EXPLOIT ANALYSIS

ANI-HILATE THIS WEEK

Peter Ferrie
Symantec Security Response, USA

The time between the announcement of a vulnerability and the exploitation of that vulnerability continues to shrink. This is especially true when the vulnerability in question is a stack overflow, since it requires very little skill to exploit. The recent ANI vulnerability is a prime example. Before we get into that, let's find out a little more about ANI files in general.

An ANI file contains animated cursors and icons – for example, the hourglass which turns upside-down during heavy processing. Internally, ANI files are *Microsoft*'s Resource Interchange File Format ('RIFF') files. They are, in fact, little-endian versions of *Electronic Arts*' Interchange File Format ('IFF') files that were introduced to the *Amiga* in 1985. IFF files themselves were inspired by *Apple*'s OSType files that were released with the *Macintosh* computer in 1984.

IT'S 'TERIFFIC'

RIFF files contain a collection of chunks, each of which begins with a four-byte type-name, followed by the size of the chunk.

The first chunk in a RIFF file is named 'RIFF'. This is one of the two types of chunk that contain a subtype and a collection of subchunks (the other one is 'LIST').

The subchunks have the same format as chunks. The idea is that the rest of the file is a collection of subchunks within the 'RIFF' chunk. The size field is not verified (most likely because it is assumed that nothing follows the 'RIFF' chunk). Instead, *Microsoft*'s parser ensures that accesses remain within the bounds of the file by calling the GetFileSize() API and comparing file offsets against the returned value.

The 'RIFF' chunk subtype has the name 'ACON' (which may be an abbreviation of 'Animated iCON'). Very few subchunk types are supported, and some of them depend on the presence of others earlier in the file. For example, until an 'anih' type is seen, only a 'LIST' type with the 'fram' subtype and 'icon' subchunk(s) is accepted. All other types are skipped. Once an 'anih' type is seen, the 'seq', 'rate', and additional 'anih', types are also accepted.

PATCHWORK QUILT

The vulnerability that is being exploited is an unbounded copy operation to a fixed-size stack buffer. Exactly the same vulnerability in exactly the same function was found in 2004. The 'anih' subchunk contains a field that specifies the size of the data. While the data are a fixed size – 36 bytes – the value in that field is used in a copy operation. This allowed an attacker to specify an arbitrary size for the block and overflow the stack buffer. At the time, *Microsoft* patched the vulnerability by adding a requirement that the first 'anih' block is exactly 36 bytes long. It's unclear why they did it that way, because the block is copied again later, using the fixed value of 36.

For subsequent 'anih' subchunks, the data are copied to the stack buffer, then 36 bytes are copied to another buffer, and *then* there is a check that the data are exactly 36 bytes long. It is possible that the reviewer thought that this check would prevent exploitation, but by the time the check is made, the buffer has already been overflowed.

This time the patch added the same kind of check for these subsequent 'anih' subchunks before they are copied. Since the memory of subsequent 'anih' subchunks is allocated dynamically, an additional piece of code was added to free any existing block prior to allocating a new one. So we go from this:

```
if (fccType == "anih")
  copy <size> bytes to stack
  block = allocate 36 bytes
  copy 36 bytes to block
```

to this:

```
if ((fccType == "anih") && (size == 36))
  copy <size> bytes to stack
  if (block != 0) free(block)
  block = allocate 36 bytes
  copy 36 bytes to block
```

instead of the more sensible:

```
if (fccType == "anih")
  if (block == 0) block = allocate 36 bytes
  copy 36 bytes to block
```

for only a single bounded copy operation, with no need to free anything.

BREAKING WINDOWS

Despite *Microsoft*'s claims of improved security, *Vista* is just as vulnerable as every other version of *Windows*. There were supposed to be three mitigating factors, but two of them proved unsuccessful in this case.

The first is the Buffer Security Check (/GS), which also exists in *Windows XP* SP2. This is a stack protection mechanism that is designed to prevent altered return addresses from being used, by checking a value that exists

on the stack below the return address. The idea is that if the return address has been altered, then the magic value must also have been altered. However, because of the performance impact of the /GS protection, it is optional – and even then it is applied only to functions that have certain characteristics. More specifically, /GS protection is applied only to functions that contain arrays of five or more characters (ASCII or Unicode), and is not applied to functions that contain only structures with small individual fields. Since the vulnerable routine contains only structures with small individual fields, the Buffer Security Check was not applied.

Vista's second mitigating factor is Address Space Layout Randomization (ASLR), which allows an ASLR-aware process to have its contents placed at a random address in memory. The idea is to make the return address unlikely to be reached in a single attempt, and thus prevent the exploit from succeeding most of the time. However, there are two methods by which an ANI exploit can defeat ASLR on *Vista*.

The first method is not specific to ANI exploits, but applies to any stack-based exploit for which /GS does not apply. It relies on the fact that, for architectural reasons, ASLR leaves the lower 16 bits of the address unchanged. It randomizes only eight bits of the 32-bit address on the 32-bit versions of *Vista*, but even if all 15 of the available upper bits were randomized (there would be a significant performance impact to that), the vulnerability would remain. All that is required is to find an appropriate instruction within the 64kb block that holds the existing return address. Then only the lower 16 bits need to be modified for exploitation to succeed, no matter where the process exists in memory.

EXCEPTIONAL BEHAVIOUR

The second method is specific to ANI exploits, and comes into play if no appropriate instruction can be found. It relies on the fact that a Structured Exception Handler (SEH) is installed by the caller of the vulnerable function.

If an exception occurs, the SEH gains control, but this particular SEH ignores the error and returns success. The process does not crash, and the user will not notice that anything went wrong. The result is that an attacker can send multiple malicious files to the vulnerable function, each with a different return address. Since there are only 256 possible combinations, it becomes a trivial matter to brute-force the correct address and compromise a vulnerable machine.

The only mitigating factor that stands any chance of success is Data Execution Protection/No eXecute (DEP/NX), which is a method for marking a region of data, such as the stack, as non-executable. The problem is that it works only if it is enabled for the process, and by default, DEP/NX it is not enabled for 32-bit *Internet Explorer* (which is the most likely attack vector), even if hardware-backed DEP/NX is present. A simple attack will attempt to execute directly from the exploited buffer, which DEP/NX will prevent. However, it is possible (though not easy) to craft the stack to execute the VirtualProtect() API on the exploited buffer first, and then to execute the exploited buffer itself. DEP/NX cannot prevent such an action.

Internet Explorer can be exploited easily because it supports animated cursors. According to a *Determina* advisory, *Firefox* is vulnerable too, despite the fact that it does not support animated cursors. However, given the fact that icons can be animated, and that there are multiple paths to the same code (LoadCursor(), LoadIcon(), and LoadImage(), and perhaps things like CopyImage(), GetCursorFrameInfo(), and SetSystemCursor()), it seems highly likely that *Firefox* can be coerced into calling an appropriate API. *Windows Explorer* is exploited automatically without user interaction when browsing a directory that contains a malicious file, because *Explorer* parses the file in order to display the icon.

OOPS I DID IT AGAIN

Continuing the long tradition of attackers who don't seem to understand what they're doing, we saw a collection of odd attempts at exploiting this vulnerability. The funniest one contained the 'LIST' chunk name spelled backwards. This may have been the result of bad disassembling, but the other chunk names were correct, which made the misspelling very perplexing.

There were also chunks with odd-sized blocks, but this is legal and perhaps was used as a detection bypass. In any case, the only requirements for exploitation are two 'anih' blocks, of which the first must be in the correct format (36 bytes long, frame and step count less than 65,536, flags bit 0 set to specify a *Windows* cursor or icon, etc.) and the second must contain the exploit.

CONCLUSION

So what have we learned from all this? The first ANI vulnerability was the result of bad code. The second ANI vulnerability was the result of more bad code. The way to patch bad code is to remove the bad code, not to add new bad code that hides the old bad code.

A more pertinent question would be: what has *Microsoft* learned from all this?