Project 1: Badly Coded Image Viewer

Hannah Burns - burn0460@umn.edu

Spring 2023

1 Threat Modeling

Badly Coded Image Viewer is currently a limited program intended to operate on a single user's Linux environment, that requires the x86-64 architecture and the GTK 3 toolkit for GUI functionality. It only processes three specific bitmap formats, all of which are Badly Coded formats: BC-Raw (.bcraw), BC-Progressive (.bcprog), and BC-Flat (.bcflat). Although this limits compatibility and versatility, it standardizes input types and allows developers to focus their development and security efforts.

The program relies on command line input from the user, including an optional tag for batch conversion mode and the image file location. The data flow between the user, bcimgview.c file, and the GTK 3 toolkit is illustrated in the data flow diagram 1, with the image file name provided by the user as the first exchange of data. There exists an important trust relationship and boundary here as the bcimgview.c file trusts that, for the most part, the data inside the provided file is accurate as it is produced by another program of theirs. However, this does not account for the possible manipulation of the file from the user or the potential for corruptions due to bugs in the file producing programs.

The data provided by the user is parsed, read, and decompressed. The internal data will either be passed into the external GTK 3 library, which supplies a widget to bcimgview.c for displaying the GUI to the user, or be written to a new file if the program is in batch conversion mode. In both cases, the correctness of the internal data relies on the functions inside of bcimgview.c functions and the correctness of the supplied image binary file.

1.1 Parsing Image Data

Regardless of the mode, user data first flows into the parse_image function. This is where the file is opened and the first 8 bytes are read to determine which of the three file types the file is, or failing if neither. There is trust that the magic number and pixel data are compatible. The opened file header will then be given as an argument to the file corresponding helper functions: parse_bcraw, parse_bcprog, and parse_bcflat.

The parse functions use read_u64_bigendian to read the image width and height from the file as unsigned 64-bit integers, ensuring compatibility of multi-byte data types. The program reads the dimensions of the file. The trust of the dimensions of the image can lead to major issues, especially since there is room for a user to manipulate the metadata. The pixel buffer is allocated based on these dimensions.

The footer, called 'info_footer', holds the information of the image metadata in the struct type image_info. The data is populated with the dimensions from the file and a pointer to the pixels buffer, as well as the created time and the cleanup, intended to be a destructor callback, is set to 0 or a null pointer. The inclusion of a callback function pointer in the struct should be handled with caution. This footer is used to call process_tagged_data and the respective read function. The parse helper function trusts these functions to preserve the struct and will copy the values from it to the dynamic info struct that will be returned to main. The entire program relies on the accuracy of this info struct.

1.2 Processing Tagged Data

All three helper functions for parsing the image data call the function process_tagged_data with the arguments of the file header and a pointer to the image_info struct. The function's intention is to recognize and assign the optional file tags; "TIME" flag reads specified time from the file to the create_time variable associated with the current struct and the "FRMT" tag signals a unique formatting for printing the file information. In this instance, a character buffer is created to read in the new string formatting and assigns it to the global variable logging_fmt which will be used later in print_log_msg. The program trusts the user supplies data for both instances. The program trusts that the supplied time is a correct time and that the string format is safe. Especially for the instance of the user supplied string format, this is a very risky trust relationship.

1.3 Internal Log Message Print

The effects of this trust relationship are shown in print_log_msg which is called for every image with the intention of printing an informative log message. The local char buffer time_str holds the time stamp. The printf call takes the global pointer logging_fmt as the format and the width and height saved in the image info struct, the time_str buffer of size 80 and the create_time struct variable. At first glance, the program only relies on trusting the external data of the image_info struct 'info' as an argument. Since struct is populated with user data and the active variables are assigned directly from the user file with minimal input size sanitation, auditing is necessary. While the created time can be manipulated by a tag, so can the string format logging_fmt. Though this variable is not a function argument, it is still another entry point of external data.

2 Code Audit

2.1 printf vulnerability

In the auditing process for programs written in C, or its derivatives, it is important to look at all printf statements. The statement:

```
printf(logging_fmt, info->width, info->height, time_str, info->create_time);
```

in the internal function *print_log_msg* seems safe at first glace. The format string for this print function, logging _fmt, is a global constant char variable with the original value of **"Displaying image of width %ld and height %ld" " from %s"**. The default value has three specifiers, meaning info− >create_time is not intended to be printed. Format strings should always match with the number and type of provided arguments.

This problem could be benign except that logging_fmt can also be an externally defined format string variable. Additionally, the extra variable, info− >create_time, can be user defined. From the threat modeling, we are shown the function process_tagged_data() reads the users-provided file for "TIME' in hexadecimal. This signals a user-defined time and assigns it to info− >create_time. Similarly, the FRMT tag signals a user-defined string format that will be read in and reassign the value of logging_fmt. The only check the system currently has on the tags is if they're the appropriate length.

Due to the lack of control to the printf format string, this vulnerability can cause unwanted reading and writing to the program file. What makes format string vulnerabilities particularly dangerous in C, and its derivatives, is the ability to write to the program with the printf specifiers of '%n' and '%hn'. These store the number of characters written so far to a pointer argument. These specifiers and write privileges associated are dangerous for their ability to target and overwrite the values of return addresses or function pointers that would cause an execution.

Looking at the architecture of the main function, there happens to be a multiple function pointers, including **per_image_callback* which executes a benign and useless counter. Additionally, every *image_info* struct contains a function pointer to a *cleanup* function that is called and executed after every process when the image is freed. These variables are exceptionally useful to an attack where an address to malicious code can be overwritten, stored, and later executed when the info struct is freed.

Taking control over a callback function pointer could be possible by exploiting the vulnerability in the printf function's user-defined format string and overwriting the address with malicious shellcode.

2.2 Buffer Overflow

The programming mistake made consistently in the bcimgview.c is the lack of input sanitation. These issues are particularly dangerous when the user-provided input is responsible for buffer sizes and memory usage. In general when auditing for memory safe vulnerabilities, it is important to test and ensure each instance of dynamically allocated memory, *xmalloc in this program.* Inappropriate sizes and carelessness can lead to buffer overflows, use-after-free vulnerabilities and even memory leaks.

In particular, the *parse_bcprog* function, the height and width of the image are read from the header of the file and checked for *if (height* $< 2||width < 1$*).* The intention is to ensure that the picture is large enough to meet all of the conditional statements in the progressive algorithm in the *read_prog_data* function. Since the read function decodes the progressive row ordering of image data in three passes. The mistake is that if height of the image is 2, then the second loop, where *row = 2*, will *do* the read of the file, and then the *while* conditional will check if the row is less than the height, and in this case it is not.

The other general issue is that the height and width of the file are user-controlled and can be completely wrong. If the values are incorrect, the *pixels* buffer will be malloced incorrectly as it depends on $num\_bytes = 3 * width * height$. In the case where this is coupled with the height being too small and of size 2, there will be an overflow of the *pixels* buffer. This doesn't just affect the size of the buffer but the placement of the footer struct, *info_footer*, of image data.

Even without malicious intent this will cause an overwrite of memory and the program will crash. This attack became obvious when the height and width of the program changed to incredibly large numbers, continued the loop, and failed when *row_start* became inaccessible. This occurs every time height and row are the same since the pointer to the buffer position to store the read pixels is determined by $*row\_start = p + row * 3 * info$-¿$width$; which, mirroring *num_bytes*, is equivalent to *pixels + pixels = info_footer placement.* The overwrite will begin exactly on the struct, hence the importance of the exception of two.

If the program can overwrite the height and width and pixels, it could also overwrite the *cleanup* variable in the info struct and be executed when the info struct is freed.

3 Attacks

3.1 Format String Attack in *print_log_msg*

This attack makes use of user-defined variables following the TIME and FRMT tags to exploit a format string vulnerability in a printf function found in print_log_msg. The first step in a format string attack is to form a basic malicious format by including more integer specifiers than arguments to expose elements on the stack. Doing so already produces an information disclosure security threat.

For example, in the example file 2, the malicious format string, which follows the FRMT tag in the binary file, includes eight **%x%x** format specifiers. This prints four additional variables, as hexadecimal, that were not intended to be read by the user. This example is shown in the output 3. This can and was exploited further. The exposed information was arbitrary to a further hack and much of it consisted of the time_str variable as seen in the figure 4 and the output 3.

At first glance this doesn't seem to be exploitable but what was exposed for possible malicious writing to the program was the info− >create_time. This can be seen in 2, where the TIME tag defines the created time to be ‘AAAA‘ and the hexadecimal representation, ‘41414141‘, is exposed as the fourth printed value. Though it is not inherently a pointer, the time_t type variable stores a long integer. Changing the time to ‘AAA‘ and the fourth specifier to ‘%n‘ will store ‘0x11‘ in the address ‘0x414141‘ in the previous example.

What is next for a successful and applicable attack is to take control of the program. The shellcode target, shellcode_target(), provided in the source code exists at the address 0x403f9c on provided machines when utilizing the ‘-no-pie‘ flag to maintain fixed memory addresses. This provided function address can be assigned to info− >create_time in the binary file but is of no use as it is never executed. Instead, the vulnerability needs to take control of a callback function or return address. Conveniently, there exists a callback function at the end of every call to an image process, whether in main or display_image().

Though this callback function is not inherently user-defined, it can still be used-exploited by assigning the value of the user-defined info− >create_time variable. The functions address, in this example it is 0x40b288, just needs to follow the TIME tag in the user-provided file. The intention now is to utilize the format string vulnerability and ‘%n‘ specifier to overwrite the pointer argument that stores a pointer argument, with the pointer to the shellcode function. Since ‘%n‘ will write the number of characters already printed, the format string just needs to ensure that the decimal representation of the shellcode, 0x403f9c = 4210588, is printed before. This process relies on provided padding.

This was achievable by printing the first two arguments of type long as the full width of 16 using ‘%16x%16x‘. The rest of the padding, being 32 less now, was absorbed by the 3rd argument and formatted to be ‘%4210556x‘. Finally, ‘%n‘ is added to the end to assign the shellcode address to the callback function stored in info− >create_time. This can be done several ways as long as the overwrite is equivalent to the address of the shellcode and the ‘%n‘ is correctly included after.

The binary file used for this attack is found in in figure 5, where the callback function pointer is highlighted in yellow after the TIME tag and the new format string is located after the FRMT tag and highlighted in blue, the special ‘%n‘ is in pink.

All together when this malicious image file is parsed into bcimgview.c, the info− >create_time is assigned to 0x40b288- or the callback function, shown in figure 6 - and the logging_fmt variable is reassigned to the string '%16x%16x%4210556x%n' in the function process_tagged_data(). This becomes useful in the function print_log_msg where the vulnerable printf function will print 0x403f9c characters and assign that number to the pointer the callback function will point to. Afterwards when '(*per_image_callback)();' calls the callback, the program's execution will now jump to shellcode_target() and the malicious file will have gained control over the program. This can be seen in figure 7.

In conclusion, the non-standardized print format in print_log_msg leads to the vulnerability. It was also the lack of input sanitization of tagged data and the unclear control flow from a poorly placed callback function that allowed for the complete control-flow hijack.

3.2 info Overwrite in bcprog Read

This attack takes advantage of the poor sanitation of userprovided data, specifically the height and width of an image file in the functions *parse_bcprog* and executed in *read_prog_data*. Without malicious intent, this would be nearly impossible as the overflow would just cause the overwrite that would crash the program, as mentioned before. It is necessary then to control how many bytes are to be overwritten, which can be controlled by the user by just changing the width.

To make it appear as nothing is different but the cleanup pointer, the height and width must be overwritten with their same values.

To only overwrite the cleanup pointer, width will be assigned to 24, the size of the width, height, and cleanup variables as they appear first in the struct. The first pass will be irrelevant and any 24 byes will be placed as padding. The same 8 bytes of the width then height follow and are highlighted yellow in figure 8. After, is where an address for a malicious function can be stored. In this example, the provided function *shellcode_target* is used and remains at *0x403f9c*. The reversed order in the binary file will appear as *9c 3f 40* and can be seen in figure 8 highlighted in pink. When the data is read from the file with *fread* and the overwrite can be seen in figure 9.

The function will return back to *parse_bcprog* as normal. Although the *footer_info* struct is not to be of any significance outside these functions and will also never call the cleanup function, the *image_info struct info* is malloced onto the heap right after the read function. Without receiving a fail code, the parse function will directly assign all of the values from *footer_info* to *info*, including the manipulated *cleanup* pointer. The *info* struct will be returned back to main, used for the remainder of the program. Near the end, *free_image_info(info)* will be called to free the struct and the cleanup callback function will execute the malicious source code. This final execution can be shown in figure

10.

In summary, the control flow hijacking was made possible by the lack of sanitation of user-provided input, specifically having the appropriate height and width. This was furthered by a mistake in the bounds checking for the progressive algorithm. The incorrect malloc size therefore allowed for a buffer overflow and an memory address overwrite to a callback function that was later executed.

4 Figures

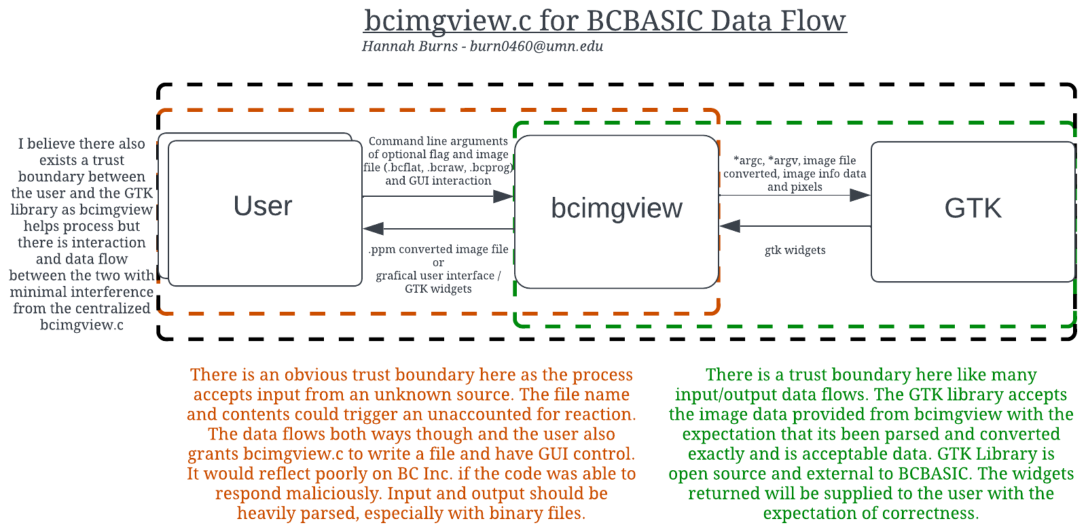
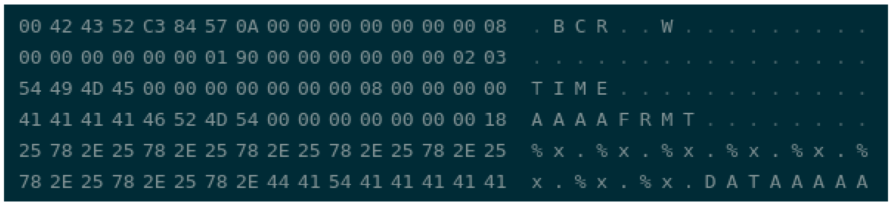
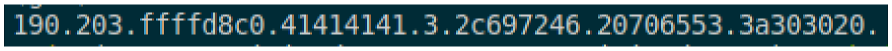

bcimgview.c for BCBASIC Data Flow
Hannah Burns - burn0460@umn.edu
I believe there also exists a trust boundary between the user and the GTK library as bcimgview helps process but there is interaction and data flow between the two with minimal interference from the centralized bcimgview.c
User
Command line arguments of optional flag and image file (.bcflat, .bcraw, .bcprog) and GUI interaction
.ppm converted image file or grafical user interface / GTK widgets
bcimgview
*argc, *argv, image file converted, image info data and pixels
gtk widgets
GTK
There is an obvious trust boundary here as the process accepts input from an unknown source. The file name and contents could trigger an unaccounted for reaction. The data flows both ways though and the user also grants bcimgview.c to write a file and have GUI control. It would reflect poorly on BC Inc. if the code was able to respond maliciously. Input and output should be heavily parsed, especially with binary files.
There is a trust boundary here like many input/output data flows. The GTK library accepts the image data provided from bcimgview with the expectation that its been parsed and converted exactly and is acceptable data. GTK Library is open source and external to BCBASIC. The widgets returned will be supplied to the user with the expectation of correctness.

Figure 1: The data flow diagram for the overview of the program architecture.



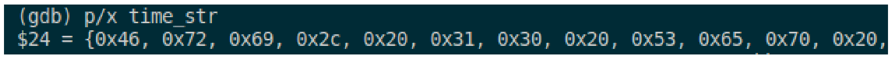
```
00 42 43 52 C3 84 57 0A 00 00 00 00 00 00 00 08   . B C R . . W . . . . . . . . .
00 00 00 00 00 00 01 90 00 00 00 00 00 00 02 03   . . . . . . . . . . . . . . . .
54 49 4D 45 00 00 00 00 00 00 00 08 00 00 00 00   T I M E . . . . . . . . . . . .
41 41 41 41 46 52 4D 54 00 00 00 00 00 00 00 18   A A A A F R M T . . . . . . . .
25 78 2E 25 78 2E 25 78 2E 25 78 2E 25 78 2E 25   % x . % x . % x . % x . % x . %
78 2E 25 78 2E 25 78 2E 44 41 54 41 41 41 41 41   x . % x . % x . D A T A A A A A
```

Figure 2: This is the binary file used for the stack dump in print_log_msg.



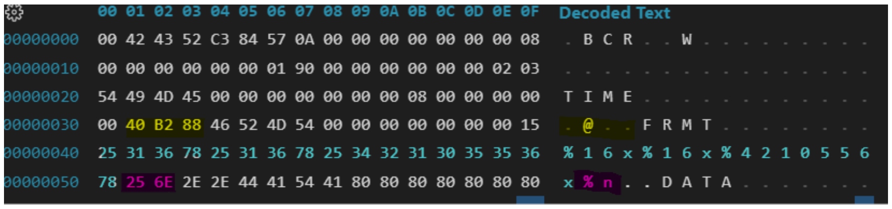
```
190.203.ffffd8c0.41414141.3.2c697246.20706553.3a303020.
```

Figure 3: This is the output of the 8 variable stack dump in print_log_msg.



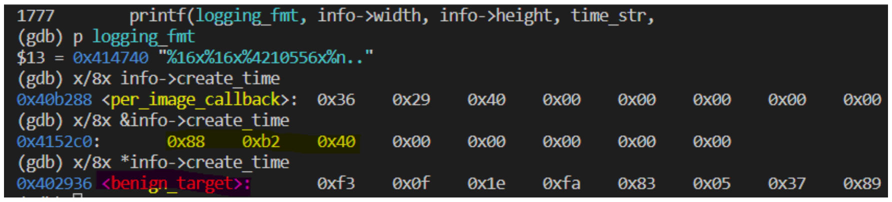
```
(gdb) p/x time_str
$24 = {0x46, 0x72, 0x69, 0x2c, 0x20, 0x31, 0x30, 0x20, 0x53, 0x65, 0x70, 0x20,
```

Figure 4: This is the time_str in hexadecimal in print_log_msg.



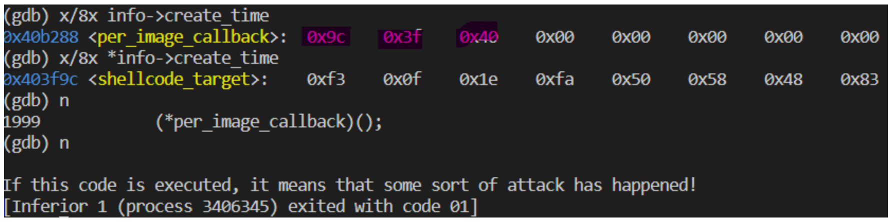
```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded Text
00000000  00 42 43 52 C3 84 57 0A 00 00 00 00 00 00 00 08  . B C R . . W . . . . . . . . .
00000010  00 00 00 00 00 00 01 90 00 00 00 00 00 00 02 03  . . . . . . . . . . . . . . . .
00000020  54 49 4D 45 00 00 00 00 00 00 00 08 00 00 00 00  T I M E . . . . . . . . . . . .
00000030  00 40 B2 88 46 52 4D 54 00 00 00 00 00 00 00 15  . @ . . F R M T . . . . . . . .
00000040  25 31 36 78 25 31 36 78 25 34 32 31 30 35 35 36  % 1 6 x % 1 6 x % 4 2 1 0 5 5 6
00000050  78 25 6E 2E 2E 44 41 54 41 80 80 80 80 80 80 80  x % n . . D A T A . . . . . . .
```

Figure 5: This is the binary file used for the printf attack.



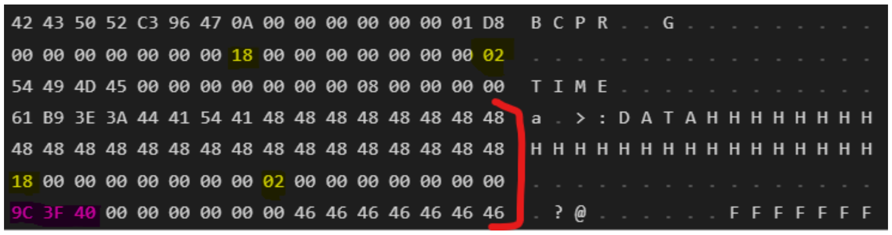
```
1777        printf(logging_fmt, info->width, info->height, time_str,
(gdb) p logging_fmt
$13 = 0x414740 "%16x%16x%4210556x%n.."
(gdb) x/8x info->create_time
0x40b288 <per_image_callback>:  0x36    0x29    0x40    0x00    0x00    0x00    0x00    0x00
(gdb) x/8x &info->create_time
0x4152c0:       0x88    0xb2    0x40    0x00    0x00    0x00    0x00    0x00
(gdb) x/8x *info->create_time
0x402936 <benign_target>:       0xf3    0x0f    0x1e    0xfa    0x83    0x05    0x37    0x89
```

Figure 6: The address to the callback function in created_time.



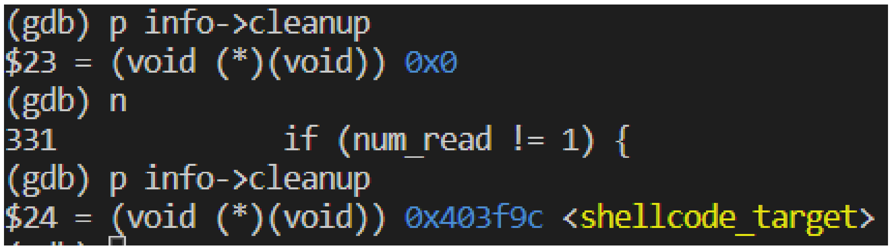
```
(gdb) x/8x info->create_time
0x40b288 <per_image_callback>:  0x9c    0x3f    0x40    0x00    0x00    0x00    0x00    0x00
(gdb) x/8x *info->create_time
0x403f9c <shellcode_target>:    0xf3    0x0f    0x1e    0xfa    0x50    0x58    0x48    0x83
(gdb) n
1999            (*per_image_callback)();
(gdb) n

If this code is executed, it means that some sort of attack has happened!
[Inferior 1 (process 3406345) exited with code 01]
```

Figure 7: The finished attack with the shellcode address in the callback function.



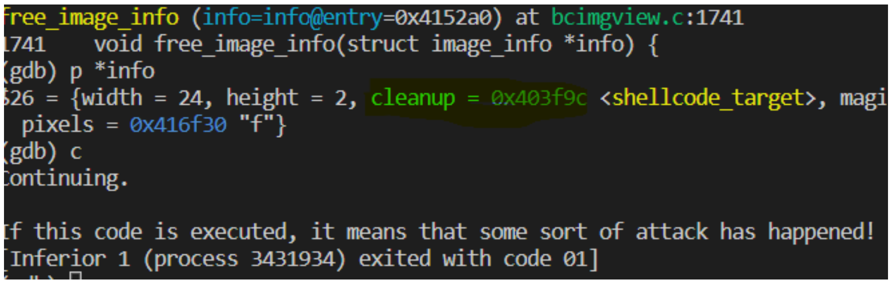
```
42 43 50 52 C3 96 47 0A 00 00 00 00 00 00 01 D8   B C P R . . G . . . . . . . . .
00 00 00 00 00 00 00 18 00 00 00 00 00 00 00 02   . . . . . . . . . . . . . . . .
54 49 4D 45 00 00 00 00 00 00 00 08 00 00 00 00   T I M E . . . . . . . . . . . .
61 B9 3E 3A 44 41 54 41 48 48 48 48 48 48 48 48   a . > : D A T A H H H H H H H H
48 48 48 48 48 48 48 48 48 48 48 48 48 48 48 48   H H H H H H H H H H H H H H H H
18 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
9C 3F 40 00 00 00 00 00 00 46 46 46 46 46 46 46   . ? @ . . . . . . F F F F F F F
```

Figure 8: This is the binary file used for the bcprog overwrite attack.



```
(gdb) p info->cleanup
$23 = (void (*)(void)) 0x0
(gdb) n
331             if (num_read != 1) {
(gdb) p info->cleanup
$24 = (void (*)(void)) 0x403f9c <shellcode_target>
```

Figure 9: This is the execution of the cleanup overwrite.



```
ree_image_info (info=info@entry=0x4152a0) at bcimgview.c:1741
741     void free_image_info(struct image_info *info) {
gdb) p *info
26 = {width = 24, height = 2, cleanup = 0x403f9c <shellcode_target>, magi
  pixels = 0x416f30 "f"}
gdb) c
ontinuing.

f this code is executed, it means that some sort of attack has happened!
Inferior 1 (process 3431934) exited with code 01]
```

Figure 10: This is the successful attack in the parse_bcprog/read_prog_data.