Intel® Software Guard Extensions Tutorial Series: Part 2, Application Design

By John M., published on July 26, 2016

The second part in the Intel® Software Guard Extensions (Intel® SGX) tutorial series is a high-level specification for the application we'll be developing: a simple password manager. Since we're building this application from the ground up, we have the luxury of designing for Intel SGX from the start. That means that in addition to laying out our application's requirements, we'll examine how Intel SGX design decisions and the overall application architecture influence one another.

Read the first tutorial in the series or find the list of all of the published tutorials in the article Introducing the Intel® Software Guard Extensions Tutorial Series.

Password Managers At-A-Glance

Most people are probably familiar with password managers and what they do, but it's a good idea to review the fundamentals before we get into the details of the application design itself.

The primary goals of a password manager are to:

- Reduce the number of passwords that end users need to remember.
- Enable end users to create stronger passwords than they would normally choose on their own.
- Make it practical to use a different password for every account.

Password management is a growing problem for Internet users, and numerous studies have tried to quantify the problem over the years. A Microsoft study published in 2007—nearly a decade ago as of this writing—estimated that the average person had 25 accounts that required passwords. More recently, in 2014 Dashlane estimated that their US users had an average of 130 accounts, while the number of accounts for their worldwide users averaged in the 90s. And the problems don't end there: people are notoriously bad at picking "good" passwords, frequently reusing the same password on multiple sites, which has led to some spectacular attacks. These problems boil down to two basic issues: passwords that are hard for hacking tools to guess are often difficult for people to remember, and having a greater number of passwords makes this problem more complex by having to remember which password is associated with which account.

With a password manager, you only need to remember one very strong passphrase in order to gain access to your password database or vault. Once you have authenticated to your password manager, you can look up any passwords you have stored, and copy and paste them into authentication fields as needed. Of course, the key vulnerability of the password manager is the password database itself: since it contains all of the user's passwords it is an attractive target for attackers. For this reason, the password database is encrypted with strong encryption techniques, and the user's master passphrase becomes the means for decrypting the data inside of it.

Our goal in this tutorial is to build a simple password manager that provides the same core functions as a commercial product while following good security practices and use that as a learning vehicle for designing for Intel SGX. The tutorial password manager, which we'll name the "Tutorial Password Manager with Intel® Software Guard Extensions" (yes, that's a mouthful, but it's descriptive), is not intended to function as a commercial product and certainly won't contain all the safeguards found in one, but that level of detail is not necessary.

Basic Application Requirements

Some basic application requirements will help narrow down the scope of the application so that we can focus on the Intel SGX integration rather than the minutiae of application design and development. Again, the goal is not to create a commercial product: the Tutorial Password Manager with Intel SGX does not need to run on multiple operating systems or on all possible CPU architectures. All we require is a reasonable starting point.

To that end, our basic application requirements are:

REQUIREMENTS AND DESIGN DECISIONS	• Must run on all platforms regardless of whether or not they are Intel SGX capable • Requires Microsoft Windows Vista*, 64-bit, or later (Windows® 10, 64-bit, or later required for Intel SGX support) • Requires an Intel® processor that supports Intel® Data Protection Technology (Intel® DPT) with Secure Key • Must not rely on third-party libraries or utilities

The first requirement may seem strange given that this tutorial series is about Intel SGX application development, but real-world applications need to consider the legacy installation base. For some applications it may be appropriate to restrict execution only to Intel SGX-capable platforms, but for the Tutorial Password Manager we'll use a less rigid approach. An Intel SGX-capable platform will receive a *hardened* execution environment, but non-capable platforms will still function. This usage is appropriate for a password manager, where the user may need to synchronize his or her password database with other, older systems. It is also a learning opportunity for implementing dual code paths.

The second requirement gives us access to certain cryptographic algorithms in the non-Intel SGX code path and to some libraries that we'll need. The 64-bit requirement simplifies application development by ensuring access to native 64-bit types and also provides a performance boost for certain cryptographic algorithms that have been optimized for 64-bit code.

The third requirement gives us access to the RDRAND instruction in the non-Intel SGX code path. This greatly simplifies random number generation and ensures access to a high-quality entropy source. Systems that support the RDSEED instruction will make use of that as well. (For information on the RDRAND and RDSEED instructions, see the Intel® Digital Random Number Generator Software Implementation Guide.)

The fourth requirement keeps the list of software required by the developer (and the end user) as short as possible. No third-party libraries, frameworks, applications, or utilities need to be downloaded and installed. However, this requirement has an unfortunate side effect: without third-party frameworks, there are only four options available to us for creating the user interface. Those options are:

- Win32 APIs
- Microsoft Foundation Classes (MFC)
- Windows Presentation Foundation (WPF)
- Windows Forms

The first two are implemented in native/unmanaged code while the latter two require .NET*.

The User Interface Framework

For the Tutorial Password Manager, we're going to be developing the GUI using Windows Presentation Foundation in C#. This design decision impacts our requirements as follows:

REQUIREMENTS AND DESIGN DECISIONS	• WPF-based GUI • Microsoft .NET Framework 4.5.1 or later • ~~Requires Microsoft Windows Vista*, 64-bit, or later (Windows® 10, 64-bit, or later required for Intel SGX support)~~ • Requires Microsoft Windows Vista with SP2, 64-bit, or later (Windows 10, 64-bit, or later required for Intel SGX support)

Why use WPF? Mostly because it simplifies the UI design while introducing complexity that we actually *want*. Specifically, by relying on the .NET Framework, we have the opportunity to discuss mixing managed code, and specifically high-level languages, with enclave code. Note, though, that choosing WPF over Windows Forms was arbitrary: either environment would work.

As you might recall, enclaves must be written in native C or C++ code, and the bridge functions that interact with the enclave must be native C (not C++) functions. While both Win32 APIs and MFC provide an opportunity to develop the password manager with 100-percent native C/C++ code, the burden imposed by these two methods does nothing for those who want to learn Intel SGX application development. With a GUI based in managed code, we not only reap the benefits of the integrated design tools but also have the opportunity to discuss something that is of potential value to Intel SGX application developers. In short, you aren't here to learn MFC or raw Win32, but you might want to know how to glue .NET to enclaves.

To bridge the managed and unmanaged code we'll be using C++/CLI (C++ modified for Common Language Infrastructure). This greatly simplifies the data marshaling and is so convenient and easy to use that many developers refer to it as IJW ("It Just Works").

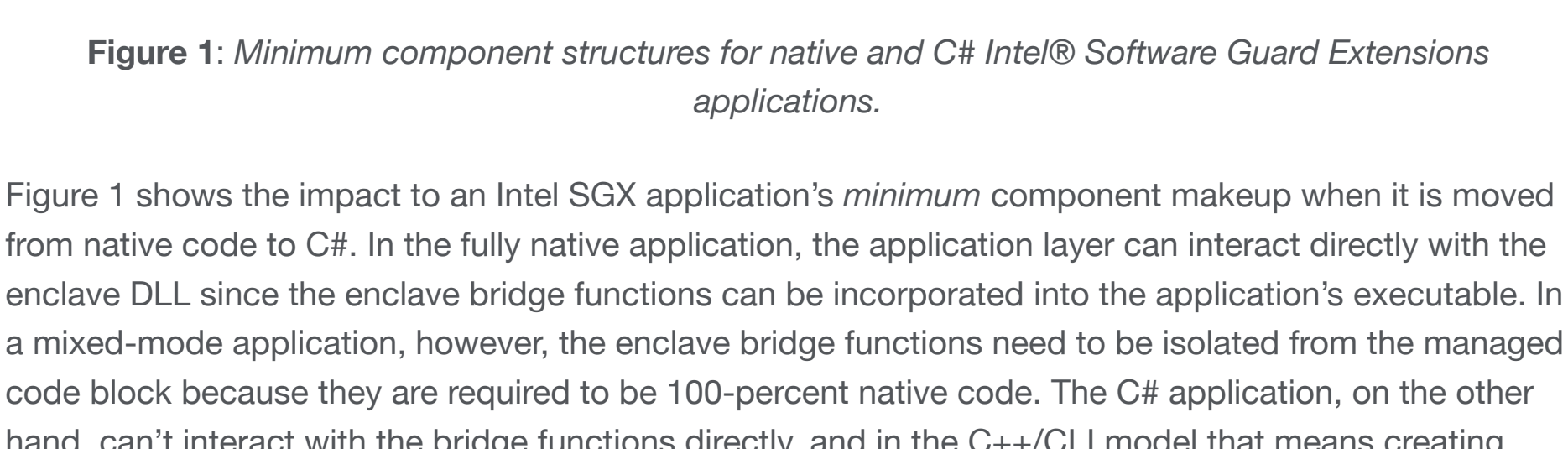

Figure 1: *Minimum component structures for native and C# Intel® Software Guard Extensions applications.*

Figure 1 shows the impact to an Intel SGX application's *minimum* component makeup when it is moved from native code to C#. In the fully native application, the application layer can interact directly with the enclave DLL since the enclave bridge functions can be incorporated into the application's executable. In a mixed-mode application, however, the enclave bridge functions need to be isolated from the managed code block because they are required to be 100-percent native code. The C# application, on the other hand, can't interact with the bridge functions directly, and in the C++/CLI model that means creating *another* intermediary: a DLL that marshals data between the managed C# application and the native, enclave bridge DLL.

Password Vault Requirements

At the core of the password manager is the password database, or what we'll be referring to as the password vault. This is the encrypted file that will hold the end user's account information and passwords. The basic requirements for our tutorial application are:

REQUIREMENTS AND DESIGN DECISIONS	• Password vault must be portable • Vault must be encrypted at rest • All encryption must use an authenticated encryption mode

The requirement that the vault be portable means that we should be able to copy the vault file to another computer and still be able to access its contents, whether or not they support Intel SGX. In other words, the user experience should be the same: the password manager should work seamlessly (so long as the system meets the base hardware and OS requirements, of course).

Encrypting the vault at rest means that the vault file should be encrypted when it is not actively in use. At a minimum, the vault must be encrypted on disk (without the portability requirement, we could potentially solve the encryption requirements by using the sealing feature of Intel SGX) and should not sit decrypted in memory longer than is necessary.

Authenticated encryption provides assurances that the encrypted vault has not been modified after the encryption has taken place. It also gives us a convenient means of validating the user's passphrase: if the decryption key is incorrect, the decryption will fail when validating the authentication tag. That way, we don't have to examine the decrypted data to see if it is correct.

Passwords

Any account information is sensitive information for a variety of reasons, not the least of which is that it tells an attacker exactly which logins and sites to target, but the passwords are arguably the most critical piece of the vault. Knowing what account to attack is not nearly as attractive as not needing to attack it at all. For this reason, we'll introduce additional requirements on the passwords stored in the vault:

REQUIREMENTS AND DESIGN DECISIONS	• Account passwords must be encrypted within the vault • Account passwords are only decrypted on-demand

This is nesting the encryption. The passwords for each of the user's accounts are encrypted when stored in the vault, and the entire vault is encrypted when written to disk. This approach allows us to limit the exposure of the passwords once the vault has been decrypted. It is reasonable to decrypt the vault as a whole so that the user can browse their account details, but displaying *all* of their passwords in clear text in this manner would be inappropriate.

An account password is only decrypted when a user asks to see it. This limits its exposure both in memory and on the user's display.

Cryptographic Algorithms

With the encryption needs identified it is time to settle on the specific cryptographic algorithms, and it's here that our existing application requirements impose some significant limits on our options. The Tutorial Password Manager must provide a seamless user experience on both Intel SGX and non-Intel SGX platforms, and it isn't allowed to depend on third-party libraries. That means we have to choose an algorithm, and a supported key and authentication tag size, that is common to both the Windows CNG API and the Intel SGX trusted crypto library. Practically speaking, this leaves us with just one option: Advanced Encryption Standard-Galois Counter Mode (AES-GCM) with a 128-bit key. This is arguably not the best encryption mode to use in this application, especially since the effective authentication tag strength of 128-bit GCM is less than 128 bits, but it is sufficient for our purposes. Remember: the goal here is not to create a commercial product, but rather a useful learning vehicle for Intel SGX development.

With GCM come some other design decisions, namely the IV length (12 bytes is most efficient for the algorithm) and the authentication tag.

REQUIREMENTS AND DESIGN DECISIONS	• Encryption will be AES-GCM
	  ○ 128-bit key
	  ○ 128-bit authentication tag
	  ○ No additional authenticated data (AAD)
	  ○ 96-bit IVs

Encryption Keys and User Authentication

With the encryption method chosen, we can turn our attention to the encryption key and user authentication. How will the user authenticate to the password manager in order to unlock their vault?

The simple approach would be to derive the encryption key directly from the user's passphrase or password using a key derivation function (KDF). But while the simple approach is a valid one, it does have one significant drawback: if the user changes his or her password, the encryption key changes along with it. Instead, we'll follow the more common practice of encrypting the encryption key.

In this method, the primary encryption key is randomly generated using a high-quality entropy source and it never changes. The user's passphrase or password is used to derive a secondary encryption key, and the secondary key is used to encrypt the primary key. This approach has some key advantages:

- The data does not have to be re-encrypted when the user's password or passphrase changes
- The encryption key never changes, so it could theoretically be written down in, say, hexadecimal notation and locked in a physically secure location. The data could thus still be decrypted even if the user forgot his or her password. Since the key never changes, it would only have to be written down once.
- More than one user could, in theory, be granted access to the data. Each would encrypt a copy of the primary key with their own passphrase.

Not all of these are necessarily critical or relevant to the Tutorial Password Manager, but it's a good security practice nonetheless.

REQUIREMENTS AND DESIGN DECISIONS	• The vault key will be generated from the RDSEED instruction or using seed-from-RDRAND on systems without RDSEED support • The vault key will be encrypted with a master key • The master key will be derived from the user's passphrase via a KDF • The KDF will be based on SHA-256

Here the primary key is called the vault key, and the secondary key that is derived from the user's passphrase is called the master key. The user authenticates by entering their passphrase, and the password manager derives a master key from it. If the master key successfully decrypts the vault key, the user is authenticated and the vault can be decrypted. If the passphrase is incorrect, the decryption of the vault key fails and that prevents the vault from being decrypted.

The final requirement, building the KDF around SHA-256, comes from the constraint that we find a hashing algorithm common to both the Windows CNG API and the Intel SGX trusted crypto library.

Account Details

The last of the high-level requirements is what actually gets stored in the vault. For this tutorial, we are going to keep things simple. Figure 2 shows an early mockup of the main UI screen.

REQUIREMENTS AND DESIGN DECISIONS	• The vault will store account information consisting of:
	  ○ The account/vendor name
	  ○ The account/vendor URL
	  ○ The login ID
	  ○ The authentication password
	• The vault will have a fixed number of accounts

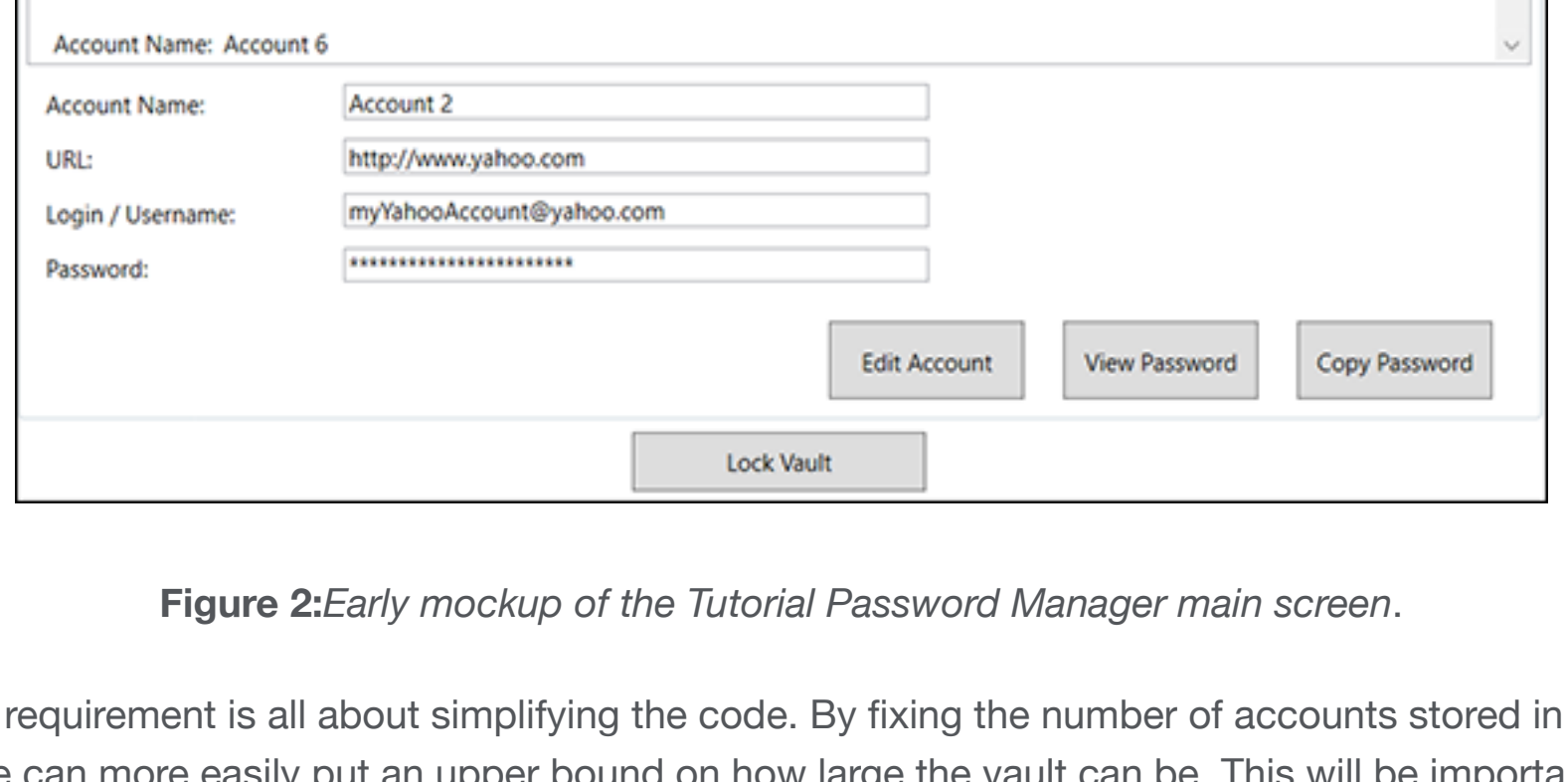

Figure 2:*Early mockup of the Tutorial Password Manager main screen*.

The last requirement is all about simplifying the code. By fixing the number of accounts stored in the vault, we can more easily put an upper bound on how large the vault can be. This will be important when we start designing our enclave. Real-world password managers do not, of course, have this luxury, but it is one that can be afforded for the purposes of this tutorial.

Coming Up Next

In part 3 of the tutorial we'll take a closer look at designing our Tutorial Password Manager for Intel SGX. We'll identify our secrets, which portions of the application should be contained inside the enclave, how the enclave will interact with the core application, and how the enclave impacts the object model. Stay tuned!

Read the first tutorial in the series, Intel® Software Guard Extensions Tutorial Series: Part 1, Intel® SGX Foundation or find the list of all the published tutorials in the article Introducing the Intel® Software Guard Extensions Tutorial Series.

Additional Resources

Get Started with the SDK

Intel® SGX Forum

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

1 comment

^Top

John M. said on Aug 19,2016

Part 3 is ready to go, but is awaiting legal approval. This will delay its release, but the good news is that this delay should only happen once (the other parts should not run into this).

We are still committed to this series, and I am already at work on Part 4.

Add a Comment

Sign in

Have a technical question? Visit our forums. Have site or software product issues? Contact support.