Talos Vulnerability Report

TALOS-2021-1413

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

FEBRUARY 28, 2022

CVE NUMBER

CVE-2021-40400

Summary

An out-of-bounds read vulnerability exists in the RS-274X aperture macro outline primitive functionality of Gerbv 2.7.0 and dev (commit b5f1eacd) and the forked version of Gerbv (commit d7f42a9a). 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 dev (commit b5f1eacd)
Gerbv forked dev (commit d7f42a9a)

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-119 - Improper Restriction of Operations within the Bounds of a Memory Buffer

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). Moreover, gerbv can be compiled and used as a shared library. For these reasons, 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, the `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*
```

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 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 to perform arithmetic operations and define variables. 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 accordingly.

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 is used to perform arithmetic operations, handle variable definitions and references via a virtual stack, and primitives.

Let’s take an outline primitive definition as example:

```
%AMX0*4,0,3,1,1,1*%
```

As discussed before, `%AM` will land us in the `parse_aperture_macro` function, and `X0` 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.

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;   // [18]
    double *lp; /* Local copy of parameters */
    double tmp[2] = {0.0, 0.0};
    ...
    /* Allocate stack for VM */
    s = new_stack(aperture->amacro->nuf_push + extra_stack_size);           // [12]
    if (s == NULL)
        GERB_FATAL_ERROR("malloc stack failed in %s()", __FUNCTION__);
    ...
    for(ip = aperture->amacro->program; ip != NULL; ip = ip->next) {
        switch(ip->opcode) {
        case GERBV_OPCODE_NOP:
            break;
        case GERBV_OPCODE_PUSH :
            push(s, ip->data.fval);                                         // [13]
            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 :                                                        // [14]
                dprintf("  Aperture macro outline [4] (");
                type = GERBV_APTYPE_MACRO_OUTLINE;
                /*
                 * 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.
                 */
                nuf_parameters = ((int)s->stack[1] + 1) * 2 + 3;            // [15]
                break;
            ...
            }

            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;           // [17]
                }

                /*
                 * 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,                            // [18]
                       sizeof(double) *  nuf_parameters);
```

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

```
%ADD09X0*
```

For `AD` to use the template, it has to execute the template in the virtual machine. To this end, a virtual stack is allocated at [12] to handle parameters. The size of this stack depends on `nuf_push`, which is incremented at [9] every time a `GERBV_OPCODE_PUSH` instruction is added to the program.

In the case of the sample macro previously discussed, our program will contain a series of `GERBV_OPCODE_PUSH` instructions (pushing the numbers `0,3,1,1,1` to the stack, at [13]) and a `GERBV_OPCODE_PRIM` instruction for primitive 4 (outline), executed at [14].

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 [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` [17]. Finally at [18] the parameters are copied from the stack into the newly allocated `sam` structure.

The problem in this whole logic is that the stack buffer (`s->stack`) created at [12] has a size that depends on `nuf_push`, while the `memcpy` happening at [18] has a size that depends on `nuf_parameters`. In the sample macro, the value of `nuf_parameters` is `3`, however an attacker could use any arbitrary number, which is taken verbatim at [15] by reading `s->stack[1]` and used to calculate `nuf_parameters`. At [17] the value of `nuf_parameters` is restricted to a maximum of 102, meaning that an attacker can set an arbitrary `nuf_parameters` value from 0 to 102, causing the `memcpy` at [18] to range from 0 to 816 (i.e. `102 * sizeof(double)`).
If `nuf_push` is smaller than `nuf_parameters`, the `memcpy` will cause an out-of-bounds read on the `s->stack` buffer, which will lead to storing data of the nearby heap chunks into `sam->parameter`. Since `sam->parameter` is used to draw the shape for the macro being evaluated, this will result in the final drawing having a different shape, coordinate points and rotation, depending on the values stored in the nearby heap chunks. Since an attacker might be able to read the rendered image at the end of the parsing (e.g. if the service using Gerbv is converting a `.gbr` file into a `.png` and returning it to the user), it would be possible to extract heap metadata or contents by reading the resulting rendered image. The quality of the information depends on the `dpi` chosen for the operation, however with careful heap manipulation, in the worst case this could result in an information leak of the process' memory.

Crash Information

```
# ./gerbv -x png -o out aperture_macro_parameters_oobr.poc

** (process:267): CRITICAL **: 15:11:07.120: Number of parameters to aperture macro (2005) are more than gerbv is able to store
(102)================================================================
==267==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf3e027b8 at pc 0xf799d8be bp 0xfff31768 sp 0xfff31338
READ of size 816 at 0xf3e027b8 thread T0
    #0 0xf799d8bd  (/usr/lib/i386-linux-gnu/libasan.so.4+0x778bd)
    #1 0x5664448d in simplify_aperture_macro ./src/gerber.c:2051
    #2 0x56646257 in parse_aperture_definition ./src/gerber.c:2272
    #3 0x56640cef in parse_rs274x ./src/gerber.c:1637
    #4 0x56634211 in gerber_parse_file_segment ./src/gerber.c:243
    #5 0x56639d97 in parse_gerb ./src/gerber.c:768
    #6 0x5664fdb3 in gerbv_open_image ./src/gerbv.c:526
    #7 0x5664d760 in gerbv_open_layer_from_filename_with_color ./src/gerbv.c:249
    #8 0x565b9528 in main ./src/main.c:932
    #9 0xf6b91f20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)
    #10 0x56577220  (./gerbv+0x16220)

0xf3e027b8 is located 0 bytes to the right of 120-byte region [0xf3e02740,0xf3e027b8)
allocated by thread T0 here:
    #0 0xf7a0c124 in calloc (/usr/lib/i386-linux-gnu/libasan.so.4+0xe6124)
    #1 0xf70165ca in g_malloc0 (/usr/lib/i386-linux-gnu/libglib-2.0.so.0+0x4e5ca)
    #2 0x566439f1 in simplify_aperture_macro ./src/gerber.c:1922
    #3 0x56646257 in parse_aperture_definition ./src/gerber.c:2272
    #4 0x56640cef in parse_rs274x ./src/gerber.c:1637
    #5 0x56634211 in gerber_parse_file_segment ./src/gerber.c:243
    #6 0x56639d97 in parse_gerb ./src/gerber.c:768
    #7 0x5664fdb3 in gerbv_open_image ./src/gerbv.c:526
    #8 0x5664d760 in gerbv_open_layer_from_filename_with_color ./src/gerbv.c:249
    #9 0x565b9528 in main ./src/main.c:932
    #10 0xf6b91f20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)

SUMMARY: AddressSanitizer: heap-buffer-overflow (/usr/lib/i386-linux-gnu/libasan.so.4+0x778bd)
Shadow bytes around the buggy address:
  0x3e7c04a0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x3e7c04b0: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
  0x3e7c04c0: 00 00 00 00 00 00 01 fa fa fa fa fa fa fa fa fa
  0x3e7c04d0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fa
  0x3e7c04e0: fa fa fa fa fa fa fa fa 00 00 00 00 00 00 00 00
=>0x3e7c04f0: 00 00 00 00 00 00 00 00[fa]fa fa fa fa fa fa fa
  0x3e7c0500: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04
  0x3e7c0510: fa fa fa fa fa fa fa fa fd fd fd fd fd fd fd fd
  0x3e7c0520: fd fd fd fd fd fd fd fd fa fa fa fa fa fa fa fa
  0x3e7c0530: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x3e7c0540: fa fa fa fa fa fa fa fa fd fd fd fd fd fd fd fd
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
==267==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.