Vulnerability analysis of the Badly Coded BASIC interpreter

Benjamin Maymir (maymi001@umn.edu)

Initial Auditing

The badly coded image viewer has two primary functionalities. The first is the ability to visually display an image file in one of three badly coded file formats. The second is the ability to decompress and translate a badly coded compressed file into a more standard type. Since image displaying was done primarily via an external library, the primary focus of the audit was on the decompression algorithm. Similarly, the user can only interact with the GUI through a file selection interface and can only interact with the console through the initial command arguments, so the audit was focused on maliciously created image files. Some initial areas of interest was the codeword verification function, as it included a comment indicating that one of the tests had false positives, the logging function, as there was a compiler warning regarding variable format strings, and the benign_target as it appeared non functional and was very close in memory to the shellcode target.

Major Vulnerabilities

The audit resulted in two high priority vulnerabilities being discovered. The first was a result of creating a malicious logging message. The steps to reproduce are as follows. First, create a bcraw image file by setting the magic bits to match the format. Second, set the row length to 0. This will allow for an arbitrary number of rows without running out of data. Third, set the number of rows equal to 0x40B148 as this is the location of free in the procedure linkage table. This is the value that will be overwritten. Finally set the logging format string to %04210588ld%ln. This will print 4210588(the location of shellcode_target) 0s to the console, then write the value 4210588 to the location specified by the number of rows. When this image is displayed by the image viewer, after parsing the metadata it will have its control hijacked and execute the shell code target.

The second vulnerability discovered has to do with the way badly coded flat images are decompressed. Per the removed test, there is a single codeword that decompresses to more than 8 times its length. By chaining these together, it is possible to create more data than the uncomp_buff can hold and overflow into other important data stores. The steps to reproduce are as follows. First Create a bcflat image file by setting the magic bits to match the

format. Second, set the row length to 876. This value is 1(the first pixel) + 8 * 109(the offset of the uncompressed buffer from the return pointer) + 3 (three codewords are used to replace $rip). Next create a chain of 109 of the codeword 0011111. This will overflow the buffer and set q to write to the instruction pointer next. Lastly append the codewords 1110111011011, 1110111100010 and 1010 to write 403f9c, the address of shellcode_target, to $rip. When this file is decompressed, it will instead have the image viewer execute the shellcode target.

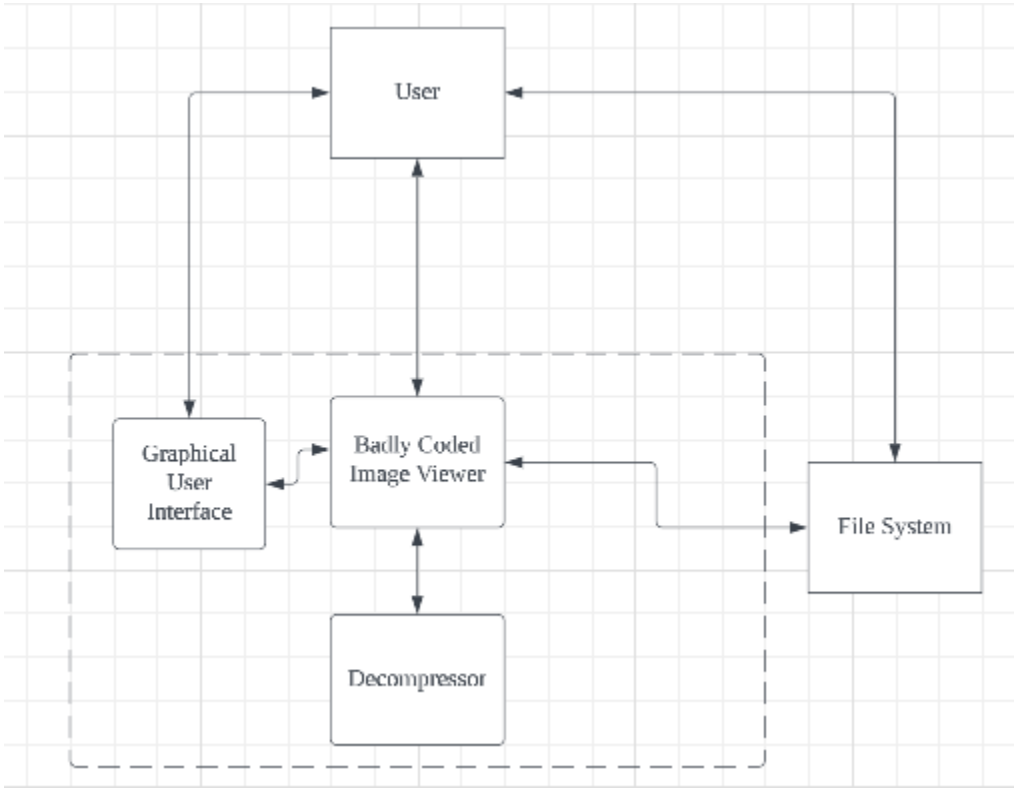
In both of these cases, the location of the control hijacking is arbitrary so instead of executing shellcode_target, an attacker could instead execute arbitrary code at a different location of their choice. This is particularly dangerous as viewing an image is not typically a dangerous activity for a user that is unfamiliar with technology.

Vulnerability Mitigation

The major vulnerabilities are a result of very small errors in the code. For the format string injection, prohibiting custom logs as part of image files would completely mitigate this attack vector. As long as unchecked user supplied format strings are printed, this vulnerability will continue to be a potential risk. Additionally, while this attack method would have worked without it, allowing raw images to have 0 pixel length is not a feature that is beneficial to a consumer and made creating the attack file significantly easier

For the decompression vulnerability, simply replacing 0x87 with 0x77 in the lookup table will prevent this particular buffer overflow. However, it is considered bad practice to ignore test results, which could have mitigated the issue before release.

Figure 1



The dataflow diagram shows the reasoning behind the auditing prioritization

Figure 2

00 42 43 52 C3 84 57 0A 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00 00 00 00 40 B1 48 54 49 4D 45 00 00 00 00 00 00 00 08 00 00 00 00 61 B8 E0 07 46 52 4D 54 00 00 00 00 00 00 00 18 25 30 34 32 31 30 35 38 38 6C 64 25 6C 6E 00 00 00 00 00 00 00 00 00 00 44 41 54 41 00

Hex representation of a bcraw file that hijacks control flow via its custom format string

Figure 3

42 43 46 4C C3 84 54 0A 00 00 00 00 00 00 0D 03 00 00 00 00 00 00 03 6C 00 00 00 00 00 00 00 01 44 41 54 41
00 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7
CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3
E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 E7 CF 9F 3E 7C F9 F3 FD DB EF 15 00

Hex representation of a file that hijacks control flow via the flat decompression algorithm