Provider Authors: Writing a Provider Skeleton

December 2023

Richard Levitte

Skeleton: Agenda

1. Fundamentals
   a. What do providers provide?
   b. Core types
2. A first attempt - a minimum nonsense provider
3. The same provider, but built-in
4. Handling parameter requests
5. Query function
6. Getting config data

Skeleton: Agenda (cont.)

7. Provider context
8. Error handling
9. Provider library context - self-contained model
10. Provider library context - inheriting child model

Skeleton: Fundamentals

- Providers have fundamentally very few dependencies on OpenSSL
  - No obligation to use OpenSSL library functions
  - No obligation to any specific C language level
  - No obligation to be written in C, even
  - Minimum dependency: `<openssl/core.h>`,`<openssl/core_dispatch.h>`
- Providers must be written in languages that are ABI compatible with C:
  - Simple symbols
  - Same function argument passing
  - Compatible structure specifications

Skeleton: What do providers provide?

- Implementations of algorithms, divided into operations (digest, cipher, signature, mac, …)
- A set of provider-wide functions

- To tie this together, OpenSSL defines (in `<openssl/core_dispatch.h>`):
  - Identifiers and function signature types for supported provider-wide
  - Identifiers and function signature types for OpenSSL "upcalls"
  - Identifiers for supported operations
  - Identifiers and function signatures for supported operation functions

Skeleton: Core types

`<openssl/core.h>` defines a set of core types:

OSSL_ALGORITHM - a structure combining a set of names, a property definition, and an implementation (an array of **OSSL_DISPATCH**).

OSSL_DISPATCH - a structure combining a function identity with a function pointer.

OSSL_ITEM - a generic structure combining an identity with a pointer.

OSSL_PARAM - a generic parameter structure, used to pass optional data to or from provider implementations.

Skeleton: A few words on the C language

There’s code written in C in the upcoming slides.

They have all been written for C99, and take full advantage of some of the language features of that language level in a few places.

If you plan on writing providers with an older C language level, you may have to adjust. If you plan on writing providers with another language, you may have to adjust a lot.

Skeleton: A minimum nonsense provider [1/2]

`provfns` is the set of functions passed back to libcrypto, currently empty.

It gets passed back to libcrypto via `*out`.

```c
#include <stdio.h>
#include <openssl/core.h>
#include <openssl/core_dispatch.h>

static const OSSL_DISPATCH provfns[] = {
    { 0, NULL }
};

/* Check function signature */
OSSL_provider_init_fn OSSL_provider_init;

int OSSL_provider_init(const OSSL_CORE_HANDLE *handle,
                       const OSSL_DISPATCH *in,
                       const OSSL_DISPATCH **out,
                       void **provctx)
{
    fprintf(stderr, "nonsense: Hello, world!\n");
    *out = provfns;
    return 1;
}
```

Skeleton: A minimum nonsense provider [2/2]

Build it:

```
$ clang -fPIC -c -o helloworld.o helloworld.c
$ clang -shared -Wl,--export-dynamic-symbol=OSSL_provider_init \
        -o helloworld.so helloworld.o
```

Try it:

```
$ OPENSSL_MODULES=. openssl list -provider helloworld -verbose
nonsense: Hello, world!
```

Skeleton: A built-in nonsense provider [1/2]

Using the exact same provider code, we can make it built in.

To do this, we need an app that registers it.

```c
#include <openssl/core.h>
#include <openssl/provider.h>

OSSL_provider_init_fn my_prov_init;
int main()
{
    OSSL_PROVIDER *prov = NULL;

    if (OSSL_PROVIDER_add_builtin(NULL, "nonsense",
                                  &my_prov_init))
        prov = OSSL_PROVIDER_load(NULL, "nonsense");
    if (prov != NULL)
        OSSL_PROVIDER_unload(prov);
}
```

Skeleton: A built-in nonsense provider [2/2]

Build them (the trick lies in the definition of `OSSL_provider_init`):

```
$ clang -c -o myapp.o myapp.c
$ clang -DOSSL_provider_init=my_prov_init -c -o helloworld.o helloworld.c
$ clang myapp.o helloworld.o  -lcrypto -o myapp
```

Try it:

```
$ ./myapp
nonsense: Hello, world!
```

Skeleton: Handling parameter requests [1/5]

Handling parameters requests is a bit more elaborate, and it can be done both with and without the help of OpenSSL library functions.

First up, with OpenSSL library functions!

(Only changes to the nonsense providers are shown)

```c
#include <openssl/core_names.h>
#include <openssl/params.h>

// ...

static OSSL_FUNC_provider_gettable_params_fn gettable_params;
static OSSL_FUNC_provider_get_params_fn get_params;
static const OSSL_PARAM *gettable_params(void *provctx)
{
    static const OSSL_PARAM params[] = {
        OSSL_PARAM_utf8_ptr(OSSL_PROV_PARAM_BUILDINFO, NULL, 0),
        OSSL_PARAM_END
    };
    return params;
}
static int get_params(void *provctx, OSSL_PARAM *params)
{
    OSSL_PARAM *p;

    p = OSSL_PARAM_locate(params, OSSL_PROV_PARAM_BUILDINFO);
    if (p != NULL) {
        if (!OSSL_PARAM_set_utf8_ptr(p, "commit " COMMITID))
            return 0;
    }
    return 1;
}
```

Skeleton: Handling parameter requests [2/5]

If you don't want to use OpenSSL library functions, `gettable_params` and `get_params` without them.

```c
static OSSL_FUNC_provider_gettable_params_fn gettable_params;
static OSSL_FUNC_provider_get_params_fn get_params;
static const OSSL_PARAM *gettable_params(void *provctx)
{
    static const OSSL_PARAM params[] = {
        { "buildinfo", OSSL_PARAM_UTF8_PTR, NULL, 0, 0 },
        { NULL, 0, NULL, 0, 0 }
    };
    return params;
}
static int get_params(void *provctx, OSSL_PARAM *params)
{
    for (; params != NULL && params->key != NULL; params++) {
        if (strcasecmp(params->key, "buildinfo") == 0) {
            if (params->data_type == OSSL_PARAM_UTF8_PTR) {
                const char **r = params->data;
                *r = "commit " COMMITID;
                params->return_size = strlen(*r);
            } else {
                return 0;
            }
        }
    }
    return 1;
}
```

Skeleton: Handling parameter requests [3/5]

All we need to do now is to hook in `gettable_params` and `get_params`.

```
diff --git a/helloworld.c b/helloworld.c
--- a/helloworld.c
+++ b/helloworld.c
@@ -56,4 +56,6 @@

 static const OSSL_DISPATCH provfns[] = {
+    { OSSL_FUNC_PROVIDER_GETTABLE_PARAMS, (void (*)(void))gettable_params },
+    { OSSL_FUNC_PROVIDER_GET_PARAMS, (void (*)(void))get_params },
     { 0, NULL }
 };
```

Skeleton: Handling parameter requests [4/5]

Build it:

```
$ clang -fPIC -DCOMMITID='"ca5eb625ac3ce2b8c99844d93656a39bf7e5980c"' -c \
        -o helloworld.o helloworld.c
$ clang -shared -Wl,--export-dynamic-symbol=OSSL_provider_init \
        -o helloworld.so helloworld.o
```

Skeleton: Handling parameter requests [5/5]

Try it:

```
$ OPENSSL_MODULES=. openssl list -provider helloworld -verbose -providers
Providers:
  helloworld
    build info: commit ca5eb625ac3ce2b8c99844d93656a39bf7e5980c
    gettable provider parameters:
      buildinfo: pointer to a UTF8 encoded string (arbitrary size)
```

Skeleton: Query function [1/4]

The query function is central! libcrypto calls it every time it wants to fetch some algorithm implementation from the provider.

The query function answers on a per operation basis.

(Only changes to the nonsense providers are shown)

```
static const OSSL_DISPATCH blargh_hash_impl[] = {
    // CONTENTS NOT SHOW!  That's a whole separate webinar...
    { 0, NULL }
};

static const OSSL_ALGORITHM hashes[] = {
    { "BLARG:id-blargh:1.2.3.4.5", "x.author=bleah", blargh_hash_impl,
      "BLARG is a fictious generic algorithm" },
    { NULL, NULL, NULL, NULL }
};

static OSSL_FUNC_provider_query_operation_fn query;
static const OSSL_ALGORITHM *query(void *provctx, int operation_id,
                                   int *no_cache)
{
    *no_cache = 0;
    switch (operation_id) {
    case OSSL_OP_DIGEST:
        return hashes;
    }
    return NULL;
}
```

Skeleton: Query function [2/4]

All we need to do now is to hook in `query`.

```
diff --git a/helloworld.c b/helloworld.c
--- a/helloworld.c
+++ b/helloworld.c
@@ -70,4 +70,5 @@

 static const OSSL_DISPATCH provfns[] = {
+    { OSSL_FUNC_PROVIDER_QUERY_OPERATION, (void (*)(void))query },
     { 0, NULL }
 };
```

Skeleton: Query function [3/4]

Build it:

```
$ clang -fPIC -c -o helloworld.o helloworld.c
$ clang -shared -Wl,--export-dynamic-symbol=OSSL_provider_init \
        -o helloworld.so helloworld.o
```

Skeleton: Query function [4/4]

Try it:

```
$ OPENSSL_MODULES=. openssl list -provider helloworld -verbose \
    -digest-algorithms
...
Provided:
  { 1.2.3.4.5, BLARG, id-blargh } @ helloworld
    description: BLARG is a fictious generic algorithm
```

Skeleton: Getting config data [1/5]

Getting config data is also about handling parameters, retrieving them this time.

This also involves finding the necessary “upcall” functions that libcrypto gives us. Let’s get to that first!

It all happens in `OSSL_provider_init`, so that’s all that’s shown here.

(Only changes to the nonsense providers are shown)

```
// ...
OSSL_FUNC_core_gettable_params_fn *c_gettable_params;
OSSL_FUNC_core_get_params_fn *c_get_params;

// Remember, |in| is the input OSSL_DISPATCH
for (; in != NULL && in->function_id; in++) {
    switch (in->function_id) {
    case OSSL_FUNC_CORE_GETTABLE_PARAMS:
        c_gettable_params = OSSL_FUNC_core_gettable_params(in);
        break;
    case OSSL_FUNC_CORE_GET_PARAMS:
        c_get_params = OSSL_FUNC_core_get_params(in);
        break;
    }
}

// For display, show all the parameters I can get from libcrypto
const OSSL_PARAM *p = c_gettable_params(handle);
for (; p != NULL && p->key != NULL; p++)
    printf("available libcrypto param: %s [type %u, size %zu]\n",
           p->key, p->data_type, p->data_size);
```

Skeleton: Getting config data [2/5]

Now, it's time to actually get parameter data from libcrypto.

"openssl-version" is a parameter that libcrypto always answers to.

"something" is a parameter that we get from configuration, if available.

(Only changes to the nonsense providers are shown)

```
// ...
const char *config_openssl_version = NULL;
const char *config_something = NULL;
OSSL_PARAM config_params[] = {
    { "openssl-version", OSSL_PARAM_UTF8_PTR,
      &config_openssl_version, 0, (size_t)-1 },
    { "something", OSSL_PARAM_UTF8_PTR,
      &config_something, 0, (size_t)-1 },
    { NULL, 0, NULL, 0, 0 }
};
if (!c_get_params(handle, config_params))
    return 0;

if (config_params[0].return_size != (size_t)-1)
    printf("libcrypto param 'openssl-version': %s\n",
           config_openssl_version);
if (config_params[1].return_size != (size_t)-1)
    printf("libcrypto param 'something': %s\n",
       config_something);
```

Skeleton: Getting config data [3/5]

The configuration file itself.

```
openssl_conf = openssl_init

[openssl_init]
providers = providers

[providers]
helloworld = helloworld_provider

[helloworld_provider]
identity = helloworld
activate = 1

something = "This is something"
somesect = helloworld_subsect

[helloworld_subsect]
somemore = "This is some more"
```

Skeleton: Getting config data [4/5]

Build it:

```
$ clang -fPIC -c -o helloworld.o helloworld.c
$ clang -shared -Wl,--export-dynamic-symbol=OSSL_provider_init \
        -o helloworld.so helloworld.o
```

Skeleton: Getting config data [5/5]

Try it:

```
$ OPENSSL_MODULES=. openssl list -provider helloworld -verbose
available libcrypto param: openssl-version [type 6, size 0]
available libcrypto param: provider-name [type 6, size 0]
available libcrypto param: module-filename [type 6, size 0]
libcrypto param 'openssl-version': 3.1.4

$ OPENSSL_CONF=./config.cnf OPENSSL_MODULES=. openssl list -verbose
available libcrypto param: openssl-version [type 6, size 0]
available libcrypto param: provider-name [type 6, size 0]
available libcrypto param: module-filename [type 6, size 0]
libcrypto param 'openssl-version': 3.1.4
libcrypto param 'something': This is something
```

Skeleton: Provider context [1/2]

So far, we have ignored `provctx`. It's time to change that.

The provider context holds everything together for a loaded instance of the provider.

(Only changes to the nonsense providers are shown)

```c
#include <stdlib.h>
// ...
struct provctx_st {
    int dummy;
};

static OSSL_FUNC_provider_teardown_fn teardown;
static void teardown(void *provctx)
{
    free(provctx);
}

static const OSSL_DISPATCH provfns[] = {
    { OSSL_FUNC_PROVIDER_TEARDOWN, (void (*)(void))teardown },
    { 0, NULL }
};
```

Skeleton: Provider context [2/2]

All we need to do now is to create and pass back the provider context.

All changes in further slides start from this point.

```
diff --git a/helloworld.c b/helloworld.c
--- a/helloworld.c
+++ b/helloworld.c
@@ -13,7 +25,9 @@ int OSSL_provider_init(const OSSL_CORE_HANDLE *handle,
                        void **provctx)
 {
-    fprintf(stderr, "nonsense: Hello, world!\n");
-
+    struct provctx_st *pctx = malloc(sizeof(struct provctx_st));
+
+    memset(pctx, 0, sizeof(*pctx));
+    *provctx = pctx;
     *out = provfns;
     return 1;
 }
```

Skeleton: Error handling [1/5]

Error handling involves an expanded provider context, and functionality to raise errors on the OpenSSL error queue.

First off, the provider context and the error raising function / macro.

The idea with this is that `ERR()` is used In code to raise errors. It works *almost* like OpenSSL's `ERR_raise_data()`, but takes the provider context instead of the library number.

```
struct provctx_st {
    const OSSL_CORE_HANDLE *handle;
    OSSL_FUNC_core_new_error_fn *up_new_error;
    OSSL_FUNC_core_set_error_debug_fn *up_set_error_debug;
    OSSL_FUNC_core_vset_error_fn *up_vset_error;
};

static void err(struct provctx_st *pctx, int reason,
                const char *file, unsigned int line, const char *func,
                const char *fmt, ...)
{
    va_list ap;

    va_start(ap, fmt);
    pctx->up_new_error(pctx->handle);
    pctx->up_set_error_debug(pctx->handle, file, line, func);
    pctx->up_vset_error(pctx->handle, reason, fmt, ap);
    va_end(ap);
}

#define ERR(pctx,reason,fmt,...) \
    err((pctx), (reason), __FILE__, __LINE__, __func__, fmt \
        __VA_OPT__(,) __VA_ARGS__)
```

Skeleton: Error handling [2/5]

Furthermore, the provider can set up its own error reason scheme, and pass back reason strings, on demand from libcrypto.

That's done with a function that returns an array of < reason code, string > tuples.

```c
#define E_UNSUPPORTED           1

static const OSSL_ITEM *get_reasons(void *provctx)
{
    static const OSSL_ITEM reasons[] = {
        { E_UNSUPPORTED, "Hello, world!  This is unsupported" },
        { 0, NULL }
    };
    return reasons;
}

// ...

static const OSSL_DISPATCH provfns[] = {
    { OSSL_FUNC_PROVIDER_GET_REASON_STRINGS,
      (void (*)(void))get_reasons },
    { OSSL_FUNC_PROVIDER_TEARDOWN, (void (*)(void))teardown },
    { 0, NULL }
};
```

Skeleton: Error handling [3/5]

To see error raising, a good spot to do so is needed.
We choose to do it in the query function we know from earlier.

(unfortunately, errors raised this way in `OSSL_provider_init()` are not rendered well, https://github.com/openssl/openssl/issues/22983)

```
static OSSL_FUNC_provider_query_operation_fn query;
static const OSSL_ALGORITHM *query(void *provctx, int operation_id,
                                   int *no_cache)
{
    ERR(provctx, E_UNSUPPORTED, "operation_id=%d", operation_id);
    return NULL;
}

static const OSSL_DISPATCH provfns[] = {
    { OSSL_FUNC_PROVIDER_GET_REASON_STRINGS,
      (void (*)(void))get_reasons },
    { OSSL_FUNC_PROVIDER_QUERY_OPERATION, (void (*)(void))query },
    { OSSL_FUNC_PROVIDER_TEARDOWN, (void (*)(void))teardown },
    { 0, NULL }
};
```

Skeleton: Error handling [4/5]

Build it:

```
$ clang -fPIC -c -o helloworld.o helloworld.c
$ clang -shared -Wl,--export-dynamic-symbol=OSSL_provider_init \
        -o helloworld.so helloworld.o
```

Skeleton: Error handling [5/5]

Try it:

```
$ OPENSSL_MODULES=. openssl enc -provider helloworld -blargh
enc: Unknown cipher: blargh
enc: Use -help for summary.
803B1CAC077F0000:error:40000001:helloworld:query:Hello, world!  This is unsupported:helloworld.c:46:operation_id=2
803B1CAC077F0000:error:0308010C:digital envelope
routines:inner_evp_generic_fetch:unsupported:../crypto/evp/evp_fetch.c:341:Global default library context, Algorithm
(blargh : 0), Properties (<null>)
```

Skeleton: Library context [1/3]

A provider that uses OpenSSL library functions, loads other providers, or fetches any algorithm implementation, **must** create a library context.

With this context, a provider works according to one of these models:

1. Self-contained
2. Inheriting the callers providers.

```c
#include <openssl/crypto.h>

struct provctx_st {
    OSSL_LIB_CTX *libctx;
};

static OSSL_FUNC_provider_teardown_fn teardown;
void teardown(void *provctx)
{
    struct provctx_st *pctx = provctx;

    OSSL_LIB_CTX_free(pctx->libctx);
    free(provctx);
}
```

Skeleton: Library context - self-contained model [2/3]

With the self-contained model, a provider:

- takes it upon itself to load all other providers it needs on its own
- doesn’t rely on the caller for anything, apart from upcalls.

This is code inserted into `OSSL_provider_init()`.

```c
// A library context that's independent of the calling application
pctx->libctx = OSSL_LIB_CTX_new_from_dispatch(handle, in);
if (pctx->libctx == NULL) {
    teardown(*provctx);
    *provctx = NULL;
    return 0;
}
```

Skeleton: Library context - inheriting model [3/3]

With the inheriting model, a provider:

- inherits all providers that the calling libcrypto loads, and can access all their implementations
- has its implementations accessible by other providers using the same model.

This is code inserted into `OSSL_provider_init()`.

```
// A library context that inherits loaded providers from the
// calling application
pctx->libctx = OSSL_LIB_CTX_new_child(handle, in);
if (pctx->libctx == NULL) {
    teardown(*provctx);
    *provctx = NULL;
    return 0;
}
```

References

All the code shown here is available in full, including its final state with all code snippets combined, in https://github.com/levitte/osslwebinar-provauthors-skeleton

There's a collection of links to diverse providers I find on GitHub, in https://github.com/provider-corner

Q&A