

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.

000000D0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
000000E0	50 45 00 00 4C 01 03 00	15 7D 10 41 00 00 00 00	PE..L...}.A..
000000F0	00 00 00 00 E0 00 0F 01	0B 01 07 0A 00 62 00 00	...à.....b
00000100	00 01 00 00 00 00 00 00	00 00 00 00 00 00 00 00

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 endianness, 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.

CPU - main thread, module msvcrt

77C4A860	53	PUSH EBX
77C4A861	56	PUSH ESI
77C4A862	57	PUSH EDI
77C4A863	E8 BDF6FEFF	CALL msvcrt.77C39F25
77C4A868	8BF0	MOV ESI,EAX
77C4A86A	837E 3C 00	CMP DWORD PTR DS:[ESI+3C],0
77C4A86E	75 14	JNZ SHORT msvcrt.77C4A884
77C4A870	6A 1A	PUSH 1A
77C4A872	E8 901BFEFF	CALL msvcrt.malloc
77C4A877	85C0	TEST EAX,EAX
77C4A879	59	POP ECX
77C4A87A	8946 3C	MOV DWORD PTR DS:[ESI+3C],EAX
77C4A87D	BA 8C23C677	MOV EDX,msvcrt.77C6238C
77C4A882	74 03	JE SHORT msvcrt.77C4A887
77C4A884	8B56 3C	MOV EDX,DWORD PTR DS:[ESI+3C]
77C4A887	8B4D 08	MOV ECX,DWORD PTR SS:[EBP+8]
77C4A88A	8B41 18	MOV EAX,DWORD PTR DS:[ECX+18]
77C4A88D	8D3C40	LEA EDI,DWORD PTR DS:[EAX+EAX*2]
77C4A890	8B41 10	MOV EAX,DWORD PTR DS:[ECX+10]
77C4A893	8955 FC	MOV DWORD PTR SS:[EBP-4],EDX
77C4A896	8D0440	LEA EAX,DWORD PTR DS:[EAX+EAX*2]
77C4A899	33F6	XOR ESI,ESI
77C4A89B	8A9C37 2C4AC177	MOV BL,BYTE PTR DS:[EDI+ESI+77C14A2C]
77C4A8A2	881A	MOV BYTE PTR DS:[EDX],BL
77C4A8A4	8A9C30 444AC177	MOV BL,BYTE PTR DS:[EAX+ESI+77C14A44]
77C4A8AB	46	INC ESI
77C4A8AC	885A 04	MOV BYTE PTR DS:[EDX+4],BL
77C4A8AF	42	INC EDX
77C4A8B0	83FE 03	CMP ESI,3
77C4A8B3	7C E6	JL SHORT msvcrt.77C4A89B
77C4A8B5	C602 20	MOV BYTE PTR DS:[EDX],20
77C4A8B8	83C2 04	ADD EDX,4
77C4A8BB	C602 20	MOV BYTE PTR DS:[EDX],20
77C4A8BE	8B41 0C	MOV EAX,DWORD PTR DS:[ECX+C]
77C4A8C1	8D72 01	LEA ESI,DWORD PTR DS:[EDX+1]

Registers (MMX)

EAX	003241B0
ECX	00000000
EDX	003241B0
EBX	00000000
ESP	0012F868
EBP	0012F878
ESI	00321E90
EDI	77C47920
EIP	77C4A88A msvcrt.77C4A88A

Stack

C 0	ES 0023 32bit 0(FFFFFFFF)
P 0	CS 001B 32bit 0(FFFFFFFF)
A 0	SS 0023 32bit 0(FFFFFFFF)
Z 0	DS 0023 32bit 0(FFFFFFFF)
S 0	FS 003B 32bit 7FFDF000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_ACCESS_DENIED (0)
EFL	00010202 (NO,NB,NE,A,NS,PO,GE,0)
MM0	7FFD F000 0012 FFB0
MM1	0000 0000 4020 027F
MM2	F260 90D8 0000 0000
MM3	804E 2490 F260 99F4
MM4	7FFD F700 F260 99B4
MM5	814F A020 BF80 0E2B
MM6	0000 0000 0000 04EA
MM7	F260 9A98 0000 0000

Stack dump

Address	Hex dump	ASCII
0040A000	00 00 00 00 00 00 00 00
0040A008	00 00 00 00 00 00 03 00
0040A010	03 00 00 00 28 00 00 80
0040A018	05 00 00 00 68 00 00 80
0040A020	0E 00 00 00 A8 00 00 80
0040A028	00 00 00 00 00 00 00 00
0040A030	00 00 00 00 00 00 01 00
0040A038	01 00 00 00 40 00 00 80
0040A040	00 00 00 00 00 00 00 00
0040A048	00 00 00 00 00 00 00 00

Access violation when reading [00000018] - use Shift+F7/F8/F9 to pass exception to program

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 time_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 – localtime() and asctime(). The next few paragraphs will describe how the localtime() 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 localtime() as declared in localtime.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);
}
```