Talos Vulnerability Report

TALOS-2021-1416

Gerbv RS-274X aperture macro multiple outline primitives out-of-bounds read vulnerability

FEBRUARY 28, 2022

CVE NUMBER

CVE-2021-40402

Summary

An out-of-bounds read vulnerability exists in the RS-274X aperture macro multiple outline primitives functionality of Gerbv 2.7.0 and dev (commit b5f1eacd), and Gerbv forked 2.7.1 and 2.8.0. A specially-crafted Gerber file can lead to information disclosure. An attacker can provide a malicious file to trigger this vulnerability.

Tested Versions

Gerbv 2.7.0
Gerbv forked 2.7.1
Gerbv forked 2.8.0
Gerbv dev (commit b5f1eacd)

Product URLs

Gerbv - https://sourceforge.net/projects/gerbv/ Gerbv forked - https://github.com/gerbv/gerbv

CVSSv3 Score

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

CWE

CWE-755 - Improper Handling of Exceptional Conditions

Details

Gerbv is an open-source software that allows users 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 RS-274X 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;
    }
    ...
    if (gerber_is_rs274x_p(fd, &foundBinary)) {                                 // [1]
        dprintf("Found RS-274X file\n");
        if (!foundBinary || forceLoadFile) {
                /* figure out the directory path in case parse_gerb needs to
                 * load any include files */
                gchar *currentLoadDirectory = g_path_get_dirname (filename);
                parsed_image = parse_gerb(fd, currentLoadDirectory);            // [2]
                g_free (currentLoadDirectory);
        }
    }
    ...
```

A file is considered of type "RS-274X" if the function `gerber_is_rs274x_p` [1] returns true. When true, `parse_gerb` is called [2] 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 an RS-274X file:

```
gboolean
gerber_is_rs274x_p(gerb_file_t *fd, gboolean *returnFoundBinary)
{
    ...
    while (fgets(buf, MAXL, fd->fd) != NULL) {
        dprintf ("buf = \"%s\"\n", buf);
        len = strlen(buf);

        /* First look through the file for indications of its type by
         * checking that file is not binary (non-printing chars and white
         * spaces)
         */
        for (i = 0; i < len; i++) {                                    // [3]
            if (!isprint((int) buf[i]) && (buf[i] != '\r') &&
                (buf[i] != '\n') && (buf[i] != '\t')) {
                found_binary = TRUE;
                dprintf ("found_binary (%d)\n", buf[i]);
            }
        }
        if (g_strstr_len(buf, len, "%ADD")) {
            found_ADD = TRUE;
            dprintf ("found_ADD\n");
        }
        if (g_strstr_len(buf, len, "D00") || g_strstr_len(buf, len, "D0")) {
            found_D0 = TRUE;
            dprintf ("found_D0\n");
        }
        if (g_strstr_len(buf, len, "D02") || g_strstr_len(buf, len, "D2")) {
            found_D2 = TRUE;
            dprintf ("found_D2\n");
        }
        if (g_strstr_len(buf, len, "M00") || g_strstr_len(buf, len, "M0")) {
            found_M0 = TRUE;
            dprintf ("found_M0\n");
        }
        if (g_strstr_len(buf, len, "M02") || g_strstr_len(buf, len, "M2")) {
            found_M2 = TRUE;
            dprintf ("found_M2\n");
        }
        if (g_strstr_len(buf, len, "*")) {
            found_star = TRUE;
            dprintf ("found_star\n");
        }
        /* look for X<number> or Y<number> */
        if ((letter = g_strstr_len(buf, len, "X")) != NULL) {
            if (isdigit((int) letter[1])) { /* grab char after X */
                found_X = TRUE;
                dprintf ("found_X\n");
            }
        }
        if ((letter = g_strstr_len(buf, len, "Y")) != NULL) {
            if (isdigit((int) letter[1])) { /* grab char after Y */
                found_Y = TRUE;
                dprintf ("found_Y\n");
            }
        }
    }
    ...
    /* Now form logical expression determining if the file is RS-274X */
    if ((found_D0 || found_D2 || found_M0 || found_M2) &&                // [4]
        found_ADD && found_star && (found_X || found_Y))
        return TRUE;

    return FALSE;

} /* gerber_is_rs274x */
```

For an input to be considered an RS-274X file, the file must first of all contain only printing characters [3]. The other requirements can be gathered by the conditional expression at [4]. An example of a minimal RS-274X file is the following:

```
%FSLAX26Y26*%
%MOMM*%
%ADD100C,1.5*%
D100*
X0Y0D03*
M02*
```

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 an RS-274X file, even though it will fail later checks in the execution flow:

```
%ADD0X0*
```

After an RS-274X file has been recognized, `parse_gerb` is called, which in turn calls `gerber_parse_file_segment`:

```
gboolean
gerber_parse_file_segment (gint levelOfRecursion, gerbv_image_t *image,
                           gerb_state_t *state,        gerbv_net_t *curr_net,
                           gerbv_stats_t *stats, gerb_file_t *fd,
                           gchar *directoryPath)
{
    ...
    while ((read = gerb_fgetc(fd)) != EOF) {
        ...
        case '%':
            dprintf("... Found %% code at line %ld\n", line_num);
            while (1) {
                    parse_rs274x(levelOfRecursion, fd, image, state, curr_net,
                                stats, directoryPath, &line_num);
```

If our file starts with “%”, we end up calling `parse_rs274x`:

```
static void
parse_rs274x(gint levelOfRecursion, gerb_file_t *fd, gerbv_image_t *image,
             gerb_state_t *state, gerbv_net_t *curr_net, gerbv_stats_t *stats,
             gchar *directoryPath, long int *line_num_p)
{
    ...
    switch (A2I(op[0], op[1])){
    ...
    case A2I('A','D'): /* Aperture Description */
        a = (gerbv_aperture_t *) g_new0 (gerbv_aperture_t,1);

        ano = parse_aperture_definition(fd, a, image, scale, line_num_p); // [6]
        ...
        break;
    case A2I('A','M'): /* Aperture Macro */
        tmp_amacro = image->amacro;
        image->amacro = parse_aperture_macro(fd);                         // [5]
        if (image->amacro) {
            image->amacro->next = tmp_amacro;
        ...
```

For this advisory, we’re interested in the `AM` and `AD` commands. For details on the Gerber format see the specification from Ucamco.

In summary, `AM` defines a “macro aperture template,” which is, in other terms, a parametrized shape. It is a flexible way to define arbitrary shapes by building on top of simpler shapes (primitives). It allows arithmetic operations and variable definition. After a template has been defined, the `AD` command is used to instantiate the template and optionally pass some parameters to customize the shape.

From the specification, this is the syntax of the `AM` command:

```
<AM command>          = AM<Aperture macro name>*<Macro content>
<Macro content>       = {{<Variable definition>*}{<Primitive>*}}
<Variable definition> = $K=<Arithmetic expression>
<Primitive>           = <Primitive code>,<Modifier>{,<Modifier>}|<Comment>
<Modifier>            = $M|< Arithmetic expression>
<Comment>             = 0 <Text>
```

While this is the syntax for the `AD` command:

```
<AD command> = ADD<D-code number><Template>[,<Modifiers set>]*
<Modifiers set> = <Modifier>{X<Modifier>}
```

For this advisory, we’re interested in the “Outline” primitive (code `4`). From the specification:

An outline primitive is an area defined by its outline or contour. The outline is a polygon, consisting of linear segments only, defined by its start vertex and n subsequent vertices.

The outline primitive should contain the following fields:

```
+-----------------+-------------------------------------------------------------------------------------+
| Modifier number | Description                                                                         |
+-----------------+-------------------------------------------------------------------------------------+
| 1               | Exposure off/on (0/1)                                                               |
+-----------------+-------------------------------------------------------------------------------------+
| 2               | The number of vertices of the outline = the number of coordinate pairs minus one.   |
|                 | An integer ≥3.                                                                      |
+-----------------+-------------------------------------------------------------------------------------+
| 3, 4            | Start point X and Y coordinates. Decimals.                                          |
+-----------------+-------------------------------------------------------------------------------------+
| 5, 6            | First subsequent X and Y coordinates. Decimals.                                     |
+-----------------+-------------------------------------------------------------------------------------+
| ...             | Further subsequent X and Y coordinates. Decimals.                                   |
|                 | The X and Y coordinates are not modal: both X and Y must be specified for all points.  |
+-----------------+-------------------------------------------------------------------------------------+
| 3+2n, 4+2n      | Last subsequent X and Y coordinates. Decimals. Must be equal to the start coordinates. |
+-----------------+-------------------------------------------------------------------------------------+
| 5+2n            | Rotation angle, in degrees counterclockwise, a decimal.                             |
|                 | The primitive is rotated around the origin of the macro definition,                 |
|                 | i.e. the (0, 0) point of macro                                                      |
+-----------------+-------------------------------------------------------------------------------------+
```

Also the specification states that “The maximum number of vertices is 5000,” which is controlled by the modified number 2. So, depending on the number of vertices, the length of this primitive will change.

In the `parse_rs274x` function, when an `AM` command is found, the function `parse_aperture_macro` is called [5]. Let’s see how this outline primitive is handled there:

```
gerbv_amacro_t *
parse_aperture_macro(gerb_file_t *fd)
{
    gerbv_amacro_t *amacro;
    gerbv_instruction_t *ip = NULL;
    int primitive = 0, c, found_primitive = 0;
    ...
    int equate = 0;

    amacro = new_amacro();

    ...
    /*
     * Since I'm lazy I have a dummy head. Therefore the first
     * instruction in all programs will be NOP.
     */
    amacro->program = new_instruction();
    ip = amacro->program;

    while(continueLoop) {

        c = gerb_fgetc(fd);
        switch (c) {
        ...
        case '*':
            ...
            /*
             * Check is due to some gerber files has spurious empty lines.
             * (EagleCad of course).
             */
            if (found_primitive) {
                ip->next = new_instruction(); /* XXX Check return value */
                ip = ip->next;
                if (equate) {
                    ip->opcode = GERBV_OPCODE_PPOP;
                    ip->data.ival = equate;
                } else {
                    ip->opcode = GERBV_OPCODE_PRIM;                          // [10]
                    ip->data.ival = primitive;
                }
                equate = 0;
                primitive = 0;
                found_primitive = 0;
            }
            break;
        ...
        case ',':
            if (!found_primitive) {                                          // [8]
                found_primitive = 1;
                break;
            }
            ...
            break;
        ...
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
        case '.':
            /*
             * First number in an aperture macro describes the primitive
             * as a numerical value
             */
            if (!found_primitive) {                                          // [7]
                primitive = (primitive * 10) + (c - '0');
                break;
            }
            (void)gerb_ungetc(fd);
            ip->next = new_instruction(); /* XXX Check return value */      // [9]
            ip = ip->next;
            ip->opcode = GERBV_OPCODE_PUSH;
            amacro->nuf_push++;
            ip->data.fval = gerb_fgetdouble(fd);
            if (neg)
                ip->data.fval = -ip->data.fval;
            neg = 0;
            comma = 0;
            break;
        case '%':
            gerb_ungetc(fd);  /* Must return with % first in string
                                 since the main parser needs it */
            return amacro;                                                   // [11]
        default :
            /* Whitespace */
            break;
        }
        if (c == EOF) {
            continueLoop = 0;
        }
    }
    free (amacro);
    return NULL;
}
```

As we can see, this function implements a set of opcodes for a virtual machine that are used to perform arithmetic operations and handle variable definitions and references via a virtual stack, as well as primitives.

Let’s take an outline primitive definition as example:

```
%AMTT*4,0,3,1,1,1*%
```

As discussed before, `%AM` will land us in the `parse_aperture_macro` function, and `TT` is the name for the macro. The macro parsing starts with `4` [7]: this is the primitive number, which is read as a decimal number until a `,` is found [8]. After that, each field separated by `,` is read as a `double` and added to the stack via `PUSH` [9]. These form the arguments to the primitive. When `*` is found [10], the primitive instruction is added, and with `%` the macro is returned.

The `%AM` command also supports defining multiple primitives, in order to produce more complex shapes:

```
%AMRR*
4,1,
4,0,0,0,1,1,1,1,0,0,0,
0*
4,1
3,1,0,2,0,1,1,1,0,
0*
%
```

This `RR` macro defines a square using the “outline” primitive (number 4), and then defines an orthogonal triangle again using the “outline” primitive which is placed at the right of the square, resulting in a right trapezoid.

For reference, these are the prototypes for the macro and the program instructions:

```
struct amacro {
    gchar *name;
    gerbv_instruction_t *program;
    unsigned int nuf_push;
    struct amacro *next;
}

struct instruction {
    gerbv_opcodes_t opcode;
    union {
        int ival;
        float fval;
    } data;
    struct instruction *next;
}
```

Back to `parse_rs274x`: When an `AD` command is found, the function `parse_aperture_definition` is called [6], which in turn calls `simplify_aperture_macro` when the `AD` command is using a template.

```
static int
simplify_aperture_macro(gerbv_aperture_t *aperture, gdouble scale)
{
    ...
    gerbv_instruction_t *ip;
    int handled = 1, nuf_parameters = 0, i, j, clearOperatorUsed = FALSE;
    double *lp; /* Local copy of parameters */
    double tmp[2] = {0.0, 0.0};
    gerbv_aperture_type_t type = GERBV_APTYPE_NONE;                         // [12]
    gerbv_simplified_amacro_t *sam;
    ...
    for(ip = aperture->amacro->program; ip != NULL; ip = ip->next) {
        switch(ip->opcode) {
        case GERBV_OPCODE_NOP:
            break;
        ...
        case GERBV_OPCODE_PRIM :
            /*
             * This handles the exposure thing in the aperture macro
             * The exposure is always the first element on stack independent
             * of aperture macro.
             */
            switch(ip->data.ival) {
            ...
            case 4 :                                                        // [13]
                dprintf("  Aperture macro outline [4] (");
                type = GERBV_APTYPE_MACRO_OUTLINE;                          // [19]
                /*
                 * Number of parameters are:
                 * - number of points defined in entry 1 of the stack +
                 *   start point. Times two since it is both X and Y.
                 * - Then three more; exposure,  nuf points and rotation.
                 *
                 * @warning Calculation must be guarded against signed integer
                 *     overflow
                 *
                 * @see CVE-2021-40394
                 */
                int const sstack = (int)s->stack[1];
                if ((sstack < 0) || (sstack >= INT_MAX / 4)) {              // [14]
                    GERB_COMPILE_ERROR(_("Possible signed integer overflow "
                            "in calculating number of parameters "
                            "to aperture macro, will clamp to "
                            "(%d)"), APERTURE_PARAMETERS_MAX);
                    nuf_parameters = APERTURE_PARAMETERS_MAX;
                } else {
                    nuf_parameters = (sstack + 1) * 2 + 3;                  // [15]
                }
                break;
            ...
            default :
                handled = 0;                                                // [20]
            }

            if (type != GERBV_APTYPE_NONE) {
                if (nuf_parameters > APERTURE_PARAMETERS_MAX) {             // [16]
                        GERB_COMPILE_ERROR(_("Number of parameters to aperture macro (%d) "
                                                        "are more than gerbv is able to store (%d)"),
                                                        nuf_parameters, APERTURE_PARAMETERS_MAX);
                        nuf_parameters = APERTURE_PARAMETERS_MAX;
                }

                /*
                 * Create struct for simplified aperture macro and
                 * start filling in the blanks.
                 */
                sam = g_new (gerbv_simplified_amacro_t, 1);
                sam->type = type;
                sam->next = NULL;
                memset(sam->parameter, 0,
                       sizeof(double) * APERTURE_PARAMETERS_MAX);
                memcpy(sam->parameter, s->stack,                            // [17]
                       sizeof(double) *  nuf_parameters);
                ...
                /*
                 * Add this simplified aperture macro to the end of the list
                 * of simplified aperture macros. If first entry, put it
                 * in the top.
                 */
                if (aperture->simplified == NULL) {                         // [18]
                    aperture->simplified = sam;
                } else {
                    gerbv_simplified_amacro_t *tmp_sam;
                    tmp_sam = aperture->simplified;
                    while (tmp_sam->next != NULL) {
                        tmp_sam = tmp_sam->next;
                    }
                    tmp_sam->next = sam;
                }
                ...
            }
```

For this advisory, all the `AD` commands have to do is utilize the macro that we just created, without special parameters. Let's consider the following macro and aperture definition:

```
%AMRR*
4,1,
4,0,0,0,1,1,1,1,0,0,0,
0*
4,1,
3,1,0,2,0,1,1,1,0,
0*
%
%ADD11RR*
```

To parse the `AD` command, the `simplify_aperture_macro` function will execute the `RR` macro in the virtual machine using the parameters given by `AD`. In this case we haven't specified any parameter since they're not relevant for this issue.

As previously discussed, our program (macro) contains a series of `GERBV_OPCODE_PUSH` instructions (pushing the numbers `1,4,0,0,0,1,1,1,1,0,0,0,0` for the first outline macro) and a `GERBV_OPCODE_PRIM` instruction for primitive 4 (outline), executed at [13].

At [15] the number of vertices is taken from the second field in the stack (as per specification), and the number of parameters for the primitive is calculated.
At [14] there is an integer overflow check that fixes a previous vulnerability (TALOS-2021-1405), which makes sure that `nuf_parameters` stays within the allowed size. Note that this fix is only present in a pull request and has not yet been merged at the time of writing. Either way, that fix wouldn't change the outcome of the issue described in this advisory. At [16] the code makes sure that `nuf_parameters` is not bigger than `APERTURE_PARAMETERS_MAX` (102), otherwise `nuf_parameters` gets limited to `APERTURE_PARAMETERS_MAX`. Then at [17] the parameters are copied from the stack into the newly allocated `sam` structure, which is added to the `aperture->simplified` linked list [18], to keep a list of all simplified aperture macros defined in the macro.

Because `sam->parameter` has a size of `102 * 8`, the checks above are important to limit `nuf_parameters` within the `sam->parameter` buffer. However, when multiple outline definitions are present, like in the `RR` macro above, any check at [14] and [15] can be bypassed.

Let's consider this macro:

```
%AMWW*
4,1,
4,0,0,0,1,1,1,1,0,0,0,
0*
99,1,
81,1,0,2,0,1,1,1,0,
0*
%
%ADD11RR*
```

The first outline is the same as the previous example. The second outline is weird: it specifies a macro code `99`, which is not supported. However, when the first outline is parsed, the `type` variable is set to `GERBV_APTYPE_MACRO_OUTLINE` [19]. When the `99` is parsed, we hit the default case at [20], which does not set any `type` (so `type` is still `GERBV_APTYPE_MACRO_OUTLINE`) but sets the `handled` variable to 0. This is used as the return value for the current function. This return value however is never checked by the caller. When the default case is matched, the loop is not interrupted. Anything happening after this will assume the current macro is an outline, and it will use the same `nuf_parameters` of the previous cycle (which has been sanitized). No out-of-bounds operations are going to happen at [17]. The line at [17] will, however, copy all the current parameters, including the `81` (which, for an outline, corresponds to the number of vertices).
Then, at [18], the new simplified aperture macro is added to the `aperture->simplified` linked list, which will be later used by the rendering code.

When the final image is drawn, the function `gerbv_draw_amacro` is called, which contains this code for handling outlines:

```
typedef enum {
                OUTLINE_EXPOSURE,
                OUTLINE_NUMBER_OF_POINTS,
                OUTLINE_FIRST_X, /* x0 */
                OUTLINE_FIRST_Y, /* y0 */
                /* x1, y1, x2, y2, ..., rotation */
                OUTLINE_ROTATION, /* Rotation index is correct if outline has
                                     no point except first */
} gerbv_aptype_macro_outline_index_t;

/* Point number is from 0 (first) to (including) OUTLINE_NUMBER_OF_POINTS */
#define OUTLINE_X_IDX_OF_POINT(number) (2*(number) + OUTLINE_FIRST_X)
#define OUTLINE_Y_IDX_OF_POINT(number) (2*(number) + OUTLINE_FIRST_Y)
#define        OUTLINE_ROTATION_IDX(param_array) \                          // [21]
                        ((int)param_array[OUTLINE_NUMBER_OF_POINTS]*2 + \
                        OUTLINE_ROTATION)
...
static int
gerbv_draw_amacro(cairo_t *cairoTarget, cairo_operator_t clearOperator,
        cairo_operator_t darkOperator, gerbv_simplified_amacro_t *s,
        gint usesClearPrimitive, gdouble pixelWidth, enum draw_mode drawMode,
        gerbv_selection_info_t *selectionInfo,
        gerbv_image_t *image, struct gerbv_net *net)
{
    ...
    case GERBV_APTYPE_MACRO_OUTLINE:
            draw_update_macro_exposure (cairoTarget,
                            clearOperator, darkOperator,
                            ls->parameter[OUTLINE_EXPOSURE]);
            cairo_rotate (cairoTarget, DEG2RAD(ls->parameter[                  // [22]
                            OUTLINE_ROTATION_IDX(ls->parameter)]));
            cairo_move_to (cairoTarget,
                            ls->parameter[OUTLINE_FIRST_X],
                            ls->parameter[OUTLINE_FIRST_Y]);

            for (int point = 1; point <
                            1 + (int)ls->parameter[                            // [23]
                                    OUTLINE_NUMBER_OF_POINTS];
                                                point++) {
                    cairo_line_to (cairoTarget,
                            ls->parameter[OUTLINE_X_IDX_OF_POINT(
                                                point)],
                            ls->parameter[OUTLINE_Y_IDX_OF_POINT(
                                                point)]);
            }
```

This function draws the objects defined in the simplified macro by reading the parameters saved at [18]. So, at [21] `param_array[OUTLINE_NUMBER_OF_POINTS]` will access the number of points of the outline, which in our example is `81`. This number is too big for the `parameter` array, and will cause out-of-bounds accesses at [21], [22] and [23]. Since attackers control this number arbitrarily, with careful heap manipulation, they could exploit this to extract the process' memory by analyzing the objects in the rendered image (e.g. the object's rotation).

Note that another spot where a similar out-of-bounds access happens is in function `gerber_parse_file_segment`, for the same reason explained before:

```
} else if (ls->type == GERBV_APTYPE_MACRO_OUTLINE) {
    int pointCounter,numberOfPoints;
    numberOfPoints = ls->parameter[OUTLINE_NUMBER_OF_POINTS] + 1;

    for (pointCounter = 0; pointCounter < numberOfPoints; pointCounter++) {
        gerber_update_min_and_max (&boundingBox,
                                   curr_net->stop_x +
                                   ls->parameter[OUTLINE_X_IDX_OF_POINT(pointCounter)],
                                   curr_net->stop_y +
                                   ls->parameter[OUTLINE_Y_IDX_OF_POINT(pointCounter)],
                                   0,0,0,0);
    }
```

Finally, because Gerbv supports the `%IF` command, which allows to include and parse any file in the system (either via relative or absolute paths), an attacker might be able to exploit this vulnerability to exfiltrate file contents by dumping specific parts of the process memory.

Crash Information

```
# ./gerbv -x png -o out.png draw_amacro_outline.min.oobr
=================================================================
==85035==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf23033bc at pc 0x5665269e bp 0xff9d4a28 sp 0xff9d4a18
READ of size 8 at 0xf23033bc thread T0
    #0 0x5665269d in gerber_parse_file_segment ./src/gerber.c:562
    #1 0x56654f27 in parse_gerb ./src/gerber.c:769
    #2 0x5666bdbc in gerbv_open_image ./src/gerbv.c:526
    #3 0x5666961f in gerbv_open_layer_from_filename_with_color ./src/gerbv.c:249
    #4 0x565d0431 in main ./src/main.c:932
    #5 0xf6b83f20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)
    #6 0x5658c290  (./gerbv+0x16290)

0xf23033bc is located 4 bytes to the right of 824-byte region [0xf2303080,0xf23033b8)
allocated by thread T0 here:
    #0 0xf79fdf54 in malloc (/usr/lib/i386-linux-gnu/libasan.so.4+0xe5f54)
    #1 0xf7008568 in g_malloc (/usr/lib/i386-linux-gnu/libglib-2.0.so.0+0x4e568)
    #2 0x56661b7c in parse_aperture_definition ./src/gerber.c:2287
    #3 0x5665c253 in parse_rs274x ./src/gerber.c:1638
    #4 0x5664eff5 in gerber_parse_file_segment ./src/gerber.c:244
    #5 0x56654f27 in parse_gerb ./src/gerber.c:769
    #6 0x5666bdbc in gerbv_open_image ./src/gerbv.c:526
    #7 0x5666961f in gerbv_open_layer_from_filename_with_color ./src/gerbv.c:249
    #8 0x565d0431 in main ./src/main.c:932
    #9 0xf6b83f20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)

SUMMARY: AddressSanitizer: heap-buffer-overflow ./src/gerber.c:562 in gerber_parse_file_segment
Shadow bytes around the buggy address:
  0x3e460620: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3e460630: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3e460640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3e460650: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3e460660: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x3e460670: 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa fa fa
  0x3e460680: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e460690: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3e4606a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3e4606b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x3e4606c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==85035==ABORTING
```

Timeline

2021-11-22 - Initial contact

2022-01-28 - 60 day follow up

2022-02-24 - 90 day public disclosure notice; vendor agreed

2022-02-28 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.