1 Threat Modeling

The architecture of Badly Coded Image Viewer (referred to as BCImgView) in this data flow diagram is split into components by logical functions rather than direct mappings to source code functions. In this context, an Attacker is someone who provides a compromised image file to a naive user (the Victim) that will cause some havoc. Depending on the situation, I will describe the user as either a user or the Victim, whether or not the situation is benign in description. The following subheadings will describe each component of the program.

1.1 Process File Input

This component represents several points in the source code where the user can select a file from the file system to read into BCImgView. It opens the file and determines which proprietary image format type it will process the file as. It is responsible for executing the functionality of Process Tagged Data and Process Pixel Data since it necessarily must occur first. An Attacker has 2 vectors of control here: the name of the file to read and the contents of the file.

In this component, an Attacker can specifically control the `flags`, `width`, and `height` fields of the `image_info` data structure used throughout the program, albeit with some restrictions.

1.2 Process Tagged Data

This component maps to the `process_tagged_data(FILE, image_info)` function in the source code. It is responsible for reading the meta "tagged" data from the image file and assigning them to relevant structures in the program. It handles several metadata identifiers: `DATA`, `TIME`, and `FRMT`, and appears to be designed to accommodate more in the future. It also verifies that tags are not too long, nor too short, or unrecognized. Notably, the `FRMT` identifier assigns the global variable `logging_fmt` to the contents of its read, affecting the functionality of Print Embedded Image Log Message.

An Attacker can control the image bytes that supply the values of these metadata. The values persist in the `image_info->create_time` and global `logging_fmt` variables.

1.3 Process Pixel Data

This component maps to several functions in the source code: `read_raw_data`, `read_flat_data`, `read_prog_data` . It is responsible for converting the proprietary image formats into a pixel buffer in memory. Each function converts the image differently, but that is not relevant to the divisions of this program into components. It begins reading the input file immediately after the location of the `DATA` metadata identifier.

An Attacker can control the image bytes that correspond to pixel data, which notably do not have to match size with the assigned pixel buffer. This is a threat because more or less data could be put in the image bytes, causing undefined behavior in the program.

1.4 Print Embedded Image Log Message

This component maps to the source code's `print_log_msg(image_info)` function. The message it prints is defined by the global variable `logging_fmt`. Its default behavior prints a log message to standard output that tells the user the width and height of the image and when it was created.

Because an Attacker can control `logging_fmt` in Process Tagged Data, this component represents a substantial threat because the use of `printf` with Attacker controlled values might allow a *Format String Injection* attack to occur.

1.5 Display Image on GUI

This component is responsible for all the parts of the program involved in displaying the image with GTK. It handles window creation, window lifetime management, and all GUI logic. It then calls Free Image Info from Memory once it is done using the pixel data. Finally, it dereferences and then calls the `per_image_callback` function pointer. This component only executes when the user executes the program standalone: as a desktop shortcut or with no command-line arguments, or with one argument that is the image file path.

The threat in this component is calling a non-static function pointer which could be writeable by an Attacker.

1.6 Convert Image to PPM

This component is responsible for converting the internal pixel buffer to a PPM file. It saves the PPM file to the same location as the input file, with the `.ppm` extension added to the filename. It then calls Free Image Info from Memory once it is done using the pixel data. Finally, it dereferences and then calls the `per_image_callback` function pointer. It only executes if the user runs the non-GUI program or uses the command-line arguments `-c <image path>`.

As this component is sort of a mirror to Display Image on GUI, the threat here is also calling a non-static function pointer which could be writeable by an Attacker.

1.7 Free Image Info from Memory

This component maps to the `free_image_info(image_info)` function in the source code. It is responsible for freeing the pixel data with the `image_info->cleanup` function pointer. After that, it frees the `image_info` data structure and returns. It appears to have been designed with extension in mind, as different internal pixel representations could be handled freed with this dynamic `cleanup` function pointer, but in the source code, only `free_pixels(image_info)` is used.

While the source code does not allow assigning the `image_info->cleanup` pointer—the parse image format functions do that explicitly—it may be possible for an Attacker to abuse another exploit to reassign this variable and execute arbitrary code.

1.8 Data Flow Diagram

See Figure 1 for the picture of the data flow diagram.

Most of the components are executed in sequence in `main`, so the flow of the arrows generally dictates both data flow and control flow. On the left side of the arrows is the data the program assumes is being passed between components. On the right side of the arrows is the potential threat flowing that the data flow implies.

There are 2 external systems that I have classified BCImgView as interacting with: The file system and the GTK window system. The 1st of the 3 trust boundaries is the program trust boundary, labeled *BCImgView program trust*, which assumes all internal components generally trust each other. The 2nd trust boundary, *Valid image metadata trust*, assumes that the image metadata—`width`, `height`, `flags`, `create_time`, and global `logging_fmt`—are valid. The 3rd trust boundary, *Valid pixel buffer trust*, assumes a valid pixel buffer around Process Pixel Data, Display Image on GUI, and Convert Image to PPM; this exists because each component trusts that the internal pixel buffer created in Process Pixel Data is valid.

2 Code Audit

2.1 AFL Testing Results

The first step in my code audit process was fuzzing. By fuzzing early, I could read the source code and find crashes with AFL simultaneously. After compiling the source code with AFL-CC, I used the sample inputs provided with the binary to fuzz the program. I fuzzed with this setup for 10 hours and 12 minutes, completing 232 cycles, and finding 15 crashes. After this round, I minimized the crashes and used them along with the original inputs as inputs for a second round of fuzzing. I ran this round for 1 hour and 51 minutes, completing 62 cycles, and finding 17 crashes (2 new unique).

2.1.1 Overlapping Memory Allocation in *BCRaw*

This crash is a result of the bug I found in Integer Overflows while Parsing *BCRaw* during static analysis, although AFL created this binary. See Figure 2 for the image file binary. I ran this crash input in GDB to analyze it. I found that when casting the product of 3, `width`, and `height` to **`int`**, it caused the assignment to `num_bytes` to be 0. This then causes the result of `trailer_location` to assign pointers `pixels` and `info_footer` to the same memory location. Initially, `info_footer->cleanup` is set to `free_pixels`, but in `read_raw_data` this gets overwritten by what gets put into `pixels[16..23]`. The image file binary that AFL generated had `0x3030'3030'3030'3030` in that section of the bytes, which caused a `SEGFAULT` by dereferencing that value as an address.

2.1.2 Parsing Pixel Data Overwrites Image Width in *BCProg*

This crash caused a `SEGFAULT` by attempting to write to a protected memory address. By looking at the stack trace, I found that this occurred in `read_prog_data` which is in the Process Pixel Data component. While reading pixel data from the image file, it can read too far and overwrite the value of `info->width` while still completing a full read. When the loop executes again, the destination of the read will be different since it is based on `info->width` (See Listing 1) and it will write image bytes to that place in memory. This is exploitable by an attacker and can be used to write *what where*. See Figure 3 for the image file binary.

2.2 Static Analysis Findings

In this subsection, I will describe potentially exploitable bugs or flawed designs I found during my static analysis of the source code. I will link these bugs to their respective component in Thread Modeling.

2.2.1 Integer Overflows while Parsing *BCRaw*

This bug occurs in the Process File Input component, in the `parse_bcraw` function, while parsing the *BCRaw* image format. As described there, the `width` and `height` fields of the `image_info` data structure are Attacker controlled. There are several areas where there are integer overflows in this component.

The type of the local variables `width` and `height` in `parse_bcraw` is **`long`**. However, the function `read_u64_bigendian` reads and returns an unsigned 64-bit integer. This can cause overflow because any unsigned 64-bit integer greater than `0x7FFF'FFFF'FFFF'FFFF` will be interpreted as a negative signed 64-bit integer. Because the only constraint for `width` and `height` is that they are not equal to -1, the allocation of the pixel buffer can be too small to handle even the image info (footer). This can lead to corruption or exploitation. I will not link to a listing of this bug, as the code does not make this much clearer. I suggest that the width and height of a BCRaw image should be constrained to unsigned 32-bit integers, which would mean the maximum area would fit in a 64-bit unsigned integer.

An Attacker can control `width` and `height` to cause an integer overflow while computing the size for a `malloc` and cause corruption later in the program when there is less memory than expected. See Listing 2 for this bug's source code.

The type of `num_bytes` is **`int`**. This is a critical error because the type of the `width` and `height` local variables in the Process File Input component is **`long`**, so if multiplication is not constrained the size of `num_bytes` must be at least as large as the square of **`long`**. This bug only manifests with the *BCRaw* image format because *BCProg* and *BCFlat* enforce size constraints. However, it is bad practice and should be adjusted and annotated to aid future security—it is always possible for the size constraints to change, and they are not stated to be there to prevent overflow. See Listing 3 for this bug's source code.

2.2.2 Size of Image Bytes Decoupled from Width and Height Parameters

This bug also occurs in the Process Pixel Data component but is a problem for all image formats. The pixel buffer is created assuming that there are the correct amount of image bytes for the `width` and `height` variables. However, this is not enforced.

2.2.3 Attacker Controlled Format String

This bug occurs in the Print Embedded Image Log Message component. The `FRMT` metadata tag allows an Attacker to control the value of the `logging_fmt` global variable with no limitations. This potentially leaves the program vulnerable to a format string injection attack. See Listing 4 for this bug's source code.

2.3 Summary of Bugs and Potential Threats

3 Forming Attacks

3.1 Format String Injection

This attack relies on the vulnerabilities found in the Print Embedded Image Log Message and Free Image Info from Memory components. I created this attack without help from an AFL crash. The *what* of the format string attack is the address of `shellcode_target` in memory: `0x40404e`. The *where* of the attack is the address of the `info->cleanup` field, or `per_image_callback` global variable, although I chose the former. To make the attack work, I set the value of the `TIME` metadata to the address of `info->cleanup` and set the value of the `FRMT` metadata to print out exactly enough characters to change its function pointer to `shellcode_target`. See Figure 4 for the image binary of this attack.

3.2 Overlapping Memory Allocation

This attack relies on the bug found in Overlapping Memory Allocation in BCRaw. I created this attack by analyzing AFL crash #6 from fuzz testing. I was able to get it to overwrite address of `info->cleanup` with the address of `shellcode_target` by inserting the address in the correct place in the image binary. This was a very simple attack, and it is all described in the code audit process where I found the bug. See Figure 5 for the image binary of this attack.

4 Recommendations

I recommend removing function pointers until they are further evaluated and deemed necessary for functionality. I also recommend removing the custom formatting for console logs and replacing them with the ability to print a custom string instead. Finally, I recommend reevaluating the types of integers and analyzing multiplications near memory allocations to prevent integer overflows and situations where a pixel buffer cannot accommodate an image.

5 Figures

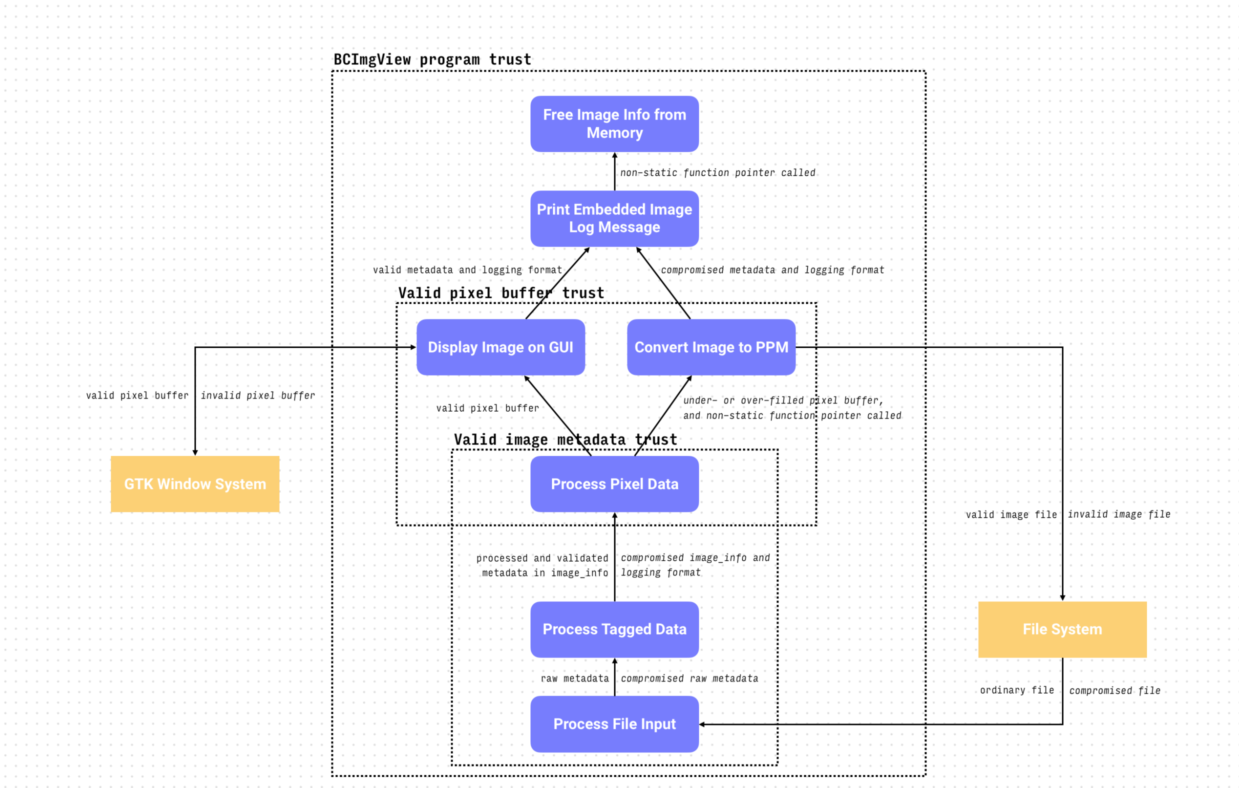

Figure 1: BCImgView Data Flow Diagram

	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	Decoded Text
00000000	00 42 43 52 C3 84 57 0A 00 00 00 00 00 00 00 08	. B C R . . W . . . .
00000010	30 30 30 30 30 00 00 00 30 30 30 30 30 30 30 30	0 0 0 0 0 . . . 0 0 0 0 0 0 0 0
00000020	44 41 54 41 00 00 00 00 00 00 00 00 30 30 30 00	D A T A . . . . 0 0 0 .
00000030	00 00 00 00 30 30 30 30 30 30 30 30	. . . . 0 0 0 0 0 0 0 0

Figure 2: AFL Crash #0 Overlapping Memory Allocation Crash

Listing 1: BCProg Overwrite of Image Width during Parse on Line 358

```
357  unsigned char *row_start = p + row * 3 * info->width;
358  num_read = fread(row_start, info->width, 1, fh);
```

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   Decoded Text
00000000  42 43 50 52 C3 96 47 0A 00 00 00 00 00 00 01 D8   B C P R . . G . . . . . . . . .
00000010  00 00 00 00 00 00 00 05 00 00 00 00 00 00 00 02   . . . . . . . . . . . . . . . .
00000020  54 49 4D 45 00 00 00 00 00 00 00 08 30 30 30 30   T I M E . . . . . . . . 0 0 0 0
00000030  30 30 30 30 44 41 54 41 30 30 30 30 30 30 30 30   0 0 0 0 D A T A 0 0 0 0 0 0 0 0
00000040  30 30 30                                          0 0 0
```

Figure 3: AFL Crash #6 Parsing Pixel Data Overwrites Image Width in *BCProg*

Listing 2: BCRaw Overflow on Unchecked Memory Allocation on Line 275

```
267 width = read_u64_bigendian(fh);
268 if (width == -1)
269     return 0;
270 height = read_u64_bigendian(fh);
271 if (height == -1)
272     return 0;
273
274 num_bytes = 3 * width * height;
275 pixels = xmalloc(num_bytes + TRAILER_ALIGNMENT + sizeof(struct image_info));
```

Listing 3: Integer Overflow: Wrong Type for num_bytes

```
248 int num_bytes, is_ok;    // should probably be long or uint64_t
249 long width, height;
```

```
275 num_bytes = 3 * width * height;
```

Listing 4: Format String Injection Vulnerability on Line 2621

```
2612 void print_log_msg(struct image_info *info) {
2613   struct tm tm_parts;
2614   char time_str[80];
2615   if (info->create_time != -1) {
2616     localtime_r(&info->create_time, &tm_parts);
2617     strftime(time_str, 80, "%a, %d %b %Y %T %z", &tm_parts);
2618   } else {
2619     strcpy(time_str, "recently");
2620   }
2621   printf(logging_fmt, info->width, info->height, time_str, info->create_time);
2622   fputc('\n', stdout);
2623 }
```

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
Open Settings 42 43 46 4C C3 84 54 0A 00 00 00 00 00 00 0D 03  B C F L . . T . . . . . . . . .
00000010  00 00 00 00 00 00 00 03 00 00 00 00 00 00 00 02  . . . . . . . . . . . . . . . .
00000020  54 49 4D 45 00 00 00 00 00 00 00 08 00 00 00 00  T I M E . . . . . . . . . . . .
00000030  00 41 6B D0 46 52 4D 54 00 00 00 00 00 40 40 32  . A k . F R M T . . . . . @ @ 2
00000040  25 30 31 36 6C 78 25 30 31 36 6C 78 25 30 31 36  % 0 1 6 l x % 0 1 6 l x % 0 1 6
00000050  6C 78 41 41 41 41 41 41 41 41 41 41 41 41 41 41  l x A A A A A A A A A A A A A A
00000060  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  A A A A A A A A A A A A A A A A
00000070  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  A A A A A A A A A A A A A A A A
00000080  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  A A A A A A A A A A A A A A A A
00404050  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  A A A A A A A A A A A A A A A A
00404060  41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41  A A A A A A A A A A A A A A A A
00404070  25 6E 44 41 54 41 00 DF D6 20 00 DF D6 20 2B EF  % n D A T A . . .   . . .   + .
00404080  2F 60 00 2B EF 2F 60 00 7F EE B7 20 7F EE B7 20  / ` . + . / ` . . . .   . . .
```

enough chars to have printed 0x40404e chars

Figure 4: Format String Attack Binary

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000  00 42 43 52 C3 84 57 0A 00 00 00 00 00 00 00 08  . B C R . . W . . . . . . . . .
00000010  30 30 30 30 30 00 00 00 30 30 30 30 30 30 30 30  0 0 0 0 0 . . . 0 0 0 0 0 0 0 0
00000020  44 41 54 41 00 00 00 00 00 00 00 00 30 30 30 00  D A T A . . . . . . . . 0 0 0 .
00000030  00 00 00 00 4E 40 40 00 00 00 00 00             . . . . N @ @ . . . . .
```

Figure 5: Overlapping Memory Allocation Attack Binary