Talos Vulnerability Report

TALOS-2021-1402

Gerbv drill format T-code tool number out-of-bounds write vulnerability

NOVEMBER 4, 2021

CVE NUMBER

CVE-2021-40391

Summary

An out-of-bounds write vulnerability exists in the drill format T-code tool number functionality of Gerbv 2.7.0, dev (commit b5f1eacd), and the forked version of Gerbv (commit 71493260). A specially-crafted drill file can lead to code execution. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

Gerbv 2.7.0
Gerbv dev (commit b5f1eacd)
Gerbv forked dev (commit 71493260)

Product URLs

https://sourceforge.net/projects/gerbv/
https://github.com/gerbv/gerbv

CVSSv3 Score

10.0 - CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

CWE

CWE-390 - Detection of Error Condition Without Action

Details

Gerbv is an open-source software that allows to view RS-274X Gerber files, Excellon drill files and pick-n-place files. These file formats are used in industry to describe the layers of a printed circuit board and are a core part of the manufacturing process.

Some PCB (printed circuit board) manufacturers use software like Gerbv in their web interfaces as a tool to convert Gerber (or other supported) files into images. Users can upload gerber files to the manufacturer website, which are converted to an image to be displayed in the browser, so that users can verify that what has been uploaded matches their expectations. Gerbv can do such conversions using the `-x` switch (export). For this reason, we consider this software as reachable via network without user interaction or privilege requirements.

Gerbv uses the function `gerbv_open_image` to open files. In this advisory we're interested in the Excellon drill file-type.

```
int
gerbv_open_image(gerbv_project_t *gerbvProject, char *filename, int idx, int reload,
                gerbv_HID_Attribute *fattr, int n_fattr, gboolean forceLoadFile)
{
    ...
    dprintf("In open_image, about to try opening filename = %s\n", filename);

    fd = gerb_fopen(filename);
    if (fd == NULL) {
        GERB_COMPILE_ERROR(_("Trying to open \"%s\": %s"),
                        filename, strerror(errno));
        return -1;
    }
    ...
    } else if(drill_file_p(fd, &foundBinary)) {                   // [1]
        dprintf("Found drill file\n");
        if (!foundBinary || forceLoadFile)
            parsed_image = parse_drillfile(fd, attr_list, n_attr, reload);
```

A file is considered of type "drill" if the function `drill_file_p` [1] returns true. When true, the `parse_drillfile` is called to parse the input file. Let's first look at the requirements that we need to satisfy to have an input file be recognized as a "drill file":

```
gboolean
drill_file_p(gerb_file_t *fd, gboolean *returnFoundBinary)
{
    ...
    while (fgets(tbuf, MAXL, fd->fd) != NULL) {
        ...
        /* First look through the file for indications of its type */
        len = strlen(buf);
        /* check that file is not binary (non-printing chars) */        // [2]
        for (i = 0; i < len; i++) {
            ascii = (int) buf[i];
            if ((ascii > 128) || (ascii < 0)) {
                found_binary = TRUE;
            }
        }

        /* Check for M48 = start of drill header */
        if (g_strstr_len(buf, len, "M48")) {
            found_M48 = TRUE;
        }

        /* Check for M30 = end of drill program */
        if (g_strstr_len(buf, len, "M30")) {
            if (found_percent) {
                found_M30 = TRUE; /* Found M30 after % = good */
            }
        }

        /* Check for % on its own line at end of header */
        if ((letter = g_strstr_len(buf, len, "%")) != NULL) {
            if ((letter[1] ==  '\r') || (letter[1] ==  '\n'))
                found_percent = TRUE;
        }

        /* Check for T<number> */
        if ((letter = g_strstr_len(buf, len, "T")) != NULL) {
            if (!found_T && (found_X || found_Y)) {
                found_T = FALSE;  /* Found first T after X or Y */
            } else {
                if (isdigit( (int) letter[1])) { /* verify next char is digit */
                    found_T = TRUE;
                }
            }
        }

        /* look for X<number> or Y<number> */
        if ((letter = g_strstr_len(buf, len, "X")) != NULL) {
            ascii = (int) letter[1]; /* grab char after X */
            if ((ascii >= zero) && (ascii <= nine)) {
                found_X = TRUE;
            }
        }
        if ((letter = g_strstr_len(buf, len, "Y")) != NULL) {
            ascii = (int) letter[1]; /* grab char after Y */
            if ((ascii >= zero) && (ascii <= nine)) {
                found_Y = TRUE;
            }
        }
    } /* while (fgets(buf, MAXL, fd->fd) */

    ...

    /* Now form logical expression determining if this is a drill file */
    if ( ((found_X || found_Y) && found_T) &&                      // [3]
         (found_M48 || (found_percent && found_M30)) )
        return TRUE;
    else if (found_M48 && found_T && found_percent && found_M30)   // [4]
        /* Pathological case of drill file with valid header
           and EOF but no drill XY locations. */
        return TRUE;
    else
        return FALSE;
} /* drill_file_p */
```

For an input to be considered a “drill file” the file must first of all contain only printing characters [2]. Then, two layouts are allowed: [3] and [4]. An example of a minimal “drill” file is the following:

```
%
M30
T1
X0
```

Even though not important for the purposes of the vulnerability itself, note that the checks use `g_strstr_len`, so all those fields can be found anywhere in the file. For example, this file is also recognized as a “drill” file, even though it will fail later checks in the execution flow:

```
--%
,,,T1M30X0
```

After the “drill file” has been recognized, `parse_drillfile` is called:

```
gerbv_image_t *
parse_drillfile(gerb_file_t *fd, gerbv_HID_Attribute *attr_list, int n_attr, int reload)
{
    drill_state_t *state = NULL;
    gerbv_image_t *image = NULL;
    ...
    ssize_t file_line = 1;

    ...
    image = gerbv_create_image(image, "Excellon Drill File");
    ...
    while ((read = gerb_fgetc(fd)) != EOF) {

        switch ((char) read) {
        ...
        case 'T':
            drill_parse_T_code(fd, state, image, file_line);   // [5]
            break;
        ...
```

If a line starting with the letter “T” is found, the function `drill_parse_T_code` is called for parsing the so-called “T-code”.

```
/* -------------------------------------------------------------- */
/* Parse tool definition. This can get a bit tricky since it can
   appear in the header and/or data section.
   Returns tool number on success, -1 on error */
static int
drill_parse_T_code(gerb_file_t *fd, drill_state_t *state,
                        gerbv_image_t *image, ssize_t file_line)
{
    int tool_num;
    gboolean done = FALSE;
    int temp;
    double size;
    gerbv_drill_stats_t *stats = image->drill_stats;
    gerbv_aperture_t *apert;
    gchar *tmps;
    gchar *string;

    dprintf("---> entering %s()...\n", __FUNCTION__);

    ...

    tool_num = (int) gerb_fgetint(fd, NULL);        // [6]
    dprintf ("  Handling tool T%d at line %ld\n", tool_num, file_line);

    if (tool_num == 0)
        return tool_num; /* T00 is a command to unload the drill */

    if (tool_num < TOOL_MIN || tool_num >= TOOL_MAX) {
        gerbv_stats_printf(stats->error_list, GERBV_MESSAGE_ERROR, -1,  // [7]
                _("Out of bounds drill number %d "
                    "at line %ld in file \"%s\""),
                tool_num, file_line, fd->filename);
    }
```

First, the `tool_num`, which is the number after “T”, is extracted [6] using `gerb_fgetint`. For `gerb_fgetint` it suffices to say that it can return any `int` value. Then this number is checked to be within `TOOL_MIN` and `TOOL_MAX` (1-9998). If not, an error is printed [7], but execution continues nonetheless:

```
    /* Set the current tool to the correct one */
    state->current_tool = tool_num;

    /* Check for a size definition */
    temp = gerb_fgetc(fd);

    /* This bit of code looks for a tool definition by scanning for strings
     * of form TxxC, TxxF, TxxS.  */
    while (!done) {
        switch((char)temp) {
        case 'C':
            size = read_double(fd, state->header_number_format, GERBV_OMIT_ZEROS_TRAILING, state->decimals);  // [8]
            dprintf ("  Read a size of %g\n", size);

            if (state->unit == GERBV_UNIT_MM) {
                size /= 25.4;                // [9]
            } else if(size >= 4.0) {
                /* If the drill size is >= 4 inches, assume that this
                   must be wrong and that the units are mils.
                   The limit being 4 inches is because the smallest drill
                   I've ever seen used is 0,3mm(about 12mil). Half of that
                   seemed a bit too small a margin, so a third it is */

                gerbv_stats_printf(stats->error_list, GERBV_MESSAGE_ERROR, -1,
                        _("Read a drill of diameter %g inches "
                            "at line %ld in file \"%s\""),
                            size, file_line, fd->filename);
                gerbv_stats_printf(stats->error_list, GERBV_MESSAGE_WARNING, -1,
                        _("Assuming units are mils"));
                size /= 1000.0;              // [10]
            }

            if (size <= 0. || size >= 10000.) {
                gerbv_stats_printf(stats->error_list, GERBV_MESSAGE_ERROR, -1,
                        _("Unreasonable drill size %g found for drill %d "
                            "at line %ld in file \"%s\""),
                            size, tool_num, file_line, fd->filename);
            } else {
                apert = image->aperture[tool_num];    // [11]
                if (apert != NULL) {
                    /* allow a redefine of a tool only if the new definition is exactly the same.
                     * This avoid lots of spurious complaints with the output of some cad
                     * tools while keeping complaints if there is a true problem
                     */
                    if (apert->parameter[0] != size
                    ||  apert->type != GERBV_APTYPE_CIRCLE
                    ||  apert->nuf_parameters != 1
                    ||  apert->unit != GERBV_UNIT_INCH) {

                        gerbv_stats_printf(stats->error_list,
                                GERBV_MESSAGE_ERROR, -1,
                                _("Found redefinition of drill %d "
                                "at line %ld in file \"%s\""),
                                tool_num, file_line, fd->filename);
                    }
                } else {
                    apert = image->aperture[tool_num] =
                                                g_new0(gerbv_aperture_t, 1);    // [12]
                    if (apert == NULL)
                        GERB_FATAL_ERROR("malloc tool failed in %s()",
                                        __FUNCTION__);

                    /* There's really no way of knowing what unit the tools
                       are defined in without sneaking a peek in the rest of
                       the file first. That's done in drill_guess_format() */
                    apert->parameter[0] = size;                                 // [13]
                    apert->type = GERBV_APTYPE_CIRCLE;
                    apert->nuf_parameters = 1;
                    apert->unit = GERBV_UNIT_INCH;
                }
            }
```

If the character after the T-code is “C”, we’ll land at [8], where a double is read and stored in `size`. This size is expected to be found after the “C” character as this is the drill diameter. At [9] and [10] this floating-point number is divided by either 25.4 or 1000, depending on the unit. If the number is within 1 and 9999, we continue at [11] where the `tool_num` variable is used to index the `image->aperture` array. This array contains only 9999 elements (`APERTURE_MAX`) and `tool_num` is unrestricted, hence this read operation goes out of bounds, and the result is stored in `apert`.

For reference, this is the definition of the `image` structure:

```
typedef struct {
  gerbv_layertype_t layertype; /*!< the type of layer (RS274X, drill, or pick-and-place) */
  gerbv_aperture_t *aperture[APERTURE_MAX]; /*!< an array with all apertures used */
  gerbv_layer_t *layers; /*!< an array of all RS274X layers used (only used in RS274X types) */
  gerbv_netstate_t *states; /*!< an array of all RS274X states used (only used in RS274X types) */
  gerbv_amacro_t *amacro; /*!< an array of all macros used (only used in RS274X types) */
  gerbv_format_t *format; /*!< formatting info */
  gerbv_image_info_t *info; /*!< miscellaneous info regarding the layer such as overall size, etc */
  gerbv_net_t *netlist; /*!< an array of all geometric entities in the layer */
  gerbv_stats_t *gerbv_stats; /*!< RS274X statistics for the layer */
  gerbv_drill_stats_t *drill_stats;  /*!< Excellon drill statistics for the layer */
} gerbv_image_t;
```

If `apert` is NULL, a new `gerbv_aperture_t` structure is allocated at [12] and stored in the spot indexed by `tool_num`. At [13] the `size` is (legitimately) written inside the allocated structure. Definition of `gerbv_aperture_t` is the following:

```
typedef struct gerbv_aperture {
    gerbv_aperture_type_t type;
    gerbv_amacro_t *amacro;
    gerbv_simplified_amacro_t *simplified;
    double parameter[APERTURE_PARAMETERS_MAX];
    int nuf_parameters;
    gerbv_unit_t unit;
} gerbv_aperture_t;
```

The result is that an attacker will be able to overwrite any 4-bytes-aligned NULL dword in memory in 32bit mode, while in 64bit the write is restricted depending on the size of `int`, but would still allow exploitation by overwriting (for example) data in the heap.

A valuable attack vector in this instance (at least for the 32bit mode) is the `drill_stats` linked-list, which gets stored in heap right after the `image` structure. In fact, after the code above is executed, each drill is added to a “drill_list”:

```
stats = image->drill_stats;
string = g_strdup_printf("%s", (state->unit == GERBV_UNIT_MM ? _("mm") : _("inch")));
drill_stats_add_to_drill_list(stats->drill_list,
                              tool_num,
                              state->unit == GERBV_UNIT_MM ? size*25.4 : size,
                              string);
```

Because the linked-list is walked by checking for the `next` field against NULL, an attacker can overwrite the `next` field and craft custom fields in the list via the write at [13]. This drill list and its contents will later be destroyed when the image destructor is called, triggering an arbitrary free which could lead to code execution.

Crash Information

```
$ gerbv -x png -o out.png drill_parse_T_code.poc

** (process:3383): CRITICAL **: 21:09:11.694: Out of bounds drill number 10000000 at line 2 in file "drill_parse_T_code.poc"

** (process:3383): CRITICAL **: 21:09:11.695: Read a drill of diameter 30 inches at line 2 in file "drill_parse_T_code.poc"

** (process:3383): WARNING **: 21:09:11.695: Assuming units are mils
ASAN:DEADLYSIGNAL
=================================================================
==3383==ERROR: AddressSanitizer: SEGV on unknown address 0xf4e25e04 (pc 0x56773a17 bp 0xf2c03100 sp 0xffb56d10 T0)
==3383==The signal is caused by a READ memory access.
    #0 0x56773a16 in drill_parse_T_code ./gerbv-git/src/drill.c:1158
    #1 0x56773a16 in parse_drillfile ./gerbv-git/src/drill.c:781
    #2 0x567e84cb in gerbv_open_image ./gerbv-git/src/gerbv.c:532
    #3 0x567e84cb in gerbv_open_layer_from_filename_with_color ./gerbv-git/src/gerbv.c:249
    #4 0x56654850 in main ./gerbv-git/src/main.c:929
    #5 0xf6bedf20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)
    #6 0x5665a23f  (gerbv+0x2223f)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV ./gerbv-git/src/drill.c:1158 in drill_parse_T_code
==3383==ABORTING
```

Timeline

2021-10-25 - Vendor Disclosure

2022-11-04 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.