Fancy use of exception handling in FormatMessage leads to repeated "discovery" of security flaw

devblogs.microsoft.com/oldnewthing/20120210-00

February 10, 2012

Raymond Chen

Every so often, somebody "discovers" an alleged security vulnerability in the `FormatMessage` function. You can try it yourself:

```
#include <windows.h>
#include <stdio.h>
char buf[2048];
char extralong[128*1024];
int __cdecl main(int argc, char **argv)
{
 memset(extralong, 'x', 128 * 1024 - 1);
 DWORD_PTR args[] = { (DWORD_PTR)extralong };
 FormatMessage(FORMAT_MESSAGE_FROM_STRING |
               FORMAT_MESSAGE_ARGUMENT_ARRAY, "%1", 0, 0,
               buf, 2048, (va_list*)args);
 return 0;
}
```

If you run this program under the debugger and you tell it to break on all exceptions, then you will find that it breaks on an access violation trying to write to an invalid address.

```
eax=00060078 ebx=fffe0001 ecx=0006fa34 edx=00781000 esi=0006fa08 edi=01004330
eip=77f5b279 esp=0006f5ac ebp=0006fa1c iopl=0         nv up ei pl nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010203
ntdll!fputwc+0x14:
77f5b279 668902           mov     [edx],ax              ds:0023:00781000=????
```

Did you just find a buffer overflow security vulnerability?

The `FormatMessage` function was part of the original Win32 interface, back in the days when you had lots of address space (two whole *gigabytes*) but not a lot of RAM (12 megabytes, or 16 if you were running Server). The implementation of `FormatMessage` reflects this historical reality by working hard to conserve RAM but not worrying too much about conserving address space. And it takes advantage of this fancy new *structured exception handling* feature.

The `FormatMessage` uses the *reserve a bunch of address space but commit pages only as they are necessary* pattern, illustrated in MSDN under the topic Reserving and Committing Memory. Except that the sample code on that page contains serious errors. For example, if the sample code encounters an exception other than `STATUS_ACCESS_VIOLATION`, it still "handles" it by doing nothing and returning `EXCEPTION_EXECUTE_HANDLER`. It fails to handle random access to the buffer or access violations caused by DEP. Though in the very specific sample, it mostly works since the protected region does only one thing, so there aren't many opportunities for the other types of exceptions to occur. (Though if you're really unlucky, you might get an `STATUS_IN_PAGE_ERROR`.) But enough complaining about that sample.

The `FormatMessage` function reserves 64KB of address space, commits the first page, and then calls an internal helper function whose job it is to generate the output, passing the start of the 64KB block of address space as the starting address and telling it to give up when it reaches 64KB. Something like this:

```
struct DEMANDBUFFER
{
  void *Base;
  SIZE_T Length;
};
int
PageFaultExceptionFilter(DEMANDBUFFER *Buffer,
                         EXCEPTION_RECORD ExceptionRecord)
{
  int Result;
  DWORD dwLastError = GetLastError();
  // The only exception we handle is a continuable read/write
  // access violation inside our demand-commit buffer.
  if (ExceptionRecord->ExceptionFlags & EXCEPTION_NONCONTINUABLE)
    Result = EXCEPTION_CONTINUE_SEARCH;
  else if (ExceptionRecord->ExceptionCode != EXCEPTION_ACCESS_VIOLATION)
    Result = EXCEPTION_CONTINUE_SEARCH;
  else if (ExceptionRecord->NumberParameters < 2)
    Result = EXCEPTION_CONTINUE_SEARCH;
  else if (ExceptionRecord->ExceptionInformation[0] &
      ~(EXCEPTION_READ_FAULT | EXCEPTION_WRITE_FAULT))
    Result = EXCEPTION_CONTINUE_SEARCH;
  else if (ExceptionRecord->ExceptionInformation[1] -
      (ULONG_PTR)Buffer->Base >= Buffer->Length)
    Result = EXCEPTION_CONTINUE_SEARCH;
  else {
    // If the memory is already committed, then committing memory won't help!
    // (The problem is something like writing to a read-only page.)
    void *ExceptionAddress = (void*)ExceptionInformation[1];
    MEMORY_BASIC_INFORMATION Information;
    if (VirtualQuery(ExceptionAddress, &Information,
                     sizeof(Information)) != sizeof(Information))
      Result = EXCEPTION_CONTINUE_SEARCH;
    else if (Information.State != MEM_RESERVE)
      Result = EXCEPTION_CONTINUE_SEARCH;
    // Okay, handle the exception by committing the page.
    // Exercise: What happens if the faulting memory access
    // spans two pages?
    else if (!VirtualAlloc(ExceptionAddress, 1, MEM_COMMIT, PAGE_READWRITE))
      Result = EXCEPTION_CONTINUE_SEARCH;
    // We successfully committed the memory - retry the operation
    else Result = EXCEPTION_CONTINUE_EXECUTION;
  }
  RestoreLastError(dwLastError);
  return Result;
}
DWORD FormatMessage(...)
{
  DWORD Result = 0;
  DWORD Error;
  DEMANDBUFFER Buffer;
  Error = InitializeDemandBuffer(&Buffer, FORMATMESSAGE_MAXIMUM_OUTPUT);
  if (Error == ERROR_SUCCESS) {
    __try {
     Error = FormatMessageIntoBuffer(&Result,
                                     Buffer.Base, Buffer.Length, ...);
    } __except (PageFaultExceptionFilter(&Buffer,
                     GetExceptionInformation()->ExceptionRecord)) {
     // never reached - we never handle the exception
    }
  }
  if (Error == ERROR_SUCCESS) {
   Error = CopyResultsOutOfBuffer(...);
  }
  DeleteDemandBuffer(&Buffer);
  if (Result == 0) {
    SetLastError(Error);
  }
  return Result;
}
```

The `FormatMessageIntoBuffer` function takes an output buffer and a buffer size, and it writes the result to the output buffer, stopping when the buffer is full. The `DEMANDBUFFER` structure and the `PageFaultExceptionHandler` work together to create the output buffer on demand as the `FormatMessageIntoBuffer` function does its work.

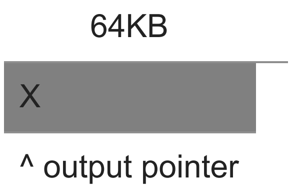
To make discussion easier, let’s say that the `FormatMessage` function merely took printf-style arguments and supported only `FORMAT_MESSAGE_FROM_STRING | FORMAT_MESSAGE_ALLOCATE_BUFFER`.

```
DWORD FormatMessageFromStringPrintfAllocateBuffer(
    PWSTR *ResultBuffer,
    PCWSTR FormatString,
    ...)
{
  DWORD Result = 0;
  DWORD ResultString = NULL;
  DWORD Error;
  DEMANDBUFFER Buffer;
  va_list ap;
  va_start(ap, FormatString);
  Error = InitializeDemandBuffer(&Buffer, FORMATMESSAGE_MAXIMUM_OUTPUT);
  if (Error == ERROR_SUCCESS) {
    __try {
     SIZE_T MaxChars = Buffer.Length / sizeof(WCHAR);
     int i = _vsnwprintf((WCHAR*)Buffer.Base, MaxChars,
                         FormatString, ap);
     if (i < 0 || i >= MaxChars) Error = ERROR_MORE_DATA;
     else Result = i;
    } __except (PageFaultExceptionFilter(&Buffer,
                   GetExceptionInformation()->ExceptionRecord)) {
     // never reached - we never handle the exception
    }
  }
  if (Error == ERROR_SUCCESS) {
   // Exercise: Why don't we need to worry about integer overflow?
   DWORD BytesNeeded = sizeof(WCHAR) * (Result + 1);
   ResultString = (PWSTR)LocalAlloc(LMEM_FIXED, BytesNeeded);
   if (ResultBuffer) {
    // Exercise: Why CopyMemory and not StringCchCopy?
    CopyMemory(ResultString, Buffer.Base, BytesNeeded);
   } else Error = ERROR_NOT_ENOUGH_MEMORY;
  }
  DeleteDemandBuffer(&Buffer);
  if (Result == 0) {
    SetLastError(Error);
  }
  *ResultBuffer = ResultString;
  va_end(ap);
  return Result;
}
```

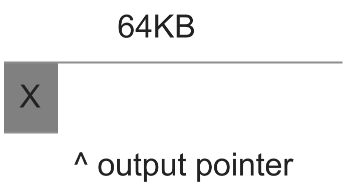
Let’s run this function in our head to see what happens if somebody triggers the alleged buffer overflow by calling

```
PWSTR ResultString;
DWORD Result = FormatMessageFromStringPrintfAllocateBuffer(
                   &ResultString, L"%s", VeryLongString);
```

After setting up the demand buffer, we call `_vsnwprintf` to format the output into the demand buffer, but telling it not to go past the buffer's total length. The `_vsnwprintf` function parses the format string and sees that it needs to copy `VeryLongString` to the output buffer. Let's say that the `DEMANDBUFFER` was allocated at address `0x00780000` on a system with 4KB pages. At the start of the copy, the address space looks like this:

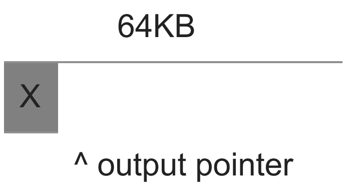


"C" stands for a committed page, "R" stands for a reserved page, and "X" stands for a page that, if accessed, would be a buffer overflow. We start copying `VeryLongString` into the output buffer. After copying 2048 characters, we fill the first committed page; copying character 2049 raises a page fault exception.

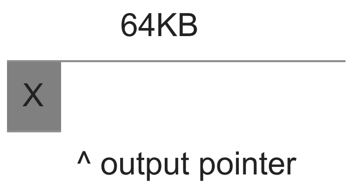


This is the point at which over-eager people observe the first-chance exception, capture the register dump above, and begin writing up their security vulnerability report, cackling with glee. (Observe that in the register dump, the address we are writing to is of the form `0x####1000`.)

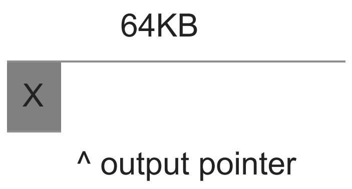
As with all first-chance exceptions, it goes down the exception chain. Our custom `PageFaultExceptionFilter` recognizes this as an access violation in a page that it is responsible for, and the page hasn't yet been committed, so it commits the page as read/write and resumes execution.



Copying character 2049 now succeeds, as does the copying of characters 2050 through 4096. When we hit character 4097, the cycle repeats:

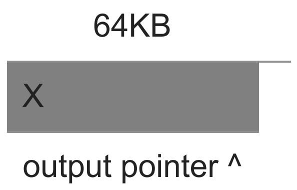


Again, the first-chance exception is sent down the chain, our `PageFaultExceptionFilter` recognizes this as a page it is responsible for, and it commits the page and resumes execution.



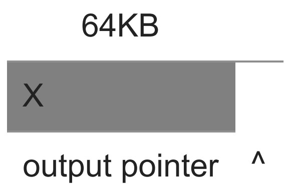
If you think about it, this is exactly what the memory manager does with memory that has been allocated but not yet accessed: The memory is not present, and the moment an application tries to access it, the not-present page fault is raised, the memory manager commits the page, and then execution resumes normally. It’s memory-on-demand, which is one of the essential elements of virtual memory. What’s going on with the `DEMANDBUFFER` is that we are simulating in user mode what the memory manager does in kernel mode. (The difference is that while the memory manager takes committed memory and makes it present on demand, the `DEMANDBUFFER` takes reserved address space and commits it on demand.)

The cycle repeats 13 more times, and then we reach another interesting part of the scenario:



We are about to write 32768th character into the `DEMANDBUFFER`. Once that’s done, the buffer will be completely full. One more byte and we will overflow the buffer. (Not even a wafer-thin byte will fit.)

Let’s write that last character and cover our ears in anticipation.



Oh noes! Completely full! Run for cover!

But wait. We passed a buffer size to the `_vsnwprintf` function, remember? We already told it never to write more than 32768 characters. As it's about to write character 32769, it realizes, "Wait a second, this would overflow the buffer I was given. I'll return a failure code instead."

The feared write of the 32769th character never takes place. We never write to the "X" page. Instead, the `_vnswprintf` call returns that the buffer was not large enough, which is converted into `ERROR_MORE_DATA` and returned to the caller.

If you follow through the entire story, you see that everything worked as it was supposed to and no overflow took place. The `_vnswprintf` function ran up to the brink of disaster but stopped before taking that last step. This is hardly anything surprising; it happens whenever the `_vnswprintf` function encounters a buffer too small to hold the output. The only difference is that along the way, we saw a few first-chance exceptions, exceptions that had nothing to do with avoiding the buffer overflow in the first place. They were just part of `FormatMessage` 's fancy buffer management.

It so happens that in Windows Vista, the fancy buffer management technique was abandoned, and the code just allocates 64KB of memory up front and doesn't try any fancy commit-on-demand games. Computer memory has become plentiful enough that a momentary allocation of 64KB has less of an impact than it did twenty years ago, and performance measurements showed that the new "Stop trying to be so clever" technique was now about 80 times faster than the "gotta scrimp and save every last byte of memory" technique.

The change had more than just a performance effect. It also removed the first-chance exception from `FormatMessage`, which means that it no longer does that thing which everybody mistakes for a security vulnerability. The good news is that nobody reports this as a vulnerability in Windows Vista any more. The bad news is that people still report it as a vulnerability in Windows XP, and each time this issue comes up, somebody (possibly me) has to sit down and reverify that the previous analysis is still correct, in the specific scenario being reported, because who knows, maybe this time they really did find a problem.

Raymond Chen

Follow