

## Talos Vulnerability Report

TALOS-2021-1375

### Accusoft ImageGear JPEG-JFIF lossless Huffman parser heap-based buffer overflow vulnerabilities

FEBRUARY 23, 2022

#### CVE NUMBER

CVE-2021-21947,CVE-2021-21946

#### Summary

Two heap-based buffer overflow vulnerabilities exist in the JPEG-JFIF lossless Huffman image parser functionality of Accusoft ImageGear 19.10. A specially-crafted file can lead to a heap buffer overflow. An attacker can provide a malicious file to trigger these vulnerabilities.

#### Tested Versions

Accusoft ImageGear 19.10

#### Product URLs

ImageGear - <https://www.accusoft.com/products/imagegear-collection/>

#### CVSSv3 Score

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

#### CWE

CWE-122 - Heap-based Buffer Overflow

#### Details

The ImageGear library is a document-imaging developer toolkit that offers image conversion, creation, editing, annotation and more. It supports more than 100 formats such as DICOM, PDF, Microsoft Office and others.

When a JPEG-JFIF with specific markers is loaded, its data is parsed by the `process_jpeg_lossless` function.

The `process_jpeg_lossless` function:

```

AT_ERRCOUNT
process_jpeg_lossless
    (jpeg_dec *jpeg_dec,SOF_object *SOF_object,short restart_interval,int max_X_sampling,
     int max_Y_sampling,lpf_allocation_jpeg_buffer lpf_allocation_jpeg_buffer)

{
    [...]

    local_8 = DAT_102bcea8 ^ (uint)0stack0xfffffffffc;
    image_width = (SOF_object->SOF_header).width;
    image_height = (SOF_object->SOF_header).height;
    precision = (SOF_object->SOF_header).precision;
    uVar1 = SOF_object->field_0x1c;
    dVar2 = jpeg_dec->old_lossless_read;
    dVar3 = SOF_object->field_0x28;
    number_of_components = (uint)*(byte *)0SOF_object->possible_num_component_or_color_channel;
    dVar4 = jpeg_dec->additional_huffman_logic;
    component_index = 0;
    single_byte = 0;
    _source_LOW = 0;
    jpeg_io_buff.size_buffer = 0;
    if (number_of_components != 0) {
        component_entry = 8*(SOF_object->nrr_component_buffer_data)[0].component_values.subsampling_X;
        parsed_component_data = horiz_component + 4;
        for (component_index_ = number_of_components; component_index_ != 0;
            component_index_ = component_index_ - 1) {
            *parsed_component_data = 0;
            parsed_component_data = parsed_component_data + 1;
        }
        do {
            X_component = *component_entry;
            horiz_component[component_index + 8] = X_component;
            horiz_component[component_index] = X_component + 1;
            component_index = component_index + 1;
            component_entry = component_entry + 0x14;
        } while (component_index < (int)number_of_components);
    }

    [... input related operations ...]

    image_height_done = 0;
    if (0 < (int)image_height) {
        do {
            if (io_buff != 0) break;
            if (number_of_components != 0) {
                component_index = 0;
                component_index_ = number_of_components;
                do {
                    jpeg_component_table_ =
                        (jpeg_component_table *)
                        ((int)0*(SOF_object->nrr_component_buffer_data)[0].field_0x0 + component_index);
                    component_index = component_index + 0x50;
                    (jpeg_component_table->component_values).buffer_working_ptr =
                        (dword)jpeg_component_table->buffer_1;
                    component_index_ = component_index_ - 1;
                } while (component_index_ != 0);
                if (number_of_components != 0) {
                    piVar10 = local_18;
                    for (component_index_ = number_of_components; component_index_ != 0;
                        component_index_ = component_index_ - 1) {
                        *piVar10 = 0;
                        piVar10 = piVar10 + 1;
                    }
                }
            }
            width_done = 0;
            if (0 < (int)image_width) {
                continue_ROW:
                if (restart_interval != 0) {
                    [...]
                }
                goto LAB_10122beb;
            }
        } while (image_height_done < image_height);
        go_to_next_ROW_or_finish:
        image_height_done_ = image_height_done;
        component_index = 0;
        if (number_of_components != 0) {
            y_comp_ptr = 0*(SOF_object->nrr_component_buffer_data)[0].component_values.subsampling_Y;
            piVar10 = local_28;
            for (component_index_ = number_of_components; component_index_ != 0;
                component_index_ = component_index_ - 1) {
                *piVar10 = 1;
                piVar10 = piVar10 + 1;
            }
            do {
                Y_component = *y_comp_ptr;
                next_component_idx = component_index + 1;
                horiz_component[component_index + 4] =
                    (int)(horiz_component[component_index + 4] + Y_component) %
                    (int)horiz_component[component_index];
                horiz_component[component_index + 8] =
                    (int)(horiz_component[component_index + 8] + Y_component) %
                    (int)horiz_component[component_index];
                y_comp_ptr = y_comp_ptr + 0x14;
                component_index = next_component_idx;
            } while (next_component_idx < (int)number_of_components);
        }
        SOF_object->image_height_done = image_height_done;
        io_buff = (*lpfn_allocation_jpeg_buffer)(2,jpeg_dec->jpeg_related,jpeg_dec,SOF_object);
        image_height_done = image_height_done_ + max_Y_sampling;
    } while ((int)image_height_done < (int)image_height);
}
IOb_done(0jpeg_io_buff);
AVar6 = kind_of_fastfail(local_8 ^ (uint)0stack0xfffffffffc);
return AVar6;
joined_r0x10122a81:
if (single_byte != 0xff) goto LAB_10122aef;
IOb_byte_read(0jpeg_io_buff,0single_byte);
if (single_byte == 0) {
    single_byte = 0xff;
    goto LAB_10122aef;
}
if (7 < (byte)(single_byte + 0x30)) goto LAB_10122aef;
IOb_byte_read(0jpeg_io_buff,0single_byte);
component_index_ = 0;
if (number_of_components != 0) {
    uVar8 = 0;

```

[1]

[2]

[3]

```

do {
    component_index_ = component_index_ + 1;
    local_28[uVar8] = 0;
    local_18[uVar8] = 0;
    uVar8 = component_index_ & 0xffff;
} while (uVar8 < number_of_components);
}
jpeg_io_buff.size_buffer = 0;
goto joined_r0x10122a81;
LAB_10122aef:
component_index_ = (uint)single_byte;
component_index = read_n_bytes(&jpeg_io_buff,6,real_read_size);
temp_var = 8;
local_68 = 8;
if (component_index != 0) {
    [... input related operations ...]
}
local_60 = component_index_ << (0x20U - (char)local_68 & 0x1f);
component_index = 0;
if (number_of_components != 0) {
    source_HIGH = 1 << (cVar5 - 1U & 0x1f);
    X_done = X_done & 0xffff | (uint)source_HIGH << 0x10;
    temp_var = 0;
    piVar10 = local_18;
    for (component_index_ = number_of_components; component_index_ != 0;
        component_index_ = component_index_ - 1) {
        *piVar10 = 0;
        piVar10 = piVar10 + 1;
    }
    piVar10 = local_28;
    for (component_index_ = number_of_components; component_index_ != 0;
        component_index_ = component_index_ - 1) {
        *piVar10 = 0;
        piVar10 = piVar10 + 1;
    }
}
do {
    Y_done_plus_X = component_index + 8;
    component_index = component_index + 1;
    *(ushort *)
        (*(int *)(&(*SOF_object->nr_component_buffer_data)[0].component_values.
            buffer_working_ptr + temp_var) +
        horiz_component[Y_done_plus_X] *
        *(int *)(&(*SOF_object->nr_component_buffer_data)[0].standardized_width + temp_var) * 2) =
        source_HIGH;
    temp_var = temp_var + 0x50;
} while (component_index < (int)number_of_components);
}
LAB_10122be5:
jpeg_io_buff.size_buffer = jpeg_io_buff.size_buffer + 1;
LAB_10122beb:
y_comp_ptr = (dword *)0x0;
if (number_of_components != 0) {
    component_index = 0;
    do {
        Y_done = 0;
        if (0 < (int)(*(SOF_object->nr_component_buffer_data)[component_index].component_values.
            subsampling_Y)) {
            mod_comp_8 = horiz_component[component_index + 8];
            mod_comp_4 = horiz_component[component_index + 4];
            do {
                Y_done_plus_X = Y_done + mod_comp_8;
                temp_var = *(int *)(&(*SOF_object->nr_component_buffer_data)[component_index].
                    standardized_width;
                pjVar17 = *SOF_object->nr_component_buffer_data + component_index;
                dst_buff = (ushort *)(&(*SOF_object->nr_component_values).buffer_working_ptr;
                component_buffer = dst_buff + ((mod_comp_4 - mod_comp_8) + Y_done_plus_X) * temp_var; [4]
                component_buffer_2 = dst_buff + temp_var * Y_done_plus_X;
                X_done = 0;
                if (0 < (int)(pjVar17->component_values).subsampling_X) {
                    local_74 = component_buffer + -1;
                    do {
                        [... read data and compute source_HIGH and _source_LOW ...]
                        shift_bit_n = (byte)dvar3;
                        if ((int)(SOF_object->SOF_header).precision < 9) {
                            component_buffer[X_done] =
                                (ushort)(byte)(((char)source_HIGH << (shift_bit_n & 0x1f)) + (char)_source_LOW); [5]
                        }
                        else {
                            component_buffer[X_done] =
                                (source_HIGH << (shift_bit_n & 0x1f)) + (short)_source_LOW; [6]
                        }
                        X_done = X_done + 1;
                        local_74 = local_74 + 1;
                    } while ((int)X_done <
                        (int)(*(SOF_object->nr_component_buffer_data)[component_index].component_values.
                            subsampling_X);
                }
                Y_done = Y_done + 1;
            } while (Y_done < (int)(*(SOF_object->nr_component_buffer_data)[component_index].
                component_values.subsampling_Y);
        }
        y_comp_ptr = (dword *)((int)y_comp_ptr + 1);
        component_entry =
            &(*SOF_object->nr_component_buffer_data)[component_index].component_values.
            buffer_working_ptr;
        *component_entry =
            *component_entry +
            (*SOF_object->nr_component_buffer_data)[component_index].component_values.subsampling_X * [7]
            2;
        component_index = (int)(short)y_comp_ptr;
    } while (component_index < (int)number_of_components);
}
width_done = width_done + max_X_sampling;
if ((int)image_width <= width_done) goto go_to_next_ROW_or_finish;
goto continue_ROW;
}

```

This function parses the JPEG data when a SOF3 segment is present. When the data is lossless, Huffman code parses the components specified in the SOS segment. This function uses, for each compent, a buffer. Each component buffer's size is calculated in the allocate\_buffer\_for\_jpeg\_decoding function, in which the buffers are also allocated:

```

AT_ERRCOUNT __cdecl
allocate_buffer_for_jpeg_decoding
(jpeg_dec *jpeg_dec, SOF_object *jpeg_object, enum_SOF_type type_of_sof,
 jpeg_component_table *jpeg_component_table)

{
    [...]

    local_10 = 0;
    size_malloc = 0;
    x_MAX_sampling_factor = (uint)jpeg_dec->x_MAX_sampling_factor;
    y_MAX_sampling_factor = (uint)jpeg_dec->y_MAX_sampling_factor;
    if (((jpeg_dec->type_of_SOF == Lossy) || (jpeg_dec->type_of_SOF == Progressive)) && [8]
        ((jpeg_dec->caller_id == 0x15 || (jpeg_dec->caller_id == 0x47)))) &&
        ((jpeg_object->SOF_header).precision == 8) {
        [...]
    }
    else {
        subsampling_X = (jpeg_component_table->component_values).subsampling_X;
        subsampling_Y = (jpeg_component_table->component_values).subsampling_Y;
        *(dword *)&jpeg_component_table->field_0x34 = subsampling_X;
        *(dword *)&jpeg_component_table->field_0x38 = subsampling_Y;
        jpeg_component_table->maybe_per_component_bits = 8;
    }
    [...]
    if (type_of_sof != Lossy) {
        if (type_of_sof == Lossless) {
            subsampling_X_ = (jpeg_component_table->component_values).subsampling_X;
            *(int *)&jpeg_component_table->standardized_width =
                (int)(jpeg_dec->x_image * subsampling_X + -1 + x_MAX_sampling_factor) /
                (int)x_MAX_sampling_factor;
            size_malloc = (subsampling_X_ + subsampling_Y) *
                *(int *)&jpeg_component_table->standardized_width * 2; [9]
            *(int *)&jpeg_component_table->standardized_height =
                (int)(jpeg_dec->y_image * subsampling_Y + -1 + y_MAX_sampling_factor) /
                (int)y_MAX_sampling_factor;
            goto LAB_101269a7;
        }
        [...]
    }
    [...]
LAB_101269a7:
    [...]
    pbVar2 = (byte *)AF_memmm_alloc(jpeg_dec->kind_of_heap, size_malloc); [10]
    jpeg_component_table->buffer_1 = pbVar2;
    pbVar2 = (byte *)AF_memmm_alloc(jpeg_dec->kind_of_heap, size_malloc);
    jpeg_component_table->buffer2 = pbVar2;
    if ((jpeg_component_table->buffer_1 == (byte *)0x0) || (pbVar2 == (byte *)0x0)) {
        local_10 = AF_err_record_set("..\..\..\Common\Formats\jpeg_dec.c", 0xec5, -1000, 0,
            size_malloc, jpeg_dec->kind_of_heap, (LPCHAR)0x0);
    }
    if (type_of_sof == Lossless) {
        *(short *)(&jpeg_component_table->buffer_1 +
            ((size_malloc >> 1) - *(int *)&jpeg_component_table->standardized_width) * 2) =
            1 << ((char)(jpeg_object->SOF_header).precision - 1U & 0x1f);
    }
    (jpeg_component_table->component_values).buffer_working_ptr =
        (dword)jpeg_component_table->buffer_1;
    jpeg_component_table->field_0x0 = 0;
    return local_10;
}

```

The function `allocate_buffer_for_jpeg_decoding` is called for each component. It calculates the required size and allocates two buffers using that size. At [9], the component's subsampling values are used in combination with values calculated at [8] to calculate the size of a single component buffer. The values at [8] are identical for every component. Indeed they are the maximum Vert and Horiz subsampling values among all the components. The size formula is summarized as:

```

standardized_width = (X_image * subsampling_X - 1 + x_MAX_sampling_factor) / x_MAX_sampling_factor
size_malloc        = (subsampling_X + subsampling_Y) * standardized_width * 2

```

This size is then used to allocate at [10] the buffer that will be later used in `process_jpeg_lossless` to process, allegedly, one "row" at the time.

In order to explain the essential points of `process_jpeg_lossless` we will first introduce a schematization of the loop structures used in `process_jpeg_lossless`. The `process_jpeg_lossless` function can be schematized as:

```

def process_jpeg_lossless_easy(X_image, Y_image, image_comps, comp_idx):
    x_comp = image_comps[comp_idx].x
    y_comp = image_comps[comp_idx].y
    component_buffer = image_comps[comp_idx].buffer

    num_comp = len(image_comps)

    for x in range(num_comp):
        x_MAX = max(image_comps[x].x, x_MAX)

    y_MAX = 0
    for x in range(num_comp):
        y_MAX = max(image_comps[x].y, y_MAX)

    standardized_width = (X_image * x_comp - 1 + x_MAX) // x_MAX # as integer

    mod_comp_4 = 0
    mod_comp_0 = x_comp + 1
    mod_comp_8 = x_comp [11]

    y_MAX_extra_idx = 0
    while y_MAX_extra_idx < Y_image:

        x_MAX_extra_idx = 0
        number_of_it = 0
        while x_MAX_extra_idx < X_image:

            for y_idx in range(y_comp):
                Y_done_comp_8 = y_idx + mod_comp_8
                buffer_offset = (mod_comp_4 - mod_comp_8 + Y_done_comp_8) * standardized_width * 2
                    + (number_of_it * x_comp * 2) [12]

                for x_idx in range(x_comp):
                    # CALCULATE the required data for sum_of_short_data or sum_of_byte_data
                    if SOF.precision < 9
                        (component_buffer + buffer_offset)[x_idx] = sum_of_byte_data
                    else:
                        (component_buffer + buffer_offset)[x_idx] = sum_of_short_data
                        # here ^ is accessing the element at position x_idx, of a word array (16bit)

                number_of_it += 1
                x_MAX_extra_idx += x_MAX

            mod_comp_4 = (mod_comp_4 + y_comp) % mod_comp_0 [13]
            mod_comp_8 = (mod_comp_8 + y_comp) % mod_comp_0 [14]

        y_MAX_extra_idx += y_MAX

```

This function does not reflect the original process\_jpeg\_lossless function. This only summarizes the loop structure for a single component. In reality there would be another loop, iterating for each component, before the one for y\_comp. Furthermore the majority of the variables in process\_jpeg\_lossless\_easy exist for each component in process\_jpeg\_lossless. This schematization is useful to understand the structure used to iterate and fill each component buffer.

The overall process repeats until y\_MAX\_extra\_idx < Y\_image, where y\_MAX\_extra\_idx starts from 0 and increases by y\_MAX. Nested there is another loop performed while x\_MAX\_extra\_idx < X\_image. The variables x\_MAX\_extra\_idx start at 0, at the beginning of the Y\_image loop, and are incremented by x\_MAX for each loop. In these loops, for each component, there is a for loop iterated Vert times, and for each of the Vert iterations, another for loop performed Horiz times.

At [12] can be seen that, for each iteration of y\_comp a buffer\_offset is calculated. This variable is used in order to "seek" the proper component's buffer position in which to write; this is performed instead of adapting the accessing index. The buffer\_offset varies based on the various already completed iterations. The corresponding instruction in process\_jpeg\_lossless is related to [4]. The three variables initialized at [11] correspond to the loop at [1] that is performed for each component. The instruction at [13] and [14] correspond to the loop at [3] that is performed every time the X\_image loop is completed. The variable number\_of\_it that is used to contribute in the buffer\_offset with (number\_of\_it \* x\_comp \* 2) corresponds to [7] when increased by one, performed each time the y\_comp loop completes. Instead, number\_of\_it resets to 0 corresponding to [2], performed each time the X\_image loop completes.

## CVE-2021-21946 - Precision lower than 9

A specially-crafted JPEG file can lead to a heap-based buffer overflow in the JPEG lossless Huffman image parser, due to a missing boundary check.

Trying to load a malicious JPEG file, we end up in the following situation:

```

(1fd4.720): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=12653ffe edx=00000000 esi=00000001 edi=12653ffe
eip=707131e1 esp=0019f940 ebp=0019fa88 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
igCore19d!IG_mpi_page_set+0xb71b1:
707131e1 66890471      mov     word ptr [ecx+esi*2],ax  ds:002b:12654000=????

```

The access violation takes place at [5] in the process\_jpeg\_lossless function, when filling a word in a component's buffer when the SOF3's precision is lower than 9.

From the "seeking" of the component's buffer at [4] to [5] there is no boundary check on accessing the element at position x\_idx.

For example with:

```

Y_image = 0x22
X_image = 0x4
precision = 0x8
nr_comp = 2
COMP = {
    Horiz, Vert = 2, 2;
    Horiz, Vert = 3, 9;
}

```

The first component would have as size:

```
standardized_width = (X_image * subsampling_X -1 + x_MAX_sampling_factor) / x_MAX_sampling_factor
malloc_size         = (subsampling_X + subsampling_Y) * standardized_width * 2
                    = (2 + 2) * ((4 * 2 -1 + 3) / 3) * 2 = 0x18 The result is '0x18' because it is firstly calculated '((4 * 2 -1 + 3)/3)' as
an integer before the value is plugged into the formula. So the buffer size, in this case, is '0x18' bytes
```

At the second iteration of Y\_image, second of X\_image, with y\_idx and x\_idx at 1, we have: - mod\_comp\_4 = 2 and mod\_comp\_8 = 1 because their values have been updated after the X\_image loop completed once - number\_of\_it = 1 because the X\_image is at the second iteration - standardized\_width = (X\_image \* x\_comp -1 + x\_MAX) / x\_MAX = (4 \* 2 -1 + 3) / 3 = 3 - Y\_done\_comp\_8 = y\_idx + mod\_comp\_8 = 1 + 1 = 2

So the buffer\_offset is equal to:

```
buffer_offset = (mod_comp_4 - mod_comp_8 + Y_done_comp_8)
               * standardized_width * 2 + (number_of_it * x_comp * 2)
               = (2 - 1 + 2) * 3 * 2 + (1 * 2 * 2) = 0x16 So we have that buffer's size at '0x18' bytes long and the offset at '0x16' bytes.
The buffer, after applying the offset, has only two bytes of space left. Since the buffer is accessed at '[5]' as a buffer of 'short', it
means that the buffer, after applying the offset, can only contains one element. Because we are accessing, after applying the offset, the
element at position '1' ('x_idx = 1') in a buffer of 'short', we have, at '[5]', a heap-based buffer overflow.
```

## CVE-2021-21947 - Precision greater or equal than 9

A specially-crafted JPEG file can lead to a heap-based buffer overflow in the JPEG lossless Huffman image parser, due to a missing boundary check.

Trying to load a malicious JPEG file, we end up in the following situation:

```
(730.9ec): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0bc9cffe ebx=00000000 ecx=00000000 edx=00000000 esi=00000001 edi=0bc9cffe
eip=707130eb esp=0019f940 ebp=0019fa88 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
igCore19d!IG_mpi_page_set+0xb70bb:
707130eb 66891c70      mov     word ptr [eax+esi*2],bx  ds:002b:0bc9d000=????
```

The access violation takes place at [6] in the process\_jpeg\_lossless function, when filling a word in a component's buffer when the SOF3's precision is greater than or equal to 9.

From the "seeking" of the component's buffer at [4] to [6] there is no boundary check on accessing the element at position x\_idx.

For example with:

```
Y_image = 0x22
X_image = 0x4
precision = 0xA
nr_comp = 2
COMP = {
    Horiz, Vert = 2, 2;
    Horiz, Vert = 3, 9;
}
```

The first component would have as size:

```
standardized_width = (X_image * subsampling_X -1 + x_MAX_sampling_factor) / x_MAX_sampling_factor
malloc_size         = (subsampling_X + subsampling_Y) * standardized_width * 2
                    = (2 + 2) * ((4 * 2 -1 + 3) / 3) * 2 = 0x18 The result is '0x18' because it is first calculated '((4 * 2 -1 + 3)/3)' as
an integer before the value is plugged into the formula.
```

At the second iteration of Y\_image, second of X\_image, with y\_idx and x\_idx at 1, we have: - mod\_comp\_4 = 2 and mod\_comp\_8 = 1 because their values have been updated after the X\_image loop completed once - number\_of\_it = 1 because the X\_image is at the second iteration - standardized\_width = (X\_image \* x\_comp -1 + x\_MAX) / x\_MAX = (4 \* 2 -1 + 3) / 3 = 3 - Y\_done\_comp\_8 = y\_idx + mod\_comp\_8 = 1 + 1 = 2

So the buffer\_offset is equal to:

```
buffer_offset = (mod_comp_4 - mod_comp_8 + Y_done_comp_8)
               * standardized_width * 2 + (number_of_it * x_comp * 2)
               = (2 - 1 + 2) * 3 * 2 + (1 * 2 * 2) = 0x16 So we have that buffer's size at '0x18' bytes long and the offset at '0x16' bytes.
The buffer, after applying the offset, has only two bytes of space left. Since the buffer is accessed at '[6]' as a buffer of 'short', the
buffer, after applying the offset, can only contain one element. Because we are accessing, after applying the offset, the element at
position '1' ('x_idx = 1') in a buffer of 'short', we have, at '[6]', an heap-based buffer overflow.
```

#### Timeline

2021-09-23 - Initial contact  
2021-09-24 - Vendor acknowledged and confirmed under review with engineering team  
2021-11-30 - 60 day follow up  
2021-12-07 - Vendor advised release planned for Q1 2022  
2021-12-07 - 30 day disclosure extension granted  
2022-01-06 - Final disclosure notification  
2022-02-23 - Public disclosure

#### CREDIT

Discovered by Francesco Benvenuto of Cisco Talos.

---

#### VULNERABILITY REPORTS

#### PREVIOUS REPORT

#### NEXT REPORT

TALOS-2021-1374

TALOS-2021-1377