

path: root/src/curve25519.c

blob: 3f57f25d5741ac3ba9d203af3a7ba9d73dc1549b (plain)

```
1  /*
2   * curve25519.c - Curve25519 ECDH functions for key exchange
3   * curve25519-sha256@libssh.org and curve25519-sha256
4   *
5   * This file is part of the SSH Library
6   *
7   * Copyright (c) 2013      by Aris Adamantiadis <aris@badcode.be>
8   *
9   * The SSH Library is free software; you can redistribute it and/or modify
10  * it under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation, version 2.1 of the License.
12  *
13  * The SSH Library is distributed in the hope that it will be useful, but
14  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
15  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
16  * License for more details.
17  *
18  * You should have received a copy of the GNU Lesser General Public License
19  * along with the SSH Library; see the file COPYING. If not, write to
20  * the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston,
21  * MA 02111-1307, USA.
22  */
23
24  #include "config.h"
25
26  #include "libssh/curve25519.h"
27  #ifdef HAVE_CURVE25519
28
29  #ifdef WITH_NACL
30  #include "nacl/crypto_scalarmult_curve25519.h"
31  #endif
32
33  #include "libssh/ssh2.h"
34  #include "libssh/buffer.h"
35  #include "libssh/priv.h"
36  #include "libssh/session.h"
37  #include "libssh/crypto.h"
38  #include "libssh/dh.h"
39  #include "libssh/pki.h"
40  #include "libssh/bignum.h"
41
42  #ifdef HAVE_LIBCRYPTO
43  #include <openssl/err.h>
44  #endif
45
46  static SSH_PACKET_CALLBACK(ssh_packet_client_curve25519_reply);
47
48  static ssh_packet_callback dh_client_callbacks[] = {
49      ssh_packet_client_curve25519_reply
50  };
51
52  static struct ssh_packet_callbacks_struct ssh_curve25519_client_callbacks = {
53      .start = SSH2_MSG_KEX_ECDH_REPLY,
54      .n_callbacks = 1,
55      .callbacks = dh_client_callbacks,
56      .user = NULL
57  };
58
59  static int ssh_curve25519_init(ssh_session session)
60  {
61      int rc;
62      #ifdef HAVE_LIBCRYPTO
63      EVP_PKEY_CTX *pctx = NULL;
64      EVP_PKEY *pkey = NULL;
65      size_t pubkey_len = CURVE25519_PUBKEY_SIZE;
66      size_t pkey_len = CURVE25519_PRIVKEY_SIZE;
67
68      pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_X25519, NULL);
69      if (pctx == NULL) {
70          SSH_LOG(SSH_LOG_TRACE,
71              "Failed to initialize X25519 context: %s",
72              ERR_error_string(ERR_get_error(), NULL));
73          return SSH_ERROR;
74      }
75
76      rc = EVP_PKEY_keygen_init(pctx);
77      if (rc != 1) {
78          SSH_LOG(SSH_LOG_TRACE,
79              "Failed to initialize X25519 keygen: %s",
80              ERR_error_string(ERR_get_error(), NULL));
81          EVP_PKEY_CTX_free(pctx);
82          return SSH_ERROR;
83      }
84
85      rc = EVP_PKEY_keygen(pctx, &pkey);
```

```
86     EVP_PKEY_CTX_free(pctx);
87     if (rc != 1) {
88         SSH_LOG(SSH_LOG_TRACE,
89             "Failed to generate X25519 keys: %s",
90             ERR_error_string(ERR_get_error(), NULL));
91         return SSH_ERROR;
92     }
93
94     if (session->server) {
95         rc = EVP_PKEY_get_raw_public_key(pkey,
96             session->next_crypto->curve25519_server_pubkey,
97             &pubkey_len);
98     } else {
99         rc = EVP_PKEY_get_raw_public_key(pkey,
100             session->next_crypto->curve25519_client_pubkey,
101             &pubkey_len);
102     }
103
104     if (rc != 1) {
105         SSH_LOG(SSH_LOG_TRACE,
106             "Failed to get X25519 raw public key: %s",
107             ERR_error_string(ERR_get_error(), NULL));
108         EVP_PKEY_free(pkey);
109         return SSH_ERROR;
110     }
111
112     rc = EVP_PKEY_get_raw_private_key(pkey,
113         session->next_crypto->curve25519_privkey,
114         &key_len);
115
116     if (rc != 1) {
117         SSH_LOG(SSH_LOG_TRACE,
118             "Failed to get X25519 raw private key: %s",
119             ERR_error_string(ERR_get_error(), NULL));
120         EVP_PKEY_free(pkey);
121         return SSH_ERROR;
122     }
123
124     EVP_PKEY_free(pkey);
125 #else
126     rc = ssh_get_random(session->next_crypto->curve25519_privkey,
127         CURVE25519_PRIVKEY_SIZE, 1);
128     if (rc != 1) {
129         ssh_set_error(session, SSH_FATAL, "PRNG error");
130         return SSH_ERROR;
131     }
132
133     if (session->server) {
134         crypto_scalarmult_base(session->next_crypto->curve25519_server_pubkey,
135             session->next_crypto->curve25519_privkey);
136     } else {
137         crypto_scalarmult_base(session->next_crypto->curve25519_client_pubkey,
138             session->next_crypto->curve25519_privkey);
139     }
140 #endif /* HAVE_LIBCRYPTO */
141
142     return SSH_OK;
143 }
144
145 /** @internal
146  * @brief Starts curve25519-sha256@libssh.org / curve25519-sha256 key exchange
147  */
148 int ssh_client_curve25519_init(ssh_session session)
149 {
150     int rc;
151
152     rc = ssh_curve25519_init(session);
153     if (rc != SSH_OK) {
154         return rc;
155     }
156
157     rc = ssh_buffer_pack(session->out_buffer,
158         "bdP",
159         SSH2_MSG_KEX_ECDH_INIT,
160         CURVE25519_PUBKEY_SIZE,
161         (size_t)CURVE25519_PUBKEY_SIZE,
162         session->next_crypto->curve25519_client_pubkey);
163
164     if (rc != SSH_OK) {
165         ssh_set_error_oom(session);
166         return SSH_ERROR;
167     }
168
169     /* register the packet callbacks */
170     ssh_packet_set_callbacks(session, &ssh_curve25519_client_callbacks);
171     session->dh_handshake_state = DH_STATE_INIT_SENT;
172     rc = ssh_packet_send(session);
173
174     return rc;
175 }
176
177 void ssh_client_curve25519_remove_callbacks(ssh_session session)
178 {
179     ssh_packet_remove_callbacks(session, &ssh_curve25519_client_callbacks);
180 }
181
182 static int ssh_curve25519_build_k(ssh_session session)
183 {
184     ssh_curve25519_pubkey k;
```

```
185     EVP_PKEY_CTX *pctx = NULL;
186     EVP_PKEY *pkey = NULL, *pubkey = NULL;
187     size_t shared_key_len = sizeof(k);
188     int rc, ret = SSH_ERROR;
189
190     pkey = EVP_PKEY_new_raw_private_key(EVP_PKEY_X25519, NULL,
191                                         session->next_crypto->curve25519_privkey,
192                                         CURVE25519_PRIVKEY_SIZE);
193     if (pkey == NULL) {
194         SSH_LOG(SSH_LOG_TRACE,
195                 "Failed to create X25519 EVP_PKEY: %s",
196                 ERR_error_string(ERR_get_error(), NULL));
197         return SSH_ERROR;
198     }
199
200     pctx = EVP_PKEY_CTX_new(pkey, NULL);
201     if (pctx == NULL) {
202         SSH_LOG(SSH_LOG_TRACE,
203                 "Failed to initialize X25519 context: %s",
204                 ERR_error_string(ERR_get_error(), NULL));
205         goto out;
206     }
207
208     rc = EVP_PKEY_derive_init(pctx);
209     if (rc != 1) {
210         SSH_LOG(SSH_LOG_TRACE,
211                 "Failed to initialize X25519 key derivation: %s",
212                 ERR_error_string(ERR_get_error(), NULL));
213         goto out;
214     }
215
216     if (session->server) {
217         pubkey = EVP_PKEY_new_raw_public_key(EVP_PKEY_X25519, NULL,
218                                              session->next_crypto->curve25519_client_pubkey,
219                                              CURVE25519_PUBKEY_SIZE);
220     } else {
221         pubkey = EVP_PKEY_new_raw_public_key(EVP_PKEY_X25519, NULL,
222                                              session->next_crypto->curve25519_server_pubkey,
223                                              CURVE25519_PUBKEY_SIZE);
224     }
225     if (pubkey == NULL) {
226         SSH_LOG(SSH_LOG_TRACE,
227                 "Failed to create X25519 public key EVP_PKEY: %s",
228                 ERR_error_string(ERR_get_error(), NULL));
229         goto out;
230     }
231
232     rc = EVP_PKEY_derive_set_peer(pctx, pubkey);
233     if (rc != 1) {
234         SSH_LOG(SSH_LOG_TRACE,
235                 "Failed to set peer X25519 public key: %s",
236                 ERR_error_string(ERR_get_error(), NULL));
237         goto out;
238     }
239
240     rc = EVP_PKEY_derive(pctx, k, &shared_key_len);
241     if (rc != 1) {
242         SSH_LOG(SSH_LOG_TRACE,
243                 "Failed to derive X25519 shared secret: %s",
244                 ERR_error_string(ERR_get_error(), NULL));
245         goto out;
246     }
247     ret = SSH_OK;
248 out:
249     EVP_PKEY_free(pkey);
250     EVP_PKEY_free(pubkey);
251     EVP_PKEY_CTX_free(pctx);
252     if (ret == SSH_ERROR) {
253         return ret;
254     }
255 #else
256     if (session->server) {
257         crypto_scalarmult(k, session->next_crypto->curve25519_privkey,
258                          session->next_crypto->curve25519_client_pubkey);
259     } else {
260         crypto_scalarmult(k, session->next_crypto->curve25519_privkey,
261                          session->next_crypto->curve25519_server_pubkey);
262     }
263 #endif /* HAVE_LIBCRYPTO */
264
265     bignum_bin2bn(k, CURVE25519_PUBKEY_SIZE, &session->next_crypto->shared_secret);
266     if (session->next_crypto->shared_secret == NULL) {
267         return SSH_ERROR;
268     }
269
270 #ifdef DEBUG_CRYPT0
271     ssh_log_hexdump("Session server cookie",
272                     session->next_crypto->server_kex.cookie, 16);
273     ssh_log_hexdump("Session client cookie",
274                     session->next_crypto->client_kex.cookie, 16);
275     ssh_print_bignum("Shared secret key", session->next_crypto->shared_secret);
276 #endif
277
278     return 0;
279 }
280
281 /** @internal
282  * @brief parses a SSH_MSG_KEX_ECDH_REPLY packet and sends back
283  * a SSH_MSG_NEWKEYS
```

```
284  */
285  static SSH_PACKET_CALLBACK(ssh_packet_client_curve25519_reply){
286      ssh_string q_s_string = NULL;
287      ssh_string pubkey_blob = NULL;
288      ssh_string signature = NULL;
289      int rc;
290      (void)type;
291      (void)user;
292
293      ssh_client_curve25519_remove_callbacks(session);
294
295      pubkey_blob = ssh_buffer_get_ssh_string(packet);
296      if (pubkey_blob == NULL) {
297          ssh_set_error(session,SSH_FATAL, "No public key in packet");
298          goto error;
299      }
300
301      rc = ssh_dh_import_next_pubkey_blob(session, pubkey_blob);
302      SSH_STRING_FREE(pubkey_blob);
303      if (rc != 0) {
304          ssh_set_error(session,
305                        SSH_FATAL,
306                        "Failed to import next public key");
307          goto error;
308      }
309
310      q_s_string = ssh_buffer_get_ssh_string(packet);
311      if (q_s_string == NULL) {
312          ssh_set_error(session,SSH_FATAL, "No Q_S ECC point in packet");
313          goto error;
314      }
315      if (ssh_string_len(q_s_string) != CURVE25519_PUBKEY_SIZE){
316          ssh_set_error(session, SSH_FATAL, "Incorrect size for server Curve25519 public key: %d",
317                                (int)ssh_string_len(q_s_string));
318          SSH_STRING_FREE(q_s_string);
319          goto error;
320      }
321      memcpy(session->next_crypto->curve25519_server_pubkey, ssh_string_data(q_s_string), CURVE25519_PUBKEY_SIZE);
322      SSH_STRING_FREE(q_s_string);
323
324      signature = ssh_buffer_get_ssh_string(packet);
325      if (signature == NULL) {
326          ssh_set_error(session, SSH_FATAL, "No signature in packet");
327          goto error;
328      }
329      session->next_crypto->dh_server_signature = signature;
330      signature=NULL; /* ownership changed */
331      /* TODO: verify signature now instead of waiting for NEWKEYS */
332      if (ssh_curve25519_build_k(session) < 0) {
333          ssh_set_error(session, SSH_FATAL, "Cannot build k number");
334          goto error;
335      }
336
337      /* Send the MSG_NEWKEYS */
338      rc = ssh_packet_send_newkeys(session);
339      if (rc == SSH_ERROR) {
340          goto error;
341      }
342      session->dh_handshake_state = DH_STATE_NEWKEYS_SENT;
343
344      return SSH_PACKET_USED;
345
346  error:
347      session->session_state=SSH_SESSION_STATE_ERROR;
348      return SSH_PACKET_USED;
349  }
350
351  #ifdef WITH_SERVER
352
353  static SSH_PACKET_CALLBACK(ssh_packet_server_curve25519_init);
354
355  static ssh_packet_callback dh_server_callbacks[] = {
356      ssh_packet_server_curve25519_init
357  };
358
359  static struct ssh_packet_callbacks_struct ssh_curve25519_server_callbacks = {
360      .start = SSH2_MSG_KEX_ECDH_INIT,
361      .n_callbacks = 1,
362      .callbacks = dh_server_callbacks,
363      .user = NULL
364  };
365
366  /** @internal
367   * @brief sets up the curve25519-sha256@libssh.org kex callbacks
368   */
369  void ssh_server_curve25519_init(ssh_session session){
370      /* register the packet callbacks */
371      ssh_packet_set_callbacks(session, &ssh_curve25519_server_callbacks);
372  }
373
374  /** @brief Parse a SSH_MSG_KEXDH_INIT packet (server) and send a
375   * SSH_MSG_KEXDH_REPLY
376   */
377  static SSH_PACKET_CALLBACK(ssh_packet_server_curve25519_init){
378      /* ECDH keys */
379      ssh_string q_c_string = NULL;
380      ssh_string q_s_string = NULL;
381      ssh_string server_pubkey_blob = NULL;
382  }
```

```
383 /* SSH host keys (rsa, ed25519 and ecdsa) */
384 ssh_key privkey = NULL;
385 enum ssh_digest_e digest = SSH_DIGEST_AUTO;
386 ssh_string sig_blob = NULL;
387 int rc;
388 (void)type;
389 (void)user;
390
391 ssh_packet_remove_callbacks(session, &ssh_curve25519_server_callbacks);
392
393 /* Extract the client pubkey from the init packet */
394 q_c_string = ssh_buffer_get_ssh_string(packet);
395 if (q_c_string == NULL) {
396     ssh_set_error(session, SSH_FATAL, "No Q_C ECC point in packet");
397     goto error;
398 }
399 if (ssh_string_len(q_c_string) != CURVE25519_PUBKEY_SIZE){
400     ssh_set_error(session,
401         SSH_FATAL,
402         "Incorrect size for server Curve25519 public key: %zu",
403         ssh_string_len(q_c_string));
404     goto error;
405 }
406
407 memcpy(session->next_crypto->curve25519_client_pubkey,
408     ssh_string_data(q_c_string), CURVE25519_PUBKEY_SIZE);
409 SSH_STRING_FREE(q_c_string);
410
411 /* Build server's key pair */
412 rc = ssh_curve25519_init(session);
413 if (rc != SSH_OK) {
414     ssh_set_error(session, SSH_FATAL, "Failed to generate curve25519 keys");
415     goto error;
416 }
417
418 rc = ssh_buffer_add_u8(session->out_buffer, SSH2_MSG_KEX_ECDH_REPLY);
419 if (rc < 0) {
420     ssh_set_error_oom(session);
421     goto error;
422 }
423
424 /* build k and session_id */
425 rc = ssh_curve25519_build_k(session);
426 if (rc < 0) {
427     ssh_set_error(session, SSH_FATAL, "Cannot build k number");
428     goto error;
429 }
430
431 /* privkey is not allocated */
432 rc = ssh_get_key_params(session, &privkey, &digest);
433 if (rc == SSH_ERROR) {
434     goto error;
435 }
436
437 rc = ssh_make_sessionid(session);
438 if (rc != SSH_OK) {
439     ssh_set_error(session, SSH_FATAL, "Could not create a session id");
440     goto error;
441 }
442
443 rc = ssh_dh_get_next_server_publickey_blob(session, &server_pubkey_blob);
444 if (rc != 0) {
445     ssh_set_error(session, SSH_FATAL, "Could not export server public key");
446     goto error;
447 }
448
449 /* add host's public key */
450 rc = ssh_buffer_add_ssh_string(session->out_buffer,
451     server_pubkey_blob);
452 SSH_STRING_FREE(server_pubkey_blob);
453 if (rc < 0) {
454     ssh_set_error_oom(session);
455     goto error;
456 }
457
458 /* add ecdh public key */
459 q_s_string = ssh_string_new(CURVE25519_PUBKEY_SIZE);
460 if (q_s_string == NULL) {
461     ssh_set_error_oom(session);
462     goto error;
463 }
464
465 rc = ssh_string_fill(q_s_string,
466     session->next_crypto->curve25519_server_pubkey,
467     CURVE25519_PUBKEY_SIZE);
468 if (rc < 0) {
469     ssh_set_error(session, SSH_FATAL, "Could not copy public key");
470     goto error;
471 }
472
473 rc = ssh_buffer_add_ssh_string(session->out_buffer, q_s_string);
474 SSH_STRING_FREE(q_s_string);
475 if (rc < 0) {
476     ssh_set_error_oom(session);
477     goto error;
478 }
479 /* add signature blob */
480 sig_blob = ssh_srv_pki_do_sign_sessionid(session, privkey, digest);
481 if (sig_blob == NULL) {
```

```
482         ssh_set_error(session, SSH_FATAL, "Could not sign the session id");
483         goto error;
484     }
485
486     rc = ssh_buffer_add_ssh_string(session->out_buffer, sig_blob);
487     SSH_STRING_FREE(sig_blob);
488     if (rc < 0) {
489         ssh_set_error_oom(session);
490         goto error;
491     }
492
493     SSH_LOG(SSH_LOG_DEBUG, "SSH_MSG_KEX_ECDH_REPLY sent");
494     rc = ssh_packet_send(session);
495     if (rc == SSH_ERROR) {
496         return SSH_ERROR;
497     }
498
499     session->dh_handshake_state = DH_STATE_NEWKEYS_SENT;
500
501     /* Send the MSG_NEWKEYS */
502     rc = ssh_packet_send_newkeys(session);
503     if (rc == SSH_ERROR) {
504         goto error;
505     }
506
507     return SSH_PACKET_USED;
508 error:
509     SSH_STRING_FREE(q_c_string);
510     SSH_STRING_FREE(q_s_string);
511     ssh_buffer_reinit(session->out_buffer);
512     session->session_state=SSH_SESSION_STATE_ERROR;
513     return SSH_PACKET_USED;
514 }
515
516 #endif /* WITH_SERVER */
517
518 #endif /* HAVE_CURVE25519 */
```