A Brief Introduction to the Standard Annotation Language (SAL)

18-May-2006

Michael Howard
http://blogs.msdn.com/michael_howard

Introduction

Even though a prior blog I wrote "Code Scanning Tools Do Not make Software Secure" at http://blogs.msdn.com/michael_howard/archive/2006/01/26/517975.aspx may have left some thinking I don't like static analysis tools, nothing could be farther from the truth. In fact, there is a code analysis technology designed by Microsoft Research which is included with Visual Studio 2005 that I simply love, and that is the Standard Annotation Language, or SAL. SAL is a meta-language that can help static analysis tools, such as the /analyze switch in Visual Studio 2005 Team System and Visual Studio 2005 Team Edition for Developers, find bugs—including security bugs—in your C or C++ code at compile time.

Using SAL is relatively easy. You simply add annotations to your function prototypes that describe more contextual information about the function being annotated. This can include annotations to function arguments and to function return values. The initial focus of SAL is to annotate functions that manipulate read and write buffers. In Windows Vista we are annotating all appropriate functions before the product is released to customers to help us find bugs as early as possible.

The main benefit of SAL is that you can find more bugs with just a little bit of upfront work. We have found that the process of adding SAL annotations to existing code can also find bugs as the developer questions the assumptions previously made about how the function being annotated works. By this I mean that as a developer adds annotations to a function, she must think about how the function works in more detail than simply assuming it was written correctly. This process finds assumption flaws.

Any bugs found in SAL annotated functions tend to be real bugs, not false positives, which has the benefit of speedier bug triage and code fixes.

Finally, SAL is highly leveraged; when you annotate a function, any code that calls that function will get the benefit of the annotation. To this end, we have annotated the majority of C Runtime functions included with Visual Studio 2005 and the Windows SDK functions. Over time we will add more annotations to more functions to help find bugs in code written to use the functions. In short, this means you will get the benefit of the annotations added by Microsoft, and you might find bugs in your code!

Digging Deeper

Let me give an example of what SAL can do. Let's say you have a C/C++ function like this:

```
void FillString(
      TCHAR* buf,
      size_t cchBuf,
```

```
        char ch) {

  for (size_t i = 0; i < cchBuf; i++)    {
    buf[i] = ch;
  }
}
```

I won't insult your intelligence by explaining what the function does, but what makes this code interesting is that two of the arguments, buf and cchBuf, are tied at the hip; buf should be at least cchBuf characters long. If buf is not as big as cchBuf claims it is, then FillString could overflow the buf buffer.

If you compile the code below with Visual Studio 2005, at warning level 4 (/W4) you will see no warnings and no errors, yet there is clearly a buffer overrun vulnerability in this code.

```
TCHAR *b = (TCHAR*)malloc(200*sizeof(TCHAR));
FillString(b,210,'x');
```

What SAL does is allow a C or C++ developer to inform the compiler of the relationship between the two arguments, buf and cchBuf, using syntax such as this:

```
void FillString(
      __out_ecount(cchBuf) TCHAR* buf,
      size_t cchBuf,
      char ch) {

  for (size_t i = 0; i < cchBuf; i++)    {
    buf[i] = ch;
  }
}
```

When both code fragments are compiled with Visual C++ in Visual Studio 2005 Team System or Visual Studio 2005 Team Edition for Developers and the /analyze compile option, you will see the following warnings:

```
c:\code\saltest\saltest.cpp(54) : warning C6203: Buffer overrun for non-stack buffer 'b' in call
to 'FillString': length '420' exceeds buffer size '400'

c:\code\saltest\saltest.cpp(54) : warning C6386: Buffer overrun: accessing 'argument 1', the
writable size is '200*2' bytes, but '420' bytes might be written: Lines: 53, 54

c:\code\saltest\saltest.cpp(54) : warning C6387: 'argument 1' might be '0': this does not adhere
to the specification for the function 'FillString': Lines: 53, 54
```

What just happened here? Note the use of __out_ecount(n) just before buf in the argument list. This is a macro that wraps some very low-level SAL constructs you should never have to worry about, but in essence __out_ecount(n) means:

“buf is an out parameter, which means it will be written to by the function, and buf cannot be NULL. The length of buf is ‘n’ elements, in this case cchBuf TCHARS”

That’s it! And as you can see, recompiling the code found the bug in the code that calls FillString. What’s really cool, is any code that uses FillString will automatically gain the benefit of the annotation.

IMPORTANT: I want to take a moment to explain something you should be aware of. SAL is in flux. More importantly, there are two versions of SAL; the first is a __declspec syntax, and the second is an attribute syntax. Visual Studio 2005 supports both, and the C Runtime today is annotated with the __declspec format. Over time, we expect to move to the attribute syntax. Both syntaxes will be supported for the near future, but innovation will occur in the attribute syntax.

The SAL macros define proper use of buffers, which are allocated regions of data represented as pointers in C/C++ code. A C/C++ pointer can be used to represent a single element buffer or a buffer of many elements. Sometimes the size is known at compile time and sometimes it’s only know at runtime. Because C/C++ pointer types are overloaded you cannot rely on the type system to help you program with buffers properly! That’s why we have SAL. It makes explicit exactly how big the buffer is that a pointer points into.

There are many other SAL macros, including:

_in

The function will only read from the single-element buffer, and the buffer must be initialized; as such __in the exactly the same as __in_ecount(1) and __in is implied if the argument is a const. The following function prototype shows how you can use __in.

```
BOOL AddElement(
      __in ELEMENT *pElement) ;
```

_out

The function fills a valid buffer, and the buffer can be dereferenced by the calling code. The following function prototype shows how you can use __out.

```
BOOL GetFileVersion(
      LPCWSTR lpsFile,
      __out FILE_VERSION *pVersion);
```

_in_opt

The function expects an optional buffer, meaning the buffer can point to NULL. The following code shows how you could use __in_opt, in this example, if szMachineName is NULL, then the code will return operating system information about the local computer.

```
BOOL GetOsType(
      __in_opt char *szMachineName,
```

```
        __out MACHINE_INFO *pMachineInfo);
```

_inout

The function expects a readable and writeable buffer, and the buffer must be initialized by the caller. Here is some sample code that shows how you might use __inout.

```
size_t EncodeStream(
        __in HANDLE hStream,
        __inout STREAM *pStream);
```

_inout_bcount_full(n)

The function expects a buffer that is n-bytes long that is fully initialized on entry and exit. Note the use of bcount rather than ecount. 'b' means bytes, and 'e' means elements, for example a Unicode string in Windows that is 12 characters (an element is SAL parlance) long is 24 bytes long. The following code example takes a BYTE * that points to a buffer to switch from big-endian format to little-endian format so it makes sense that the incoming buffer be fully initialized, and is a fully initialized buffer on function exit. You'll also see another SAL macro in the function prototype, __out_opt, which means the data will be written to by the function, but it can be NULL. In the case of a NULL exception point, the function will not return exception data to the caller.

```
void ConvertToLittleEndian(
    __inout_bcount_full(cbInteger) BYTE *pbInteger,
    DWORD cbInteger,
    __out_opt EXCEPTION *pException);
```

_deref_out_bcount(n)

The function whose dereference will be set to an uninitialized buffer of 'n' bytes, in other words, *p is initialized, but **p is not.

```
HRESULT StringCbAlloc(
  size_t cb,
  __deref_out_bcount(cb) char **ppsz) {

      *ppsz = (char*)LocalAlloc(LPTR, cb);
      return *ppsz ? S_OK : E_OUTOFMEMORY;
}
```

And there are many more such annotations.

SAL's usefulness extends beyond function arguments. It can also be used to detect errors on function return. If you look closely at the list of warnings earlier in this document, you'll notice a third warning:

```
c:\code\saltest\saltest.cpp(54) : warning C6387: 'argument 1' might be '0': this does not adhere
to the specification for the function 'FillString': Lines: 53, 54
```

This bug really has little to do with the function argument, rather it occurs because the code calls malloc() and does not check the return value is non-NULL. If you look at the function prototype for malloc() in malloc.h, you'll see this:

```
_checkReturn __bcount_opt(_Size)
    void *__cdecl malloc(__in size_t _Size);
```

Because the return from malloc() could be NULL we use a __bcount_opt(n) macro (note the use of opt in the macro name.) If we change the code that calls malloc() to check the return is not NULL prior to calling FillString, the warning goes away. Don't confuse an optional NULL return value with __checkReturn, the latter means you ignored the result altogether, for example:

```
size_t cb = 10 * 12;
malloc(cb);
```

This code will yield this warning when compiled with /analyze:

```
c:\code\saltest\saltest.cpp(30) : warning C6031: Return value ignored: 'malloc'
```

The Future of SAL

This section is important for completeness and to set expectations about the future of SAL. I have already mentioned that __inout and the like are actually macros that wrap low-level SAL constructs. Presently, there is one set of macros and two low-level SAL primitives; one is a __declspec form, and the other is an attribute form. As I write this, the macros that ship with Visual Studio 2005 map to the __declspec form. For example, __out_ecount(n) maps to:

```
__pre
  __notnull
  __elem_writableTo(n)
__post
  __valid
  __deref
  __notreadonly
```

The good news is that you do not, indeed you *should not* use these low-level SAL primitives unless you absolutely must do so. To be honest, I doubt you will need to use them. Stick with using the macros. As you can probably guess, __pre, __notnull and so forth are the declspec SAL annotations. But in the future we will move to an attribute syntax, which looks a little like this. This is the same declspec annotation about, but using attribute syntax.

```
[SA_Pre(WritableElements="n", Null=SA_No)]
[SA_Post(Valid=SA_Yes, Deref=1, Access=SA_Write)]
```

Now here's the bad news. Today, if you want to use attribute-based SAL, you have to enter all these low-level attribute SAL annotations. Moving forward, however, we will wrap the most commonly used SAL constructs into macros. The plan is to provide these macros in Visual Studio "Orcas", but like all non-

released products, this is subject to change! Presently, the headers in Visual Studio 2005 are annotated with the __declspec macros, but we will update these to use attribute macros over time also.

Action Items

SAL is a powerful mechanism to help find real security bugs in your code, and you should take advantage of it as soon as possible. If you simply use the updated C-runtime and Windows SDK headers and compiling with the /analyze option in Visual Studio 2005 Team System or Visual Studio 2005 Team Edition for Developers will probably find bugs in your code with no additional work on your behalf!

Better yet, you should annotate all functions that take writeable buffers that you create. You do so by adding SAL macros to your function prototypes. Today, that will mean using the __declspec macro form.

Best, annotate all functions that take writeable and readable buffers.

Once you have performed these steps, compile with /analyze and find some bugs. It really is that simple!

Other Resources

That was a brief tour of SAL. You can learn more by looking at the comments at the top of sal.h which includes a summary of the current SAL constructs. The strsafe.h (a set of safer string handling functions) header file also offers a good smattering of sample SAL usage in real-life. Below are some links to other references you should look at to learn more about SAL.

Header Annotations (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/header_annotations.asp)

SAL Annotations (http://msdn2.microsoft.com/en-us/library/ms235402.aspx)

Annotation Overview (http://msdn2.microsoft.com/en-us/library/ms182033.aspx)

Annotation Properties (http://msdn2.microsoft.com/en-us/library/ms182037.aspx)

Walkthrough: Analyzing C/C++ Code for Defects (http://msdn2.microsoft.com/en-us/library/ms182028.aspx)

Visual Studio 2005 Security Features and Tools (http://msdn.microsoft.com/security/vs2005security/default.aspx)

Windows SDK (http://windowssdk.msdn.microsoft.com/library/)

A big thanks to the many people who are actively involved in the development of SAL and reviewed this document: Hunter Hudson and Daniel Wang from Windows , Hannes Ruescher from Office, Dave Lubash from Enterprise Developer Tools and Eric Bidstrup and Steve Lipner in my group, Security Engineering.