Intel® Software Guard Extensions Part 3: Design an Application

By John M., Isayah R., published on August 29, 2016

In Part 3 of the Intel® Software Guard Extensions (Intel® SGX) tutorial series we'll talk about how to design an application with Intel SGX in mind. We'll take the concepts that we reviewed in Part 1, and apply them to the high-level design of our sample application, the Tutorial Password Manager, laid out in Part 2. We'll look at the overall structure of the application and how it is impacted by Intel SGX and create a class model that will prepare us for the enclave design and integration.

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

While we won't be coding up enclaves or enclave interfaces just yet, there is source code provided with this installment. The non-Intel SGX version of the application core, without its user interface, is available for download. It comes with a small test program, a console application written in C#, and a sample password vault file.

Designing for Enclaves

This is the general approach we'll follow for designing the Tutorial Password Manager for Intel SGX:

1. Identify the application's secrets.
2. Identify the providers and consumers of those secrets.
3. Determine the enclave boundary.
4. Tailor the application components for the enclave.

Identify the Application's Secrets

The first step in designing an application for Intel SGX is to identify the application's secrets.

A secret is anything that is not meant to be known or seen by others. Only the user or the application for which it is intended should have access to a secret, and it should not be exposed to others users or applications regardless of their privilege level. Potential secrets can include financial information, medical records, personally identifiable information, identity data, licensed media content, passwords, and encryption keys.

In the Tutorial Password Manager, there are several items that are immediately identifiable as secrets, shown in Table 1.

Secret
The user's account passwords
The user's account logins
The user's master password or passphrase
The master key for the password vault
The encryption key for the account database

Table 1: *Preliminary list of application secrets.*

These are the obvious choices, but we're going to expand this list by including *all* of the user's account information and not just their logins. The revised list is shown in Table 2.

Secret
The user's account passwords
The user's account ~~logins~~ information
The user's master password or passphrase
The master key for the password vault
The encryption key for the account database

Table 2: *Revised list of application secrets.*

Even without revealing the passwords, the account information (such as the service names and URLs) is valuable to attackers. Exposing this data in the password manager leaks valuable clues to those with malicious intent. With this data, they can choose to launch attacks against the services themselves, perhaps using social engineering or password reset attacks, to obtain access to the owner's account because they know exactly who to target.

Identify the Providers and Consumers of the Application's Secrets

Once the application's secrets have been identified, the next step is to determine their origins and destinations.

In the current version of Intel SGX, the enclave code is not encrypted, which means that anyone with access to the application files can disassemble and inspect it. By definition, something cannot be a secret if it is open to inspection, and that means that secrets should never be statically compiled into enclave code. An application's secrets must originate from outside its enclaves and be loaded into them at runtime. In Intel SGX terminology, this is referred to as provisioning secrets into the enclave.

When a secret originates from a component outside of the Trusted Compute Base (TCB), it is important to minimize its exposure to untrusted code. (One of the main reasons why remote attestation is such a valuable component of Intel SGX is that it allows a service provider to establish a trusted relationship with an Intel SGX application, and then derive an encryption key that can be used to provision encrypted secrets to the application that only the trusted enclave on that client system can decrypt.) Similar care must be taken when a secret is exported out of an enclave. As a general rule, an application's secrets should not be sent to untrusted code without first being encrypted inside of the enclave.

Unfortunately for the Tutorial Password Manager application, we *do* need to send secrets into and out of the enclave, and those secrets will have to exist in clear text at some point. The end user will be entering his or her account information and password via a keyboard or touchscreen, and recalling it at a future time as needed. Their account passwords will need to be shown on the screen, and even copied to the Windows* clipboard on request. These are core requirements for a password manager application to be useful.

What that means for us is that we can't completely eliminate the attack surface: we can only minimize it, and we'll need some mitigation strategy for dealing with secrets when they exist outside the enclave in plain text.

Secret	Source	Destination
The user's account passwords	User input* Password vault file	User interface* Clipboard* Password vault file
The user's account information	User input* Password vault file	User interface* Password vault file
The user's master password or passphrase	User input	Key derivation function
The master key for the password vault	Key derivation function	Database key crypto
The encryption key for the password database	Random generation Password vault file	Password vault crypto Password vault fil

Table 3: *Application secrets, their sources, and their destinations. Potential security risks are denoted with an asterisk (*).*

Table 3 adds the sources and destinations for the Tutorial Password Manager's secrets. Potential problems—areas where secrets may be exposed to untrusted code—are denoted with an asterisk (*).

Determine the Enclave Boundary

Once the secrets have been identified, it's time to determine the boundary for the enclave. Start by looking at the data flow of secrets through the application's core components. The enclave boundary should:

- Encompass the minimum set of critical components that act on your application's secrets.
- Completely contain as many secrets as is feasible.
- Minimize the interactions with, and dependencies on, untrusted code.

The data flows and chosen enclave boundary for the Tutorial Password Manager application are shown in Figure 1.

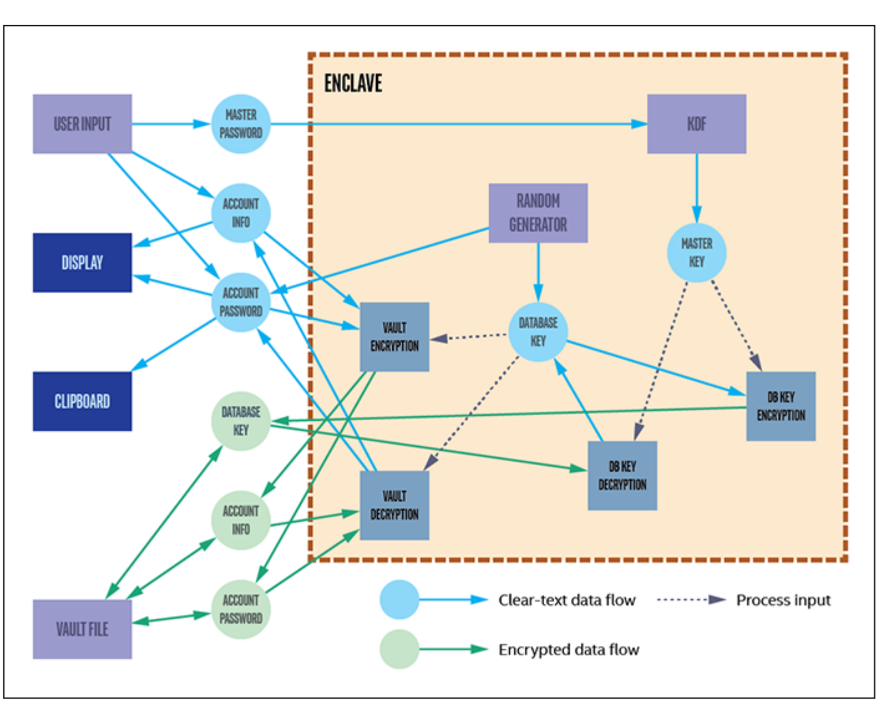

Figure 1: *Data flow for secrets in the Tutorial Password Manager.*

Here, the application secrets are depicted as circles, with blue circles representing secrets that will exist in plain text (unencrypted) at some point during the application's execution and green circles representing secrets that are encrypted by the application. The enclave boundary has been drawn around the encryption and decryption routines, the key derivation function (KDF) and the random number generator. This does several things for us:

1. The database/vault key, which is used to encrypt some of our application's secrets (account information and passwords), is generated within the enclave and is never sent outside of it in clear text.
2. The master key is derived from the user's passphrase inside the enclave, and used to encrypt and decrypt the database/vault key. The master key is ephemeral and is never sent outside the enclave in any form.
3. The database/vault key, account information, and account passwords are encrypted inside the enclave using encryption keys that are not visible to untrusted code (see #1 and #2).

Unfortunately, we have issues with unencrypted secrets crossing the enclave boundary that we simply can't avoid. At some point during the Tutorial Password Manager's execution, a user will have to enter a password on the keyboard or copy a password to the Windows clipboard. These are insecure channels that can't be placed inside the enclave, and the operations are absolutely necessary for the functioning of the application. This is potentially a huge problem, which is compounded by the decision to build the application on top of a managed code base.

Protecting Secrets Outside the Enclave

There are no complete solutions for securing unencrypted secrets outside the enclave, only mitigation strategies that reduce the attack surface. The best we can do is minimize the amount of time that this information exists in a form that is easily compromised.

Here is some general advice for handling sensitive data in untrusted code:

- Zero-fill your data buffers when you are done with them. Be sure to use functions such as *SecureZeroMemory* (Windows) and *memzero_explicit* (Linux) that are guaranteed to not be optimized out by the compiler.
- Do not use the C++ standard template library (STL) containers to store sensitive data. The STL containers have their own memory management, which makes it difficult to ensure that the memory allocated to an object is securely wiped when the object is deleted. (By using custom allocators you can address this issue for some containers.)
- When working with managed code such as .NET, or languages that feature automatic memory management, use storage types that are specifically designed for holding secure data. Other storage types are at the mercy of the garbage collector and just-in-time compilation, and may not be cleared or freed on demand (if at all).
- If you must place data on the clipboard be sure to clear it after a short length of time. In particular, don't allow it to remain there after the application has exited.

For the Tutorial Password Manager project, we have to work with both native and managed code. In native code, we'll allocate **wchar_t** and **char** buffers, and use *SecureZeroMemory* to wipe them clean before freeing them. In the managed code space, we'll employ .NET's **SecureString** class.

When sending a SecureString to unmanaged code, we'll use the helper functions from System::Runtime::InteropServices to marshal the data.

```
using namespace System::Runtime::InteropServices;

LPWSTR PasswordManagerCore::M_SecureString_to_LPWSTR(SecureString ^ss)
{
    IntPtr wsp= IntPtr::Zero;

    if (!ss) return NULL;

    wsp = Marshal::SecureStringToGlobalAllocUnicode(ss);
    return (wchar_t *) wsp.ToPointer();
}
```

When marshaling data in the other direction, from native code to managed code, we have two methods. If the **SecureString** object already exists, we'll use the *Clear* and *AppendChar* methods to set the new value from the **wchar_t** string.

```
password->Clear();
for (int i = 0; i < wpass_len; ++i) password->AppendChar(wpass[i]);
```

When creating a new **SecureString** object, we'll use the constructor form that creates a **SecureString** from an existing **wchar_t** string.

```
try {
    name = gcnew SecureString(wname, (int) wcslen(wname));
    login = gcnew SecureString(wlogin, (int) wcslen(wlogin));
    url = gcnew SecureString(wurl, (int) wcslen(wurl));
}
catch (...) {
    rv = NL_STATUS_ALLOC;
}
```

Our password manager also supports transferring passwords to the Windows clipboard. The clipboard is an insecure storage space that can potentially be accessed by other users and for this reason Microsoft recommends that sensitive data never be placed on there. The point of a password manager, though, is to make it possible for users to create strong passwords that they do not have to remember. It also makes it possible to create lengthy passwords consisting of randomly generated characters which would be difficult to type by hand. The clipboard provides much needed convenience in exchange for some measure of risk.

To mitigate this risk, we need to take some extra precautions. The first is to ensure that the clipboard is emptied when the application exits. This is accomplished in the destructor in one of our native objects.

```
PasswordManagerCoreNative::~PasswordManagerCoreNative(void)
{
    if (!OpenClipboard(NULL)) return;
    EmptyClipboard();
    CloseClipboard();
}
```

We'll also set up a clipboard timer. When a password is copied to the clipboard, set a timer for 15 seconds and execute a function to clear the clipboard when it fires. If a timer is already running, meaning a new password was placed on the clipboard before the old one was expired, that timer is cancelled and the new one takes its place.

```
void PasswordManagerCoreNative::start_clipboard_timer()
{
    // Use the default Timer Queue

    // Stop any existing timer
    if (timer != NULL) DeleteTimerQueueTimer(NULL, timer, NULL);

    // Start a new timer
    if (!CreateTimerQueueTimer(&timer, NULL,
(WAITORTIMERCALLBACK)clear_clipboard_proc,
        NULL, CLIPBOARD_CLEAR_SECS * 1000, 0, 0)) return;
}

static void CALLBACK clear_clipboard_proc(PVOID param, BOOLEAN fired)
{
    if (!OpenClipboard(NULL)) return;
    EmptyClipboard();
    CloseClipboard();
}
```

Tailor the Application Components for the Enclave

With the secrets identified and the enclave boundary drawn, it's time to structure the application while taking the enclave into account. There are significant restrictions on what can be done inside of an enclave, and these restrictions will mandate which components live inside the enclave, which live outside of it, and when porting an existing applications, which ones may need to be split in two.

The biggest restriction that impacts the Tutorial Password Manager is that enclaves cannot perform any I/O operations. The enclave can't read from the keyboard or write to the display so all of our secrets—passwords and account information—must be marshaled into and out of the enclave. It also can't read from or write to the vault file: the components that parse the vault file must be separated from components that perform the physical I/O. That means we are going to have to marshal more than just our secrets across the enclave boundary: we have to marshal the file contents as well.

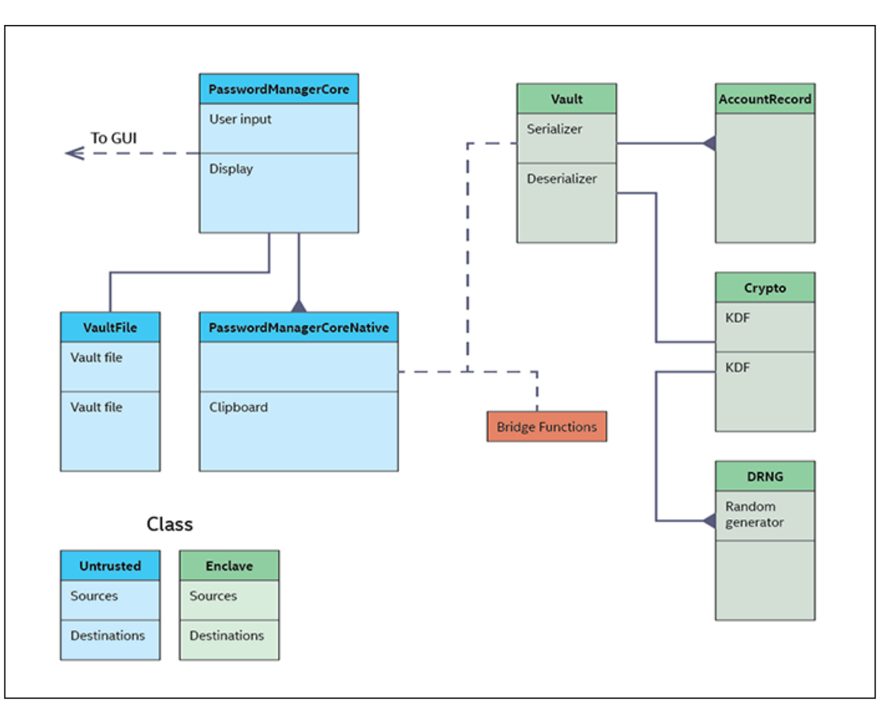

Figure 2: *Class diagram for the Tutorial Password Manager.*

Figure 2 shows the basic class diagram for the application core (excluding the user interface), including which classes serve as the sources and destinations for our secrets. Note that the **PasswordManagerCore** class is considered the source and destination for secrets which must interact with the GUI in this diagram for simplicity's sake. Table 4 briefly describes each class and its purpose.

Class	Type	Function
PasswordManagerCore	Managed	Interact with the C# graphical user interface (GUI) and marshal data to the native layer.
PasswordManagerCoreNative	Native, Untrusted	Interact with the managed **PasswordManagerCore** class. Also responsible for converting between Unicode and multibyte character data (this will be discussed in more detail in Part 4).
VaultFile	Managed	Reads and writes from the vault file.
Vault	Native, Enclave	Stores the password vault data in **AccountRecord** members. Deserializes the vault file on reads, and reserializes it for writing.
AccountRecord	Native, Enclave	Stores the account information and password for each account in the user's password vault.
Crypto	Native, Enclave	Performs cryptographic functions.
DRNG	Native, Enclave	Interface to the random number generator.

Table 4: *Class descriptions.*

Note that we had to split the handling of the vault file into two pieces: one that does the physical I/O, and one that stores its contents once they are read and parsed. We also had to add serialization and deserialization methods to the **Vault** object as intermediate sources and destinations for our secrets. All of this is necessary because the **VaultFile** class can't know anything about the structure of the vault file itself, since that would require access to cryptographic functions that are located inside the enclave.

We've also drawn a dotted line when connecting the **PasswordManagerCoreNative** class to the **Vault** class. As you might recall from Part 2, enclaves can only link to C functions. These two C++ classes cannot directly communicate with one another: they must use an intermediary which is denoted by the Bridge Functions box.

The Non-Intel® Software Guard Extensions Code Path

The diagram in Figure 2 is for the Intel SGX code path. The **PasswordManagerCoreNative** class cannot link directly to the **Vault** class because the latter is inside the enclave. In the non-Intel SGX code path, however, there is no such restriction: **PasswordManagerCoreNative** can directly contain a member of class **Vault**. *This is the only shortcut we'll take in the application design for the non-Intel SGX code path.* To simplify the enclave integration, the non-enclave code path will still separate the vault processing into the **Vault** and **VaultFile** classes.

Another key difference between the two code paths is that the cryptographic functions in the Intel SGX path will come from the Intel SGX SDK. The non-Intel SGX code path can't use these functions, so they will draw upon Microsoft's Cryptography API: Next Generation* (CNG). That means we have to maintain two, distinct copies of the **Crypto** class: one for use in enclaves and one for use in untrusted space. (We'll have to do the same with other classes, too; this will be discussed in Part 5.)

Sample Code

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 DLL, prior to enclave integration. In other words, this is the non-Intel SGX version of the application core. There is no user interface provided, but we have included a rudimentary test application written in C# that runs through a series of test operations. It executes two test suites: one that creates a new vault file and performs various operations on it, and one that acts on a reference vault file that is included with the source distribution. As written, the test application expects the test vault to be located in your Documents folder, though you can change this in the **TestSetup** class if needed.

This source code was developed in Microsoft Visual Studio* Professional 2013 per the requirements stated in the introduction to the tutorial series. It does not require the Intel SGX SDK at this point, though you will need a system that supports Intel® Data Protection Technology with Secure Key.

Coming Up Next

In Part 4 of the tutorial we'll develop the enclave and the bridge functions. Stay tuned!

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