Report

Integer Overflow Vulnerability

CVE-2013-4233

Anastasia Bolotnikova

Tartu 2014
Content

Content .................................................................................................................................................. 2
Vulnerability ........................................................................................................................................... 3
  What is CVE-2013-4233 ...................................................................................................................... 3
  How it happens and why is it dangerous .............................................................................................. 3
Solution ................................................................................................................................................. 4
  How was it fixed .................................................................................................................................. 4
  How does this fix work ......................................................................................................................... 4
References ............................................................................................................................................. 6
Vulnerability

What is CVE-2013-4233
This vulnerability have been found in the product called libmodplug and it have been published on 9th August 2013 [1]. The libmodplug is a library that allows to load and render music module files of many different types. One of the music module files that interests us in particular when talking about CVE-2013-4233 is the ABC file in which we use the ABC notation [2] to represent the notes and other elements of a musical composition.

The vulnerability itself consist in an integer overflow in ‘j’ variable in the abc_set_parts function in load_abc.cpp file in libmodplug 0.8.8.4 and some earlier versions. In simple terms, the main problem is that there is a ‘j’ variable of type integer in the code which is supposed to parse ABC file and sometimes this variable wants to be greater than it can be. How does it happen and why this is dangerous will be described in the next chapter.

How it happens and why is it dangerous
Everything starts when there is a ‘P:’ element in the header of an ABC file that is supposed to be parsed. This element indicates the order in which the parts (P:) of the musical composition are played, for example ‘P:ABA’, ‘P:BAA’ (here A and B are some certain parts of a tune). And while parsing the string of the ‘P:’ element the function abc_set_parts is supposed to 'decode constructs like "((AB)2.(CD)2).(AB)E2" to “ABABCDABABCDABABCDABABCDABE” ’ as stated in the code of the program [3].

When the program meets a ‘P:’ element it calls a abc_set_parts function which starts to count how much storage it will need to allocate for the decoded string. While doing so it calls another function – abc_getnumber that covert a decimal string of an input to an integer value. And with particularly crafted P headers it is possible to make ‘j’ variable equals the maximum value of an integer. And when allocating the storage they use a value ‘j+1’ for indicating the amount of memory to be allocated. The problem is that in the case of a maximum value of a variable of type integer, ‘j+1’ will become negative (-0 to be precise) and so allocated memory for the decoded string will be zero byte. Then not enough allocated storage space will lead to a heap-based buffer overflow when we will try to decode the string and save it in the dynamic memory. This will lead to the unintended behavior of the program and will cause a denial of service condition, application will crash.

Potentially the condition of a heap-based buffer overflow may also be exploited to execute some other (dangerous) code on the system. But in the case of CVE-2013-4233 it is quite complicated due to the following input check in the first for-loop of the abc_set_parts function:

```c
if (!strchr("ABCDEFGHIJKLMNOPQRSTUVWXYZ().0123456789 ",p[i]) ) {
    abc_message("invalid characters in part string scanning P:%s", p);
}
```

So the only characters that attacker can use are uppercase letters and numbers. Which makes it much harder to come up with the code that can do something useful for the attacker and bad to the system.
Solution

How was it fixed
Here is a commit with CVE-2013-4233 fix [4]:

```plaintext
--- a/libmodplug/src/load_abc.cpp
+++ b/libmodplug/src/load_abc.cpp
@@ -1814,7 +1814,7 @@
 static void abc_set_parts(char **d, char *p)
 {
- int i,j,k,m,n;
+ int i,j,k,m,n,size;
   char *q;
   #ifdef NEWMIKMOD
   static MM_ALLOC *h;
@@ -1852,10 +1852,11 @@
      i += n-1;
   }
- q = (char *)_mm_calloc(h, j+1, sizeof(char));  // enough storage for the worst case
+ size = (j + 1) 0 ? j+1 : j;
+ q = (char *)_mm_calloc(h, size, sizeof(char));  // enough storage for the worst case
   // now copy bytes from p to *d, taking parens and digits in account
   j = 0;
- for( i=0; p[i] && p[i] != '%'; i++ ) {
+ for( i=0; p[i] && p[i] != '%' && j < size; i++ ) {
     if( isdigit(p[i]) || isupper(p[i]) || p[i] == '(' || p[i] == ')' ) {
       if( p[i] == ')' ) {
         for( n=j; n > 0 && q[n-1] != '('; n-- ); // find open paren in q

These code changes have been implemented in libmodplug version 0.8.8.5. So to resolve this problem on your machine it is necessary to upgrade the libmodplug to the version 0.8.8.5 or later.

How does this fix work
As can be seen in the code above the additional variable ‘size’ has been implemented and used instead of a ‘j+1’ value when allocating the memory for the uncompressed string.

Variable ‘j’ is initially zero and then it changes its value in the second for-loop of the function. The value of the ‘size’ variable is set correspondingly to the value of ‘j’ after the second for-loop and is defined as `size = (j + 1) 0 ? j+1 : j;`, which means that if ‘j+1’ is not negative, everything is ok and we can use ‘j+1’ value when allocating the memory, but if adding 1 to a positive value of a variable ‘j’ will give us a negative number, then something must have gone wrong and in this case we will use ‘j’ value to allocate the memory for the uncompressed string.

In order to understand why we get a negative value as a result of adding 1 to a positive ‘j’, we need to go into the details about how the addition process happens on a level of binary numbers.
First, we need to say that it is impossible to define exactly what is the maximum value of ‘j’ variable as the size of each data type in C++ varies depending on the processor we use. But for this example, let’s say we use a 32 bit system, so the size of an integer in this case will be 2 bytes. The maximum value for the variable of type integer will then be 32767, which in the binary code is represented as 0111111111111111, here the first bit is zero which means that the number is positive. And now if we try to add 1 to the maximum value of an integer, here’s what happens: 0111111111111111 + 0000000000000001 = 1000000000000000. Translating it into decimal format will look like 32767 + 1 = -0. If we were to write something like this for example in Java, we would simply get an error, but since it is C++ the program would think that the programmers know what they are doing and they actually want to get zero as a result of a given computation.

As a result of this integer overflow, when using the ‘j+1’ for allocating the memory the function _mm_calloc allocates 0 byte for the uncompressed string, so we have no storage for the result of the function. But with the corrections of the code we will avoid this situation and in the worst case allocated memory will be the size of a maximum integer value. And in the third for-loop of the abc_set_parts function the ‘size’ variable will be used to control whether the string that we decode can fit to the memory we allocated for it or not. The for-loop will only continue if the string fits.
References


