C++ Call to FieldShield Encryption Routine

The C++ program shown in the subsequent pages (**wrapper.cpp**) is a sample application calling the AES-256 format preserving encryption (FPE) routine through the C executable that IRI FieldShield (sortcl.exe) uses to run this script, *fpe_field.fcl:*

```
/INFILE=stdin
/PROCESS=CONCH                  # named for the shell format
/FIELD=(IN, POSITION=1)
/STREAM
/OUTFILE=stdout
/FIELD=(OUT=enc_fp_aes256_alphanum(IN), POSITION=1)
```

As you see, *fpe_field.fcl* is a 4GL definition of data streaming from an application into the masking (FPE) function and sending the resulting ciphertext in memory back into the calling program.

The program below is based on a C++ wrapper adapted from:
https://docs.microsoft.com/en-us/windows/win32/procthread/creating-processes
https://www.installsetupconfig.com/win32programming/windowsthreadsprocessapis7_18.html
to run on Windows.

In such a program, expected output, for example, would be:

```
'Carl' replaced with 'Jufc'
'Frank' replaced with 'Mxbxo'
'John' replaced with 'Qcll'
```

It is important that the called function script above, *fpe_field.fcl* is in the same place as the executable. It is advised to set the absolute path to the *fpe_field.fcl* file in the **wrapper.cpp** code (line 92) just in case.

There is now only one known limitation in this example:

1. No support wide characters (the two links above show the correct way, however).

wrapper.cpp

```
/*
Simple wrapper for executing sortcl CONCH process scripts (streaming stdin and
stdout). This program creates a child process for sortcl and pipes its stdin and
stdout to the user. You can adapt this program to utilize other sources and targets
for the stdin and stdout of the child process.
*/

#include <windows.h>
#include <stdio.h>
#include <iostream>
#include <string>
using namespace std;

#define BUFSIZE 512

// Global variables
HANDLE g_hChildStd_IN_Rd = NULL;
HANDLE g_hChildStd_IN_Wr = NULL;
HANDLE g_hChildStd_OUT_Rd = NULL;
HANDLE g_hChildStd_OUT_Wr = NULL;

// Prototypes, needed for C++
void CreateChildProcess(char const *);
void CreateReadThread(void);
void ErrorExit(const char*);
DWORD WINAPI ReadFromChildProcess(LPVOID);

int main(int argc, char const *argv[])
{
       if (argc != 2)
              ErrorExit("usage: wrapper <scriptPath>");

    SECURITY_ATTRIBUTES saAttr;
       DWORD dwWritten;
       BOOL bSuccess = FALSE;

    // Set the bInheritHandle flag so pipe handles are inherited
    saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
    saAttr.bInheritHandle = TRUE;
    saAttr.lpSecurityDescriptor = NULL;

    // Create a pipe for the child process's STDOUT
    if (!CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0))
        ErrorExit("Failed to create STDOUT pipe.");

    // Ensure the read handle to the pipe for STDOUT is not inherited
    if (!SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0))
        ErrorExit("STDOUT read handle failed for inheritance.");

    // Create a pipe for the child process's STDIN
    if (!CreatePipe(&g_hChildStd_IN_Rd, &g_hChildStd_IN_Wr, &saAttr, 0))
        ErrorExit("Failed to create STDIN pipe.");

    // Ensure the write handle to the pipe for STDIN is not inherited
    if (!SetHandleInformation(g_hChildStd_IN_Wr, HANDLE_FLAG_INHERIT, 0))
        ErrorExit("STDIN write handle failed for inheritance.");
```

```
    CreateChildProcess(argv[1]);
       CreateReadThread();
       printf("Enter input (ctrl-c to cancel):\n");

       for (string line; getline(cin, line);)
       {
              line += '\n'; // Required for sortcl to identify the input as a record
              bSuccess = WriteFile(g_hChildStd_IN_Wr, line.c_str(), line.length(),
&dwWritten, NULL);
              if (!bSuccess)
                     ErrorExit("Error writing to sortcl");
       }
    return 0;
}

// Create a child process that uses the previously created pipes for STDIN and STDOUT.
void CreateChildProcess(char const * scriptPath)
{
       string strCommandLine = "sortcl /spec=" + string(scriptPath);
       LPSTR cSCommandLine = _strdup(strCommandLine.c_str());
    STARTUPINFO siStartInfo;
    PROCESS_INFORMATION piProcInfo;
    BOOL bSuccess = FALSE;

    // Set up members of the PROCESS_INFORMATION structure
    ZeroMemory(&piProcInfo, sizeof(PROCESS_INFORMATION));

    // Set up members of the STARTUPINFO structure
    // This structure specifies the STDIN and STDOUT handles for redirection
    ZeroMemory(&siStartInfo, sizeof(STARTUPINFO));
    siStartInfo.cb = sizeof(STARTUPINFO);
    siStartInfo.hStdError = g_hChildStd_OUT_Wr;
    siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
    siStartInfo.hStdInput = g_hChildStd_IN_Rd;
    siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

    bSuccess = CreateProcess(NULL,      // No module name (use command line)
              cSCommandLine,                                  // Command Line
        NULL,                         // Process handle not inheritable
        NULL,                         // Thread handle not inheritable
        TRUE,                         // Set handle inheritance to TRUE
        0,                            // No creation flags
        NULL,                         // Use parent's environment block
        NULL,                         // Use parent's starting directory
        &siStartInfo,                 // Pointer to STARTUPINFO structure
        &piProcInfo);                 // Pointer to PROCESS_INFORMATION structure

    if (!bSuccess)
        ErrorExit("CreateProcess failed.\n");

       if(CloseHandle(piProcInfo.hProcess) == 0)
              ErrorExit("Closing handle to child process has failed.");

       if (CloseHandle(piProcInfo.hThread) == 0)
              ErrorExit("Closing handle to child process thread has failed.");
}

// Creates a thread function that constantly reads from the stdout of the child
process.
```

```
// The thread is cleaned up when this program terminates, but for long running
programs
// you would need to terminate the thread manually when you no longer expect output
from
// the child process.
void CreateReadThread()
{
	HANDLE hThread;
	DWORD dwThreadId;

	// We need to create a separate thread for reading from the process to avoid a
potential deadlock.
	hThread = CreateThread(
		NULL,			// default security attributes
	  0,			// use default stack size
	  ReadFromChildProcess,	// thread function name
	  NULL,				// argument to thread function
	  0,			// use default creation flags
	  &dwThreadId);			// returns the thread identifier

	// If CreateThread fails, terminate execution.
    // This will automatically clean up threads and memory.
	if(hThread == NULL)
		ErrorExit("Create thread failed");
}

void ErrorExit(const char* error)
{
    printf(error);
    ExitProcess(1);
}


// Thread function created in CreateReadThread().
DWORD WINAPI ReadFromChildProcess(LPVOID lpParam)
{
	CHAR chBuf[BUFSIZE];
	DWORD dwRead;
	BOOL bSuccess = FALSE;

	for(;;)
	{
		bSuccess = ReadFile(g_hChildStd_OUT_Rd, chBuf, BUFSIZE, &dwRead, NULL);

		if (!bSuccess)
			ErrorExit("Error reading from sortcl");

		chBuf[dwRead] = '\0';
		cout << chBuf;
	}
}
```