     110.818   0.22   syscall_mstate
   95.097   0.19    2042.399   3.99   read
```