A low risk DoS condition exists in a popular PE analyzer, such that it crashes and drops out of process space when supplied with invalid input. This is local-only and currently is not known to result in execution of arbitrary code.

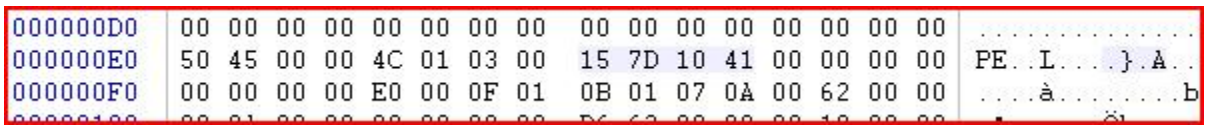
PE files are portable executables – the native format for 32-bit Windows binaries. A PE analyzer is a program to read and display information about other PE files. Each PE file contains a header which holds metadata about the executable (ie entry point address, sections, imports, exports, etc). Among these fields is the link/compile date stamp. The value is designated type DWORD according to the official specification on MSDN:

```
typedef struct _IMAGE_FILE_HEADER {
  WORD Machine;
  WORD NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD SizeOfOptionalHeader;
  WORD Characteristics;
} IMAGE_FILE_HEADER,
 *PIMAGE_FILE_HEADER;
```

The TimeDateStamp represents the number of seconds that have elapsed since epoch. On a 32-bit x86 architecture, a DWORD is 32 bits long, or 4 bytes. Systems that treat this value as a signed integer are only capable of reading time up to 03:14:07 on 01/19/2038 UTC (a 31 bit integer). If the integer which corresponds to this date is increased, it will end up flipping the most significant (signed-ness) bit and the system will read it as negative.

The DoS condition exists because one of the time conversion functions returns a null pointer when it encounters this negative number. A subsequent function is then called for further processing of the time format, however no error checking is done to see if the first function returned a valid pointer or not. The null pointer is entered into a register, and then the system tries to access the memory address at which it points. The program essentially tries to read an invalid region of memory and terminates due to access violation.

To elicit the behavior, I took the standard ftp.exe program from an XP system and viewed the TimeDateStamp. The 4 bytes appear at offset E8-EB of the program, just after the well-known “PE” signature. Here, they are 15 7D 10 41.



According to the target program, this value is early August 2004.

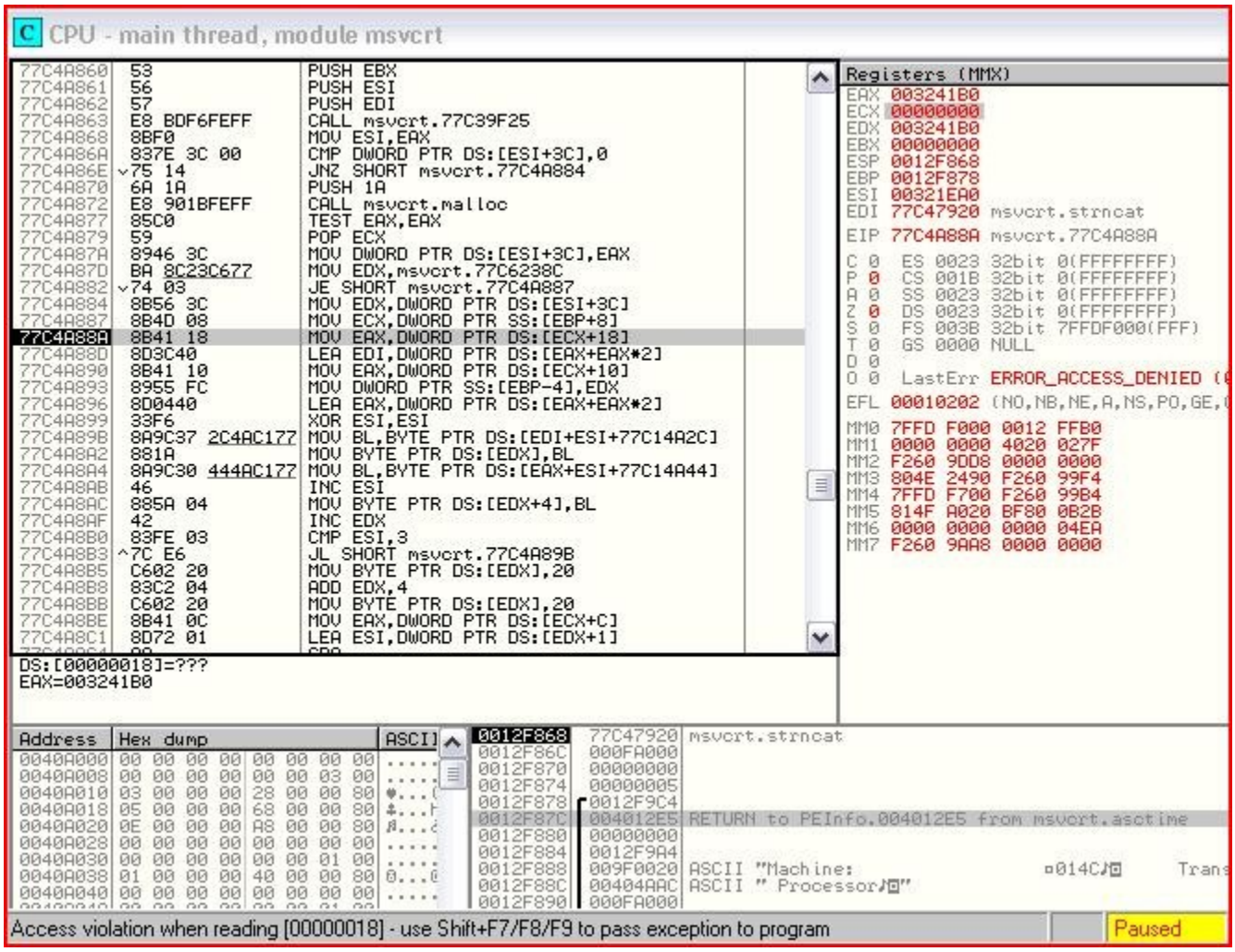
```
TimeDateStamp       :         41107D15
     Created (GMT): Wed Aug 04 06:07:17 2004
```

A pair of functions called by the target program convert the DWORD value into a human readable 26-byte string (24 characters + CR + LF) as output. I began investigating this, because when I manually change the original DWORD value to something else (a much higher number in unsigned terms), my analysis program crashes. By “crashes” I mean it terminates and drops out of the process list with no

indication of what went wrong.

This can be demonstrated by changing 15 7D 10 41 to 15 7D 10 FF (note that due to endian-ness, FF is most significant). The same reaction occurs with any other binary (not just ftp.exe) and can likely also be generalized to any other target programs that use the particular sequence of functions (identified next) for the date conversion on systems where 1) time_t is a signed 32-bit integer and 2) no mechanisms are employed to check the return value of the first function.

I loaded the target program into OllyDbg to find out more about its crash. Once it was loaded, I used it to open the modified version of ftp.exe and let it begin parsing the PE headers. It halted execution on the instruction at 77C4A88A (see top left partition of the image below). The top right shows the registers (note ECX is 00000000), the bottom right shows the stack contents, and the bottom bar notes that an access violation occurred while trying to read memory at address 0x00000018.



Although all panes are significant, the stack contents was most interesting to me first. It contains the name of a function which had just returned – msvcrt.asctime(). Since this function seems related to time conversion, it seemed like a good place to start looking further. The function, as declared in time.h returns a pointer to a 26-byte string that holds the converted date/time stamp. It accepts a pointer to a structure of type tm.

```
char *asctime(
   const struct tm *timeptr
);
```

So, assuming we want to trace the steps that the program itself takes, we start out with a 4-byte hexadecimal value (the DWORD) that can easily be converted to a base10 integer. How does this integer then become a structure of type tm so that it can be given to asctime() for a human-readable string? One answer involves using the time() and localtime() functions. The time() function returns an integer of type time_t and it contains the number of seconds since epoch. Localtime() accepts a pointer to a type_t integer and returns a structure of type tm – perfect! For reference, here are the function prototypes:

```
time_t time(
   time_t *timer
);

struct tm *localtime(
   const time_t *timer
);
```

The purpose of exploring these functions was to gain the ability to write a small program which has the same coding flaw as the target program. This will make analysis much easier, since the vulnerability won't be obscured by all the other functions called by ftp.exe. The only requirements for my sample program are that it must call localtime() and asctime(), in that order. The time() function will not be required, because I will supply the time_t value manually in order to make it negative. The value I supply will represent the TimeDateStamp field of a PE file.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

struct tm *newtime;
time_t tdstamp;

int main( void ) {
   tdstamp = -1596619; // this is FF107D15 in decimal
   newtime = localtime( &tdstamp );
   printf( "Date and time: %s", asctime( newtime ) );
   exit(0);
}
```

As commented in the code, the value of tdstamp (-1596619) corresponds to FF107D15. Remember this is the value of the TimeDateStamp field after making the small change described on page 2. When this sample program is executed, it crashes on the same instruction as the target program (see either screen shot or below for the assemble). This worked extremely well, because it was able to narrow the problem to an interaction between only 2 functions – localhost() and asctime(). The next few paragraphs will describe how the localhost() function's return value affects asctime() in a negative way when no error checking is implemented.

I set a breakpoint on the last instruction within localhost() as declared in localtim.c and stepped through the program with the debugger. Before returning control to main(), the function fills ECX with its return value – a pointer to the structure of type tm that asctime() must use for conversion.

For reference, when localtime() returns after reading a positive number, the registers may look like this:

```
EAX = 0042A76C EBX = 7FFDE000 ECX = 0042A76C
```

On the contrary, when localtime() returns after reading a negative number, the registers may look like this:

```
EAX = 00000000 EBX = 7FFD4000 ECX = 00000000
```

Note the 0s which indicate a null pointer, rather than a valid address. When asctime() begins to execute, it issues the following instruction:

```
MOV EAX, DWORD PTR DS:[ECX+18]
```

This tries to move the 4-byte value that starts at [ECX+18] into the EAX register. Currently ECX is 00000000. Therefore, ECX+18 is 00000018. Unfortunately for the program, it cannot access memory at 00000018, so it cannot read the bytes that reside there in order to move them into EAX. The attempt to read ECX+18 is ultimately what causes the program to crash.

In summary, this is only a DoS vulnerability. Even if we were able to write to addresses in the 00000018 range, we would also need a way to overwrite the function's return address so that the program resumes execution with our supplied code at 00000018 (rather than just reading a value for use in a relatively safe mathematical formula). To fix this flaw, there should be a check for null return values between localtime() and asctime(). It can be accomplished by adding code similar to what appears in red below.

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

struct tm *newtime;
time_t tdstamp;

int main( void ) {
   tdstamp = -1596619; // this is FF107D15 in decimal
   newtime = localtime( &tdstamp );
   if (newtime == NULL) {
     printf("it's null!\n");
     exit(1);
   }
   printf( "Date and time: %s", asctime( newtime ) );
   exit(0);
}
```