

Audit Process

I started by reading the code and looking for a function pointer; those are often vulnerable to executing code unexpectedly. I found a function pointer in the info struct used for cleanup, and this became the target for my attacks. I then started running example files through the program in order to follow the path using gdb. I started with a bcflat and while doing this I realized that there was no checking if tags occurred multiple times so I looked into what that might do. However, this search yielded nothing I could exploit to execute unintended code.

Following the code using the debugger, there was a printf statement where everything that it printed out was controlled by the user. This was clearly an example of a format string vulnerability. By exploiting this vulnerability it is possible to do a write what where attack where any data can be written anywhere in memory. Using a format string attack it is possible to overwrite a jump instruction in order to jump to some unintended bit of code.

I then ran the program through afl and looked at where errors occurred and saw that the bcraw files would sometimes completely overwrite what was written in the info struct. Following the code through the debugger I saw that there was no limit to the size of width or height in the parse_bcraw function but that both values were multiplied together. This was clear evidence of an integer overflow attack. By exploiting this, it is possible to allocate a small amount of data and read beyond the expected boundaries, thereby overwriting the info struct.

I ran afl again and found a crash in the bcprog and found that in one of the functions a pointer became a random value. Investigating further I found that the program was reading data into the wrong position and by increasing width it would increase the amount of data written into the wrong position. Using this I found a vulnerability to exploit that will run any function given by the user.

Format String Attack

Giving the user access to the format string in `printf` is unsafe; if it is possible to control the format string, it is possible to read information from or write to a specific spot in memory. The theory behind this attack is that when `%n` is used in a format string, `printf` will save the number of characters printed to the subsequent pointer. If the number of characters printed was equal to the value of a function pointer and the user had been given access to the subsequent values for the specifier it is possible to put the value of a function pointer virtually anywhere.

A `printf` function can be used to read data from the stack. If `%x` is used in a `printf` statement it will read 4 bytes of hex data and then skip 4. The `printf` will then take this data from the stack and print it to the output stream. This will move the stack pointer to the next spot on the stack and if this is repeated enough it could be possible to find where a variable the user has control of exists, or even where format string is stored. If `%n` is used the `printf` function will write the number of bytes printed to the location pointed to by the next value on the stack. By using enough `%x` to get to a spot on the stack the user controls it is possible to write directly to a spot in memory. This can be used to overwrite what a standard library function jumps to, or in the case of `bcimgview` the function pointer stored in the `info` struct.

There is one spot in the code that has a `printf` where the format string can be controlled and it is always run with no safety checks. `Print_log_message` is always called during normal execution of the program and it uses information from the file to print out a log message. The line in question is, `printf(logging_fmt, info->width, info->height, time_str, info->create_time);`. All of these variables come from the file: `width` and `height` are just read, `time` has its own keyword, and logging format is set by the `FRMT` tag. Since there are 4 variables and the user is given control of all of them, one of them can be used as a pointer to the cleanup function. In this example, a `bcflat` is used which has bounds checking on `width` and `height`. This has input checking, by using the section with input checking it is proof that this attack could be reproduced on any file format as they all have the same or less checks. If you were using a `bcraw` file, `width` or `height` could be used as a pointer to the cleanup function instead.

To create such a file, it is essential to generate a functional file that the program will accept. This can be accomplished by taking an existing file, modifying the time value, and adding a FRMT tag to meet the desired specifications. If the file were to be created from scratch, setting the file tag to define the file type would be crucial. The actual data within the file can be arbitrary, but to minimize file size, the height and width can be set to 1. The program reads tagged data, so the inclusion of a FRMT tag followed by the hex value for 16 ensures the entire string is read in, followed by `%4204449x%x%s%n` to configure the format string. The DATA tag signifies the end of tagged data, and the inclusion of random data ensures the file is read and accepted, resulting in a functioning attack.

Integer overflow

To exploit an integer overflow in the program, it's crucial to understand that integers have a finite number of bytes to store values. When addition or multiplication causes the integer to exceed its representable size, it overflows and the result will wrap around to the lowest possible value for that integer type, effectively starting over from zero. The result of this is that only the lower bytes of the operation are read and the given result is smaller than the actual result of a computation. Although this alone might not be dangerous, an integer overflow can disrupt other functions in the program, especially if they rely on the overflowed result, such as bounds checking. In this program, the size of memory allocated to pixels is determined by `width * height * 3`. Overflowing this value results in a smaller allocation for pixel data. Given that the width is substantial, the program will read more bytes than allocated, potentially overwriting subsequent data in memory.

The program checks the size of pixels and then allocates the resulting value as the number of pixels, if this value overflows, the program will still read the height and width of pixels but only allocate space for part of that. The function reading data from `bcrw` does so in 24-byte chunks, ensuring that the program won't read or write non-allocated data unless it's the attacker's goal. When reading the program will overwrite the `info` struct that contains data about the amount of allocated data and also the cleanup

function pointer. Using this to overwrite the height, width, and function pointer it is possible to read an acceptable amount of data and set the function pointer to the desired attack function. Then execution will continue as normal until the program hits the cleanup function which will activate the given attack function that info was overwritten with.

Creating a file to execute this attack involves starting with a bcrow header of 16 bytes, beginning with the magic number. Subsequently, the correct flags are set in the following bytes as expected by the program. The next two sets of 8 bytes represent width and height, these should be set to the values 0x1200000000000003 for width and 2 for the height; this causes an overflow when they are both multiplied together by 3 resulting in a value of 18. Following this, there are 24 bytes of padding which can be arbitrary. The purpose is to provide sufficient space so that the data is written into the info section. The next 24 bytes write into width, height, and cleanup within info, requiring two sets of 8 bytes, both equal to 1. Finally, the next 8 bytes represent the cleanup function pointer, set to the value of the attack function.

Buffer overflow

While parsing a bcprog file, the code checks if the height is not less than 2; this specific condition allows for the success of the attack. The read_prog_data function reads files in a pattern, first multiples of 4, then odd multiples of two and then odd lines. The multiples of 4 loop starts at 0, the first pass of reading doesn't do anything dangerous so 24 characters of padding should be added. Then multiples of two start at row 2 which is beyond the length of the allocated bytes.

The process used to read in data takes a pointer to pixels, this is allocated before the info struct, and then the program reads in data to row * 3 * width. Since the value of row is 2, this function reads in data to 2 * 3 * width bytes from the start of pixels. The problem occurs because the amount of allocated pixels was height * 3 * width. Since 2 was used for height, the program will reach the end of the allocated pixels data and read data into the next bytes, which happens to be the info struct.

Now that there is a way to read data into the struct, there needs to be enough data to read into the info struct to overwrite the cleanup pointer. This can be done by setting width to 24, this is the exact amount of bytes to fully overwrite the height, width, and cleanup function pointer. Setting height and width to 1 each, and since the row number is bigger than the height read_prog_data breaks out of that loop. Following that, the program initiates the next loop, which reads one byte before exiting. The program writes some standard messages and then during the cleanup function it will call (*info->cleanup)() which will activate the desired attack function.

The file will start with an identifying tag, 8 bytes for bcprog, then set the flags for the bcprog, 6 bytes of 0s then a 1 and then a d8. After these tags comes two sets of 8 bytes to set width and height, these should have the values 24 for width and 2 for height. Next, the file needs the word data in order to signify the start of the data that will be read into the pixels.

The second pass will overwrite the info struct so setting width and height to small values is beneficial. This isn't strictly necessary, but by setting them both to 1, read_prog_data will only read one byte in the next pass, this helps the file stay small. Width and height are both 8 bytes, so from byte 0 -17 The width and height are set, and from byte 17-24 the cleanup function pointer is being set. The bytes from 17-24 should be a pointer to the desired attack function. Finally, since the height and width were set to 1, the program reads 1 more byte. In order to make a valid file add one more byte of data.

Concluding the Analysis

In conclusion, the extensive analysis of the bcimgview code has unveiled several critical vulnerabilities that demand immediate attention before the program can be considered secure for public use. The identified vulnerabilities include a format string vulnerability, integer overflow vulnerability, and buffer overflow vulnerability pose significant risks to the integrity and security of the system.

Addressing these issues is paramount to ensuring the program's robustness and safeguarding users from potential exploits. The format string vulnerability can be addressed by sanitizing the data read by the FRMT tag to parse only 3 %x tags and remove any %n tags used. This will stop reading from the stack

beyond what is expected and stop any writing of data using printf. The integer overflow vulnerability can be fixed by checking that the width and height values are not so big that the integer would overflow when multiplied. Finally, the buffer overflow can be fixed by introducing more stringent checking on height in bcprog in order to fix where the program reads from.

By looking at the supplied threat model it gives some idea of where the system is vulnerable to attack. The user input is checked by the sections of the code that parse the image and after the image is parsed very few checks are used to verify the integrity of the data. This is where most attacks will likely be activated. More time should be spent looking into possible vulnerabilities in the code before it is released as it likely isn't complete.

Fixes to bugs

In order to fix the Integer overflow I implemented a check in order to make sure that the $(width * height * 3)$ execution wouldn't overflow an integer. I did this by checking "*width* > 0 && *height* > 0 && (*width* > $INTMAX_MAX / height / 3 || height > INTMAX_MAX / width / 3$)". The first section makes sure that both height and width are greater than 0, both necessary for a real image to display correctly. The program then checks for a possible overflow. Since multiplication can be undone by dividing, taking the maximum value of an integer and dividing it by $3 * height$ and comparing it to width will allow checking if the next computation will overflow an integer. If width is greater than the result of this operation it means that $height * width * 3$ will be larger than $INTMAX_MAX$ which will cause an overflow.


In order to fix the buffer overflow from bcprog, I changed all the do while loops to while loops. The issue was caused by doing a computation before checking if the computation is valid. The program would read data to a spot beyond what was allocated for pixels and would overwrite the info struct. By using while loops instead the program will check before the computation.

In order to fix the format string attack, I made sure to parse the incoming format string to check for an attack. If the format string contains more than 4 format specifiers, determined by counting % symbols then we replace the format string with a default one. If the format string had a % followed by an


n anywhere not separated by a space it would replace the given format string with a default before running printf. By checking for a space it is possible to tell when a format specifier ends and thus if the format specifier contains an n. This will fix the issue of %hn or %hhn or just %n while still allowing there to be a format specifier followed by text that contains n somewhere in the body of text.

A final safety check, every time the program is about to read data from a file a check occurs to see if the amount of data is within the desired boundaries. This is done by checking if the amount read + the current spot the program is reading to is greater than pixels but less than info struct. This will make sure that the program is reading into the data allocated to pixels and not overwriting the info struct. This is not to be relied on as there might be other ways to overwrite data not using fread.

Diagrams

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded Text
00000000	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	01	00	00	00	00	00	00	00	01
00000020	54	49	4D	45	00	00	00	00	00	00	00	08	00	00	00	00	T I M E
00000030	00	40	B5	10	46	52	4D	54	00	00	00	00	00	00	00	0F	. @ . . . F R M T
00000040	25	34	32	30	34	34	34	39	78	25	78	25	73	25	6E	44	% 4 2 0 4 4 4 9 x % x % s % n D
00000050	41	54	41	00	DF	D6	20	00	DF	D6	20	2B	EF	2F	60	00	A T A + . / ` .
00000060	2B	EF	2F	60	00	7F	EE	B7	20	7F	EE	B7	20				+ . / `

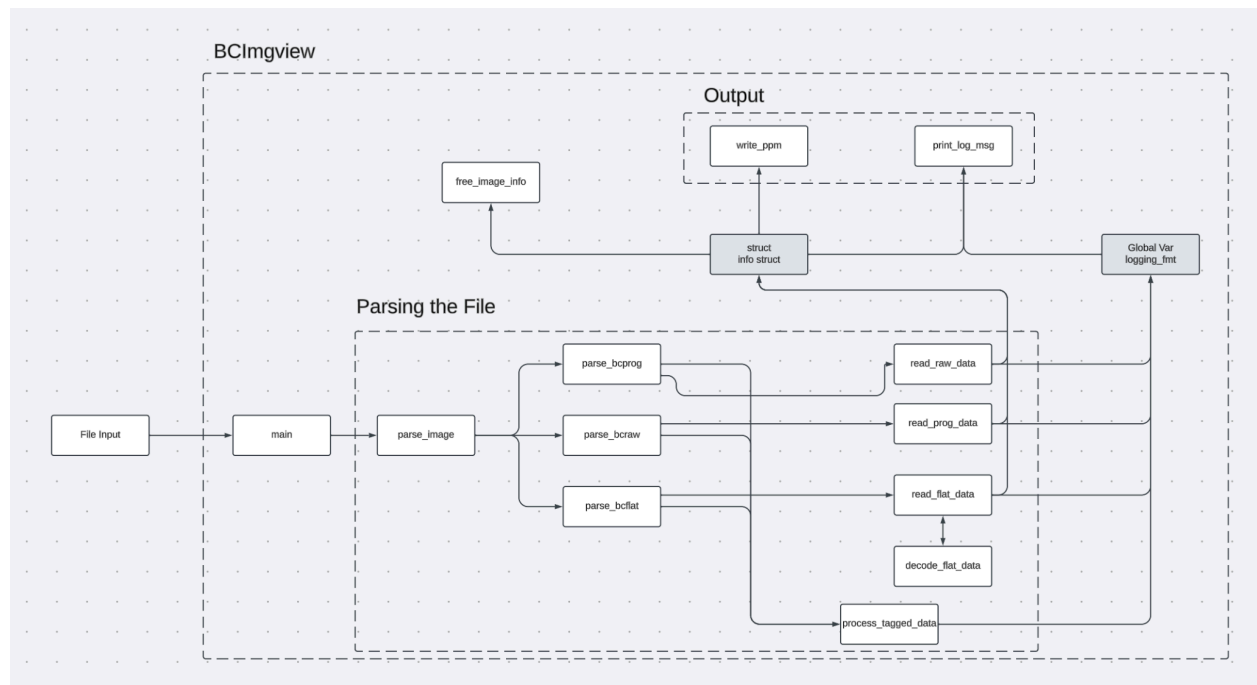
Format String

	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	12	00	00	00	00	00	00	03	00	00	00	00	00	00	00	02
00000020	44	41	54	41	20	70	61	64	64	69	6E	67	20	74	6F	20	D A T A p a d d i n g t o
00000030	67	65	74	20	74	6F	20	69	6E	66	6F	20	01	00	00	00	g e t t o i n f o
00000040	00	00	00	00	01	00	00	00	00	00	00	00	C1	27	40	00 ' @ .
00000050	00	00	00	00												

Int overflow

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	18	00	00	00	00	00	00	00	02
00000020	44	41	54	41	67	61	72	62	61	67	65	20	74	6F	20	72	D A T A g a r b a g e t o r
00000030	65	61	63	68	20	74	68	65	20	62	75	66	01	00	00	00	e a c h t h e b u f
00000040	00	00	00	00	01	00	00	00	00	00	00	00	00	C1	27	40 ' @ .
00000050	00	00	00	00	69											 i

Buffer overflow



Threat model

Tools used: spell check, chat gpt for essay checking, gdb

Note: if the headers are removed the essay is 7 pages, they are just used to make it easier to skim through