

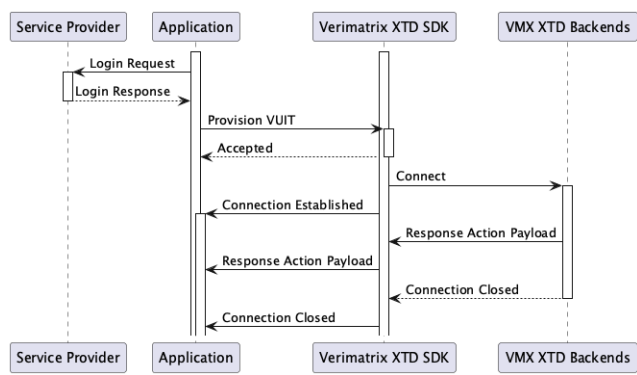
XTD SDK

Version	Description
1	Initial Draft
2	Client Integration
3	Server API
4	Updated to remove 'version' from V1 format. Added character sets allowed.
5	Updated to correctly reflect VUIT
6	Major rewrite for XTD SDK
7	Improve iOS & Android integration

Introduction

Verimatrix XTD is an application protection and threat analytics system. It protects mobile applications by preventing tampering and intrusions into application code. In addition, it collects security/threat analytics data and creates a risk assessment of the environment that the application is running in. The risk assessment can be retrieved from APIs exposed by XTD. XTD also provides operations to interact with the protected application, allowing the service provider to take actions such as suspending the application remotely. To support this, the protection process injects an analytics agent into the application, which runs in parallel to the protected application code.

The Verimatrix XTD SDK is a software developer kit that allows closer integration with Verimatrix XTD services. It provides an asynchronous API, enabling a bi-directional channel for communicating with the XTD infrastructure, both on-device and from the XTD analytics backend.



The current abilities of the API provide the following:

- Basic ion estaconnectivity callbacks
- Provisioning Verimatrix User Transaction Identifier (VUIT)
- Response actions

Basic Connectivity Callbacks

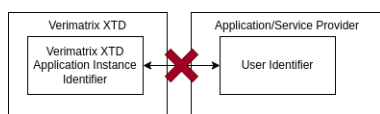
The XTD SDK provides functionality to signal that the underlying analytics agent has connectivity with the Verimatrix XTD security analytics backend. This is sent to the application layer to allow it to take action if the XTD security analytics agent does not regularly signal connections to the backend have been established.

- Connection Established
- Connection Closed

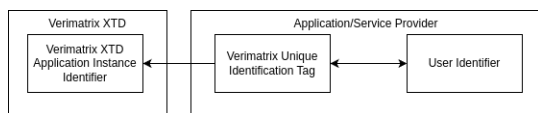
The connectivity events are delivered via the main callback, registered as part of integrating the SDK.

Verimatrix User Transaction Identifier (VUIT)

Verimatrix XTD's analytics agent internally uses an application instance identifier (AIID) to identify a specific application instance running on a specific device. This unique identifier ensures that the same application, running on two different devices, appear as different instances in the Verimatrix XTD ecosystem. The AIID is used in all Verimatrix XTD analytics operations to tie a variety of data together in the system, but it is not exposed to the protected application. The AIID is not exposed to make sure device specific information is not leaked outside of the Verimatrix XTD ecosystem. As a result, the service provider cannot link the application instance to the user that is logged into their service via the application.



To provide a capability to connect an instance of a protected application to a real user, Verimatrix allows the service provider to optionally provision an application with a Verimatrix User Identity Tag (VUIT). The VUIT is generated and provisioned by the service provider themselves and is not related to the Verimatrix application instance identifier in any way.



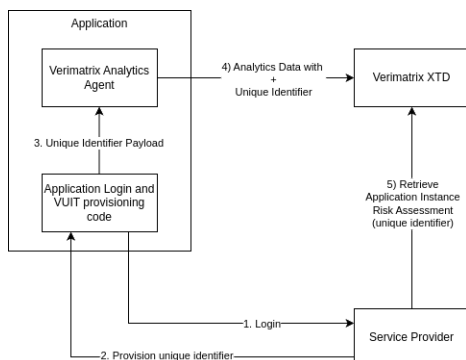
The use of VUIT is currently optional and the system will continue to work as expected even if no VUIT is provisioned to the application instances. If the service provider chooses not to use VUIT, some functionality of APIs might be limited and/or unusable without a VUIT. In particular, this affects all APIs in which the service provider expects to be able to interact with and/or query information relating to a particular application instance.

GDPR

In addition to the above, the Verimatrix unique identification tag aims to maintain application user anonymity while allowing the service provider to use the new VUIT system to infer known end user identifiers. The provisioned identifier provides no identifiable information about the actual user. This ensures that any identity inference can be made only by the service provider and their internal system, should they choose to generate a VUIT that is linked to their user and application.

Integration Points

The following integration points are used when interacting with Verimatrix XTD while using VUITs./



Provisioning

The provisioning integration point establishes the link between the application instance and the user identity and is the responsibility of the service provider (server side) and application (at runtime).

The typical provisioning flow is:

1. User logs into the service
2. Service provider generates provisioning information (VUIT) and provides it back to the application based on subscriber information
3. Application shares the provisioning information (VUIT) with the analytics agent
4. Analytics agent ingests the provisioning information (VUIT) and shares this with Verimatrix XTD analytics
5. Service provider can use the VUIT to request a risk assessment of the application instance

This flow assumes provisioning is done on the server side (within the service provider infrastructure). An alternative provisioning flow allows the application itself to provide the provisioning information (generated client side). This is considered less secure and would allow potential attackers to tamper with the identity of the application instance and should be avoided if possible.

Application Driven Provisioning

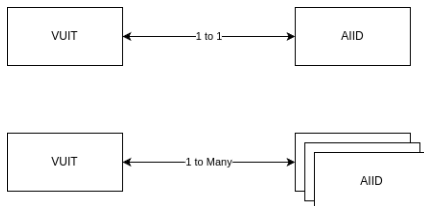
In the case where the application is responsible for providing the provisioning information, the application would need to be changed to generate the required unique identifier and share it with the analytics agent in the same way. The drawback with this approach is that there is no assurance that the value will not be used intercepted and/or tampered with. It can also provide means for an attacker to inject fake identities into the system.

Requirements for Generating Identifiers

To ensure consistency when using VUIT in the XTD ecosystem, there are some constraints and requirements on the VUIT value itself. Meeting these criteria is solely under the responsibility of the service provider, Verimatrix has no control of the generation or provisioning flows implemented in the application.

Entity Relationships

Under ideal conditions, the mapping between a VUIT and the application instance should be a one-to-one relationship. However, Verimatrix XTD also supports the case of mapping a single VUIT to many application instances.



The one-to-one mapping is the preferred solution and the service provider should strive to use this mode if possible. All operations using the VUIT will only affect a single application instance in this case.

When using the one-to-many relationship, the service provider is assigning a single VUIT to act as the identifier for a group of application instances. All operations using that VUIT will apply to all application instances that have been provisioned with it.

⚠ The use case where the application allows the customer to login with different usernames and the VUIT provisioned changes for each logged in user is currently not supported by Verimatrix XTD.

GDPR compliance

The service provider is fully responsible for generating the VUIT used when provisioning the identifier to the Verimatrix analytics agent. Care must be taken by the service provider to ensure that there is no sharing of Personal Identifiable Information (PII) from the service provider using VUIT.

When generating the VUIT, the service provider must make sure there is no PII included, for example:

- usernames
- email addresses
- names

In addition to ensuring no PII is included in the VUIT, service providers must also ensure there is no PII related information included or derivable from the generated VUIT.

A simple way to avoid leaking any information is to apply a forward hash to the input information being used.

Uniqueness

When generating a VUIT, service providers must also make sure it is unique. It should be considered to be unique across all applications of the service provider, which means that for ideal usage the VUIT must also include information that differentiates it across applications, for example, the package id of the application.

Properties that will add uniqueness are:

- Application package identifiers
- Device models
- Already globally unique subscriber identifiers

Using the above in conjunction with an appropriate and low collision forward hash function will provide a more unique identifier.

Format

The format of the VUIT is restricted to the following:

```
1 SP::<Up to 252 characters>
```

The provisioned value for the VUIT can not exceed 252 characters (256 including the SP:: prefix). The character encoding of the VUIT is US/ASCII with the following characters allowed.

```
1 [a-zA-Z0-9-]
```

For example, using a UUID v7 as the assigned VUIT, the value would be:

```
1 SP::018Cf454-b814-7CA2-8e43-6a13d7a33cfb
```

Provisioning Protocol Versions

Version 1

In version 1 of the specification, the format of the JSON document is as follows:

```
1 {
2   "version" : 1,
3   "spuid" : "SP::<Up to 252 characters>"
4 }
```

The properties are described further below:

Property	Description
spuid	Provisioned VUIT value. Maximum 256 characters. Should start with SP:: and be encoded in US/ASCII.
version	Version of protocol, currently always 1

Response Actions

The response actions form an important part of the XTD SDK and provide a way for the analytics back-end to deliver actions to the application layer of the protected application.

Currently, the following callbacks are supported:

- Degrade

The response actions are delivered via the same callback action that the connectivity events are delivered via.

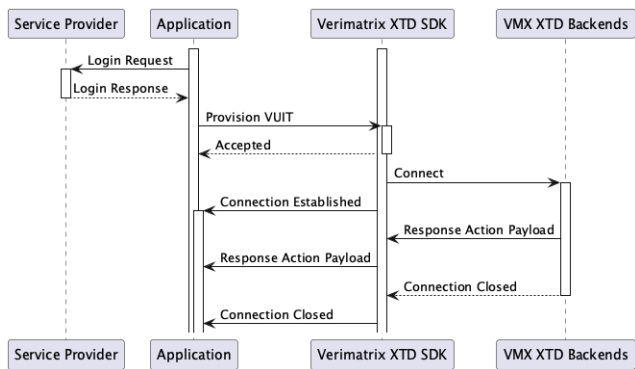
SDK Integration

This library is provided in platform specific implementations and is available in Java (Android) as an AAR and in Swift (iOS) as a static library.

Typical Application Usage

All interactions with SDK are made via JSON payloads being passed to and/or from the application/SDK respectively. A JSON schema is provided below to ensure the JSON payloads are valid and the data within them are of the correct type.

A typical integration flow is shown below.



API Overview

Due to language differences, the API will differ slightly between platforms, but in general, it provides the following methods:

Function	Parameters	Description
registerCallback	Message callback function/instance	Registers a callback with the analytics library which will be invoked when response action messages arrive from Verimatrix XTD. The same callback will also be used to signal errors back to the protected application.
sendMessage	JSON message to pass to Verimatrix XTD	Passes the specified JSON message to the analytics agent for processing and possible forwarding to Verimatrix XTD. The payloads use the same action types as the callback.

Message Callback

The message callback is a callback function that the analytics library will invoke when there is a response action payload available from the security analytics library. It defines a single method which when invoked, will be passed the response action payload JSON string.

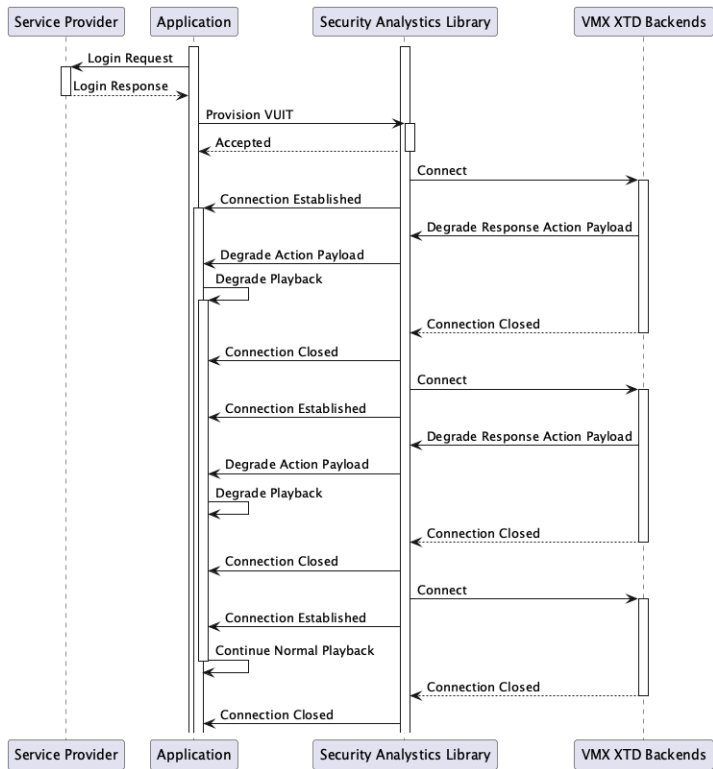
Function	Parameters
<code>MessageCallback#onMessage(@Nonnull String messageJSON)</code> (Android) <code>typedef void (*XTDStubLibCallback)(const char *messageJSON);</code> (iOS)	Response action payload (JSON string)

Response Action Processing and Application State Handling

Due to the nature of the response actions and how they arrive from the Verimatrix XTD analytics backend, some consideration is needed in the application layer.

Response actions can be divided into 2 categories: an immediate, single action or a long-term, persistent action. If the action is an immediate, single action, the application can process the response action immediately and then not store any state. Long-term, persistent actions should be persisted in the session and the action enforced until a successful pair of connection established/closed events have

been received without receiving another action, of the same type, in that connection window. Below shows an example of how this would work for the degrade action.



Error Handling & Error Codes

The `sendMessage` API will throw exceptions encapsulating XTD error codes if there is something wrong with the JSON payload being sent. In addition, the message callback, mostly used to deliver response actions from the analytics agent, also delivers error messages back to the protected application if they happen asynchronously within the application.

XTD error codes returned by the `sendMessage` API (encapsulated in the exception):

XTD error	Code	Description
XTD_SUCCESS	0	Indicates that the message has been successfully processed
XTD_GENERAL_ERROR	1	General error (no memory, library not initialized, etc)
XTD_JSON_PARSING_FAILED	2	Message JSON can't be parsed
XTD_INVALID_MESSAGE	3	Message contains invalid/missing objects
XTD_MESSAGE_VERSION_MISMATCH	4	Message version doesn't match
XTD_UNRECOGNIZED_COMMAND	5	Message command is not recognized
XTD_INVALID_PAYLOAD	6	Message payload is not in the expected format
XTD_CRYPTO_ERROR	7	Message payload cryptographic verification failed

Request/Response Action Payloads

Payload Format

The general format of the requests/response action payloads are plain JSON strings, passed directly into the XTD SDK.

```
1 {
2   "c":1,
3   "v":1,
4   "p" : { "some" : "command specific payload data"}
5 }
```

Property	Description
c	Message payload type
v	Payload format version (currently always 1)
p	Additional payload, command specific

JSON Schema

The following defines the JSON schema for the action request/response payloads.

```
1 {
2   "$schema": "https://json-schema.org/draft/2020-12/schema",
3   "$id": "https://verimatrix.com/xtd/response-actions-payloads/2025-01/schema.json",
4   "title": "Verimatrix XTD Response Actions Payloads",
5   "description": "Describes response actions payloads used when interacting with Verimatrix secure analytics library (SAIL)",
6   "properties": {
7     "c": {
8       "description": "Defines the management action this payload carries. ",
9       "enum": [1, 2, 3, 6]
10    },
11    "v": {
12      "description": "Indicate the version of the payload being carried",
13      "enum": [1]
14    },
15    "p": {
16      "oneOf": [
17        {
18          "type": "object",
19          "properties": {
20            "spuid" : { "type": "string" },
21            "version" : { "type": "number" }
22          }
23          "required" : ["spuid", "version"]
24          "additionalProperties": false
25        },
26        {
27          "type": "object",
28          "properties": {
29            "id" : { "type": "number" },
30            "name" : { "type": "string" },
31            "details": { "type": "object",
32              "properties": {
33                "version": { "type": "integer" },
34                "names": {
35                  "type": "array",
36                  "items": { "type": "string" }
37                }
38              },
39              "required": ["version", "names"],
40              "additionalProperties": false
41            }
42          }
43          "required" : ["id", "name"]
44          "additionalProperties": false
45        },
46      ],
47    },
48    "additionalProperties": false
49  },
50  "required": ["c", "v"],
51  "additionalProperties": false
52 }
```

Response Actions

Response Action Types

The following action types are currently in use:

Payload Type	Management Action Name	Action Sent By	Description	Additional Payload Data
0	Error Occurred	XTD SDK	An asynchronous error occurred within the XTD SDK	No
1	Connection Established	XTD SDK	A connection has been established to the XTD Analytics Server	No
2	Connection Closed	XTD SDK	A connection to the XTD Analytics Server has been closed	No
3	Degrade	XTD SDK	Backend has signaled that any on-going playback session should be degraded (in quality or otherwise) until further notice.	No
6	Provision VUIT	Customer Application	Provisions the VUIT from the application into the analytics agent.	Yes, see VUIT payload example
7	Event Detected	XTD SDK	A security event has been detected by the XTD SDK	Yes, see Event Detected payload example
8	Get Version	Customer Application	Application requests the analytics agent version	No
9	Analytics Agent Version	XTD SDK	Response to Get Version request containing the agent version	Yes, see Get Version Response payload example
10	Provision VUIT Response	XTD SDK	VUIT has been provisioned successfully	No

Event Detected (7) Types

Event ID	Event Name	OS	Additional info in the “details” object of the payload
100	Bootloader Detected	Android	-
101	Debugger Detected	Android	-
102	Emulator Detected	Android	-
103	Overlay Detected	Android	-
104	Rooting/Jailbreak Detected	Android, iOS	Android: - iOS: <pre> 1 { 2 "version" : 1, 3 "names" : ["Checkra1n", "Dopamine"], 4 }</pre>
105	Side Loading Detected	Android	-
106	Tampering Detected	Android, iOS	-
107	Hooking Detected	Android, iOS	Android: - <pre> 1 { 2 "version" : 1, 3 "names" : ["Frida"], 4 }</pre>
108	Hostname Whitelist Violation	Android, iOS	

109	Installer Mismatch	Android	
-----	--------------------	---------	--

Response Action Payload Details

The following are example payload details for each of the action types listed above.

Connection Established

```
1 {
2   "v" : 1,
3   "c" : 1
4 }
```

Connection Closed

```
1 {
2   "v" : 1,
3   "c" : 2
4 }
```

Degrade

```
1 {
2   "v" : 1,
3   "c" : 3
4 }
```

Event Detected

```
1 {
2   "c" : 7,
3   "v" : 1,
4   "p" : {
5     "id" : 104,
6     "name" : "RootingDetected"
7     "details" : {
8       "version" : 1,
9       "names" : ["Checkra1n","Dopamine"],
10    }
11 }
```

Get Version Response

```
1 {
2   "c" : 9,
3   "v" : 1,
4   "p" : {
5     "agentVersion" : "1.50.0"
6   }
7 }
```

Provision VUIT Response

```
1 {
2   "c" : 10,
3   "v" : 1
4 }
```

Request Actions

Request Actions Types

The following request action types are currently in use:

Payload Type	Management Action Name	Action Sent By	Description	Additional Payload Data
6	Provision VUIT	Customer Application	Provisions the VUIT from the application into the analytics agent.	Yes, see VUIT payload example
8	Get Version	Customer Application	Requests the analytics agent version.	No, see Get Version example

Provision VUIT

```
1 {
2   "v" : 1,
3   "c" : 6,
4   "p" : {
```

```

5     "version" : 1,
6     "spuid" : "SP::<VUIT Data>"
7   }
8 }

```

Get Version

```

1 {
2   "v" : 1,
3   "c" : 8
4 }

```

Integrating the SDK into your project

To start using the XTD SDK, you have to make some changes to your project. Android and iOS specific details follow below.

Android

Package contents

docs	- Integration guide and installation guide
android/aar	- VMX XTD Android library (vmx-xt-d-sdk.aar)
android/example	- Example Java functions (VmxXtdExample.java)

Copy Library

Copy the `vmx-xt-d-sdk.aar` from the VMX XTD SDK package into the project's `app/libs`.

If you do not have one, create the libs folder in the the app folder of your project.

build.gradle.kts

If your project uses a build.gradle.kts file, follow these steps.

Make the changes shown in the following code snippets in the settings.gradle.kts file and the app/build.gradle.kts file.

settings.gradle.kts

```

1 pluginManagement {
2   repositories {
3     ...
4     flatDir {
5       dirs("libs")
6     }
7   }
8 }

```

app/build.gradle.kts

```

1 ...
2 dependencies {
3   implementation(files("libs/vmx-xt-d-sdk.aar"))
4   ...
5 }
6 ....

```

build.gradle

If your project uses a build.gradle file, follow these steps.

Include the following in the project's `build.gradle`

```

1 repositories {
2   flatDir {
3     dirs 'libs'
4   }
5 }
6
7 dependencies {
8   implementation(name: 'stublib', ext: 'aar')
9 }

```

API Overview

The VMX XTD API is a Java interface that allows an app to send and receive messages to/from VMX XTD.


The API is accessible by the use of the following classes:

- `com.verimatrix.stublib.StubLib`, main API entry point and the only one mandatory for XTD API usage.

Receive XTD SDK messages

To be able to receive commands and other messages, the app MUST register its own callback function. After registering, the registered callback function will be invoked whenever XTD SDK needs to pass any command and operation response results or errors. It is application responsibility to handle those messages accordingly. The given example is just a simplified demo where the app logs all the received messages.

```
1 // Import the VMX XTD SDK
2 import com.verimatrix.stublib.StubLib;
3
4 /**
5  * Receive XTD messages example.
6  */
7 public static void registerCallback() {
8     try {
9         StubLib.registerCallback(message -> {
10             // Parse and process messages received from the VMX XTD
11             // ...
12             Log.i(TAG, "Received message: " + message);
13         });
14     } catch (StubLib.StubLibException e) {
15         Log.e(TAG, "Error registering callback: " + e.getMessage());
16     }
17 }
```

 The callback is invoked on the UI thread to allow UI interactions if needed. As a result, care should be taken to not do too much work in the callback.

Send messages to VMX XTD SDK

To be able to uniquely identify the running instance, the app needs to define and set the VUIT value. The example below contains the basic implementation that sends the provision VUIT command to VMX XTD.

Similarly, other command can be constructed and sent by using the `sendMessage` API function.

```
1 // Import the XTD SDK
2 import com.verimatrix.stublib.StubLib;
3 import com.verimatrix.stublib.StubLibMessageBuilder;
4
5 /** Example of how to use the XTD SDK to provision the VUIT with the security analytics agent */
6 public class SetVUITExample {
7
8     /** Example method of creating the VUIT payload and provisioning this with the security analytics agent using XTD SDK
9     *
10     * @param vuit The VUIT to provision
11     */
12     public static void provisionVUIT(String vuit) {
13
14         // Create the VUIT payload using the StubLibMessageBuilder. This creates the protocol equivalent of
15         //
16         // {"v": 1, "c" : 6, "p" : {"version" : 1, "spuid" : "SP::example"}}
17         //
18         // where:
19         // "v" - payload format version (MUST be 1)
20         // "c" - payload type (provision VUIT is 6)
21         // "p" - additional payload (required for provision VUIT payload)
22         // "version" - version of VUIT payload
23         // "spuid" - required element for provision VUIT payload
24         // "SP::example" - required value for "spuid" element, MUST contain the VUIT value, pre-fixed with "SP::" string literal
25         //
26         final String vuitPayload = StubLibMessageBuilder.createNewSetVUITCommand(vuit).build();
27
28
29         try {
30             StubLib.sendMessage(vuitPayload);
31         } catch (StubLib.StubLibException e) {
32             Log.e(TAG, "Error registering callback: " + e.getMessage());
33         }
34     }
35 }
```

```
35
36 }
37
```

iOS

Package contents

```
docs                - Integration guide and installation guide
ios/inc              - VMX XTD library include file
ios/lib-iphoneos     - VMX XTD library static library for device
ios/lib-iphonesimulator - VMX XTD library static library for simulator
ios/example          - Example swift functions
```

Copy Library and Header Files

Copy the `ios/inc`, `ios/lib-iphoneos` and `ios/lib-iphonesimulator` folders to your project folder (\$SRCROOT) this is the folder where your Xcode project folder is stored. The following instructions assume you copied the files there, adjust as needed if you decide to put the files somewhere else.

If your project uses xcconfig files, skip to Project Configuration with xcconfig file. Else the next section shows you how to update your build settings.

Xcode Build Settings

For Xcode to find the header-file, the header search paths have to be updated. Go to **Header Search Paths** in your build settings and add `$(SRCROOT)/inc`

To link the library add the library path and add the library in the **Link Binary With Libraries Build Phase**.

Go to Build Settings and find the **Library Search Paths**. Open this setting and add

```
$(SRCROOT)/lib$(EFFECTIVE_PLATFORM_NAME)
```

 note that the value of `EFFECTIVE_PLATFORM_NAME` is either -
iphoneos or -iphonesimulator, including the dash.

Then add the library from the **Link Binary With Libraries Build Phase**. Click the + button and navigate to one of the static libraries. You need to add only one, the Library Search Paths change above will select the appropriate one for device or simulator.

Project Configuration with xcconfig file

- Add paths to the `USER_HEADER_SEARCH_PATHS` and the `LIBRARY_SEARCH_PATHS` in your project's .xcconfig file.
- Link the VMX XTD static library (`libvmx-xtd-sdk.a`) to your project. Also update the value of `OTHER_LDFLAGS` in your project's .xcconfig file to link with this library.

```
1 Example .xcconfig section:
2
3 ...
4 // Search paths for headers and libraries
5 USER_HEADER_SEARCH_PATHS = $(inherited) $(SRCROOT)/inc
6 LIBRARY_SEARCH_PATHS = $(inherited) $(SRCROOT)/lib$(EFFECTIVE_PLATFORM_NAME)
7
8 // Other Linker Flags
9 OTHER_LDFLAGS = $(inherited) -ObjC -lvmx-xtd-sdk
10 ...
```

Use the XTD SDK from Swift

To call the API functions from Swift, import the `vmx-xtd-sdk.h` header in your project bridging header.

```
#import "vmx-xtd-sd.h"
```

If you do not have a bridging header yet, you can induce Xcode to create one by adding a temporary C source file to your project, you can remove the file from your project and discard the file after Xcode has generated the bridging header.

Refer the file `ios/example/StublibIntegrationExample.swift` for example functions on how to use the VMX XTD API.

Use the XTD SDK from Objective C

Import the header where needed.

```
#import "vmx-xtd-sdk.h"
```

API Overview

The VMX XTD API provides a few simple Objective C methods in `vmx-xtd-sdk.h` for keeping compatibility with both Swift and Objective C projects:

```
1 // Registers a callback for receiving messages from the XTD system
2 // The registered callback can be called at any time and from background threads.
3 (void) registerCallback:(XTDStubLibCallback) callback
4
5 // Sends a command to the server XTD ARC server
6 (BOOL) sendMessage:(NSString *) messageJSON
7         error:(out NSError **) outError;
8
9 // Sets the VUIT value for the running instance
10 // Note: This API is deprecated, use the sendMessage() function to set the VUIT (previously called SPUID).
11 (BOOL) setSPUID:(NSString *) spuidJSON
12         error:(out NSError **) outError;
```

Note:

- Swift translates the above "BOOL return code with NSError** error" pattern to exceptions, therefore it's advised to use a do/try/catch block when calling the XTD library methods from Swift code. The returned error will be in a generic variable named `error` after catching errors.
- Verimatrix provides the JSON schema of messages in this integration guide, including the available commands with parameters.

API Usage examples in Swift

Registering Message Callback

```
1 // Example callback function to receive commands from the XTD ARC server in JSON format
2 let cbFunc: XTDStubLibCallback = {(messageJSON: Optional<UnsafePointer<CChar>>) -> () in
3
4     let messageFromXTD = String(cString: UnsafePointer<CChar>(messageJSON!))
5     DispatchQueue.main.async {
6         // Process the message. Note that the callback can be called at any time
7         // and almost always from a background thread.
8     }
9 }
10
11 // Register a callback function in the XTD library for receiving commands in JSON format from the XTD ARC server
12 func registerCallback()
13 {
14     XTDStubLib.registerCallback(cbFunc)
15 }
```

Provisioning VUIT

```
1 /** Example method of creating the VUIT payload and provisioning this with the security analytics agent using XTD SDK
2  *
3  * @param vuit The VUIT to provision
4  */
5 func provisionVUIT(_ vuit: String) {
6     // Create the message payload {"v": 1, "c" : 6, "p" : {"spuid":"SP:example"}}
7     let vuitPayload = "{\"v\": 1,\"c\":6,\"p\":{\"spuid\":\"\\(vuit)\"}}";
8     do {
9         try XTDStubLib.sendMessage(vuitPayload)
10     } catch {
11         // Send message failed
12         print("XTD stub lib error: \\(error)")
13     }
14 }
```