

### **Digitālā paraksta ģenerācijas funkcija (izmantojot OpenSSL bibliotēku)**

```
int sign_(const char *data, const char *key, TLogClient& log)
{
    const char *ptr= strlen(key) > 0 ? key : cert;
    log.add("in: KEY=%s\n", ptr);

    FILE *fp= fopen(ptr, "r");
    if (!fp) {
        log.add("fopen '%s'= %d\n", ptr, errno);
        return 1;
    }

    EVP_PKEY *pkey= (EVP_PKEY*)PEM_ASN1_read((char *(*))d2i_PrivateKey,
    PEM_STRING_EVP_PKEY, fp, NULL, NULL);

    fclose (fp);

    if (!pkey) {
        log.add ("can't load private key from '%s'\n", ptr);
        return 2;
    }

    EVP_MD_CTX md_ctx;
    unsigned char sig_buf[4096];
    unsigned int sig_len= sizeof(sig_buf);

    EVP_SignInit(&md_ctx, EVP_sha1());
    EVP_SignUpdate(&md_ctx, (unsigned char *)data, strlen(data));
    int ret= EVP_SignFinal(&md_ctx, sig_buf, &sig_len, pkey);
    EVP_PKEY_free(pkey);

    b64_ntop (sig_buf, sig_len, b64_enc_mac, sizeof (b64_enc_mac));
    return 0;
}
```

### **Digitālā paraksta pārbaudes funkcija (izmantojot OpenSSL bibliotēku)**

```
int check_(const char *data, const char *mac, const char *certfile, TLogClient& log)
{
    char mac_b64_dec[0x1000];
    int len= b64_pton(mac, mac_b64_dec, sizeof(mac_b64_dec));
    FILE *fp= fopen(certfile, "r");
    if (!fp) {
        return 2;
    }
    X509 *x509= (X509 *)PEM_ASN1_read((char *(*))d2i_X509, PEM_STRING_X509, fp,
    NULL, NULL);
    fclose(fp);

    if (!x509) {
        return 3;
    }
    EVP_PKEY *pkey= X509_extract_key (x509);
    if (!pkey) {
        return 4;
    }
}
```

```

    EVP_MD_CTX md_ctx;

    EVP_VerifyInit(&md_ctx, EVP_sha1());
    EVP_VerifyUpdate(&md_ctx, (unsigned char *) data, strlen (data));
    int ret= EVP_VerifyFinal(&md_ctx, (unsigned char *)mac_b64_dec, len, pkey);    //
1 = ok
    EVP_PKEY_free (pkey);

    if (ret == 1) { // ok
        return 0;
    }
    return 1;
}

```

### Base64 kodēšanas funkcijas

```

#include <ctype.h>
#include <string.h>

static const char Base64[]=
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
static const char Pad64= '=';

int b64_pton (const char *src, char *target, int targsize)
{
    int tarindex, state, ch;
    char *pos;

    state = 0;
    tarindex = 0;

    while ((ch = *src++) != '\0')
    {
        if (isspace(ch)) /* Skip whitespace anywhere. */
            continue;

        if (ch == Pad64)
            break;

        pos = strchr(Base64, ch);
        if (pos == 0) /* A non-base64 character. */
            return -1;

        switch (state)
        {
            case 0:
                if (target)
                {
                    if (tarindex >= targsize)
                        return -1;
                    target[tarindex] = (pos - Base64) << 2;
                }
                state= 1;
                break;
            case 1:
                if (target)

```

```

        {
            if (tarindex + 1 >= targsize)
                return -1;
            target[tarindex] |= (pos - Base64) >> 4;
            target[tarindex+1] = ((pos - Base64) & 0x0f) << 4;
        }
        tarindex++;
        state = 2;
        break;
    case 2:
        if (target)
        {
            if (tarindex + 1 >= targsize)
                return -1;
            target[tarindex] |= (pos - Base64) >> 2;
            target[tarindex+1] = ((pos - Base64) & 0x03) << 6;
        }
        tarindex++;
        state = 3;
        break;
    case 3:
        if (target)
        {
            if (tarindex >= targsize)
                return (-1);
            target[tarindex] |= (pos - Base64);
        }
        tarindex++;
        state = 0;
        break;
    default:
        return -1;
    }
}

/*
 * We are done decoding Base-64 chars. Let's see if we ended
 * on a byte boundary, and/or with erroneous trailing characters.
 */

if (ch == Pad64)          /* We got a pad char. */
{
    ch = *src++;          /* Skip it, get next. */
    switch (state)
    {
        case 0:          /* Invalid = in first position */
        case 1:          /* Invalid = in second position */
            return -1;

        case 2:          /* Valid, means one byte of info */
            /* Skip any number of spaces. */
            for (; ch != '\0'; ch = *src++)
                if (!isspace(ch))
                    break;
            /* Make sure there is another trailing = sign. */
            if (ch != Pad64)
                return -1;
    }
}

```

```

        ch = *src++;          /* Skip the = */
        /* Fall through to "single trailing =" case. */
        /* FALLTHROUGH */

    case 3:          /* Valid, means two bytes of info */
        /*
         * We know this char is an =. Is there anything but
         * whitespace after it?
         */
        for (; ch != '\0'; ch = *src++)
            if (!isspace(ch))
                return -1;

        /*
         * Now make sure for cases 2 and 3 that the "extra"
         * bits that slopped past the last full byte were
         * zeros. If we don't check them, they become a
         * subliminal channel.
         */
        if (target && target[tarindex] != 0)
            return -1;
    }
}
else
{
    /*
     * We ended by seeing the end of the string. Make sure we
     * have no partial bytes lying around.
     */
    if (state != 0)
        return -1;
}

return tarindex;
}

int b64_ntop(const unsigned char *src, int srclength, char *target, int targsize)
{
    int datalength, i;
    unsigned char input[3];
    unsigned char output[4];

    datalength= 0;

    while (2 < srclength)
    {
        input[0]= *src++;
        input[1]= *src++;
        input[2]= *src++;
        srclength-= 3;

        output[0]= input[0] >> 2;
        output[1]= ((input[0] & 0x03) << 4) + (input[1] >> 4);
        output[2]= ((input[1] & 0x0f) << 2) + (input[2] >> 6);
        output[3]= input[2] & 0x3f;

        if (datalength + 4 > targsize)

```

```

        return -1;

        target[datalength++] = Base64[output[0]];
        target[datalength++] = Base64[output[1]];
        target[datalength++] = Base64[output[2]];
        target[datalength++] = Base64[output[3]];
    }

    /* Now we worry about padding. */
    if (0 != srclength)
    {
        /* Get what's left. */
        input[0] = input[1] = input[2] = '\0';
        for (i = 0; i < srclength; i++)
            input[i] = *src++;

        output[0] = input[0] >> 2;
        output[1] = ((input[0] & 0x03) << 4) + (input[1] >> 4);
        output[2] = ((input[1] & 0x0f) << 2) + (input[2] >> 6);

        if (datalength + 4 > targsize)
            return -1;

        target[datalength++] = Base64[output[0]];
        target[datalength++] = Base64[output[1]];
        if (srclength == 1)
            target[datalength++] = Pad64;
        else
            target[datalength++] = Base64[output[2]];

        target[datalength++] = Pad64;
    }
    if (datalength >= targsize)
        return -1;

    target[datalength] = '\0';          /* Returned value doesn't count \0. */
    return datalength;
}

```