FFM and OpenSSL

Using OpenSSL in Apache Tomcat™
Rémy Maucherat
Software engineer at Red Hat
Apache Tomcat committer since 2000
ASF member
Contents

- FFM basics
- Tooling
- OpenSSL in Tomcat
Foreign Function and Memory API
The Panama project

Foreign Function and Memory API

Project developed by Oracle, part of java.base module

Incubation followed by preview JEPs since Java 14

Final in Java 22

Included in LTS with Java 25 due 09/2025
Objectives

Reflection style code for native functions, replacement for JNI

Foreign (non heap) memory access, replacement for Unsafe

Greatly improve safety and reliability
**Arena (MemorySession)**

- Handles native access lifecycle
- Allocations and deallocations
- Explicit close or GC close
MemorySegment

The main API since Java 20 (previously MemoryAddress and Addressable)

Pointer to native or heap memory, with associated size

May be tied to an associated session

Does size and lifecycle checks

Can have associated cleanup action
Memory layouts and native types

Allows modelling types and structures

Simple types are easy

Valhalla will provide support for additional types
Function descriptors and Method handles

FunctionDescriptor API allows describing the native calls to the JVM

Associate a native symbol to a matching method handle

Use the Linker API for that
Downcalls

Calls from Java to native

Use lookup to get the native symbol

Use the linker to get the method handle

Simply call the method handle with the right arguments
Downcall example

System.loadLibrary("ssl");

MemorySegment OpenSSL_versionSymbol = SymbolLookup.loaderLookup().find("OpenSSL_version").get();

MethodHandle OpenSSL_version = Linker.nativeLinker().downcallHandle(OpenSSL_versionSymbol, FunctionDescriptor.of(ValueLayout.ADDRESS, ValueLayout.JAVA_INT));

System.out.println("Hello " + ((MemorySegment) OpenSSL_version.invokeExact(0)).getString(0));
Upcalls

Calls from native to Java

Get a method handle from your Java method

The linker gives a memory segment containing a function pointer

Call the appropriate native downcall to set the function pointer
Upcall example

FunctionDescriptor openSSLCallbackVerifyFunctionDescriptor = FunctionDescriptor.of(ValueLayout.JAVA_INT, ValueLayout.JAVA_INT, ValueLayout.ADDRESS); /* typedef int (*SSL_verify_cb)(int preverify_ok, X509_STORE_CTX *x509_ctx); */

MethodHandle openSSLCallbackVerifyHandle = MethodHandles.lookup().findStatic(OpenSSLContext.class, "openSSLCallbackVerify", MethodType.methodType(int.class, int.class, MemorySegment.class));

MethodHandle SSL_CTX_set_verify = ...; // The OpenSSL downcall to set the function pointer

MemorySegment openSSLCallbackVerify = Linker.nativeLinker().upcallStub(openSSLCallbackVerifyHandle, openSSLCallbackVerifyFunctionDescriptor, state.contextMemorySession);

SSL_CTX_set_verify(state.sslCtx, /* int */ validationMode, openSSLCallbackVerify);
Using jextract
Jextract

Uses C header files

Generates boilerplate code for:

- Downcalls
- Upcalls
- Structures
Using jextract

Easy to build from https://github.com/openjdk/jextract

Binaries available at https://jdk.java.net/jextract/

Generates Java sources

Skips functional macros

Big library means huge and very verbose sources
After jextract run

List native APIs actually used

Limit jextract to these APIs

Handle bitfields if needed (hopefully not ...)

Add Java methods for any missing functional macros

Use generated class for downcalls and upcalls (call static allocate implementing the inner Function interface)
Downcall and upcall example using jextract

```java
System.out.println("Hello " + ((MemorySegment) OpenSSL_version(0)).getString(0));

// static class VerifyCallback implements SSL_CTX_set_verify$callback.Function
//    public int apply(int preverify_ok, MemorySegment /*X509_STORE_CTX*/ x509ctx)
SSL_CTX_set_verify(state.sslCtx, /* int */ validationMode,
    SSL_CTX_set_verify$callback.allocate(new OpenSSLEngine.VerifyCallback(), myArena));
```
OpenSSL in Tomcat
OpenSSL API style

Factories and destructors for everything

Accessors and setters for everything

Many callbacks needed

**Functional macros needed for compatibility**

OpenSSL headers work well with jextract
TLS support: Translate tomcat-native code

Translate from C to Java code using FFM

Then integrate into the Tomcat OpenSSL code

Lots of wrapper code, needless structures and state tracking

Surprisingly large amount of logic inside the JNI layer in some places (certs handling, init, OCSP)

Supports OpenSSL 1.1+, no support for LibreSSL
Why FFM turned out better

MemorySegment sslCtxAddress = SSL_CTX_new(TLS_server_method());
// This is only a 0 len address segment

// Reinterpret in a real session and a cleanup
Arena stateArena = Arena.ofShared();
MemorySegment sslCtx = sslCtx.reinterpret(ValueLayout.ADDRESS.byteSize(), stateArena,
   (MemorySegment t) -> SSL_CTX_free(t));

// Now use sslCtx in the code, close the arena on GC using a cleaner,
// and the arena will offer full protection
QUIC support

Nice C examples in OpenSSL 3.x

API based on SSL objects, create sub SSL objects with SSL_accept_stream

Non blocking requires doing socket polling (ex: using epoll)
Current Status and Conclusion

TLS 1.3 with PHA

Support for additional key formats, ciphers, protocols

Good performance

High level API for QUIC