Intel® Software Guard Extensions Part 6: How to Create Dual Code Paths

By John M., published on October 28, 2016

In Part 6 of the Intel® Software Guard Extensions (Intel® SGX) tutorial series, we set aside the enclave to address an outstanding design requirement that was laid out in Part 2, Application Design: provide support for dual code paths. We want to make sure our Tutorial Password Manager will function on hosts both with and without Intel SGX capability. Much of the content in this part comes from the article, Properly Detecting Intel® Software Guard Extensions in Your Applications.

You can find the list of all of the published tutorials in the article Introducing the Intel® Software Guard Extensions Tutorial Series.

There is source code provided with this installment of the series.

All Intel® Software Guard Extensions Applications Need Dual Code Paths

First it's important to point out that all Intel SGX applications must have dual code paths. Even if an application is written so that it should *only* execute if Intel SGX is available and enabled, a fallback code path must exist so that you can present a meaningful error message to the user and then exit gracefully.

In short, *an application should never crash or fail to launch solely because the platform does not support Intel SGX.*

Scoping the Problem

In Part 5 of the series we completed our first version of our application enclave and tested it by hardcoding the enclave support to be on. That was done by setting the *_supports_sgx* flag in PasswordCoreNative.cpp.

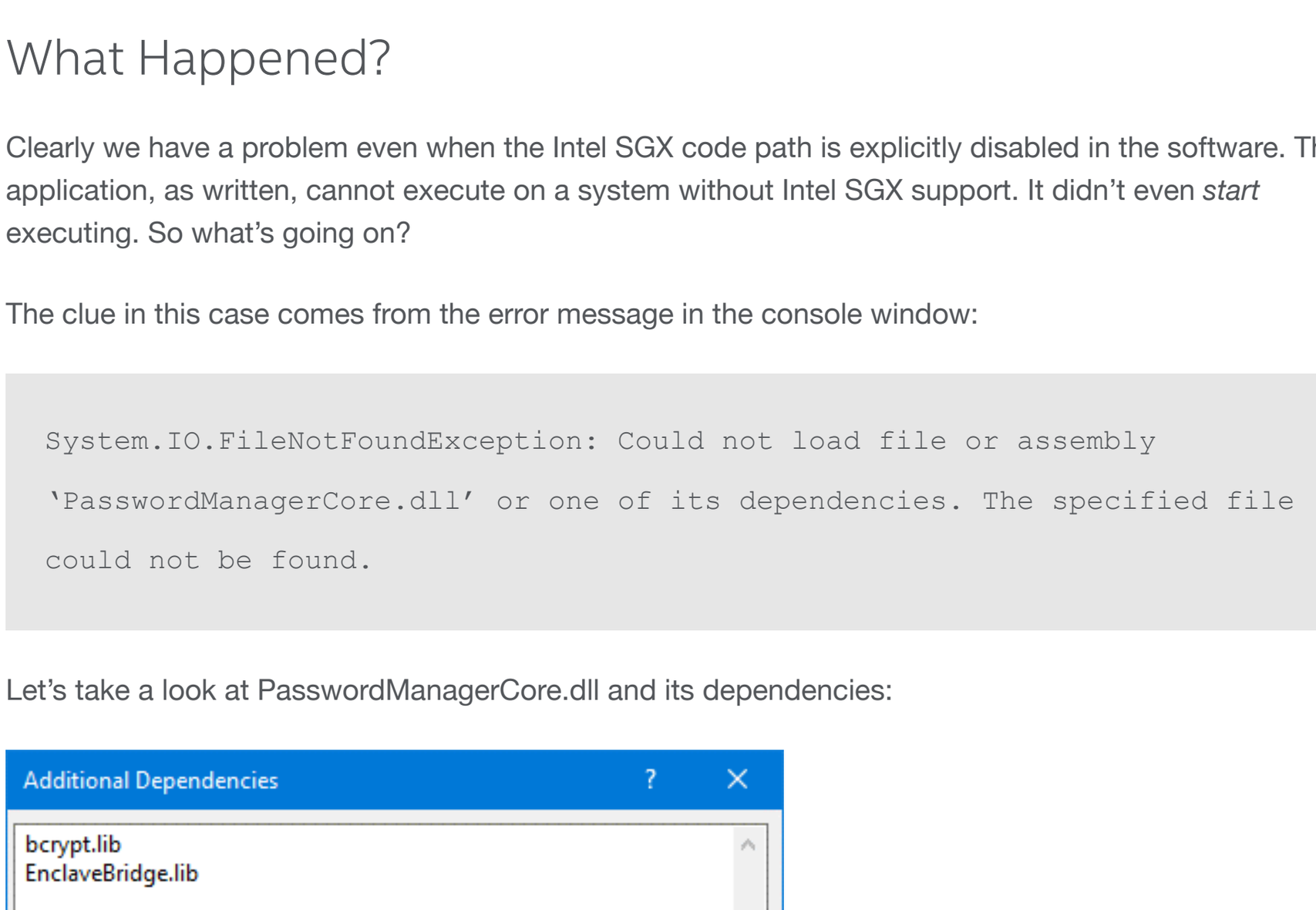
```
PasswordManagerCoreNative::PasswordManagerCoreNative(void)
{
    _supports_sgx= 1;
    adsize= 0;
    accountdata= NULL;
    timer = NULL;
}
```

Obviously, we can't leave this on by default. The convention for feature detection is that features are off by default and turned on if they are detected. So our first step is to undo this change and set the flag back to 0, effectively disabling the Intel SGX code path.

```
PasswordManagerCoreNative::PasswordManagerCoreNative(void)
{
    _supports_sgx= 0;
    adsize= 0;
    accountdata= NULL;
    timer = NULL;
}
```

However, before we get into the feature detection procedure, we'll give the console application that runs our test suite, CLI Test App, a quick functional test by executing it on an older system that does not have the Intel SGX feature. With this flag set to zero, the application will not choose the Intel SGX code path and thus should run normally.

Here's the output from a 4th generation Intel® Core™ i7 processor-based laptop, running Microsoft Windows* 8.1, 64-bit. This system does not support Intel SGX.



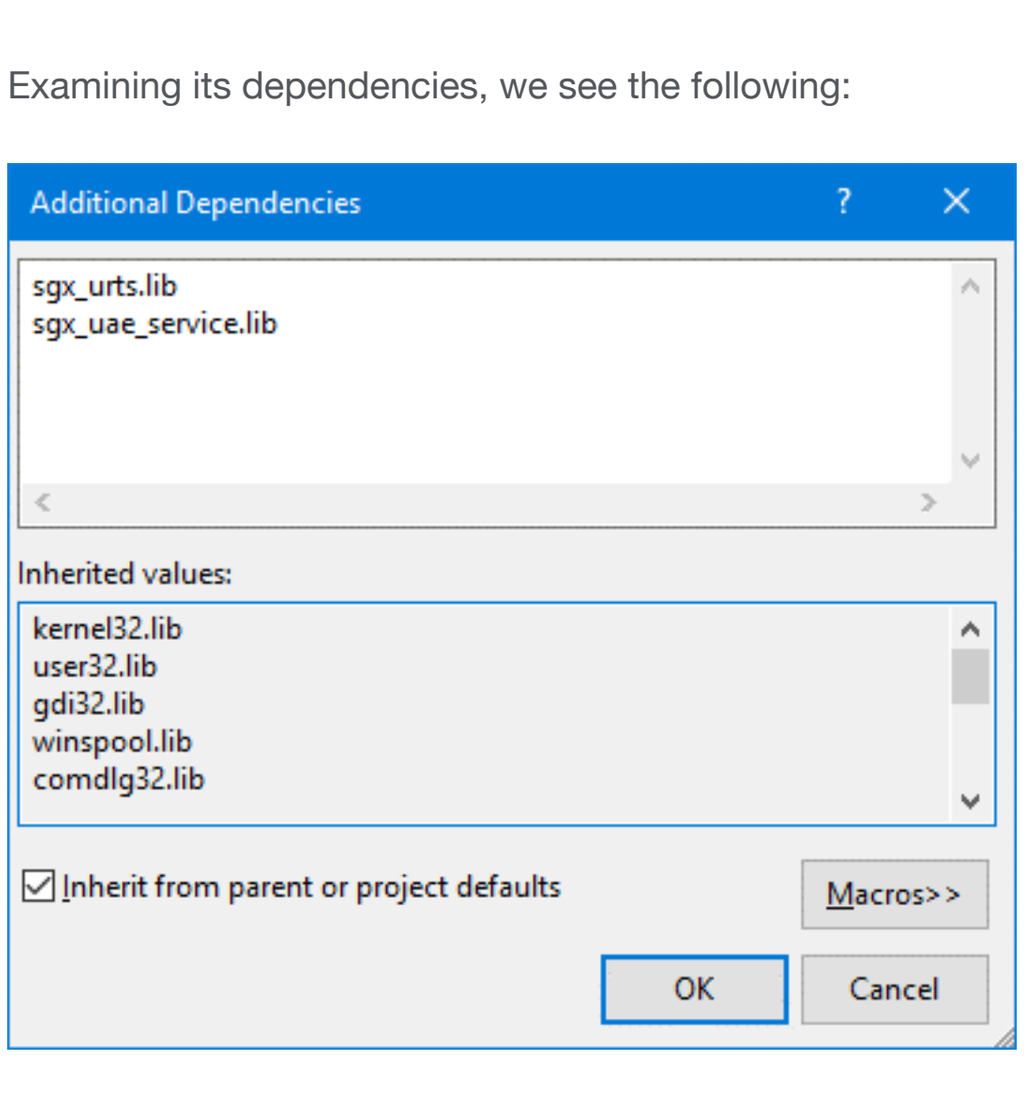
What Happened?

Clearly we have a problem even when the Intel SGX code path is explicitly disabled in the software. This application, as written, cannot execute on a system without Intel SGX support. It didn't even *start* executing. So what's going on?

The clue in this case comes from the error message in the console window:

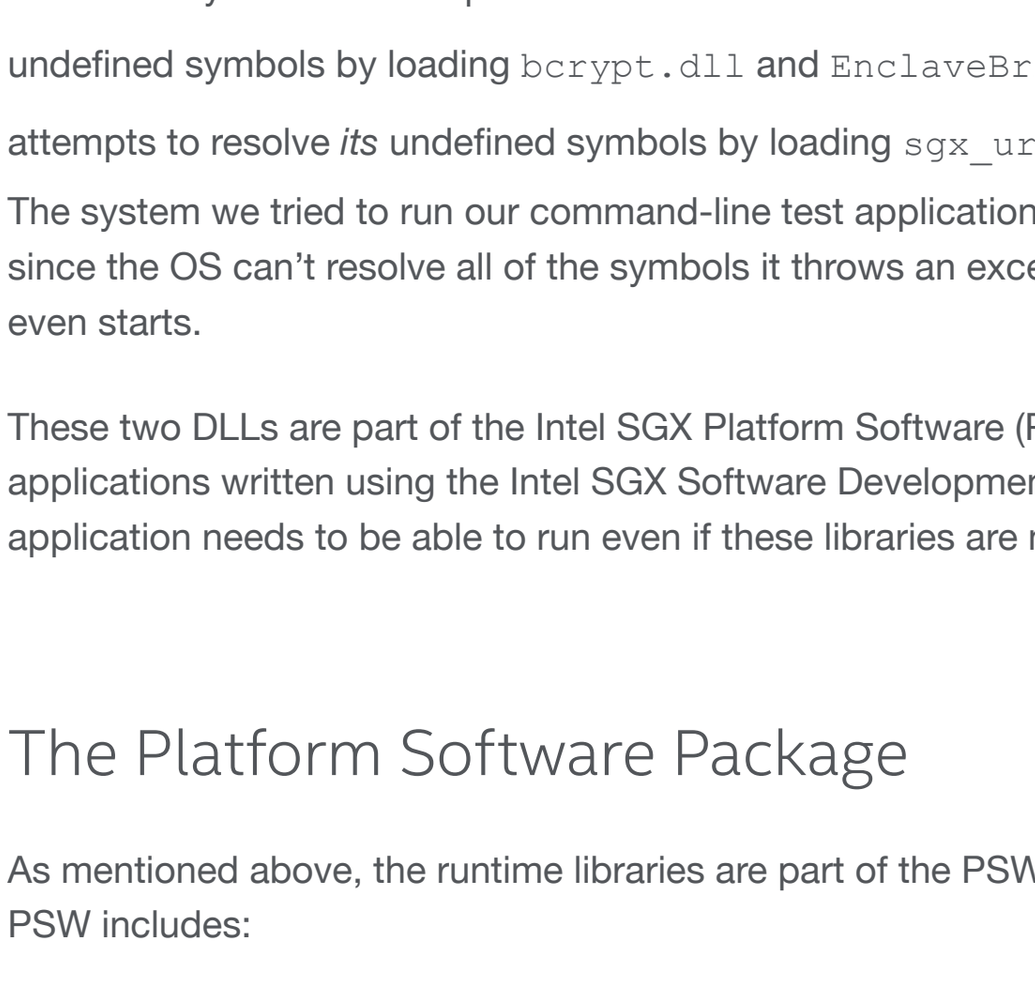
```
System.IO.FileNotFoundException: Could not load file or assembly
'PasswordManagerCore.dll' or one of its dependencies. The specified file
could not be found.
```

Let's take a look at PasswordManagerCore.dll and its dependencies:



In addition to the core OS libraries, we have dependencies on `bcrypt.lib` and `EnclaveBridge.lib`, which will require `bcrypt.dll` and `EnclaveBridge.dll` at runtime. Since `bcrypt.dll` comes from Microsoft and is included in the OS, we can reasonably assume its dependencies, if any, are already installed. That leaves `EnclaveBridge.dll`.

Examining its dependencies, we see the following:



This is the problem. Even though we have the Intel SGX code path explicitly disabled, `EnclaveBridge.dll` still has references to the Intel SGX runtime libraries. All symbols in an object module *must* be resolved as soon as it is loaded. It doesn't matter if we disable the Intel SGX code path: undefined symbols are still present in the DLL. When `PasswordManagerCore.dll` loads, it resolves its undefined symbols by loading `bcrypt.dll` and `EnclaveBridge.dll`, the latter of which, in turn, attempts to resolve *its* undefined symbols by loading `sgx_urts.dll` and `sgx_uae_service.dll`. The system we tried to run our command-line test application on does not have these libraries, and since the OS can't resolve all of the symbols it throws an exception and the program crashes before it even starts.

These two DLLs are part of the Intel SGX Platform Software (PSW) package, and without them Intel SGX applications written using the Intel SGX Software Development Kit (SDK) cannot execute. Our application needs to be able to run even if these libraries are not present.

The Platform Software Package

As mentioned above, the runtime libraries are part of the PSW. In addition to these support libraries, the PSW includes:

- Services that support and maintain the trusted compute block (TCB) on the system
- Services that perform and manage certain Intel SGX operations such as attestation
- Interfaces to platform services such as trusted time and the monotonic counters

The PSW must be installed by the application installer when deploying an Intel SGX application, because Intel does not offer the PSW for direct download by end users. Software vendors must not assume that it will already be present and installed on the destination system. In fact, the license agreement for Intel SGX specifically states that licensees must re-distribute the PSW with their applications.

We'll discuss the PSW installer in more detail in a future installment of the series covering packaging and deployment.

Detecting Intel Software Guard Extensions Support

So far we've focused on the problem of just *starting* our application on systems without Intel SGX support, and more specifically, without the PSW. The next step is to detect whether or not Intel SGX support is present and enabled once the application is running.

Intel SGX feature detection is, unfortunately, a complicated procedure. For a system to be Intel SGX capable, four conditions must be met:

1. The CPU must support Intel SGX.
2. The BIOS must support Intel SGX.
3. In the BIOS, Intel SGX must be explicitly enabled or set to the "software controlled" state.
4. The PSW must be installed on the platform.

Note that the CPUID instruction, alone, is not sufficient to detect the usability of Intel SGX on a platform. It can tell you whether or not the CPU supports the feature, but it doesn't know anything about the BIOS configuration or the software that is installed on a system. Relying solely on the CPUID results to make decisions about Intel SGX support can potentially lead to a runtime fault.

To make feature detection even more difficult, examining the state of the BIOS is not a trivial task and is generally not possible from a user process. Fortunately the Intel SGX SDK provides a simple solution: the function sgx_enable_device will both check for Intel SGX capability and attempt to enable it if the BIOS is set to the software control state (the purpose of the software control setting is to allow applications to enable Intel SGX without requiring users to reboot their systems and enter their BIOS setup screens, a particularly daunting and intimidating task for non-technical users).

The problem with sgx_enable_device, though, is that it is part of the Intel SGX runtime, which means the PSW must be installed on the system in order to use it. So before we attempt to call sgx_enable_device, we must first detect whether or not the PSW is present.

Implementation

With our problem scoped out, we can now lay out the steps that must be followed, in order, for our dual-code path application to function properly. Our application must:

1. Load and begin executing even without the Intel SGX runtime libraries.
2. Determine whether or not the PSW package is installed.
3. Determine whether or not Intel SGX is enabled (and attempt to enable it).

Loading and Executing without the Intel Software Guard Extensions Runtime

Our main application depends on PasswordManagerCore.dll, which depends on EnclaveBridge.dll, which in turn depends on the Intel SGX runtime. Since all symbols need to be resolved when an application loads, we need a way to prevent the loader from trying to resolve symbols that come from the Intel SGX runtime libraries. There are two options:

Option #1: Dynamic Loading

In dynamic loading, you don't explicitly link the library in the project. Instead you use system calls to load the library at runtime and then look up the names of each function you plan to use in order to get the addresses of where they have been placed in memory. Functions in the library are then invoked indirectly via function pointers.

Dynamic loading is a hassle. Even if you only need a handful of functions, it can be a tedious process to prototype function pointers for every function that is needed and get their load address, one at a time. You also lose some of the benefits provided by the integrated development environment (such as prototype assistance) since you are no longer explicitly calling functions by name.

Dynamic loading is typically used in extensible application architectures (for example, plug-ins).

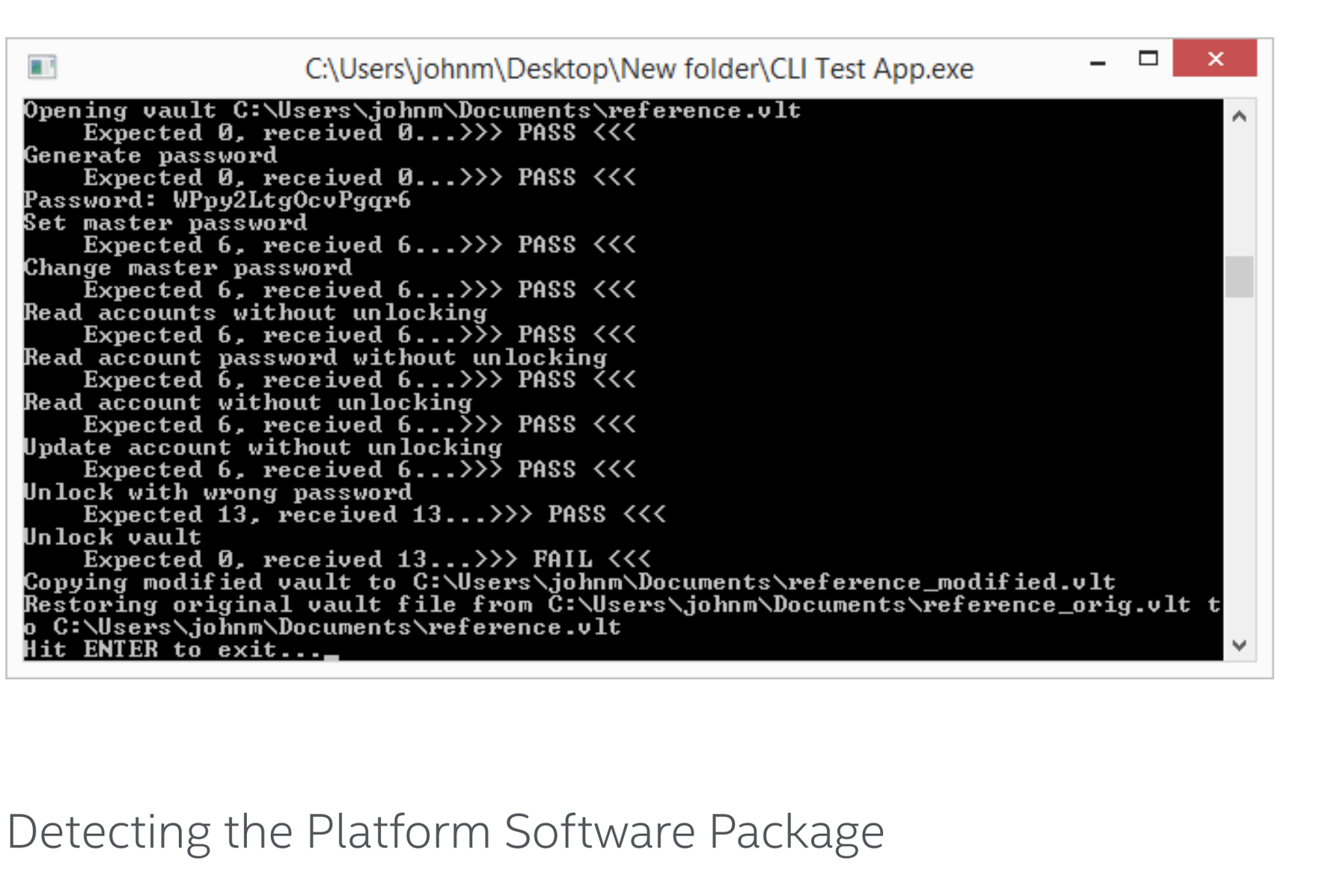
Option #2: Delayed-Loaded DLLs

In this approach, you dynamically link all your libraries in the project, but instruct Windows to do delayed loading of the problem DLL. When a DLL is delay-loaded, Windows does not attempt to resolve symbols that are defined by that DLL when the application starts. Instead it waits until the program makes its first call to a function that is defined in that DLL, at which point the DLL is loaded and the symbols get resolved (along with any of *its* dependencies). What this means is that a DLL is not loaded until the application needs it. A beneficial side effect of this approach is that it allows applications to reference a DLL that is not installed, so long as no functions in that DLL are ever called.

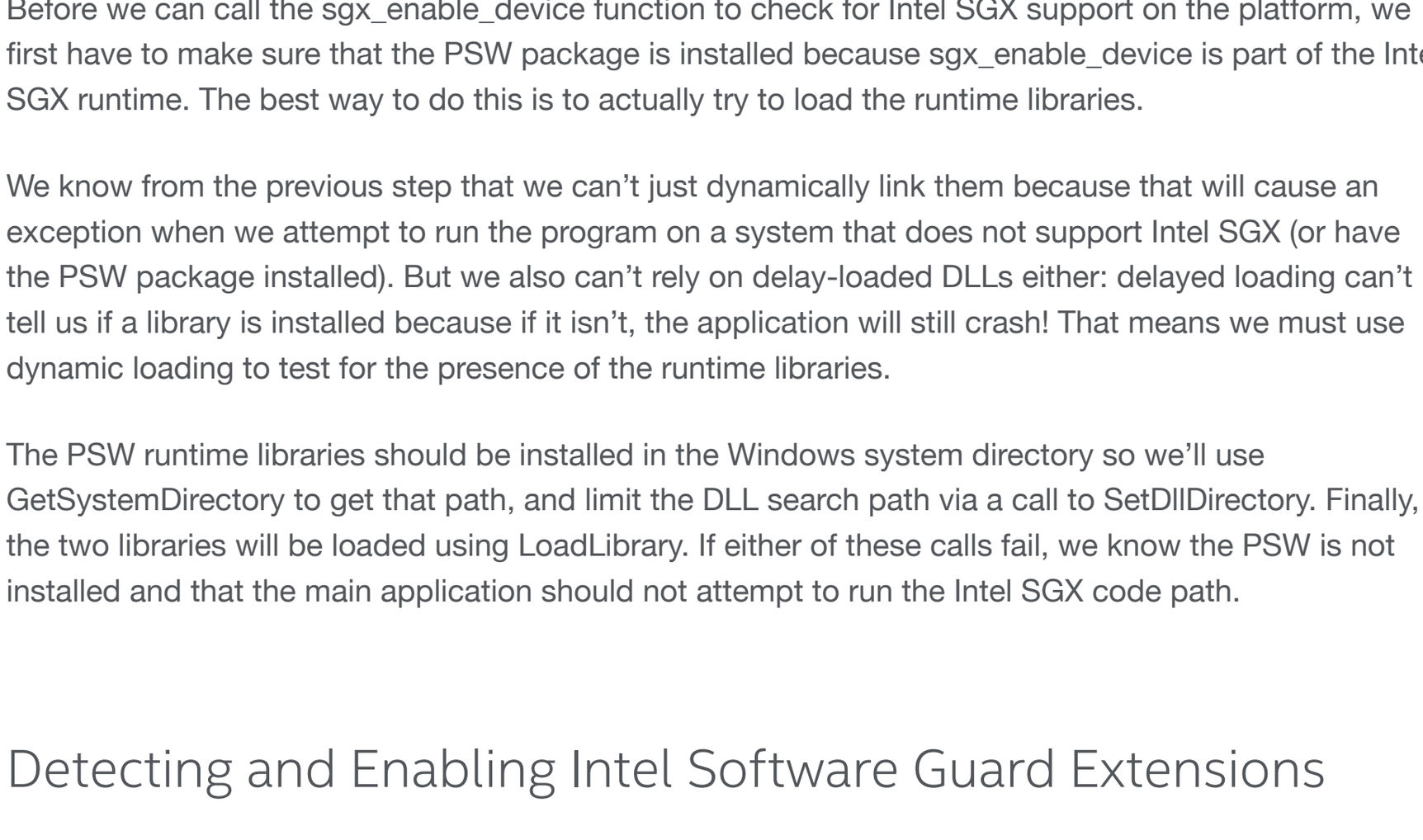
When the Intel SGX feature flag is off, that is exactly the situation we are in so we will go with option #2.

You specify the DLL to be delay-loaded in the project configuration for the dependent application or DLL. For the Tutorial Password Manager, the best DLL to mark for delayed loading is EnclaveBridge.dll as we only call this DLL if the Intel SGX path is enabled. If this DLL doesn't load, neither will the two Intel SGX runtime DLLS.

We set the option in the **Linker -> Input** page of the PasswordManagerCore.dll project configuration:



After the DLL is rebuilt and installed on our 4th generation Intel Core processor system, the console test application works as expected.



Detecting the Platform Software Package

Before we can call the sgx_enable_device function to check for Intel SGX support on the platform, we first have to make sure that the PSW package is installed because sgx_enable_device is part of the Intel SGX runtime. The best way to do this is to actually try to load the runtime libraries.

We know from the previous step that we can't just dynamically link them because that will cause an exception when we attempt to run the program on a system that does not support Intel SGX (or have the PSW package installed). But we also can't rely on delay-loaded DLLs either: delayed loading can't tell us if a library is installed because if it isn't, the application will still crash! That means we must use dynamic loading to test for the presence of the runtime libraries.

The PSW runtime libraries should be installed in the Windows system directory so we'll use GetSystemDirectory to get that path, and limit the DLL search path via a call to SetDllDirectory. Finally, the two libraries will be loaded using LoadLibrary. If either of these calls fail, we know the PSW is not installed and that the main application should not attempt to run the Intel SGX code path.

Detecting and Enabling Intel Software Guard Extensions

Since the previous step dynamically loads the PSW runtime libraries, we can just look up the symbol for sgx_enable_device manually and then invoke it via a function pointer. The result will tell us whether or not Intel SGX is enabled.

Implementation

To implement this in the Tutorial Password Manager we'll create a new DLL called FeatureSupport.dll. We can safely dynamically link this DLL from the main application since it has no explicit dependencies on other DLLs.

Our feature detection will be rolled into a C++/CLI class called FeatureSupport, which will also include some high-level functions for getting more information about the state of Intel SGX. In rare cases, enabling Intel SGX via software may require a reboot, and in rarer cases the software enable action fails and the user may be forced to enable it explicitly in their BIOS.

The class declaration for FeatureSupport is shown below.

```

    // Function pointers

    fp_sgx_enable_device_t fp_sgx_enable_device;

    int is_psw_installed(void);
    void check_sgx_support(void);
    void load_functions(void);

public:
    FeatureSupport();
    ~FeatureSupport();

    UINT get_sgx_support(void);
    int is_enabled(void);
    int is_supported(void);
    int reboot_required(void);
    int bios_enable_required(void);

    // Wrappers around SGX functions

    sgx_status_t enable_device(sgx_device_status_t *device_status);

};
```

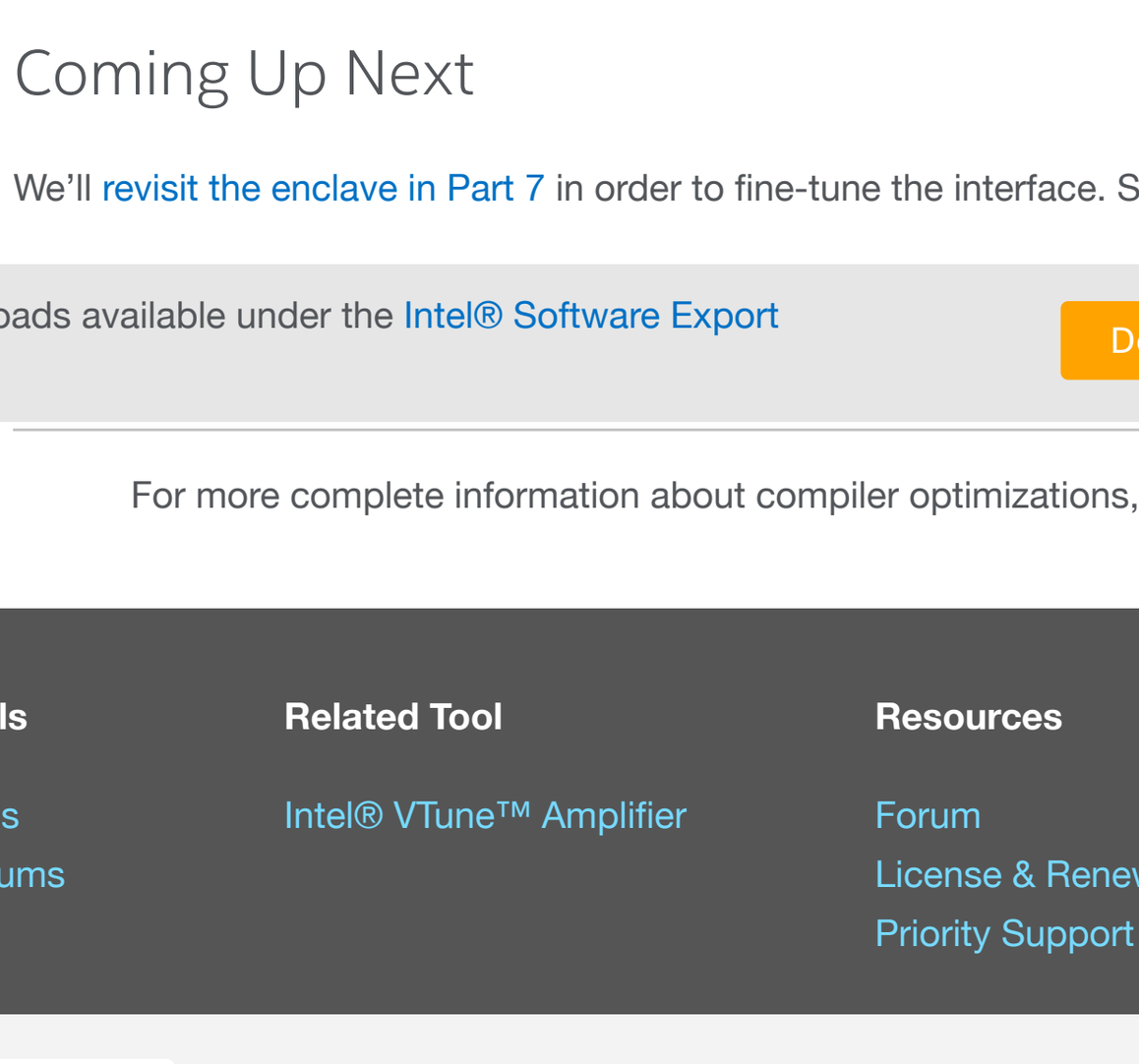
Here are the low-level routines that check for the PSW package and attempt to detect and enable Intel SGX.

```
int FeatureSupport::is_psw_installed()
{
    _TCHAR *systemdir;
    UINT rv, sz;

    // Get the system directory path. Start by finding out how much space we need
    // to hold it.

    sz = GetSystemDirectory(NULL, 0);
    if (sz == 0) return 0;

    systemdir = new _TCHAR[sz + 1];
    rv = GetSystemDirectory(systemdir, sz);
    if (rv == 0 || rv > sz) return 0;

    // Set our DLL search path to just the System directory so we don't accidentally
    // load the DLLs from an untrusted path.

    if (SetDllDirectory(systemdir) == 0) {
        delete systemdir;
        return 0;
    }
```

Wrapping Up

With these code changes, we have integrated Intel SGX feature detection into our application! It will execute smoothly on systems both with and without Intel SGX support and choose the appropriate code branch.

As mentioned in the introduction, there is sample code provided with this part for you to download. The attached archive includes the source code for the Tutorial Password Manager core, including the new feature detection DLL. Additionally, we have added a new GUI-based test program that automatically selects the Intel SGX code path, but lets you disable it if desired (this option is only available if Intel SGX is supported on the system).



The console-based test program has also been updated to detect Intel SGX, though it cannot be configured to turn it off without modifying the source code.

Coming Up Next

We'll revisit the enclave in Part 7 in order to fine-tune the interface. Stay tuned!

There are downloads available under the Intel® Software Export Warning license. Download Now

For more complete information about compiler optimizations, see our Optimization Notice.