guardtime

KSI C SDK Tutorial

Disclaimer

The SDK is still under development and the following tutorial may not yet describe the final version.

Prerequisites

To use this tutorial you'll need to have at least basic knowledge of the C programming language and need to be comfortable with the linux command line. This document does not cover the installation of the required software:

- KSI C SDK
- Curl
- OpenSSL

The KSI Context

The KSI context is the central object while using the KSI C SDK. The context and all other objects created using it may not be shared between threads, but a single thread may have many KSI contexts. As most of the functionality of the SDK, the context (`KSI_CTX`) is declared in the `ksi/ksi.h` header file, which has to be included. The two following functions are used to create a new instance of the context and to free up the memory:

```
int KSI_CTX_new(KSI_CTX **ctx);
void KSI_CTX_free(KSI_CTX *ctx);
```

It is the callers responsibility to call the free-method on the context and to make sure there are no KSI functions are performed on it or any other object created using it afterwards.

Data hashing

As the signing mechanism does not sign the documents directly but instead signs the calculated imprint[1] of the document, we will start with by calculating it. The first example will be a simple program, which hashes a single message and outputs the imprint in hexadecimal to the console.

NB! For clarity we have omitted most of the error handling.

First we need add couple of include files. As `stdio.h` and `string.h` should be familiar to the reader, the `ksi/ksi.h` is needed to use the SDK.

```
#include<stdio.h>   /* For printf() */
#include<string.h>  /* For strlen() */
#include<ksi/ksi.h> /* KSI base functions */
```

The second step will be to declare and initialize the KSI context.

```
KSI_CTX *ctx = NULL;
KSI_CTX_new(&ctx);
```

To start calculating an imprint or hash value, we need a pointer to a `KSI_DataHasher`. This object holds the state of the calculation – not all the hashed data needs to be available at once as it can be added incrementally. When there is no more data to hashed, the hasher can be closed with `KSI_DataHash_close`, which returns a pointer to a `KSI_DataHash`.

```
KSI_DataHasher *hasher = NULL;
KSI_DataHash *hash = NULL;
char *data1 = "Hello ";
char *data2 = "nerd!";

/* Open a new hasher, using SHA2-256 algorithm. */
KSI_DataHasher_open(ctx, KSI_HASHALG_SHA2_256, &hasher);

/* Add some data to the hasher. */
KSI_DataHasher_add(hasher, data1, strlen(data1));
KSI_DataHasher_add(hasher, data2, strlen(data2));

/* Close the hasher, and get the hash */
KSI_DataHasher_close(hasher, &hash);
```

The hash object will be an input for the actual signing, which we will cover below. But if you need to access the actual imprint value, we can extract a constant pointer to it. The fact that the pointer is a constant, means that the user does not have to (actually even may not) free it. We will use `KSI_DataHash_getImptint()` to extract the binary value of the imprint. Now it is possible to output the value in hexadecimal to the console.

1 An imprint is a binary hash value preceded with a single byte for hash algorithm id.

```
const unsigned char *raw = NULL;
unsigned len;
unsigned i;

KSI_DataHash_getImprint(hash, &raw, &len);
printf("Hash: ");
for (i = 0; i < len; i++) {
    printf("%02x", raw[i]);
}
printf("\n");
```

Now all thaths left to do is free the allocated memory.

```
KSI_DataHasher_free(hasher);
KSI_DataHash_free(hash);
KSI_CTX_free(ctx);
```

Signing the Hash

In this part we will assume, that we have a valid pointer to `KSI_CTX *ctx` and to a `KSI_DataHash *hash`. Before we can start singing anything we need to configure the context's network client. A network client is an abstraction layer to communicate with the gateway (or aggregator). A network client can send signing and extending request and receive the responses and download the publications file.

In this tutorial we will be using the HTTP network provider (`KSI_HttpClient`). The client is described in `ksi/net_http.h` header file, which needs to be included. The http client's main functions are:

```
int KSI_HttpClient_new(
    KSI_CTX *ctx, KSI_HttpClient **http);

void KSI_HttpClient_free(KSI_HttpClient *http);

int KSI_HttpClient_setPublicationUrl(
    KSI_HttpClient *client, const char *val);

int KSI_HttpClient_setExtender(
    KSI_HttpClient *client, const char *url,
    const char *user, const char *pass);

int KSI_HttpClient_setAggregator(
    KSI_HttpClient *client, const char *url,
    const char *user, const char *pass);
```

We will configure the network client only for signing and extending using “anon” for the user name and for password. By default the publications file will be downloaded from the Guardtime public location[2].

```
KSI_HttpClient *http = NULL;
KSI_HttpClient_new(ctx, &http);

/* Configure the signing service. */
KSI_HttpClient_setAggregator(
    http, “192.168.11.123”, “anon”, “anon”);

/* Configure the extending service. */
KSI_HttpClient_setExtender(
    http, “192.168.11.123:8081”, “anon”, “anon”);
```

Now, the http client of the context needs to be updated with the newly created context. The implicit cast to `KSI_NetworkClient *` is added to avoid compiler warnings.

```
/* Set the network client. */
KSI_setNetworkProvider(
    ctx, (KSI_NetworkClient *)http);
```

By calling the setter method, the user looses the ownership of the http client and may not free it – the context will take care of it.

Finally, we can create our first KSI signature. For a signature we need a pointer to `KSI_Signature`. It is created while signing and the caller is responsible for freeing the memory with the function `KSI_Signature_free()`. In the following we will check the return value of the function, as there may be several reasons why signing may fail (gateway is down, network problems).

```
KSI_Signature *signature = NULL;
int res;
/* Sign and capture the error code. */
res = KSI_createSignature(ctx, hash, &signature);
if (res != KSI_OK) {
    fprintf(stderr, “Unable to sign\n”);
}
```

Assuming everything went well, and we got a valid response from the gateway, the least we can do, is to save the signature into a file. For this we need to serialize the signature. Note, that this time the output parameter is not constant, and the caller is responsible for freeing the memory.

```
unsigned char *rawSignature = NULL;
unsigned rawSignature_len;
FILE *f = NULL;

/* Serialize the signature. */
```

2 `http://verify.guardtime.com/ksi-publications.bin`

```
KSI_Signature_serialize(
    signature, &rawSignature, &rawSignature_len);

/* Write the raw signature into a file. */
f = fopen ("signature.ksi");
fwrite(rawSignature, rawSignature_len, f);
fclose(f);

/* Cleanup. */
KSI_free(rawSignature);
```

Now we have a file called `signature.ksi` which contains a KSI signature. We can observe the contents of is using gttlvdump, which is a simple tool for viewing TLV[3] encoded data.

Parsing a Signature

In this section we are going to read the signature from file and parse it into a `KSI_Signature`. We are going to use a statically allocated buffer whith size `0xffff + 4`, which is the maximum size of a single (uni-) signature, due to the TLV encoding.

```
KSI_Signature *signature = NULL;
FILE *f = NULL;
unsigned char buffer[0xffff + 4];
unsigned len;

/* Read the file into buffer. */
f =  fopen("signature.ksi", "rb");
len = fread(buffer, 1, sizeof(buffer), f);

/* Parse the signature. */
KSI_Signature_parse(ctx, buffer, len, &signature);
```

Verifying

The signature itself can be verified without having the original document (or the hash/imprint), but it is not required. The process of verifying the document verifies the signature automatically.

Again, we will assume we have a signature `KSI_Signature *signature` and a `KSI_DataHash *hash`, which is created using the document we wish to verify. The next example is the easiest way to verify a document.

```
int res;
```

3 TLV - Type Length Value.

```
res = KSI_Signature_verifyDataHash(sig, ctx, hash);
switch (res) {
    case KSI_OK:
        printf("Document and signature verifyed.\n");
        break;
    case KSI_VERIFICATION_FAILURE:
        printf("Verification failed.\n");
        break;
    default:
        fprintf(stderr, "Technical error (%s)\n",
            KSI_getErrorString(res));
}
```

The following function can be used to verify the signature without having the initial document hash. It acts the same way as the function in the previous example, but does not require the document hash as a parameter.

```
int KSI_Signature_verify(KSI_Signature *sig, KSI_CTX *ctx);
```

Extending the Signature

While extending an existing signature object, a new signature object is created, thus the original is preserved. The following example illustrates the process. It is the responsibility of the caller to free the original and the extended signature.

```
int res = KSI_extendSignature(ctx, sig, &ext);
if (res != KSI_OK) {
    if (res == KSI_EXTEND_NO_SUITABLE_PUBLICATION) {
        printf("No suitable publication to extend to.\n");
    } else {
        fprintf(stderr, "Unable to extend signature.\n");
    }
}
```