Binding the Bytecode.byc file to the Interpreter

This process is kind of tricky requiring a minimum level of complexity, but worth it for anyone who may want to utilize AGK for Windows game projects where code signing and some form of basic integrity protection is needed.

A disclaimer and recommendation: This example is provided as-is, freely open to use and modify. It is based on my work implementing such a system for my own projects, but has been altered to operate more broadly so others can incorporate the same option into their projects. You may spot some things that could be done better and/or find things you want to change to meet your preferences. This is only one approach, you may want to shift things around to other points in the codebase to behave differently. This example is designed to be pretty flexible to be moved or changed in such ways.

First, the OpenSSL libraries need to be plugged into Visual Studio. You can use the git utility to place things (https://git-scm.com/downloads) which you may already have installed if you've worked with the AGK repo before, here are some example command line operations to install the vcpkg utility:

git clone https://github.com/Microsoft/vcpkg
cd vcpkg
bootstrap-vcpkg.bat
vcpkg integrate install

This will install and integrate the vcpkg utility. The 'cd vcpkg' line may need to be adapted to whatever custom folder might apply (often 'users\YourUserName\vcpkg'). Once vcpkg is installed, OpenSSL needs to be installed:

vcpkg search ssl
vcpkg install openssl:x64-windows-static

It may be necessary to also execute 'vcpkg install openssl-windows --triplet x64-windows' prior to the static install step, but such a step did not seem necessary. Once that is done, then the prerequisites are in place and they just need to be linked. In Visual Studio 2022, load the interpreter project from the AGK Repo, then right click on it in the Solution Explorer menu, then click on 'Properties', then apply these link conditions:

C/C++ > General > Additional Include Directories > vcpkg\installed\x64-windows-static\include\openssl
Linker > General > Additional Library Directories > vcpkg\installed\x64-windows-static\lib
Linker > Input > Additional Dependencies > add libcrypto.lib

Again, the 'vcpkg' directory may be something different on your system depending on where things got routed for install. Be sure to align the specified folder with the location on your system. Once these steps are complete, needed code can be added. First, we need to add some functions:

```
// player hash function
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
```

```
#include <algorithm>

std::string readStringFromPlayer(const std::string& filePath, size_t stringSize) {
	std::ifstream file(filePath, std::ios::binary | std::ios::ate);
	if (!file.is_open()) {
		return "";
	}

	std::streamsize fileSize = file.tellg();
	if (fileSize < stringSize) {
		return "";
	}

	int buffSize = 100000;
	std::vector<char> buffer(buffSize);  // create a static buffer large enough to handle all
ending data
	file.seekg(fileSize - buffSize, std::ios::beg);
	if (!file.read(buffer.data(), buffSize)) {
		return "";
	}

	file.close();

	// copy char array to std::string
	std::string text(buffer.begin(), buffer.end());

	// define what we're looking for as markers before and after the hash
      // these are custom string values you set for what you want to look for
	std::string begin_text("MyCustomStartingString");
	std::string end_text("MyCustomEndingString");

	// find the start and end of the text we need to extract
	size_t begin_pos = text.find(begin_text) + begin_text.length();
	size_t end_pos = text.find(end_text);

	// create a substring from the positions
	std::string extract = text.substr(begin_pos, end_pos);

	extract = extract.substr(0, 128);  // remove ending characters

	return extract;
}


// bytehash function, this uses openssl
#include <sstream>
#include <iomanip>
#include <openssl/sha.h>
```

```
std::string sha512_file(const std::string& filename) {
	std::ifstream file(filename, std::ios::binary);
	if (!file.is_open()) {
		return ""; // Return empty string if file can't be opened
	}

	// SHA512_CTX, SHA512_Init, SHA512_Update, SHA512_Final available through
OpenSSL
	SHA512_CTX sha512_ctx;
	SHA512_Init(&sha512_ctx);

	const int buffer_size = 4096;
	char buffer[buffer_size];
	while (file.read(buffer, buffer_size) || file.gcount() > 0) {
		SHA512_Update(&sha512_ctx, buffer, file.gcount());
	}

	unsigned char hash[SHA512_DIGEST_LENGTH];
	SHA512_Final(hash, &sha512_ctx);

	std::stringstream ss;
	for (int i = 0; i < SHA512_DIGEST_LENGTH; ++i) {
		ss << std::hex << std::setw(2) << std::setfill('0') << (int)hash[i];
	}

	return ss.str();
}
```

Make sure to change the 'MyCustomStartingString' and ' MyCustomEndingString' values above to the search strings you want to use as markers for finding the hash in the interpreter file. These should be relatively long values unique enough to not conflict with any other string values that may be present inside the EXE interpreter file itself.

With these functions in place, we can add the additional code required to compare hash values in the startup sequence of the interpreter/player. So in that section of the code (find '**void app::Begin(void)**'), find these lines:

```
	if ( agk::GetFileExists( "bytecode.byc" ) == 1 )
	{
		agk::SetWindowAllowResize(0);

		m_iStandAlone = 1;
		m_iDebugMode = 0;
		m_iStepMode = 0;
		m_iAppControlStage = APP_RUNNING;
```

Then below those lines add:

```
		// first retrieve the hash stored with the player/interpreter (should be appended at the end)
		uString PlayerHash;

	// This is the final output filename for your game.
	// So a separate player is built and kept with each game individually.
	// After compiling, you may want to keep a copy of this interpreter in a \player folder
	// within your game's work folder, then copy it over to complete the hash addition step
	// later.  Creating a batch file to simplify this process is recommended.
		std::string filePath = "MyGamesFilename.exe";

		size_t stringSize = 128;  // not really used any more, just left in place to avoid breaking anything

	//
		std::string extractedString = readStringFromPlayer(filePath, stringSize);

		if (!extractedString.empty()) {
			PlayerHash = extractedString.c_str();
		}
		else {
			PlayerHash = "";
		}


		// generate hash from bytecode file for comparison
		uString ByteHash;

		std::string filename = "media/bytecode.byc";
		std::string hash = sha512_file(filename);
		ByteHash = hash.c_str();

		// check to see if hashes match, exit if not
		if ( PlayerHash != ByteHash ) {  // check strings in their uString format
			agk::Message("File integrity check failed!");
			AppQuit();
		}
```

Make sure to change the 'MyGamesFilename.exe' value above to match whatever the final name of your game's EXE file will be. This will be the filename the routine searches for the hash value, so it needs to match exactly. You'll likely want to compile a unique player for each project you implement this system for and store it separately (more information and suggestions below).

That's basically it for adding the necessary code to the interpreter. After changing the three custom string values in the sample code, you can compile the interpreter. Next, a method to generate a hash from the bytecode file and attach it to the player is needed. To do that, I recommend keeping a copy of the custom interpreter in a folder with your project, then copy it over every time you are ready to make a release build. This way, you keep your original in a safe location without any hash attached so you

can reuse it every time you make a release build. For this example, we will apply things with the player located inside a folder named '\player' within your project's work folder and we'll create a batch file to simplify the process of copying and hashing the player. We'll keep the filename for the custom player as the default 'Windows64.exe' while inside the '\player' folder, then rename it to the game name's filename whenever we make a release build within the batch file. Here is a sample batch to complete these steps:

```
echo off
setlocal
set file=Windows64.exe
set newfile=MyGamesFilename.exe

copy player\Windows64.exe Windows64.exe

rem get sha512 hash of bytcode file to add to the player
for /f "delims=" %%A in ('certutil -hashfile media\bytecode.byc SHA512 ^| find /v ":"') do set "hash=%%A"

echo MyCustomStartingString%hash%MyCustomEndingString >> Windows64.exe

rem rename file to game's filename
rename %file% %newfile%

echo Ready for release!
pause
exit
```

And that's basically it. The batch file will copy the Windows64.exe file from the '\player' folder and place it in the root work folder of your project. Then it will use the Windows 'certutil' program to generate a hash file from the 'bytecode.byc' file located in the '\media' folder. It then stores that value to a variable named 'hash'. Next, it takes that hash value and combines it with the starting and ending string values. Remember, these string values need to match the values you specified in the interpreter's code as they are used as the markers for the interpreter to find the self-contained hash value. Once the hash value is assembled with the markers, it is then added to the Windows64.exe file placed in the project's root folder. The file is then renamed to the game's filename as specified in the 'newfile' value in line 4. The player will now only work with the one matching bytecode file. Once code signed, the player can't be altered without breaking the code signing, leaving it locked to the bytecode file.

This approach also allows AGK to function normally using the players that come with it by default. You can hit F5 to compile and run your projects as normal while you work just like before. Then when you are ready to distribute, you can copy over your custom player, generate and attach the hash, code sign, and then it's ready for release.