Matthew Long – [Long0418@umn.edu](mailto:Long0418@umn.edu)

CSCI 4271W

Spring 2024

Project 1

**BCIMGVIEW Code Audit**

Overall, the code provided in BCIMGVIEW.c is mostly secure, however, some vulnerabilities need to be addressed. First, I reviewed and analyzed the code to determine how data flows throughout the program and where might an attacker be able to influence that data flow. After several days of thorough auditing and review of the code, I have found three security vulnerabilities, each of which can lead to control flow hijacking. The three vulnerabilities I found are a format string vulnerability, and two heap buffer overflows. Each of these vulnerabilities are distinct but equally as dangerous if not addressed.

I began my audit by analyzing the code and making a threat model diagram. The most important parts of the diagram (Figure 1) are where there is a data flow connecting an object from outside the trust boundary to an object inside the trust boundary. After examining the threat model, there are 5 points in which data flows across this trust boundary. However, after examining these data flows, only 3 of them are relevant. The three relevant data flows are from the "BCPROG Input File", "BCFLAT Input File", and "BCRAW Input File". These three are of concern because in all three cases, data flows into the program. Although data flows from the "on\_open\_image" to the "file system" this is less of a concern because data cannot flow from the file system back into the program, it is only used by the program to view directories and select a new input file, the act of which is not dangerous. The last remaining data flow is from the user to the Linux terminal. The data flow from user to terminal is not dangerous, in the context of this program, because the inputs from the user are validated; all they can do is launch the program and select an input file. Therefore, when it comes to auditing the program, the focus will primarily be on the functions that use the three input files. This is because, in the absence of other data flows, the only way to exploit a vulnerability in this program is through some malicious input file.

After conducting my threat modeling, I completely audited the code, focusing on how those three input files are read and used by the program. I began my audit by first reading the entirety of the code, ensuring that I had a solid understanding of the purpose of each function, and how the three input files were structured. The first time I read through the code, I read through it solely to understand it, not to find vulnerabilities. After understanding the code, I read through it again, this time taking note of any location that used a function pointer, used a printf, malloc'd memory, array declarations, or anywhere that did multiplication/addition without bounds checking. I was looking for function pointers because they can enable control flow hijack if corrupted. I was looking for printf calls because if there is any printf without a declared variable type, an attacker might be able to exploit that printf. Next, I looked for any malloc calls and array declarations because if the amount of memory allocated is not sufficient for its requirements, a buffer overflow could exist. Lastly, I looked for multiplication and additions without bounds checking, because then there would exist the possibility for integer overflow, which could be used with malloc to declare an array smaller than it should be. After taking note of these locations in the code, I then analyzed each of these locations to see if they were a vulnerability or not.

After looking at all the pointers, I determined that two such pointers are of concern. The first pointer is the per\_image\_callback() function. This is of concern because although it is initially set to &count\_image if there is a way to overwrite it so it points to another memory address, whenever per\_image\_callback() is called, which occurs when an image is displayed, any function could be called instead. The next pointer I determined was of concern was the cleanup destructor callback in the image\_info struct. This is of concern because if there exists any way to overflow the data in the struct, an attacker could overwrite cleanup to instead hold another memory address, they could control what function is called instead of free\_pixels whenever the destructor is called. Next, I analyzed all of the printf calls and determined that there was one instance in which an input file could control a variable in the printf call, and that is the printf in print\_log\_msg() that uses logging\_fmt. This is of concern because logging\_fmt only declares the first three variables, %ld, %ld, and %s for the height, width, and start\_time variables respectively. That leaves the fourth variable, info->create\_time potentially vulnerable because an attacker could control what the variable type is and potentially overwrite info->create\_time to hold another memory address. After looking at all of the malloc calls in the program, I determined that the only malloc of concern is the malloc for the pixels buffer in the image\_info struct although the vulnerability exists in the processing of all three input file types. This malloc is concerning because it multiplies the declared height and width by 3, with no bound checking. Therefore, if the height and width are sufficiently large, they could overflow when multiplied by 3, resulting in a pixel buffer that is smaller than required. I found no declared arrays of concern. The only multiplication/addition of concern in the code, as mentioned previously, exists in all three file types with the multiplication of num\_bytes for the creation of the pixel buffer. While completing my audit, although I was not explicitly looking for it initially, I encountered some do-while loops in the processing of bcprog files that could potentially lead to an off-by-one bug because if the height is less than 3, extra data could be read in than should be, potentially overflowing data in the info\_footer->cleanup. Once I completed this audit, I went in-depth into each of these potential vulnerabilities to determine if they could be exploited.

The first vulnerability I explored in depth is a heap buffer overflow. This vulnerability is a combination of two concerns I noted in my audit above, the integer overflow in the calculation of num\_bytes, and the malloc of the pixels buffer. This is a vulnerability because when num\_bytes is calculated, (3 \* width \* height), there are no bounds checking on width and height, when it is multiplied by 3, the value of num\_bytes could overflow and be much smaller than it should be (see Figure 2). This is a problem because num\_bytes is then used to determine the size of the pixels buffer for the malloc call. If an attacker were to set the height and width to be sufficiently large, to cause an overflow of num\_bytes, they could overwrite the values in the info\_footer because of the pixels buffer. To understand this, we have to look at how the info\_footer is created. The info\_footer is created using the location of the pixels buffer with an offset of the value of num\_bytes. Therefore, if num\_bytes is at least 24 bytes smaller than it should be because of an int overflow, the starting address of info\_footer, will be shifted 24 bytes to the left of what it should be, creating an overlap with the last 24 bytes of the pixels buffer and the first 24 bytes of the info\_footer struct. Using this information, an attacker could overwrite the first three variables in the info\_footer struct with the values of the last 24 bytes read into the pixel buffer from the DATA section of the input file. This would overwrite the width, height, and cleanup function call in the info\_footer struct. What's important here is that the cleanup function can be overwritten because it can be changed from &free\_pixels to any memory address the attacker wants. To demonstrate this is a vulnerability that can be exploited, I wrote a bcraw input file that accomplishes this by overwriting the cleanup function call to instead call the shellcode\_target function, viewable in Figure 3. To achieve this, I declared the initial width to be 9, and the height to be 159,180,280, which results in num\_bytes to be 2,900,264 instead of 4,297,867,560 like it should be. I chose these values for the height and width because after using AFL, I determined that the width needs to be significantly smaller than the height. After all, although it achieves the same overwrite, it later fails the gdk assertions or seg\_faults and therefore doesn't achieve the same result. Next, I filled the DATA section so that the last 24 bytes of the DATA section are "4E40400000000000" + "0100000000000000" + "0100000000000000". By Doing so, the entire pixels buffer is filled with meaningless data except that the last 24 bytes overlap and overwrite the first three variables in the info\_footer. This sets the height and width both to be 1, although the values could theoretically be anything greater than 0, and info\_footer->cleanup to hold the memory address of shellcode\_target 0x40404E (note that is read in little-endian). Now when you run the program with this input file, shellcode\_target() is called, as demonstrated in Figure 4. Given this apparent vulnerability, it is clear that these three functions that have the num\_bytes calculation (parse\_bcprog(), parse\_bcraw(), parse\_bcflat()) all need to be modified to ensure this exploit is no longer possible. To do so is quite simple an addition to the code. In each of those functions, if you add a 64-bit check on the calculation, you can ensure that the value of num\_bytes matches the 64-bit calculation. If when you compare the value of num\_bytes to that 64-bit variable, they are different, there must have been an overflow in num\_bytes, whereas if they are the same, the dimensions are within the bounds that prevent an overflow. To implement this fix, I did the same calculation but stored it in a long long since that is 64-bits long (see Figure 5). Testing the new code with the same input file indicated that this vulnerability was no longer present, as it returned an error in the program without crashing. Therefore, with this rather simple addition to the code, this vulnerability will no longer be present.

Next, I investigated in-depth the format string vulnerability. This vulnerability stems from the fact that the variable logging\_fmt is instantiated with 3 variable types (see Figure 6) but uses 4 variables (see Figure 7) when used in the printf of the print\_log\_msg function. This means that the fourth variable type is potentially able to be controlled by an attacker. Once the attacker has controlled the input to the fourth variable, they could use the fact that %n allows you to write into a printf. Using this, if we write into the fourth variable the number of bytes of padding that we want, equivalent to the decimal address of the target function we want to call, and then insert at that location in the hex file, the address of the per\_image\_callback() function, then info\_footer->create\_time will now hold the address of per\_image\_callback() but per\_image\_callback() holds the address of our target function, in this case shellcode\_target(). To help make sense of this, we can visualize what this looks like with Figure 8. In this example, I took the sample file flag.bcflat and overwrote the 4 bytes of data before the FRMT tag, the memory address of per\_image\_callback. This was done because if you run the flag.bcflat file in GDB and print the address of info\_footer->create\_time it prints the bytes that correspond with the 4 bytes of data before the FRMT tag; so by overwriting those 4 bytes, we change what info\_footer->create\_time holds. Now if we go immediately after the first FRMT tag, H, and replace the next bytes with "%4210766p%4$ln", we manipulate it so that 4210766 bytes will be printed out as the first argument after the format string, and then the number 4210766 will be written to the address pointed by info\_footer->create\_time. This is significant because 4210766 is the decimal address of shellcode\_target, so when combined with the fact that info\_footer->create\_time holds the per\_image\_callback() address and the bytes of padding for shellcode\_target, per\_image\_callback() will instead call shellcode\_target(). Now when we run this file described, shellcode\_target() is executed as expected (see Figure 9). Certainly, this is a vulnerability exploit not intended by the developers and must be addressed to prevent control flow hijacking by any would-be attacker. Fixing this is even simpler than the previous vulnerability. By looking at the output that is printed whenever an image is displayed, nothing gets printed for info->create\_time, therefore, if we simply remove that argument from the printf call in print\_log\_msg, the user experience remains unchanged, and the vulnerability no longer exists. To see what this looks like in practice, see Figure 10 where I implemented this. Testing the attack file mentioned previously with the newly compiled code indicated that this exploit is no longer possible as it now results in an error, without crashing the program or calling the shellcode\_target. Therefore, it is clear that with this small adjustment to the code, the program is more secure, and the format string injection vulnerability no longer exists.

Lastly, I dived deep into the heap buffer overflow that is possible because of the off-by-one bug present from BCPROG files. This vulnerability stems from the fact that in the do-while loops of read\_prog\_data (see Figure 11) if the height is a value less than 3, there is an off-by-one bug in which one row of the data doesn't get read into the pixels buffer. This is a problem because after the pixels buffer in the image\_info struct is the cleanup destructor callback. This vulnerability is rather similar to the first vulnerability I discovered but it is unique in its implementation in that achieves the overwrite of cleanup in a completely different manner. This vulnerability can be exploited by picking dimensions that cause the last 24 bytes of the DATA to hold a desired new width, height, and the memory address of our target function, shellcode\_target(). We can visually see what this looks like with the input file I made (see Figure 12). In this file, I chose a height of 2 and a width of 24; I chose a height of 2 because it's less than 3, so the off-by-one bug will be present, and 24 because BCIMGVIEW reads data from the file in chunks of 24 bytes at a time. Now using these values, num\_bytes is going to be 144 (see Figure 2 for num\_bytes calculation). Knowing that num\_bytes is 144 is critical because that means the pixel buffer is going to be 144 bytes long. If we then go to the DATA tag and insert 127 meaningless bytes, and then insert at the 128th bit, our new intended width, height, and shellcode\_target() address, we will have completed the attack. This works because the do-while is going to skip the last row (24 bytes from the width) of the image when reading into the pixels buffer, the info\_footer->width, info\_footer->height, and info\_footer->cleanup are all stored sequentially immediately after the pixels buffer. Therefore, by putting "0100000000000000" + "0100000000000000" + "4E40400000000000" at the end of the DATA tag, we will have changed the height and width to both be 1 and the info\_footer->cleanup function to instead call the shellcode\_target function. To see how this attack works, see Figure 13. Certainly, this heap buffer overflow is not intentional and to prevent this the developers should fix their do-while loops so that the off-by-one bug is no longer present. To implement this fix, all the developer must do is modify the second pass of the do-while loop to instead only enter if the row is greater than or equal to 3. This way, it doesn’t leave a row unread into the pixel buffer. To see what this looks like in practice, see Figure 14 in which I implemented this solution. Compiling the adjusted code and running the program with the original malicious input file, the attack is no longer possible as it now gives an error, without killing the program or calling shellcode\_target(). Testing the program with non-malicious input files indicates that the program still executes entirely as expected and displays images correctly with no lost data.

Given the vulnerabilities identified during my audit of the program, it is imperative for the developers of BCIMGVIEW.c to take immediate action to address these issues. All three of these vulnerabilities: format string, and both heap buffer overflows; present a clear and present danger to the security of this program. In conclusion, while the code audit revealed several vulnerabilities, the issues identified are certainly all rectifiable with rather simple adjustments to the code that still ensure easy readability of the code and without refactoring much of the code. With the necessary changes provided for the vulnerabilities present in the initial state of the program, the program will undoubtedly become more secure and robust in its defense against control flow hijacking.

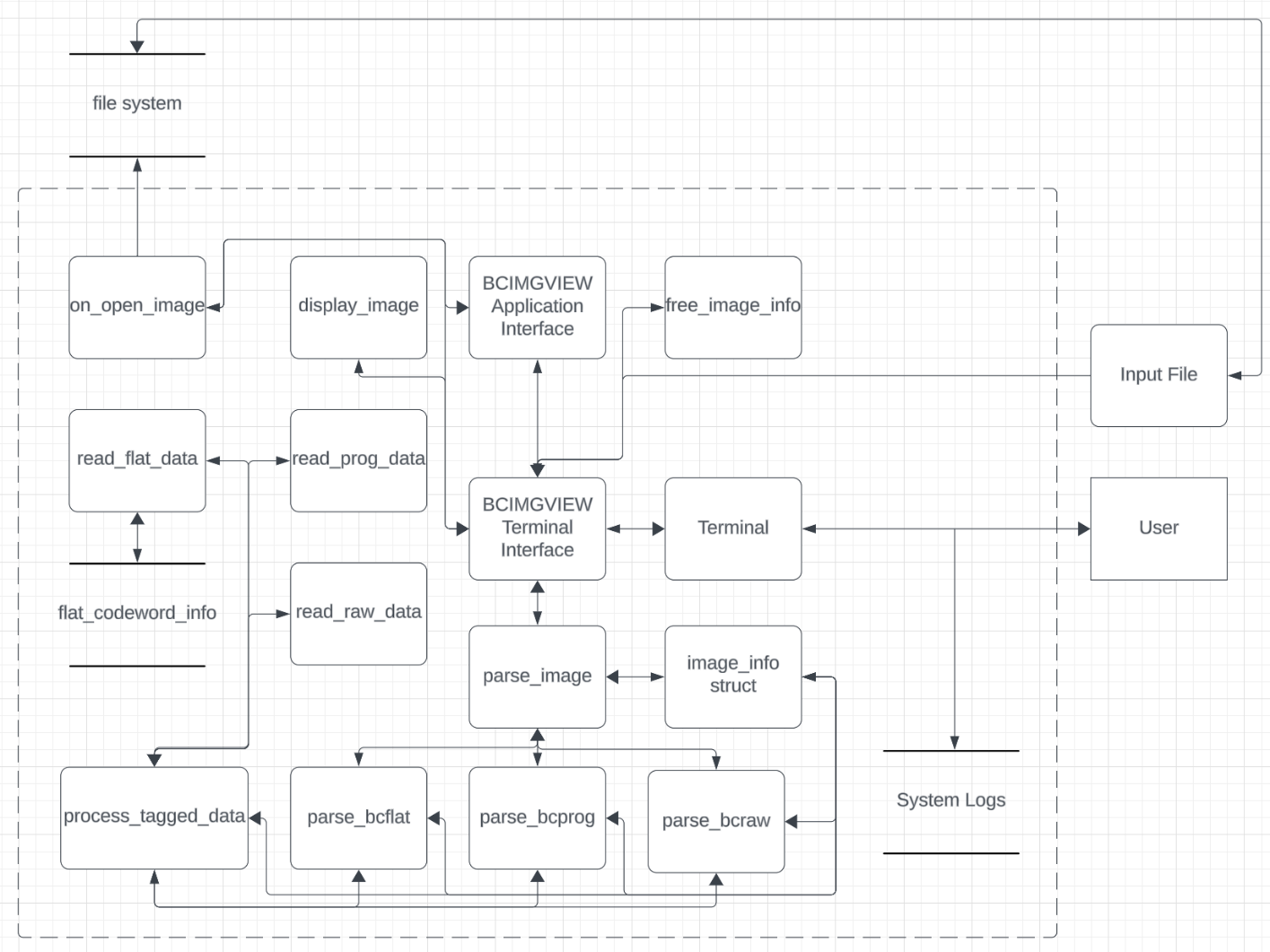


Figure 1 Threat Model Diagram of BCIMGVIEW.c

A screen shot of a computer program

Description automatically generated

Figure 2 integer overflow in calculation of num\_bytes leads to heap buffer overflow in malloc of pixels buffer

A screenshot of a computer

Description automatically generated

Figure 3 num\_bytes.bcraw input file for a heap buffer overflow vulnerability

A screenshot of a computer

Description automatically generated

Figure 4 Output from running file num\_bytes.bcraw on BCIMGVIEW.cA computer screen with colorful text

Description automatically generated

Figure 5 Modified code with test for int overflow in num\_bytes calculation

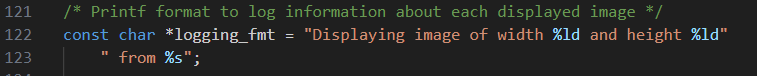


Figure 6 logging\_fmt declaration

A screen shot of a computer code

Description automatically generated

Figure 7 print\_log\_msg function

A screenshot of a computer

Description automatically generated

Figure 8 fmt.bcflat input file for a format string vulnerability

A screenshot of a computer error

Description automatically generated

Figure 9 output from running fmt.bcflat on BCIMGVIEW.cA computer screen shot of text

Description automatically generated

Figure 10 Modified code without format string injection possibility

A screenshot of a computer program

Description automatically generated

Figure 11 read\_prog\_data - off by one bug leads to a heap overflow buffer vulnerability

A screenshot of a computer

Description automatically generated

Figure 12 offByOne.bcprog input file for heap buffer overflow vulnerability

A screenshot of a computer

Description automatically generated

Figure 13 output from running offByOne.bcprog on BCIMGVIEW.cA screen shot of a computer program

Description automatically generated

Figure 14 Modified code that ensures no heap overflow vulnerability via off by one bug

I primarily only used GDB and Grammarly Spellchecker for my work. I did, primarily out of curiosity, when I first started the project try asking ChatGPT if it found any vulnerabilities in the program, but it was not very helpful and stated the program was fully secure with no vulnerabilities.