Part 8 of the Intel® Software Guard Extensions (Intel® SGX) tutorial series integrates the graphical user interface (GUI) with the back end. In this part, we’ll examine the implications of mixing managed code with enclaves and the potential for undermining the security that is gained from Intel SGX. We’ll scope out the risks to our secrets and then develop mitigation strategies to limit their exposure.

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

Source code is provided with this installment of the series.

The User Interface

When we developed our application requirements back in Part 2 of the tutorial series, we made a decision to use the Microsoft*.NET Framework* to develop the user interface. That decision was based on the need to limit the complexity of the application development as well as limit dependencies on external software libraries.

The main window for the UI is shown in Figure 1. It is based on the Windows Presentation Foundation* (WPF), and makes use of standard WPF controls.

![The Tutorial Password Manager user interface.](image)

This decision came at a cost, however. .NET applications execute under the supervision of the Common Language Runtime, and the CLR includes a memory manager with garbage collection.

The Problem

In a traditional application—one built on a native code base—the developer is responsible for allocating memory as it’s needed, and must keep track of the addresses of allocated memory so that it can be freed when it’s no longer needed. Failing to free memory that has been allocated leads to memory leaks, which in turn causes the application to consume more and
more memory over time. In a high-performance application, the developer may even need to consider how the variables and objects in their application are *arranged* in memory, because some hardware instructions are sensitive to byte boundaries, and some algorithms depend on efficient use of the CPU cache.

In a managed application, however, the developer is abstracted from these details. The developer doesn’t allocate memory directly when an object is created, and that memory doesn’t have to be explicitly freed by the developer when it’s no longer needed. The runtime keeps track of the number of references to an object, and when the reference count drops to zero it is flagged for cleanup at a later time by a subsystem known as the garbage collector. The garbage collector has several jobs, but its primary ones are freeing up unused memory and compacting allocated memory into contiguous blocks for efficient usage.

Memory managed applications have a number of advantages, and the biggest of these is that they eliminate entire categories of software bugs. Unfortunately, they also have a very big *disadvantage* when it comes to application security: the developer is no longer in control over the low-level memory management, which means there is no way to guarantee that sensitive areas of memory can be securely erased or overwritten. The garbage collector can, at any time during a program’s execution, manipulate all the memory under its control. It can move objects around, make copies of them, delay writes to memory until they are needed, and even eliminate writes that it deems unnecessary. For a security application that needs to guarantee that a region of memory is erased or overwritten, memory managed environments are bad news.

**How Big is the Problem, Really?**

We can scope the extent of the problem by looking at the *String* class in .NET. Nearly every UI widget that displays text output to the screen in .NET does so through a *String*, and we can use that to assess just how big of a security problem we are creating if we put sensitive information into a UI control. To do that, we’ll write a simple Windows Forms* application that merely updates a text box, empties it, and then resets it.

Figure 2 shows the test application window, and Figure 3 shows the relevant code that modifies the string in the *TextBox* control.

![String Demo application window](image)

**Figure 2.** The *String Demo* application window.

```csharp
private void button2_Click(object sender, EventArgs e)
{
    textBox1.Text = "furniture";
    textBox1.Text = "compound";
    textBox1.Text += "-complex";
    textBox1.Clear();
    textBox1.Text = "Click 'Run' to go again";
}
```

**Figure 3.** *String operations on the TextBox control.*

We’ll compile this application in Release mode, execute it, and then attach to it using the Windows Debugger*, WinDbg, which is part of the Debugging Tools for Windows* from Microsoft. One of the features of WinDbg is that you can load other
DLLs to extend its functionality, and the SOS.DLL from the .NET Framework exposes a command to dump classes of a specified type from the memory heap (for more information on this topic, see the article .NET Debugging: Dump All Strings from a Managed Code Process at CodeProject*).

When we dump the String objects in the heap right at the start of the application, before clicking the “Run” button, we see all the Strings currently in use by the application. There are a lot of Strings on the heap, but the most relevant ones are highlighted in Figure 4.

![Command - Pid 43736 - WinDbg:10.0.14321.1024 AMD64](image)

Figure 4. Strings in the String Demo application. Highlighted strings correspond to the user interface controls.

Here we can see the names of our Windows Form widgets, and the starting text for the TextBox widget, “Click ‘Run’ to start”. If we scroll down toward the bottom, we see “String Demo”, the name of our main form, repeated a few dozen times as shown in Figure 5.
Now, we resume the program and click the “Run” button. This cycles through the operations shown in Figure 3, which returns almost instantly. Returning to WinDbg, we dump the String objects once again, and get the output shown in Figure 6.

**Figure 5.** Duplicates of the same string in the String Demo application.

**Figure 6.** Strings in the String Demo application, post execution.
Note that this String dump is from a single execution of the button2_Click() event handler, and the debugger wasn’t attached until a full minute after the execution had completed. This means that, one minute after running through a command sequence in which the execution:

1. overwrote the textbox String with the word “furniture”
2. immediately overwrote it again, this time with the word “compound”
3. appended the text “-complex”
4. cleared the textbox by calling the Clear() method
5. reset the text to the phrase “Click ‘Run’ to restart”

the memory heap for the application contained multiple Strings with the contents “furniture”, “compound”, and “compound-complex”. Clearly, any controls that depend on the String class are going to be leaking our secrets, and doing so to multiple places in memory!

Mitigation and Solution

Fortunately, most managed code frameworks provide classes that are specifically designed to store sensitive data. These usually work by placing their contents in unmanaged memory, outside of the control of the garbage collector. In the .NET Framework this capability is provided by the SecureString class, but we are hamstrung by having only one WPF control that can work with the SecureString class out of the box, the PasswordBox.

This may not seem like a significant issue since our application’s secrets include passwords, and the PasswordBox control is designed specifically for password entry. For most applications this is sufficient, but our application is a password manager, and it must be able to reveal the password to the user on demand. In the Universal Windows Platform* runtime, the PasswordBox class provides a property named PasswordRevealMode, which can be set to either “Hidden”, “Visible”, or “Peek” (this last value reveals the password while the user is pressing the “reveal” button on the control.

Figure 7. The Universal Windows Platform version of the PasswordBox control supports a “peek” capability that the WPF control in .NET does not support.

The .NET Framework version of the PasswordBox control does not have this property, however, which means that the PasswordBox control is effectively write-only (from the user’s perspective). PasswordBox is fine for entering a password, but it’s not a solution for revealing a password when the user asks to view it. We need to create our own solution.

Requirements and Constraints

In Part 3 of the series, we identified two secrets that exit the enclave in clear text so that they can be displayed via the user interface:

- The login information for an account (a name for the account, a login or user name, and URL associated with the account).
- The account password.

While the login information for a user’s account is sensitive data, accidental exposure does not carry tremendous risk. The entire application could be re-written to make use of native windows, but that would be a huge investment for arguably little gain. A future release of the .NET Framework may contain the functionality that we need in the PasswordBox control, so we
can accept the risk today and look for the problem to be fixed for us in the future. If later we decide the risk is too great, we could always revisit the decision and implement a native solution in a later release.

For account passwords, however, the day of reckoning has arrived. *There is no scenario where it is appropriate to expose a password to the String class in .NET.* As a password manager, we need the ability to show passwords to the user, but we must not use the .NET Framework to do it.

**The Solution**

We must use a native window when revealing a password to the end user. With a native window we are placing the password in unmanaged memory, and because it’s unmanaged memory we can guarantee that the password is securely erased by making a call to `SecureZeroMemory()`. The problem with a native window, though, is the same one that drove our design to .NET and WPF in the first place: the default tools for building native windows either force you to build the entire window and messaging handlers from the ground up using the Win32 API, or rely on the large and complex Microsoft Foundation Classes*, which is extremely heavy-handed for our simple application.

To solve this problem we’re going to take a shortcut. Though the Win32 API does not provide convenience methods for building generic windows, it does provide one function for displaying a very simple, modestly configurable dialog box that meets our needs: `MessageBox()`. The `MessageBox()` function lets you specify a window title, some caption text, and a button configuration from a number of preset options. It takes care of the window formatting and the event handlers, and it returns an integer value that tells us which button the user clicked.

Figure 8 shows the dialog box created by our password manager application using this approach. To implement it, we added a method called `accounts_view_password()` to the `PasswordManagerCoreNative` class. The code listing is given in Figure 9.

---

**Figure 8.** The "view password" window generated by `accounts_view_password()`.

```cpp
int PasswordManagerCoreNative::accounts_view_password(UINT32 idx, HWND hWnd)
{
    int rv = NL_STATUS_UNKNOWN;
    LPWCH wpass;
    UINT16 wlen;
    static const wchar_t empty[] = L"(no password stored)";

    // First get the password

    rv = this->accounts_get_password(idx, &wpass, &wlen);
    if (rv != NL_STATUS_OK) return rv;

    // Display our MessageBox
```
```c++
MessageBox(hWnd, (wlen>1) ? wpass : empty, L"Password", MB_OK);

this->accounts_release_password(wpass, wlen);

return rv;
}
```

**Figure 9.** *The accounts_view_password() method in PasswordManagerCoreNative.*

The first argument is the index of the account in our vault whose password we want to view. Internally, it makes a call to `accounts_get_password()` to fetch that password from the vault, and then it calls `MessageBox()` to display it (with logic to display a canned message if the password string is empty) with a single OK button. Then we free up the password and return.

The second argument to this function is the handle of the owner window. In native code, that handle is a type `HWND`, but our owner window originates from managed code. Getting this handle is a two-step procedure. **PasswordManagerCore** can marshal an `IntPtr` to a native pointer that `accounts_get_password()` can use, and this is shown in Figure 10, but the `IntPtr` that refers to the owning WPF window has to come from the UI layer of the application.

```c++
int PasswordManagerCore::accounts_view_password(UInt32 idx, IntPtr hptr)
{
    int rv;
    UINT32 index = idx;
    HWND hWnd = static_cast<HWND>(hptr.ToPointer());

    rv = _nlink->accounts_view_password(index, hWnd);
    if (rv != NL_STATUS_OK) return rv;

    return rv;
}
```

**Figure 10.** *The accounts_view_password() method in PasswordManagerCore.*

Figure 11 shows the code from EditDialog.xaml.cs. Here, we get a managed handle to the current window and use the `WindowInteropHelper` class from WPF to produce an `IntPtr`.

```c#
private void btnView_Click(object sender, RoutedEventArgs e)
{
    WindowInteropHelper wih;
    Window wnd;
    IntPtr hptr;
    int rv = 0;

    wnd = Window.GetWindow(this);
    wih = new WindowInteropHelper(wnd);
    hptr = wih.Handle;
}
```
rv = mgr.accounts_view_password(pNewItem.index, hptr);
if (rv != PasswordManagerStatus.OK)
{
    MessageBox.Show("Couldn't view password: " + MainWindow{returnErrorcode(rv)});
    return;
}

Figure 11. The event handler for viewing a password.

Viewing the password isn’t the only place where this is needed. Our Tutorial Password Manager application also provides the user with the option to randomly generate a password, as shown in Figure 12. It stands to reason that the user should be able to review that password before accepting it.

![EditPassword Interface](image)

Figure 12. The interface for setting an account's password.

We follow the same approach of using `MessageBox()` to show the password, only with a slightly different twist. Instead of presenting a dialog box with a single OK button, we prompt the user, asking them if they accept the randomly generated password that is shown. Their options are:

- **Yes.** Accepts and saves the password.
- **No.** Generates a new password using the same character requirements, and then reprompts.
- **Cancel.** Cancels the operation entirely, and does not change the password.

This dialog box is shown in Figure 13.
Accepting or rejecting a randomly generated password.

The code behind this dialog box, which implements the `accounts_generate_and_view_password()` method, is shown in Figure 14. The approach is very similar to `accounts_view_password()`, except there is additional logic to handle the button options in the dialog box.

```cpp
int PasswordManagerCoreNative::accounts_generate_and_view_password(UINT16 length, UINT16 flags, LPWSTR *wpass, UINT16 *wlen, HWND hWnd)
{
    int rv;
    char *cpass;
    int dresult;

    cpass = new char[length + 1];
    if (cpass == NULL) return NL_STATUS_ALLOC;

    if (supports_sgx()) rv = ew_accounts_generate_password(length, flags, cpass);
    else rv = vault.accounts_generate_password(length, flags, cpass);
    cpass[length] = NULL;

    if (rv != NL_STATUS_OK) return rv;

    *wpass = towchar(cpass, length + 1, wlen);

    SecureZeroMemory(cpass, length);
    delete[] cpass;

    if (*wpass == NULL) return NL_STATUS_ALLOC;

    // Show the message box

    dresult = MessageBox(hWnd, *wpass, L"Accept this password?", MB_YESNOCANCEL);
    if (dresult == IDNO) return NL_STATUS_AGAIN;
    else if (dresult == IDCANCEL) return NL_STATUS_USER_CANCEL;

    return NL_STATUS_OK;
}
```

**Figure 13.** Accepting or rejecting a randomly generated password.
The dialog result of IDNO is mapped to NL_STATUS_AGAIN, which is the “retry” option. A result of IDCANCEL is mapped to NL_STATUS_USER_CANCEL.

Because the wpass pointer is passed in from the parent method in PasswordManagerCore, we implement the retry loop there instead of in the native method. This is slightly less efficient, but it keeps the overall program architecture consistent. The code for accounts_generate_and_view_password() in PasswordManagerCore is shown in Figure 15.

```c++
int PasswordManagerCore::generate_and_view_password(UInt16 mlength, UInt16 mflags, SecureString ^%password, IntPtr hptr)
{
    int rv = NL_STATUS_AGAIN;
    LPWSTR wpass;
    UINT16 wlen;
    UINT16 length = mlength;
    UINT16 flags = mflags;
    HWND hWnd = static_cast<HWND>(hptr.ToPointer());

    if (!length) return NL_STATUS_INVALID;

    // Loop until they accept the randomly generated password, cancel, or an error occurs.
    while (rv == NL_STATUS_AGAIN) {
        rv = _nlink->accounts_generate_and_view_password(length, flags, &wpass, &wlen, hWnd);
        // Each loop through here allocates a new pointer.
        if (rv == NL_STATUS_AGAIN) _nlink->accounts_release_password(wpass, wlen);
    }

    if (rv != NL_STATUS_OK) return rv;

    // They accepted this password, so assign it and return.
    password->Clear();
    for (int i = 0; i < length; ++i) {
        password->AppendChar(wpass[i]);
    }

    _nlink->accounts_release_password(wpass, wlen);

    return rv;
}
```

**Figure 14.** Code listing for accounts_generate_and_view_password() in PasswordManagerCoreNative.


Summary
Mixing enclaves with managed code is a tricky, and potentially risky, business. As a general rule, secrets should not cross the enclave boundary unencrypted, but there are applications such as our Tutorial Password Manager where this cannot be avoided. In these cases, you must take proper care to not undermine the security that is provided by Intel SGX. Exposing secrets to unprotected memory can be a necessary evil, but placing them in managed memory can have disastrous consequences.

The takeaway from this part of the series should be this: **Intel SGX does not eliminate the need for secure coding practices.** What it gives you is a tool (and a very powerful one) for keeping your application’s secrets away from malicious software, but that tool, alone, does not and cannot build a secure application.

Sample Code
The code sample for this part of the series builds against the Intel SGX SDK version 1.7 using Microsoft Visual Studio* 2015.

Coming Up Next
In Part 9 of the series, we’ll add support for power events. Stay tuned!
Notices

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

This sample source code is released under the Intel Sample Source Code License Agreement.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

*Other names and brands may be claimed as the property of others.

© 2017 Intel Corporation