```
/* $OpenBSD: jpake.c,v 1.6 2010/09/20 04:54:07 djm Exp $ */
/*
 * Copyright (c) 2008 Damien Miller.  All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose with or without fee is hereby granted, provided that the above
 * copyright notice and this permission notice appear in all copies.
 *
 * THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
 * WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
 * MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
 * ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
 * WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
 * ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
 * OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
 */

/*
 * Shared components of zero-knowledge password auth using J-PAKE protocol
 * as described in:
 *
 * F. Hao, P. Ryan, "Password Authenticated Key Exchange by Juggling",
 * 16th Workshop on Security Protocols, Cambridge, April 2008
 *
 * http://grouper.ieee.org/groups/1363/Research/contributions/hao-ryan-2008.pdf
 */

#include <sys/types.h>

#include <stdio.h>
#include <string.h>
#include <stdarg.h>

#include <openssl/bn.h>
#include <openssl/evp.h>

#include "xmalloc.h"
#include "ssh2.h"
#include "key.h"
#include "hostfile.h"
#include "auth.h"
#include "buffer.h"
#include "packet.h"
#include "dispatch.h"
#include "log.h"
#include "misc.h"

#include "jpake.h"
#include "schnorr.h"

#ifdef JPAKE

/* RFC3526 group 5, 1536 bits */
#define JPAKE_GROUP_G "2"
#define JPAKE_GROUP_P \
        "FFFFFFFFFFFFFFFFC90FDAA22168C234C4C6628B80DC1CD129024E088A67CC74" \
        "020BBEA63B139B22514A08798E3404DDEF9519B3CD3A431B302B0A6DF25F1437" \
        "4FE1356D6D51C245E485B576625E7EC6F44C42E9A637ED6B0BFF5CB6F406B7ED" \
        "EE386BFB5A899FA5AE9F24117C4B1FE649286651ECE45B3DC2007CB8A163BF05" \
        "98DA48361C55D39A69163FA8FD24CF5F83655D23DCA3AD961C62F356208552BB" \
        "9ED529077096966D670C354E4ABC9804F1746C08CA237327FFFFFFFFFFFFFFFF"

struct modp_group *
jpake_default_group(void)
```

```
{
        return modp_group_from_g_and_safe_p(JPAKE_GROUP_G, JPAKE_GROUP_P);
}

struct jpake_ctx *
jpake_new(void)
{
        struct jpake_ctx *ret;

        ret = xcalloc(1, sizeof(*ret));

        ret->grp = jpake_default_group();

        ret->s = ret->k = NULL;
        ret->x1 = ret->x2 = ret->x3 = ret->x4 = NULL;
        ret->g_x1 = ret->g_x2 = ret->g_x3 = ret->g_x4 = NULL;
        ret->a = ret->b = NULL;

        ret->client_id = ret->server_id = NULL;
        ret->h_k_cid_sessid = ret->h_k_sid_sessid = NULL;

        debug3("%s: alloc %p", __func__, ret);

        return ret;
}

void
jpake_free(struct jpake_ctx *pctx)
{
        debug3("%s: free %p", __func__, pctx);

#define JPAKE_BN_CLEAR_FREE(v)                  \
        do {                                    \
                if ((v) != NULL) {              \
                        BN_clear_free(v);       \
                        (v) = NULL;             \
                }                               \
        } while (0)
#define JPAKE_BUF_CLEAR_FREE(v, l)              \
        do {                                    \
                if ((v) != NULL) {              \
                        bzero((v), (l));        \
                        xfree(v);               \
                        (v) = NULL;             \
                        (l) = 0;                \
                }                               \
        } while (0)

        JPAKE_BN_CLEAR_FREE(pctx->s);
        JPAKE_BN_CLEAR_FREE(pctx->k);
        JPAKE_BN_CLEAR_FREE(pctx->x1);
        JPAKE_BN_CLEAR_FREE(pctx->x2);
        JPAKE_BN_CLEAR_FREE(pctx->x3);
        JPAKE_BN_CLEAR_FREE(pctx->x4);
        JPAKE_BN_CLEAR_FREE(pctx->g_x1);
        JPAKE_BN_CLEAR_FREE(pctx->g_x2);
        JPAKE_BN_CLEAR_FREE(pctx->g_x3);
        JPAKE_BN_CLEAR_FREE(pctx->g_x4);
        JPAKE_BN_CLEAR_FREE(pctx->a);
        JPAKE_BN_CLEAR_FREE(pctx->b);

        JPAKE_BUF_CLEAR_FREE(pctx->client_id, pctx->client_id_len);
        JPAKE_BUF_CLEAR_FREE(pctx->server_id, pctx->server_id_len);
        JPAKE_BUF_CLEAR_FREE(pctx->h_k_cid_sessid, pctx->h_k_cid_sessid_len);
        JPAKE_BUF_CLEAR_FREE(pctx->h_k_sid_sessid, pctx->h_k_sid_sessid_len);
```

```
#undef JPAKE_BN_CLEAR_FREE
#undef JPAKE_BUF_CLEAR_FREE

        bzero(pctx, sizeof(pctx));
        xfree(pctx);
}

/* dump entire jpake_ctx. NB. includes private values! */
void
jpake_dump(struct jpake_ctx *pctx, const char *fmt, ...)
{
        char *out;
        va_list args;

        out = NULL;
        va_start(args, fmt);
        vasprintf(&out, fmt, args);
        va_end(args);
        if (out == NULL)
                fatal("%s: vasprintf failed", __func__);

        debug3("%s: %s (ctx at %p)", __func__, out, pctx);
        if (pctx == NULL) {
                free(out);
                return;
        }

#define JPAKE_DUMP_BN(a)        do { \
                if ((a) != NULL) \
                        JPAKE_DEBUG_BN(((a), "%s = ", #a)); \
        } while (0)
#define JPAKE_DUMP_BUF(a, b)    do { \
                if ((a) != NULL) \
                        JPAKE_DEBUG_BUF((a, b, "%s", #a)); \
        } while (0)

        JPAKE_DUMP_BN(pctx->s);
        JPAKE_DUMP_BN(pctx->k);
        JPAKE_DUMP_BN(pctx->x1);
        JPAKE_DUMP_BN(pctx->x2);
        JPAKE_DUMP_BN(pctx->x3);
        JPAKE_DUMP_BN(pctx->x4);
        JPAKE_DUMP_BN(pctx->g_x1);
        JPAKE_DUMP_BN(pctx->g_x2);
        JPAKE_DUMP_BN(pctx->g_x3);
        JPAKE_DUMP_BN(pctx->g_x4);
        JPAKE_DUMP_BN(pctx->a);
        JPAKE_DUMP_BN(pctx->b);

        JPAKE_DUMP_BUF(pctx->client_id, pctx->client_id_len);
        JPAKE_DUMP_BUF(pctx->server_id, pctx->server_id_len);
        JPAKE_DUMP_BUF(pctx->h_k_cid_sessid, pctx->h_k_cid_sessid_len);
        JPAKE_DUMP_BUF(pctx->h_k_sid_sessid, pctx->h_k_sid_sessid_len);

        debug3("%s: %s done", __func__, out);
        free(out);
}

/* Shared parts of step 1 exchange calculation */
void
jpake_step1(struct modp_group *grp,
    u_char **id, u_int *id_len,
    BIGNUM **priv1, BIGNUM **priv2, BIGNUM **g_priv1, BIGNUM **g_priv2,
    u_char **priv1_proof, u_int *priv1_proof_len,
    u_char **priv2_proof, u_int *priv2_proof_len)
{
        BN_CTX *bn_ctx;

        if ((bn_ctx = BN_CTX_new()) == NULL)
                fatal("%s: BN_CTX_new", __func__);

        /* Random nonce to prevent replay */
        *id = xmalloc(KZP_ID_LEN);
        *id_len = KZP_ID_LEN;
        arc4random_buf(*id, *id_len);

        /*
         * x1/x3 is a random element of Zq
         * x2/x4 is a random element of Z*q
         * We also exclude [1] from x1/x3 candidates and [0, 1] from
         * x2/x4 candiates to avoid possible degeneracy (i.e. g^0, g^1).
         */
        if ((*priv1 = bn_rand_range_gt_one(grp->q)) == NULL ||
            (*priv2 = bn_rand_range_gt_one(grp->q)) == NULL)
                fatal("%s: bn_rand_range_gt_one", __func__);

        /*
         * client: g_x1 = g^x1 mod p / server: g_x3 = g^x3 mod p
         * client: g_x2 = g^x2 mod p / server: g_x4 = g^x4 mod p
         */
        if ((*g_priv1 = BN_new()) == NULL ||
            (*g_priv2 = BN_new()) == NULL)
                fatal("%s: BN_new", __func__);
        if (BN_mod_exp(*g_priv1, grp->g, *priv1, grp->p, bn_ctx) == -1)
                fatal("%s: BN_mod_exp", __func__);
        if (BN_mod_exp(*g_priv2, grp->g, *priv2, grp->p, bn_ctx) == -1)
                fatal("%s: BN_mod_exp", __func__);

        /* Generate proofs for holding x1/x3 and x2/x4 */
        if (schnorr_sign_buf(grp->p, grp->q, grp->g,
            *priv1, *g_priv1, *id, *id_len,
            priv1_proof, priv1_proof_len) != 0)
                fatal("%s: schnorr_sign", __func__);
        if (schnorr_sign_buf(grp->p, grp->q, grp->g,
            *priv2, *g_priv2, *id, *id_len,
            priv2_proof, priv2_proof_len) != 0)
                fatal("%s: schnorr_sign", __func__);

        BN_CTX_free(bn_ctx);
}

/* Shared parts of step 2 exchange calculation */
void
jpake_step2(struct modp_group *grp, BIGNUM *s,
    BIGNUM *mypub1, BIGNUM *theirpub1, BIGNUM *theirpub2, BIGNUM *mypriv2,
    const u_char *theirid, u_int theirid_len,
    const u_char *myid, u_int myid_len,
    const u_char *theirpub1_proof, u_int theirpub1_proof_len,
    const u_char *theirpub2_proof, u_int theirpub2_proof_len,
    BIGNUM **newpub,
    u_char **newpub_exponent_proof, u_int *newpub_exponent_proof_len)
{
        BN_CTX *bn_ctx;
        BIGNUM *tmp, *exponent;

        /* Validate peer's step 1 values */
        if (BN_cmp(theirpub1, BN_value_one()) <= 0)
                fatal("%s: theirpub1 <= 1", __func__);
        if (BN_cmp(theirpub1, grp->p) >= 0)
```

```
                fatal("%s: theirpub1 >= p", __func__);
        if (BN_cmp(theirpub2, BN_value_one()) <= 0)
                fatal("%s: theirpub2 <= 1", __func__);
        if (BN_cmp(theirpub2, grp->p) >= 0)
                fatal("%s: theirpub2 >= p", __func__);

        if (schnorr_verify_buf(grp->p, grp->q, grp->g, theirpub1,
            theirid, theirid_len, theirpub1_proof, theirpub1_proof_len) != 1)
                fatal("%s: schnorr_verify theirpub1 failed", __func__);
        if (schnorr_verify_buf(grp->p, grp->q, grp->g, theirpub2,
            theirid, theirid_len, theirpub2_proof, theirpub2_proof_len) != 1)
                fatal("%s: schnorr_verify theirpub2 failed", __func__);

        if ((bn_ctx = BN_CTX_new()) == NULL)
                fatal("%s: BN_CTX_new", __func__);

        if ((*newpub = BN_new()) == NULL ||
            (tmp = BN_new()) == NULL ||
            (exponent = BN_new()) == NULL)
                fatal("%s: BN_new", __func__);

        /*
         * client: exponent = x2 * s mod p
         * server: exponent = x4 * s mod p
         */
        if (BN_mod_mul(exponent, mypriv2, s, grp->q, bn_ctx) != 1)
                fatal("%s: BN_mod_mul (exponent = mypriv2 * s mod p)",
                    __func__);

        /*
         * client: tmp = g^(x1 + x3 + x4) mod p
         * server: tmp = g^(x1 + x2 + x3) mod p
         */
        if (BN_mod_mul(tmp, mypub1, theirpub1, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_mul (tmp = mypub1 * theirpub1 mod p)",
                    __func__);
        if (BN_mod_mul(tmp, tmp, theirpub2, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_mul (tmp = tmp * theirpub2 mod p)", __func__);

        /*
         * client: a = tmp^exponent = g^((x1+x3+x4) * x2 * s) mod p
         * server: b = tmp^exponent = g^((x1+x2+x3) * x4 * s) mod p
         */
        if (BN_mod_exp(*newpub, tmp, exponent, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_mul (newpub = tmp^exponent mod p)", __func__);

        JPAKE_DEBUG_BN((tmp, "%s: tmp = ", __func__));
        JPAKE_DEBUG_BN((exponent, "%s: exponent = ", __func__));

        /* Note the generator here is 'tmp', not g */
        if (schnorr_sign_buf(grp->p, grp->q, tmp, exponent, *newpub,
            myid, myid_len,
            newpub_exponent_proof, newpub_exponent_proof_len) != 0)
                fatal("%s: schnorr_sign newpub", __func__);

        BN_clear_free(tmp); /* XXX stash for later use? */
        BN_clear_free(exponent); /* XXX stash for later use? (yes, in conf) */

        BN_CTX_free(bn_ctx);
}

/* Confirmation hash calculation */
void
jpake_confirm_hash(const BIGNUM *k,
    const u_char *endpoint_id, u_int endpoint_id_len,
    const u_char *sess_id, u_int sess_id_len,
    u_char **confirm_hash, u_int *confirm_hash_len)
{
        Buffer b;

        /*
         * Calculate confirmation proof:
         *     client: H(k || client_id || session_id)
         *     server: H(k || server_id || session_id)
         */
        buffer_init(&b);
        buffer_put_bignum2(&b, k);
        buffer_put_string(&b, endpoint_id, endpoint_id_len);
        buffer_put_string(&b, sess_id, sess_id_len);
        if (hash_buffer(buffer_ptr(&b), buffer_len(&b), EVP_sha256(),
            confirm_hash, confirm_hash_len) != 0)
                fatal("%s: hash_buffer", __func__);
        buffer_free(&b);
}

/* Shared parts of key derivation and confirmation calculation */
void
jpake_key_confirm(struct modp_group *grp, BIGNUM *s, BIGNUM *step2_val,
    BIGNUM *mypriv2, BIGNUM *mypub1, BIGNUM *mypub2,
    BIGNUM *theirpub1, BIGNUM *theirpub2,
    const u_char *my_id, u_int my_id_len,
    const u_char *their_id, u_int their_id_len,
    const u_char *sess_id, u_int sess_id_len,
    const u_char *theirpriv2_s_proof, u_int theirpriv2_s_proof_len,
    BIGNUM **k,
    u_char **confirm_hash, u_int *confirm_hash_len)
{
        BN_CTX *bn_ctx;
        BIGNUM *tmp;

        if ((bn_ctx = BN_CTX_new()) == NULL)
                fatal("%s: BN_CTX_new", __func__);
        if ((tmp = BN_new()) == NULL ||
            (*k = BN_new()) == NULL)
                fatal("%s: BN_new", __func__);

        /* Validate step 2 values */
        if (BN_cmp(step2_val, BN_value_one()) <= 0)
                fatal("%s: step2_val <= 1", __func__);
        if (BN_cmp(step2_val, grp->p) >= 0)
                fatal("%s: step2_val >= p", __func__);

        /*
         * theirpriv2_s_proof is calculated with a different generator:
         * tmp = g^(mypriv1+mypriv2+theirpub1) = g^mypub1*g^mypub2*g^theirpub1
         * Calculate it here so we can check the signature.
         */
        if (BN_mod_mul(tmp, mypub1, mypub2, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_mul (tmp = mypub1 * mypub2 mod p)", __func__);
        if (BN_mod_mul(tmp, tmp, theirpub1, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_mul (tmp = tmp * theirpub1 mod p)", __func__);

        JPAKE_DEBUG_BN((tmp, "%s: tmp = ", __func__));

        if (schnorr_verify_buf(grp->p, grp->q, tmp, step2_val,
            their_id, their_id_len,
            theirpriv2_s_proof, theirpriv2_s_proof_len) != 1)
                fatal("%s: schnorr_verify theirpriv2_s_proof failed", __func__);

        /*
```

```
         * Derive shared key:
         *     client: k = (b / g^(x2*x4*s))^x2 = g^((x1+x3)*x2*x4*s)
         *     server: k = (a / g^(x2*x4*s))^x4 = g^((x1+x3)*x2*x4*s)
         *
         * Computed as:
         *     client: k = (g_x4^(q - (x2 * s)) * b)^x2 mod p
         *     server: k = (g_x2^(q - (x4 * s)) * b)^x4 mod p
         */
        if (BN_mul(tmp, mypriv2, s, bn_ctx) != 1)
                fatal("%s: BN_mul (tmp = mypriv2 * s)", __func__);
        if (BN_mod_sub(tmp, grp->q, tmp, grp->q, bn_ctx) != 1)
                fatal("%s: BN_mod_sub (tmp = q - tmp mod q)", __func__);
        if (BN_mod_exp(tmp, theirpub2, tmp, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_exp (tmp = theirpub2^tmp) mod p", __func__);
        if (BN_mod_mul(tmp, tmp, step2_val, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_mul (tmp = tmp * step2_val) mod p", __func__);
        if (BN_mod_exp(*k, tmp, mypriv2, grp->p, bn_ctx) != 1)
                fatal("%s: BN_mod_exp (k = tmp^mypriv2) mod p", __func__);

        BN_CTX_free(bn_ctx);
        BN_clear_free(tmp);

        jpake_confirm_hash(*k, my_id, my_id_len, sess_id, sess_id_len,
            confirm_hash, confirm_hash_len);
}

/*
 * Calculate and check confirmation hash from peer. Returns 1 on success
 * 0 on failure/mismatch.
 */
int
jpake_check_confirm(const BIGNUM *k,
    const u_char *peer_id, u_int peer_id_len,
    const u_char *sess_id, u_int sess_id_len,
    const u_char *peer_confirm_hash, u_int peer_confirm_hash_len)
{
        u_char *expected_confirm_hash;
        u_int expected_confirm_hash_len;
        int success = 0;

        /* Calculate and verify expected confirmation hash */
        jpake_confirm_hash(k, peer_id, peer_id_len, sess_id, sess_id_len,
            &expected_confirm_hash, &expected_confirm_hash_len);

        JPAKE_DEBUG_BUF((expected_confirm_hash, expected_confirm_hash_len,
            "%s: expected confirm hash", __func__));
        JPAKE_DEBUG_BUF((peer_confirm_hash, peer_confirm_hash_len,
            "%s: received confirm hash", __func__));

        if (peer_confirm_hash_len != expected_confirm_hash_len)
                error("%s: confirmation length mismatch (my %u them %u)",
                    __func__, expected_confirm_hash_len, peer_confirm_hash_len);
        else if (timingsafe_bcmp(peer_confirm_hash, expected_confirm_hash,
            expected_confirm_hash_len) == 0)
                success = 1;
        bzero(expected_confirm_hash, expected_confirm_hash_len);
        xfree(expected_confirm_hash);
        debug3("%s: success = %d", __func__, success);
        return success;
}

/* XXX main() function with tests */

#endif /* JPAKE */
```