

NASA's Software Security Vulnerabilities Found For Fun, Not Profit

How to discover more than dozen severe and critical security vulnerabilities in various NASA's in house developed software in only 4 hours

Leon Juranic (leon.juranic@threatleap.com)

Published: 05/27/2025

Long time ago in a galaxy far, far away...

Long time ago, in a galaxy far, far away, fifteen years ago (in 2009), I was a 25 year old hacker and cofounder of my first cybersecurity startup Infigo and just finished a one year long side project security research collaboration with NASA Goddard Space Flight Center. During the security research I discovered 12 dangerous security vulnerabilities in Common Data Format (CDF) software library (some of them critical severity).

NASA's CDF software library (<https://cdf.gsfc.nasa.gov/>) is according to its documentation developed and used by NASA and hundreds other government agencies, academic community and various organizations for the purpose (in very simple words) - of tracking objects locations in space.

When checking how widely CDF is used we can find detailed information in their FAQ section page.

<https://cdf.gsfc.nasa.gov/html/faq.html>

6. How widely used is CDF?

The CDF software package is used by hundreds of government agencies, universities, and private and commercial organizations as well as independent researchers on both national and international levels. CDF has been adopted by the International Solar-Terrestrial Physics (ISTP) project as well as the Central Data Handling Facilities (CDHF) as their format of choice for storing and distributing key parameter data.

CDF software is open source and available for anyone to download and use from NASA's website. Long time ago in 2009 I downloaded the CDF library and started to look around for security vulnerabilities in it.

Since it is written in C, back then in 2009, I was mostly looking for memory corruption vulnerabilities, most interesting and most dangerous vulnerabilities.

Combination of fuzzing and manual source code review resulted in total of 12 security vulnerabilities and at least 3 of them critical memory corruption / remote code execution, all of which I reported to NASA's CDF software developers and also wrote an exploit for one of the remote code execution vulnerabilities, just as Proof Of Concept.

This collaboration started with a single security vulnerability reported to NASA's CDF team and afterwards extended with more fuzzing and source code audit which I did. More vulnerabilities started popping up. Reporting them, verifying NASA's CDF developer's fixes for the reported vulnerabilities, confirmed fixes or suggested improvements, collaboration which lasted over one year and resulted in a total of 12 security vulnerabilities which were fixed.

In the end it resulted in a much more fortified and secure NASA's CDF software library and after all the fixes were implemented on their side, in coordination with NASA's CDF team with responsible full disclosure, I published a security advisory about CDF security vulnerabilities, covering in detail one of the discovered critical security vulnerabilities and briefly mentioning some other vulnerabilities.

Security advisory was published on Bugtraq mailing list - <https://seclists.org/bugtraq/2009/Jul/142> .

For the record, I still miss Bugtraq :-(even though in its final years it was far away from what it was during its golden years.

I recall that NASA's CDF developers were very friendly and cooperative when I was approaching and helping them to fortify the CDF software library.

I even acknowledged their collaboration efforts in previously mentioned security advisory published in July of 2009:

"We would like to thank the whole NASA CDF team, and especially Michael Liu for cooperation, and good work in dealing with reported vulnerabilities."

And NASA's CDF team also acknowledged my contributions to the CDF software security:

<https://cdf.gsfc.nasa.gov/html/acknowledgements.html> :

"The CDF team greatly appreciates the efforts of Mr. Leon Juranic, from Infigo Information Security <http://www.infigo.hr> , for reporting and testing the vulnerability in our code."

In The Meantime

Not long after publishing the previously mentioned NASA CDF security advisory in 2009, I parted ways with my first co-founded cybersecurity startup Infigo in order to start my next cybersecurity startup DefenseCode. Dedicated to building state-of-the-art SAST and DAST automated software tools.

During the following 12 years, I was - all around the clock - focused and dedicated to developing state-of-the-art SAST and DAST automated tools in DefenseCode.

Since there was, back then, a serious trend in going to “shift-left” and automated SAST and DAST tools along with software security overall and especially web applications becoming more and more popular, I saw my chance and place there.

Following that, there were 12 years during which I coded more than 18 Megabytes of raw source code for Static Application Security Testing software of full blown taint/data flow SAST analysis engines with thousands purely security-based rules for Java, C#, Python, Ruby, PHP, JavaScript, Groovy, PL/SQL, Golang, C/C++, VB.Net, ASP Classic, VBScript, Android Java, Objective-C, ColdFusion, Cobol, ABAP, Swift, Xamarin, Kotlin, R, Salesforce APEX and Visual Basic.

Beside SAST, there was also a DAST tool which I've developed, with support for classic web but also latest web technologies, like Web 2.0, HTML 5, etc. with usually more than 200-300 different security tests with advanced heuristics per single web application or API parameter/argument/header/path for mostly all of the vulnerabilities which could be found via DAST approach.

Even though we were a relatively small but high performing company, we also had Fortune 100 logos on our clients list.

So you could definitely say I paid my dues to Application Security.

Three years ago my Application Security SAST and DAST startup DefenseCode was acquired by a global AppSec leader/vendor in 2022.

Ok, so why are you writing this SAST/DAST stuff? Who cares about your AppSec journey?

Simply to point out that I know a thing or two about AppSec :)

Nowadays, my daily focus is on a new startup ThreatLeap (<https://www.threatleap.com/>) where I'm fully engaged in the position of technical and business advisor. ThreatLeap is an all around proactive security monitoring SaaS solution for organizations digital security posture and digital footprint with mission of discovering threats and vulnerabilities of different kinds and types in a proactive manner in order for organizations to stay one step ahead of threat actors.

Now, let's get back to NASA

Three months ago during the still of the night, I got bored listening to music, didn't feel like taking a guitar in my hands. Instead I did what I used to do a long time ago, but much more rarely nowadays, I downloaded some software from Github and started to look for security vulnerabilities in it, purely by accident it was related to NASA again.

Actually not developed by NASA, but software which was mentioned by NASA - NetCDF, written in C++ and originating in the academic community, IIRC.

After quick googling, it turned out that NetCDF software was fuzzed to death by OSS-Fuzz, Google backed project - <https://nvd.nist.gov/vuln/detail/CVE-2019-25050>.

Decision was quick - I gave up. :) It's important to know how to choose your battles. I'm looking for something with low hanging fruits (vulnerabilities which are easier to find but of a high/critical risk) because I'm doing it for fun, not looking into something stressed out to death by Google backed fuzzing.

I'm not as eager as I was 15 years ago to spend a year looking for vulnerabilities in something that I probably won't even get paid for, even though I'm doing it for fun this time. :)

During the 15 years from 2009 till 2024 there was an enormous change and shift in what hackers are mostly looking for and exploiting these days. Along with "good old pals", brand new classes of security vulnerabilities and adversary techniques emerged during those 15 years. Technology has advanced significantly and the security posture of companies and organizations is much more different these days than what it was a decade and half ago.

So, Instead of beating the dead horse - NetCDF, I turned to NASA's public software repositories published on Github under NASA's official account (<https://github.com/NASA>) which was referenced from <https://code.nasa.gov/> and <https://software.nasa.gov/>.

Software developed by NASA and used by NASA and anyone else who cloned or forked these repositories from NASA's Github account.

In the first run, I've downloaded roughly 10 NASA's applications and started to manually check the source code for security vulnerabilities. Just as an empirical test to see how long it will take me to potentially find some (if any) vulnerability in NASA's software published on Github, 15 years later after the NASA CDF research, whatever vulnerability it could be.

I've set a time limit of two hours of code audit and as you can guess, in the beginning I was focused mostly on web applications, with a slight twist at the half of the analysis process, when I decided to invest two more hours into auditing NASA's software written in C/C++ :)

I used a combination of manual source code review and quick pure grep-ing for stuff which I know that could be security questionable in some given technology stack.

At the web site <https://code.nasa.gov/> there is a catalog of a NASA's software with a reference to an official GitHub repository (<https://github.com/nasa/>) and description of NASA's Open Source Software which was approved to be published online on the NASA's GitHub account.



Before publishing source code on GitHub, NASA's source code has to get approval by NASA's SRA (Software Release Authority) at <https://softwarerelease.ndc.nasa.gov/>. The whole process is described on the following URL: <https://code.nasa.gov/#/guide>. So I started to look into NASA's software referenced from <https://code.nasa.gov/> and published on GitHub.

Some of these software applications which I audited are older than me. NASA often uses the same in house developed software over many decades, like for example Voyager space probe which was sent to space in 1977 is still, to this very day, based and operated on FORTRAN and (allegedly) assembler.

What I discovered in the end were vulnerabilities of various classes and types, ranging from Cross Site Scripting and Secrets Leaks to Buffer Overflows. You can find more details about my vulnerability findings below, chronologically as I discovered them, not necessarily sorted by severity and/or risk/criticality level.

NASA's GeoRef Software - Reflected Cross Site Scripting Security Vulnerability

After quickly checking couple of NASA's applications, soon I stumbled across NASA's GeoRef software

So, what is it all about?

Taken from the NASA's web page (<https://software.nasa.gov/software/ARC-17943-1>):

"GeoRef is a Web-based software application designed to increase the efficiency and precision in geo-locating photographs taken by astronauts from the International Space Station. GeoRef provides highly automated processes for: (1) calculating the latitude and longitude coordinates of the center point of the image, and (2) producing geo-referenced map overlays for the image. The georeferenced images produced by GeoRef are designed to support the needs of educational, Earth science, and disaster response users."

Download link for GeoRef v1 software from:

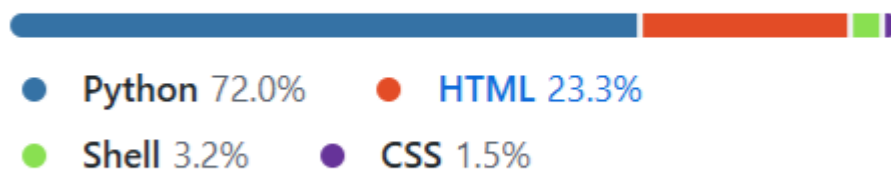
<https://software.nasa.gov/software/ARC-17943-1>

- is pointing to NASA's Github account, here:

<https://github.com/nasa/georef>

NASA's GeoRef software is mostly written in Python and HTML.

Languages

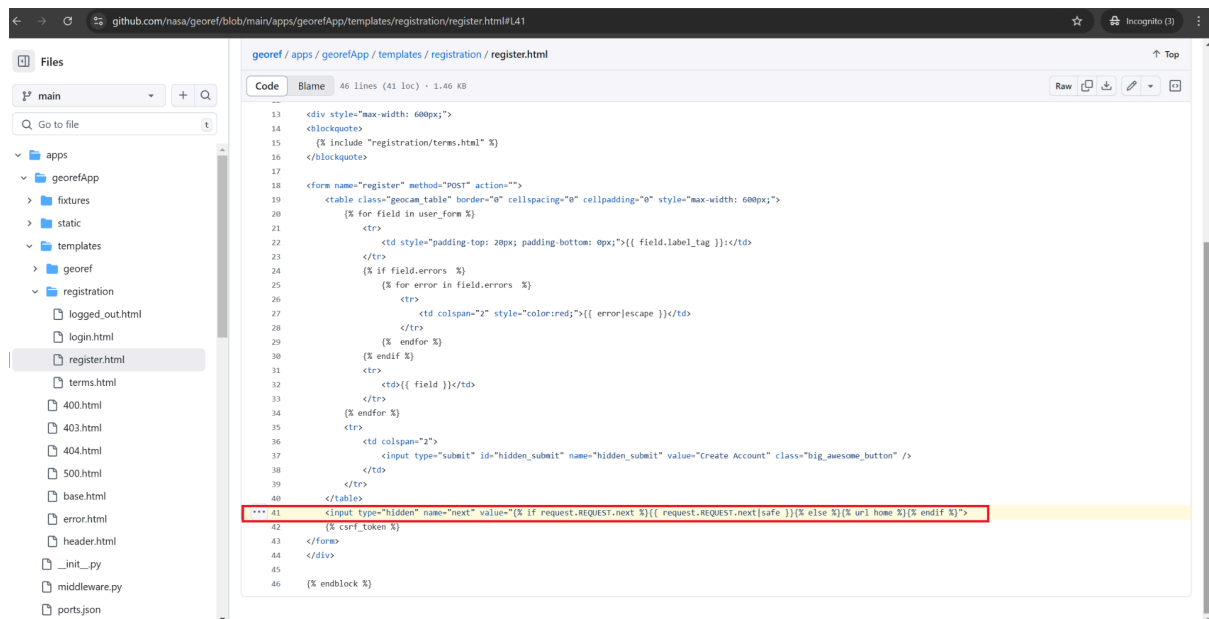


So I downloaded software from Github and the first thing which I did was to search for common Python pitfalls.

Looking through the source code and grep-ing stuff. Nothing more but VIM and grep, party like it's the 80's :)

It took me 30 minutes to find this interesting piece of the source code.

<https://github.com/nasa/georef/blob/main/apps/georefApp/templates/registration/register.html#L41>



```
13 <div style="max-width: 600px;">
14 <blockquote>
15 {% include "registration/terms.html" %}
16 </blockquote>
17
18 <form name="register" method="POST" action="">
19 <table class="geocam_table" border="0" cellspacing="0" cellpadding="0" style="max-width: 600px;">
20 {%- for field in user_form %}
21 <tr>
22 <td style="padding-top: 20px; padding-bottom: 0px;"{{ field.label_tag }}</td>
23 </tr>
24 {%- if field.errors %}
25 {%- for error in field.errors %}
26 <tr>
27 <td colspan="2" style="color: red;"{{ error|escape }}</td>
28 </tr>
29 {%- endfor %}
30 {%- endif %}
31 <tr>
32 <td{{ field }}</td>
33 </tr>
34 {%- endfor %}
35 <tr>
36 <td colspan="2">
37 <input type="submit" id="hidden_submit" name="hidden_submit" value="Create Account" class="big_awesome_button" />
38 </td>
39 </tr>
40 </table>
41 *** 41 <input type="hidden" name="next" value="{% if request.REQUEST.next %}{{
42 request.REQUEST.next|safe }}{% else %}{{ url home }}{% endif %}">
43 {%- csrf_token %}
44 </form>
45 </div>
46 {%- endblock %}
```

Line 41 of template file register.html contains this interesting piece of the source code:

```
<input type="hidden" name="next" value="{% if request.REQUEST.next %}{{
request.REQUEST.next|safe }}{% else %}{{ url home }}{% endif %}">
```

So, in the end, it took me 30 minutes of overall time spent on a few NASA's applications to find a Reflected Cross Site Scripting security vulnerability in NASA's GeoRef v1 software purely by manual inspection.

Previous source code line will check if HTTP GET/POST request parameter "next" is set and if it is, it will place it in a Jinja or Django template response with one very important markup - JavaScript/HTML - "|safe".

Someone would maybe assume that this "|safe" markup is making this code secure, but it's quite the opposite, sometimes technical things can be quite misleading and non/counter intuitive.

This peculiar little "|safe" thing will automatically disable any Python (by default enabled) Cross Site Scripting vulnerability prevention measures and protections and allow any dangerous HTML characters to be reflected in the server's response, since the Python templating engine is declaring tainted user input parameter named "next" as Cross Site Scripting concerns - Safe.

What it means - it means that it is possible to set the GET/POST parameter "next" to a variation of "<script>alert(1)</script>" and it will be reflected exactly as it is and execute given JavaScript alert(1).

Of course, nowadays it depends on the client's browser environment and settings, but in cases when the client's browser doesn't have Reflected XSS protection in place, alert(1) window will popup.

Even with client's browser protection for JavaScript, HTML code will still go through without a problem, so it is a classic Reflected Cross Site Scripting.

Since this is a register.html template used to register accounts which probably doesn't require any form of prior authentication, these sorts of Cross Site Scripting vulnerabilities are usually classified as High Severity Reflected Cross Site Scripting vulnerabilities.

Moreover, when I searched Github for a code sample which is the root cause of this Cross Site Scripting vulnerability in GeoRef software, I've found this vulnerable code line in 5 more repositories/locations (probably somehow related to NASA's GeoRef software but didn't do more research on it).

<https://github.com/geocam/geocamMapFasten/blob/faa8f0c963a76cd72d0e694385e0e58e35b6500e/apps/mapFastenApp/templates/registration/register.html>

<https://github.com/geocam/geocamResponderMaps/blob/19a87410bc0887b8c01ae565a6ab15f2e2a7c26e/apps/responderMaps/templates/registration/register.html>

https://github.com/ep-infosec/28_nasa_georef/blob/0d3e722d66eb721449d0133e472826e97130dc63/apps/georefApp/templates/registration/register.html

<https://github.com/afcarl/georef/blob/25d3e7ae6dd3ea2a40e8ecd7c74d1a86b9404e54/apps/georefApp/templates/registration/register.html>

<https://github.com/geocam/geocamResponderMaps/blob/19a87410bc0887b8c01ae565a6ab15f2e2a7c26e/apps/responderMaps/templates/landing/index.html>

NASA's CMR-OpenSearch - "Secrets Leak"

Software Security Issue

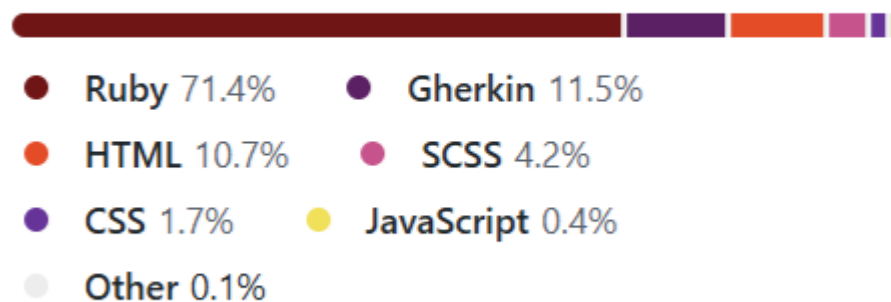
After checking a few more software applications from NASA's Github repositories, I've stumbled across another interesting <https://github.com/nasa/cmr-opensearch/> repository - CMR-OpenSearch application.

As described on the Github page:

"CMR-OpenSearch is a web application developed by NASA EOSDIS to enable data discovery, search, and access across the CMR Earth Science data holdings via the OpenSearch standard."

NASA's CMR-OpenSearch software is mostly written in Ruby:

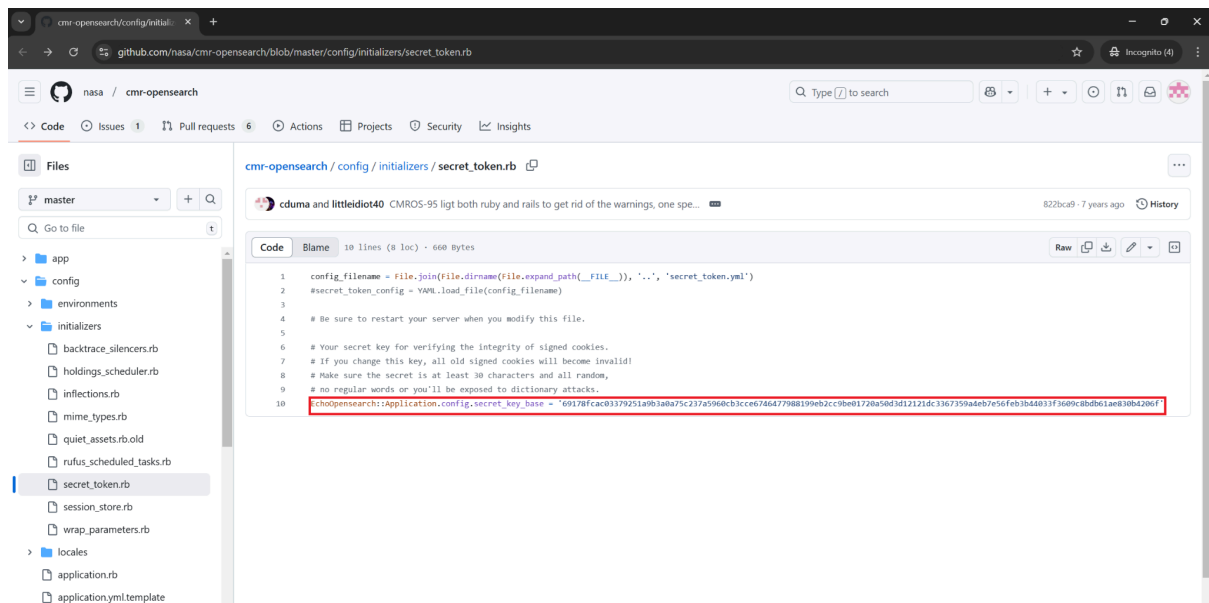
Languages



Again, with VIM and grep I've looked the old-fashioned way for common Ruby applications pitfalls that could endanger the security of this software or NASA's systems and organizations using this software.

Pretty quickly I've stumbled across this file:

https://github.com/nasa/cmr-opensearch/blob/master/config/initializers/secret_token.rb#L10



It took me only about 10 minutes of auditing this application before stumbling across a security issue/data/secret leak.

On line 10 of secret_token.rb we can see secret_key_base value is hardcoded in Ruby code and publicly disclosed in the application's Github repository. It's unclear if this is secret_key_base used by NASA, their software developers or their contractors, but these kinds of vulnerabilities are classified as "Secrets Leaks".

```
# Your secret key for verifying the integrity of signed cookies.  
# If you change this key, all old signed cookies will become invalid!  
# Make sure the secret is at least 30 characters and all random,  
# no regular words or you'll be exposed to dictionary attacks.
```

```
EchoOpensearch::Application.config.secret_key_base =  
'69178fcac03379251a9b3a0a75c237a5960cb3cce6746477988199eb2cc9be01720a  
50d3d12121dc3367359a4eb7e56feb3b44033f3609c8bdb61ae830b4206f'
```

It doesn't say without a reason in the comments above the hardcoded key - "Your secret key". It is supposed to be secret, not publicly available on Github.

Public "secrets leaks" security issues became mostly popular with the rise of Github, and it happens when developers or companies/organizations publish their source code along with (accidentally) publishing hardcoded sensitive/secret key values publicly on Github, usually without intention of doing so, for anyone to see it.

"secret_key_base" token is used for session encryption, message verifiers, password resets and signed/encrypted cookies.

Leaked secret_key_base tokens values can be used for session hijacking, tampering session cookies, password reset token exploits, encrypted data breaches and replay attacks.

If secret_key_base is not changed in the production and it doesn't have to be, especially since it is hardcoded in the Ruby code and also publicly disclosed, it can pose a significant danger to every system which is using these applications, both NASA and other organizations using this software.

Danger of leaked sensitive Ruby secret_key_base key can be found explained in details in the following link:

<https://www.gitguardian.com/remediation/rails-secret-key-base>

NASA's CMR-CSW Software - "Secrets Leak" Security Issue

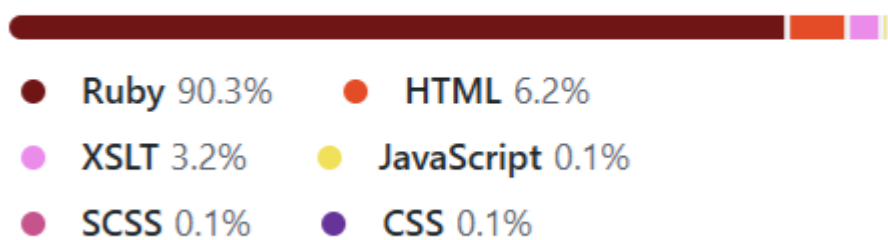
Since there was another NASA's Ruby application which I've downloaded, CMR-CSW <https://github.com/nasa/cmr-csw/> I've looked for similar problems as in the previous Ruby application.

Description from the Github:

"CMR-CSW is a web application developed by NASA EOSDIS to enable data discovery, search, and access across the CMR Earth Science data holdings via the OpenGIS Catalogue Service for the Web (CSW) standard."

Again, mostly Ruby language.

Languages



Quick search for hardcoded keys again resulted in an interesting discovery.

https://github.com/nasa/cmr-csw/blob/master/config/initializers/secret_token.rb#L7

The screenshot shows the GitHub repository for NASA's CMR-CSW. The file `secret_token.rb` is open, showing a hardcoded secret key. The key is a long alphanumeric string, and the comment above it states: "Make sure the secret is at least 30 characters and all random, no regular words or you'll be exposed to dictionary attacks." The key is highlighted with a red box.

```
1 # Be sure to restart your server when you modify this file.
2
3 # Your secret key for verifying the integrity of signed cookies.
4 # If you change this key, all old signed cookies will become invalid!
5 # Make sure the secret is at least 30 characters and all random,
6 # no regular words or you'll be exposed to dictionary attacks.
7 *** 7 csw:Application.config.secret_key_base = "2befSecF972567631e8c0b45d27c1503128a0d70bf8035c0241cbe0b40df624b7fa391f3a40d37538c29d6c83c1adfe6003926435abd12cfa13a211a"
```

On line 7 of secret_token.rb there is again hardcoded secret_key_base value:

```
# Your secret key for verifying the integrity of signed cookies.  
# If you change this key, all old signed cookies will become invalid!  
# Make sure the secret is at least 30 characters and all random,  
# no regular words or you'll be exposed to dictionary attacks.  
Csw::Application.config.secret_key_base =  
'2bef5ecf97256763b1e8c9b45d27c1503128a08d7bbf8835c0241cbce80b40df624b7f  
a391f3ae403d37538c29d6c83c1adfeb6803926435abd12cafa13a211a'
```

Another “Secrets Leaks” security issue/data leak of secret_key_base value, same as in previous case.

Same principles and security implications applies to this one as to previous issue:

<https://www.gitguardian.com/remediation/rails-secret-key-base>

NASA's Portable Environment for Quick Image Processing (QulP) Remote Buffer Overflow

Since I've decided to invest an additional two hours in NASA's GitHub software audit, I've started to look for C/C++ applications and memory corruption issues.

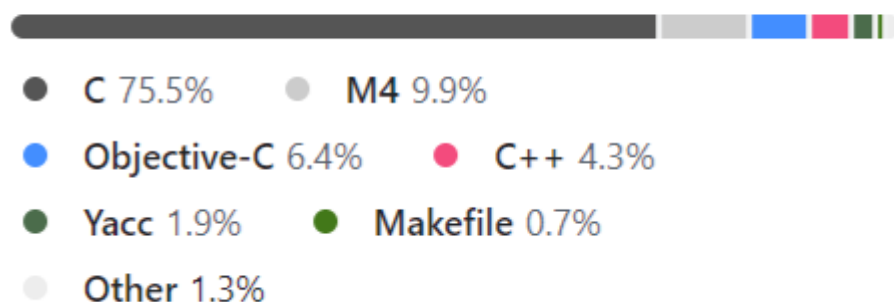
I've stumbled across <https://github.com/nasa/QulP> repository which is Portable Environment for Quick Image Processing (QulP). Image data parsing and processing overall can be quite complicated and challenging and has a long history of memory corruption issues in all sorts of platforms, software, libraries and Operating Systems, so I started to dig.

QulP software as described on <https://software.nasa.gov/software/ARC-16295-1A> or <https://code.nasa.gov/> :

"The QulP interpreter, a software environment for QUick image processing, uses an interactive scripting language designed to facilitate use by non-expert users, through features such as context-sensitive automatic response completion and integrated documentation. The package includes a number of script packages that implement high-, medium-, and low-level functions (e.g., analysis of eye images for human gaze tracking, feature tracking, and image filtering). The environment also includes facilities for displaying images on screen, drawing and overlaying graphics, and constructing graphical user interfaces using the scripting language. Currently supported platforms are *NIX (tested on Mac OS X and Linux), and Apple iOS."

QulP software is mostly written in C language:

Languages



So I've started to look for some stack buffers and inherently insecure C/C++ functions and memory operations. It took me about 45 minutes of grepping and manual review to stumble across an interesting code construct.

I've stumbled across HIPS file format processing code. Since I've never heard of HIPS / HIPS2 file format I've asked ChatGPT what it stands for and here is ChatGPT's answer:



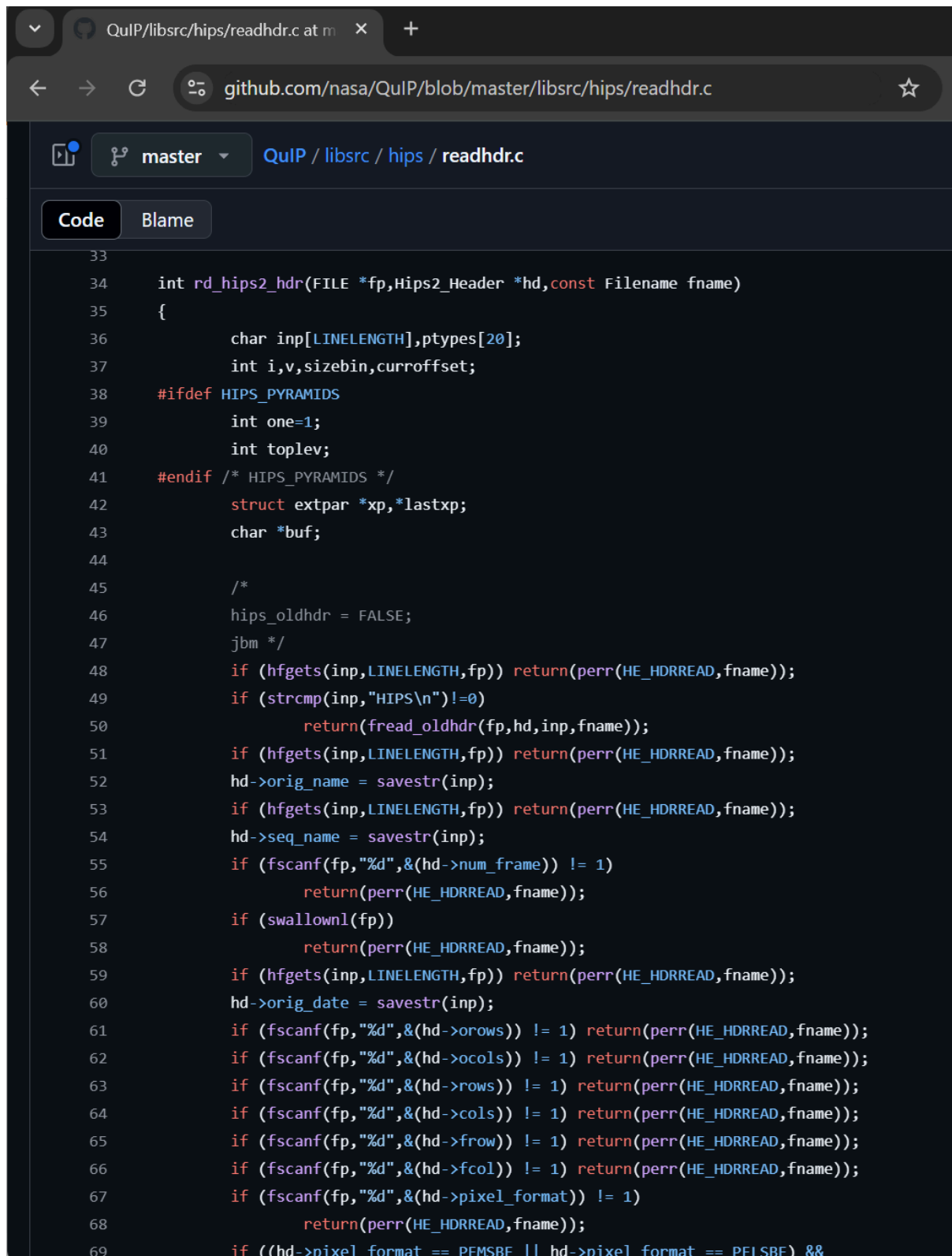
The HIPS-2 file format is likely related to the **Hierarchical Image Processing System (HIPS-2)**, an image processing and computer vision framework used in research and medical imaging. However, detailed documentation on the specific **HIPS-2 file format** is limited.

Possible Characteristics of HIPS-2 File Format

- **Image Processing Focus:** Used for storing image data in hierarchical structures for image analysis and manipulation.
- **Medical & Scientific Use:** Could be related to medical imaging, remote sensing, or computer vision research.
- **Raw or Encoded Data:** Might store images in an uncompressed or minimally processed format, making it suitable for high-performance processing.
- **Successor to HIPS Format:** HIPS-2 may be an updated version of the **original HIPS (Hierarchical Image Processing System)** format, used for handling sequences of images efficiently.

When I checked the source code, yes, it looks like ChatGPT was right.

File <https://github.com/nasa/QuIP/blob/master/libsrc/hips/readhdr.c> was reading HIPS file header from the file with pointer to FILE structure of previously opened HIPS/HIPS2 file.



```
33
34  int rd_hips2_hdr(FILE *fp,Hips2_Header *hd,const Filename fname)
35  {
36      char inp[LINELENGTH],ptypes[20];
37      int i,v,sizebin,curroffset;
38      #ifdef HIPS_PYRAMIDS
39          int one=1;
40          int toplev;
41      #endif /* HIPS_PYRAMIDS */
42      struct extpar *xp,*lastxp;
43      char *buf;
44
45      /*
46      hips_oldhdr = FALSE;
47      jbm */
48      if (hfgets(inp,LINELENGTH,fp)) return(perr(HE_HDRREAD,fname));
49      if (strcmp(inp,"HIPS\n")!=0)
50          return(fread_oldhdr(fp,hd,inp,fname));
51      if (hfgets(inp,LINELENGTH,fp)) return(perr(HE_HDRREAD,fname));
52      hd->orig_name = savestr(inp);
53      if (hfgets(inp,LINELENGTH,fp)) return(perr(HE_HDRREAD,fname));
54      hd->seq_name = savestr(inp);
55      if (fscanf(fp,"%d",&(hd->num_frame)) != 1)
56          return(perr(HE_HDRREAD,fname));
57      if (swallownl(fp))
58          return(perr(HE_HDRREAD,fname));
59      if (hfgets(inp,LINELENGTH,fp)) return(perr(HE_HDRREAD,fname));
60      hd->orig_date = savestr(inp);
61      if (fscanf(fp,"%d",&(hd->orows)) != 1) return(perr(HE_HDRREAD,fname));
62      if (fscanf(fp,"%d",&(hd->ocols)) != 1) return(perr(HE_HDRREAD,fname));
63      if (fscanf(fp,"%d",&(hd->rows)) != 1) return(perr(HE_HDRREAD,fname));
64      if (fscanf(fp,"%d",&(hd->cols)) != 1) return(perr(HE_HDRREAD,fname));
65      if (fscanf(fp,"%d",&(hd->frow)) != 1) return(perr(HE_HDRREAD,fname));
66      if (fscanf(fp,"%d",&(hd->fcol)) != 1) return(perr(HE_HDRREAD,fname));
67      if (fscanf(fp,"%d",&(hd->pixel_format)) != 1)
68          return(perr(HE_HDRREAD,fname));
69      if ((hd->pixel format == PFMSBF || hd->pixel format == PFLSBF) &&
```

On line 34, we can see rd_hips2_hdr() function

```
int rd_hips2_hdr(FILE *fp,Hips2_Header *hd,const Filename fname)
```

In the beginning of the file header processing we can see some common operations like reading the file header, reading of various data, and it's all stored in a buffer on stack 'char inp[LINELENGTH]' and in various variables in data structure Hips2_Header.

We can also see the 'char ptypes[20]' buffer also declared on stack, which we will come back to again later.

From a quick overview, everything seems safe so far.

Code is using a custom hfgets() function which is actually a wrapper around a "safe" fgets() function which has boundary limits when reading from file into a buffer, so everything is fine so far.

inp buffer LINELENGTH is defined in file:

<https://github.com/nasa/QuIP/blob/master/include/hips/hip2hdr.h#L88>

```
#define LINELENGTH 400
```

Some things in HIPS2 header code reading data potentially (fleeting glimpse) looks like integer overflows but hey, I have a limited time span of only 2 hours :)

But things gets interesting when going little bit below initial header data reading:

<https://github.com/nasa/QuIP/blob/master/libsrc/hips/readhdr.c#L125>

```
116         for (i=0;i<hd->numparam;i++) {
117             xp = (struct extpar *)getbuf(sizeof(*xp));
118             if (i==0)
119                 lastxp = hd->params = xp;
120             else {
121                 lastxp->nextp = xp;
122                 lastxp = xp;
123             }
124             xp->nextp = NULLPAR;
125             if (fscanf(fp,"%s %s %d",inp,ptypes,&(xp->count)) != 3)
126                 return(perr(HE_HDRPREAD,fname));
127             xp->name = savestr(inp);
128             switch(ptypes[0]) {
129                 case 'c': xp->format = PFASCII; break;
130                 case 'b': xp->format = PFBYTE; break;
131                 case 'i': xp->format = PFINT; break;
132                 case 'f': xp->format = PFFLOAT; break;
133                 case 's': xp->format = PFSHORT; break;
134                 default: return(perr(HE_HDRPTYPES,ptypes,fname));
```

On line 125 of file readhdr.c

(<https://github.com/nasa/QuIP/blob/master/libsrc/hips/readhdr.c#L125>) we can see plain simple, inherently insecure fscanf() function reading data with %s format strings into previously mentioned buffers on stack 'inp' and 'ptypes' and extpar struct 'count' variable.

```
if (fscanf(fp,"%s %s %d",inp,ptypes,&(xp->count)) != 3)
```

That means that this `fscanf()` function will read as much data as it can from file into 'inp' and 'ptypes' buffers, before it stumbles across blank space or newline character, without respecting any (inp or ptypes) destination buffers memory boundaries.

What we see here is **plain old, straight from a textbook, vanilla stack based buffer overflow**.

[illegible]

This line in the correctly formatted HIPS2 file (HIPS2 file header should be correct) which will pass the initial checks will overflow both 'inp' and 'ptypes' buffers on stack causing stack based buffer overflow.

If a malicious HIPS2 file comes from an untrusted source as in over an email or from the web, QulP software can potentially be exploited to achieve stack based buffer overflow and consequently remote arbitrary code execution when processing a malformed and malicious HIPS2 file.

What is interesting, QuIP software has been in development by NASA for roughly four decades. The development started even before I was born :) Something like The Rolling Stones would say, but a little bit different - *Pleased to meet you, hope you guess my age :)*

Just out of curiosity, when asking ChatGPT if NASA's QuIP software is still in use nowadays, this is the response:



Yes, NASA's **QuIP** (Quick Image Processing) software is still in use and actively maintained. It is an interactive environment for image processing and related applications, such as eye movement analysis. QuIP provides tools for computing and presenting images and image sequences, manipulating and storing arbitrary data, and general scientific computing and plotting. The software supports Unix-like operating systems (tested on Linux and macOS) and Apple's iOS mobile operating system. Additionally, QuIP supports GPU acceleration with either CUDA or OpenCL. [GITHUB.COM](#)

The QuIP interpreter features an interactive scripting language designed to facilitate use by non-expert users through features such as context-sensitive automatic response completion and integrated documentation. The package includes a number of script packages that implement high-level functions such as analysis of eye images for human gaze tracking, medium-level functions such as feature tracking, and low-level functions such as image filtering. [SOFTWARE.NASA.GOV](#)

QuIP is freely available under the NASA Open Source Agreement and can be accessed through its GitHub repository. [GITHUB.COM](#)

For more information and to download QuIP, you can visit the [NASA Software Catalog](#).

Sources



Conclusion - I have strong feelings that there are more vulnerabilities in QuIP software than just this one described, but as stated, I've limited my total overall time focus to two hours to see what I can find in NASA's repos in such a short time period.

NASA's Vehicle Sketch Pad (VSP) Remote Buffer Overflow

After finding basic vanilla stack buffer overflow in file processing in the previous example **I was curious if there are any other similar cases**, so I searched NASA's GitHub account for similar cases of file processing operations with insecure fscanf() functions.

Main motive and reasoning behind that was that all over different NASA's code in GitHub repositories there is a whole bunch of NASA's specific file formats processing, and **maliciously constructed data files can easily be slipped to the victim over an email or over the web**. Along with the fact that inherently insecure fscanf() with %s format string is a 90% probability **security failure**.

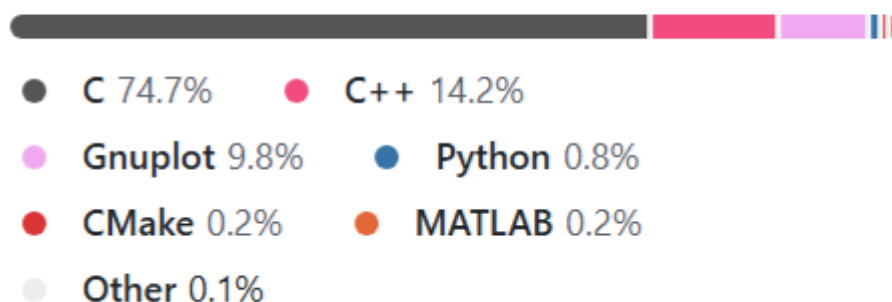
I quickly stumbled across NASA's Vehicle Sketch Pad (VSP) code repository: <https://github.com/nasa/OpenVSP/>

As described on <https://code.nasa.gov/> :

"The Vehicle Sketch Pad (VSP) is an aircraft geometry tool for rapid evaluation of advanced design concepts. Fast and accurate geometry modeling allows the designer to use more complex analysis methods earlier in the design process and reduces reliance on empiricism in conceptual design. VSP includes tools to model and export the internal structural layout."

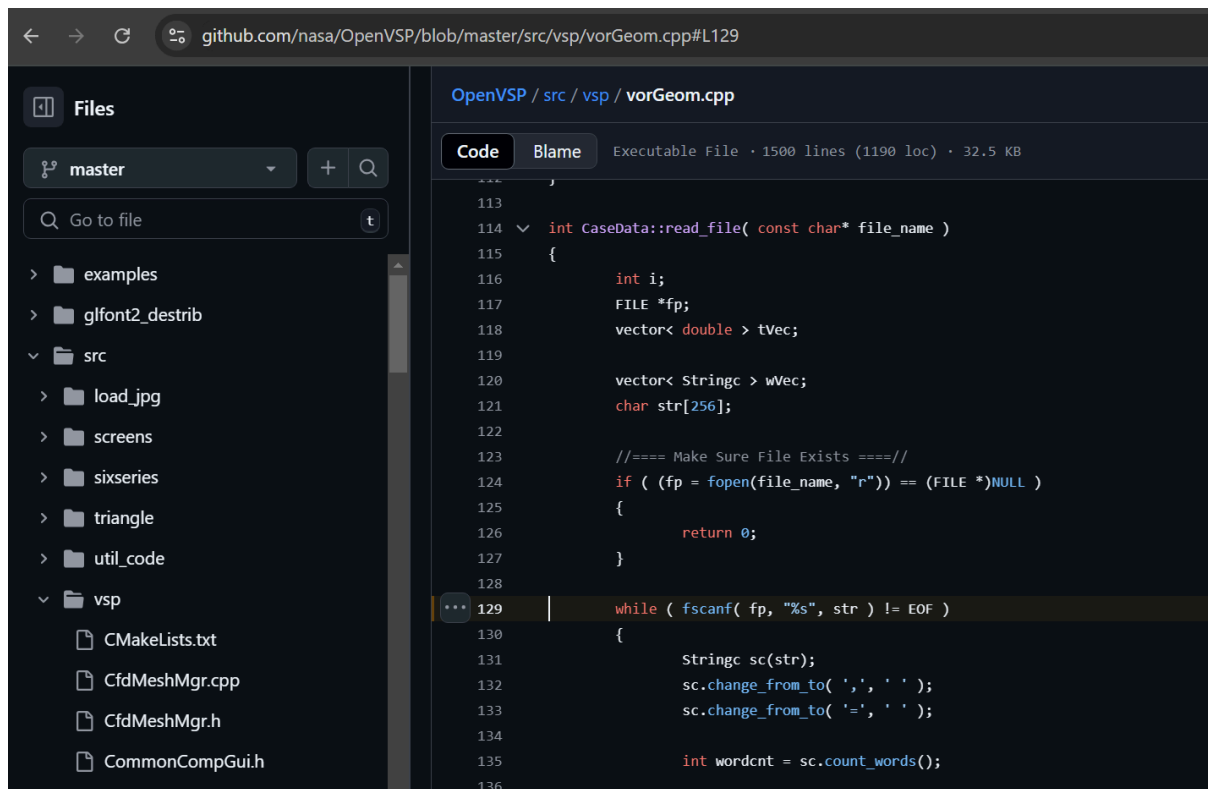
OpenVSP is mostly written in C and C++ languages.

Languages



After a five minute looking around the code, I quickly stumbled across another stack based buffer overflow in processing files, similar to previously described.

In following file <https://github.com/nasa/OpenVSP/blob/master/src/vsp/vorGeom.cpp#L114> there is a read_file() method/function declared which reads a file and processes its content.



```
113
114  int CaseData::read_file( const char* file_name )
115  {
116      int i;
117      FILE *fp;
118      vector< double > tVec;
119
120      vector< Stringc > wVec;
121      char str[256];
122
123      //==== Make Sure File Exists ====
124      if ( (fp = fopen(file_name, "r")) == (FILE *)NULL )
125      {
126          return 0;
127      }
128
129      while ( fscanf( fp, "%s", str ) != EOF )
130      {
131          Stringc sc(str);
132          sc.change_from_to( ',', ' ' );
133          sc.change_from_to( '=', ' ' );
134
135          int wordcnt = sc.count_words();
136
```

Again, we can see a stack buffer on line 121, named 'str' declared as 'char str[256]' which is later used on code line 129 in file vorGeom.cpp - <https://github.com/nasa/OpenVSP/blob/master/src/vsp/vorGeom.cpp#L129> as destination buffer for fscanf() function reading the file data without checking any buffer boundaries.

```
while ( fscanf( fp, "%s", str ) != EOF )
```

When looking around the code calling read_file() it seems that these files are expected to have ".cas" file extension.

If .cas file has a data line longer than 256 characters it will overflow the 'str' buffer causing **another vanilla stack based buffer overflow**.

If a malicious CAS file comes from an untrusted source as in over an email or from the web, OpenVSP software can potentially be exploited to achieve stack based buffer overflow and consequently remote arbitrary code execution when processing a malformed and malicious CAS file.

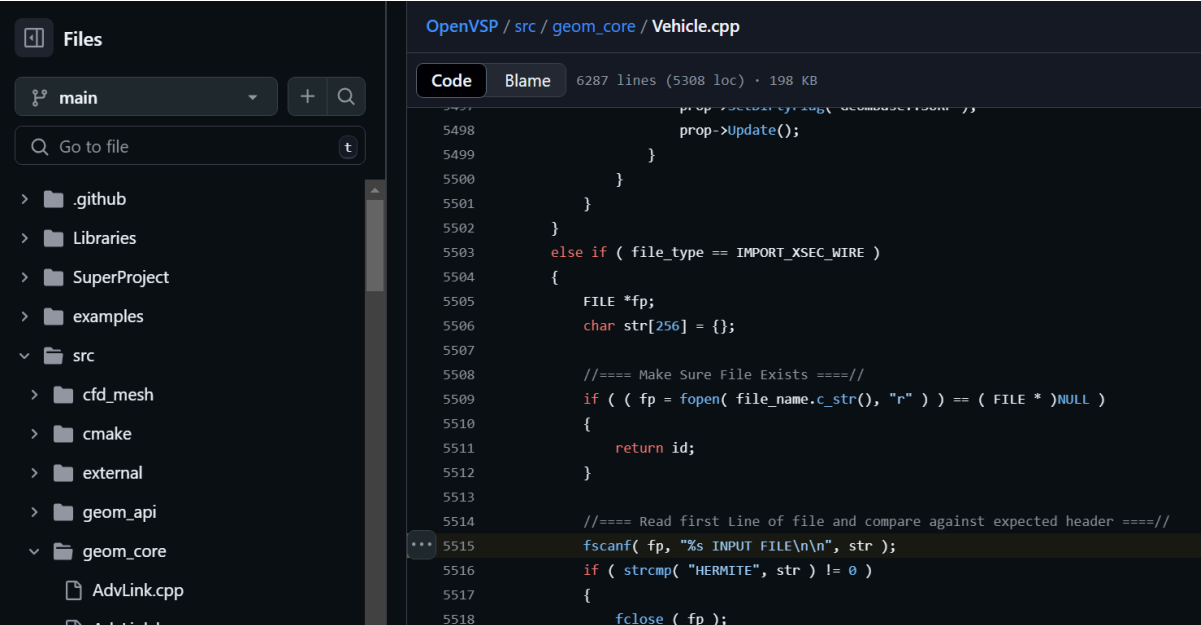
Apparently, there is a notice in a previous repository on GitHub which I didn't see before - "This repository has been archived by the owner on Sep 8, 2018. It is now read-only."

However it says that <https://openvsp.org/> is now the official place for new versions of OpenVSP. Little bit of a look around it. Ok, source code link is pointing to another GitHub repository: <https://github.com/OpenVSP/OpenVSP>

Codebase seems to be updated recently and it took me exactly 1 minute to find another fscanf() stack based buffer overflow in the file processing code.

In [file https://github.com/OpenVSP/OpenVSP/blob/main/src/geom_core/Vehicle.cpp#L5515](https://github.com/OpenVSP/OpenVSP/blob/main/src/geom_core/Vehicle.cpp#L5515) on line 5515, there is again vanilla stack based buffer overflow when reading from file data into 'char str[256]' stack buffer, causing again buffer overflow.

```
fscanf( fp, "%s INPUT FILE\n\n", str );
```



```
OpenVSP / src / geom_core / Vehicle.cpp
Code Blame 6287 lines (5308 loc) · 198 KB
5498         prop->SetData( prop->GetGeometry() );
5499         prop->Update();
5500     }
5501 }
5502 }
5503 else if ( file_type == IMPORT_XSEC_WIRE )
5504 {
5505     FILE *fp;
5506     char str[256] = {};
5507
5508     //==== Make Sure File Exists ====//
5509     if ( ( fp = fopen( file_name.c_str(), "r" ) ) == ( FILE * )NULL )
5510     {
5511         return id;
5512     }
5513
5514     //==== Read first line of file and compare against expected header ====//
5515     fscanf( fp, "%s INPUT FILE\n\n", str );
5516     if ( strcmp( "HERMITE", str ) != 0 )
5517     {
5518         fclose ( fp );
```

If a malicious file comes from an untrusted source as in over an email or from the web, latest OpenVSP software can potentially be exploited to achieve stack based buffer overflow and consequently remote arbitrary code execution when processing a malformed and malicious file.

When asking ChatGPT if NASA's OpenVSP software is still in use, here is ChatGPT's response:



Yes, NASA's **OpenVSP** (Open Vehicle Sketch Pad) software is still actively used and maintained. As of February 4, 2025, the latest version, **OpenVSP 3.42.2**, was released, addressing critical issues and enhancing stability. [OPENVSP.ORG](https://openvsp.org)

OpenVSP is a parametric aircraft geometry tool that allows users to create 3D models of aircraft defined by common engineering parameters. These models can be processed into formats suitable for engineering analysis. [OPENVSP.ORG](https://openvsp.org)

To support users of all experience levels, NASA offers the **OpenVSP Ground School**, an online training program providing tutorials, demonstrations, and example files. This resource is designed to help users learn the basics and explore advanced features of OpenVSP. [VSPU.LARC.NASA.GOV](https://vspu.larc.nasa.gov)

Additionally, annual **OpenVSP Workshops** are held to educate users about ongoing developments and features, providing hands-on experience with the software. [VSPU.LARC.NASA.GOV](https://vspu.larc.nasa.gov)

The active development, recent releases, and ongoing educational initiatives indicate that OpenVSP remains a vital tool in the aerospace design community.

Sources  



NASA's Regional Hydrologic Extremes Assessment System (RHEAS) Software Framework Remote Buffer Overflow

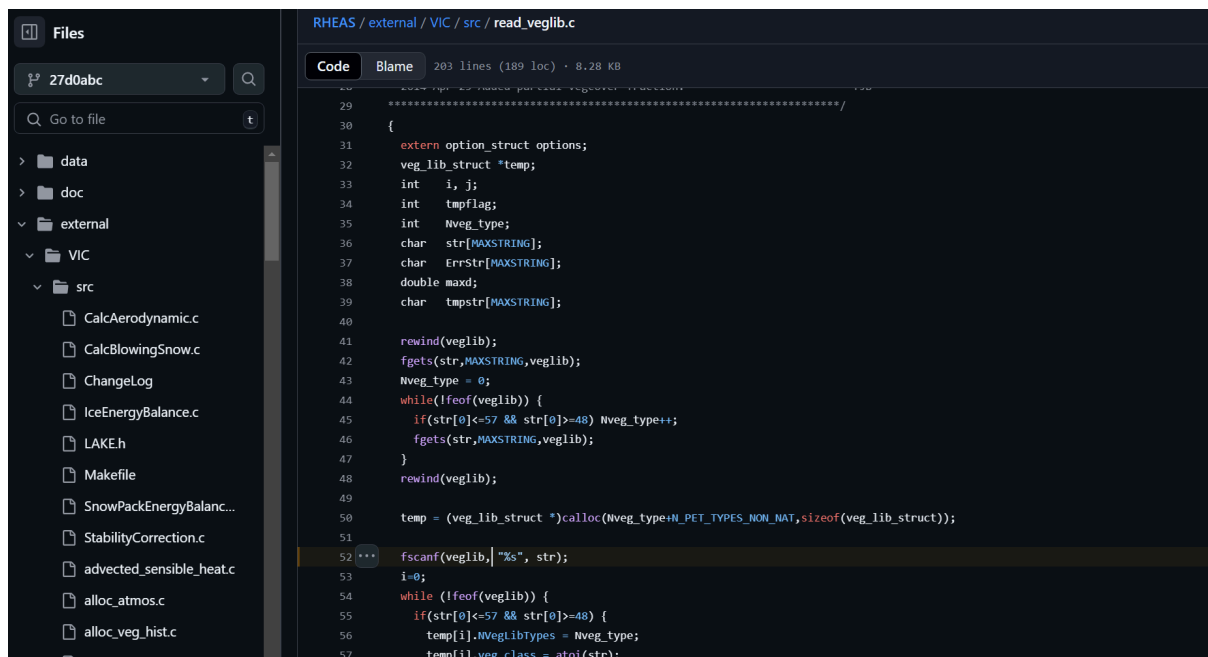
Looking, searching and grepping for more fscanf() stack based buffer overflows I've stumbled across NASA's RHEAS software framework - <https://github.com/nasa/RHEAS/>.

RHEAS software as described on <https://code.nasa.gov/> :

“Automates the deployment of nowcasting and forecasting hydrologic simulations and ingests satellite observations (through data assimilation). Allows coupling of other environmental models. Also facilitates delivery of data products to users via a GIS-enabled database. users of project outputs through CCAFS network of partners. Our project objectives, and approach are integral with the goals of SERVIR.”

Long story short, another fscanf() stack based buffer overflow from file data.

https://github.com/nasa/RHEAS/blob/master/external/VIC/src/read_veglib.c#L52



```
29
30 {
31     extern option_struct options;
32     veg_lib_struct *temp;
33     int i, j;
34     int tmpflag;
35     int Nveg_type;
36     char str[MAXSTRING];
37     char ErrStr[MAXSTRING];
38     double maxd;
39     char tmpstr[MAXSTRING];
40
41     rewind(vglib);
42     fgets(str, MAXSTRING, vglib);
43     Nveg_type = 0;
44     while(!feof(vglib)) {
45         if(str[0] <= 57 && str[0] >= 48) Nveg_type++;
46         fgets(str, MAXSTRING, vglib);
47     }
48     rewind(vglib);
49
50     temp = (veg_lib_struct *)calloc(Nveg_type+N_PET_TYPES_NON_NAT, sizeof(veg_lib_struct));
51
52     fscanf(vglib, \"%s\", str);
53     i=0;
54     while (!feof(vglib)) {
55         if(str[0] <= 57 && str[0] >= 48) {
56             temp[i].NvegLibTypes = Nveg_type;
57             temp[i].veg_class = atoi(str);
```

On line 52 of read_veglib.c file we can see following code:

```
fscanf(vglib, \"%s\", str);
```

Buffer 'str' is declared on stack as 'char str[MAXSTRING]', while MAXSTRING is defined as:

src/vicNI_def.h: **#define MAXSTRING 2048**

String in a VegLib data file longer than 2048 bytes will overflow the 'str' buffer while reading the file, causing stack based buffer overflow.

Asking ChatGPT what is VegLib file format, here is ChatGPT's response:

The NASA VegLib file is a vegetation parameter library used in land surface models (LSMs), particularly in climate and hydrology simulations. It is part of NASA's Land Information System (LIS) and other environmental modeling frameworks.

Key Features of NASA's VegLib File

- Contains biophysical and radiative properties of different vegetation types.
- Used in land surface and climate models to simulate energy, water, and carbon fluxes.
- Provides parameters like leaf area index (LAI), albedo, roughness length, and stomatal resistance.
- Helps improve the accuracy of Earth system models in predicting land-atmosphere interactions.

Usage in NASA's Land Models

NASA's Land Information System (LIS) and Noah Land Surface Model (Noah LSM) utilize VegLib data to simulate land-atmosphere processes such as:

- Evapotranspiration
- Surface runoff
- Soil moisture dynamics
- Carbon exchange

Sources and Applications



- Used in hydrological and climate studies for weather forecasting and drought monitoring.

If a malicious VegLib file comes from an untrusted source as in over an email or from the web, RHEAS software can potentially be exploited to achieve stack based buffer overflow and consequently remote arbitrary code execution when processing a malformed and malicious VegLib file.

More Buffer Overflow Vulnerabilities in NASA's Software

Quick GitHub code search of NASA's official account on GitHub resulted in 8 more potential buffer overflows in NASA's file processing software with an inherently insecure use of `fscanf()` function. To keep this article sane in terms of its length, I will just list them here, with GitHub repo, brief info and vulnerable code snippets. I'll leave detailed analysis as an exercise for a curious reader. :)

Software: Omnia's Stack Based Buffer Overflow

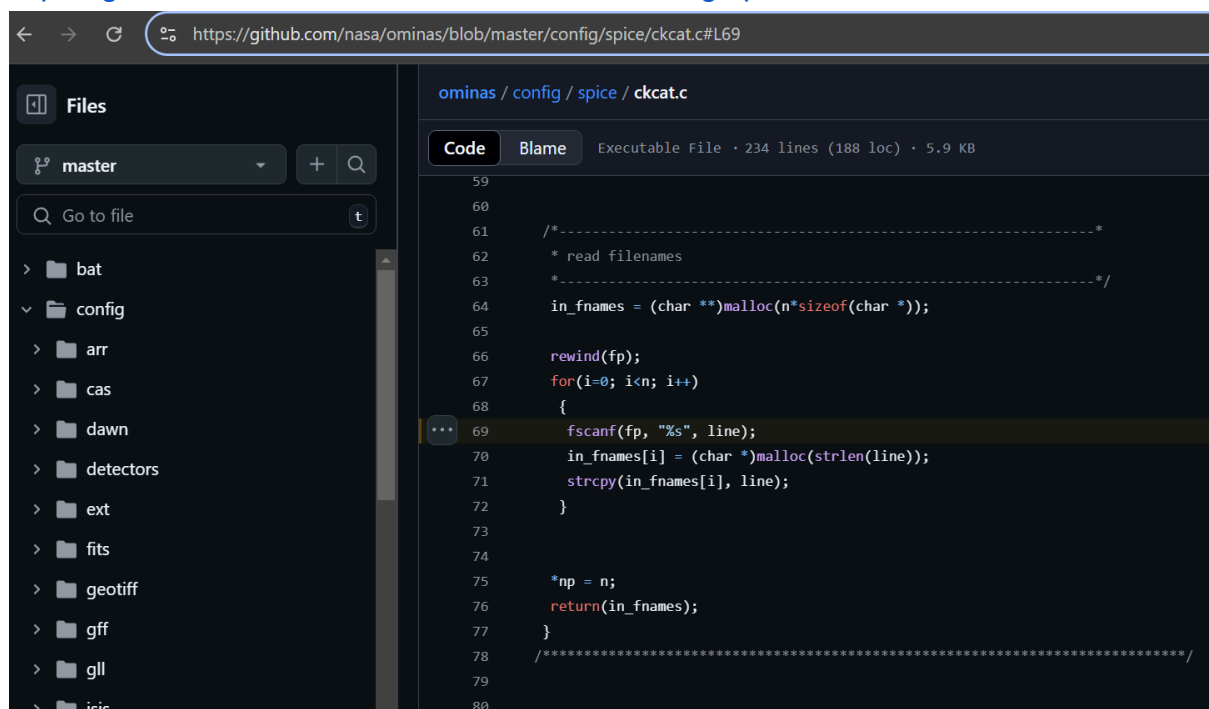
Description: Opensource Multi-INstrument Analysis Software

Repository: <https://github.com/nasa/omina>

Vulnerable file and lines:

<https://github.com/nasa/omina/blob/master/config/spice/ckcat.c#L58>

<https://github.com/nasa/omina/blob/master/config/spice/ckcat.c#L69>



```
59
60
61  /*-----*
62   * read filenames
63   *-----*/
64   in_fnames = (char **)malloc(n*sizeof(char *));
65
66   rewind(fp);
67   for(i=0; i<n; i++)
68   {
69   *** fscanf(fp, "%s", line);
70       in_fnames[i] = (char *)malloc(strlen(line));
71       strcpy(in_fnames[i], line);
72   }
73
74
75   *np = n;
76   return(in_fnames);
77   }
78   /*****
79
80
```

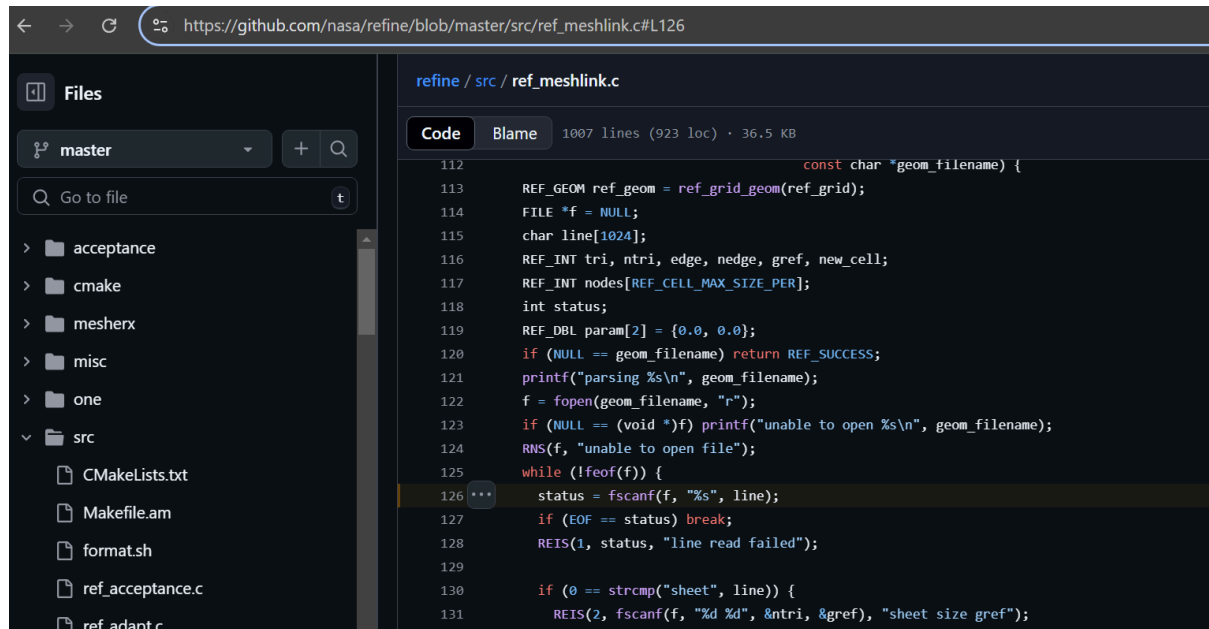
Software: Refine Stack Based Buffer Overflows

Description: N/A

Repository: <https://github.com/nasa/refine/>

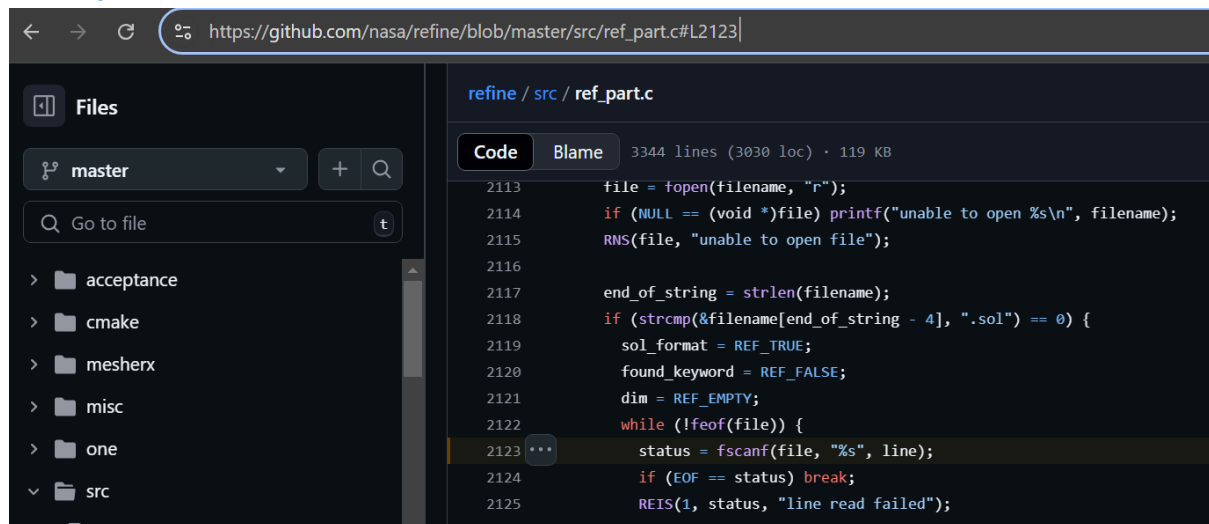
Vulnerable files and lines:

https://github.com/nasa/refine/blob/master/src/ref_meshlink.c#L126



```
refine / src / ref_meshlink.c
Code Blame 1007 lines (923 loc) · 36.5 KB
112 const char *geom_filename) {
113     REF_GEOM ref_geom = ref_grid_geom(ref_grid);
114     FILE *f = NULL;
115     char line[1024];
116     REF_INT tri, ntri, edge, nedge, gref, new_cell;
117     REF_INT nodes[REF_CELL_MAX_SIZE_PER];
118     int status;
119     REF_DBL param[2] = {0.0, 0.0};
120     if (NULL == geom_filename) return REF_SUCCESS;
121     printf("parsing %s\n", geom_filename);
122     f = fopen(geom_filename, "r");
123     if (NULL == (void *)f) printf("unable to open %s\n", geom_filename);
124     RNS(f, "unable to open file");
125     while (!feof(f)) {
126     ... status = fscanf(f, "%s", line);
127     if (EOF == status) break;
128     REIS(1, status, "line read failed");
129
130     if (0 == strcmp("sheet", line)) {
131         REIS(2, fscanf(f, "%d %d", &ntri, &gref), "sheet size gref");
```

https://github.com/nasa/refine/blob/master/src/ref_part.c#L2123



```
refine / src / ref_part.c
Code Blame 3344 lines (3030 loc) · 119 KB
2113 file = fopen(filename, "r");
2114 if (NULL == (void *)file) printf("unable to open %s\n", filename);
2115 RNS(file, "unable to open file");
2116
2117 end_of_string = strlen(filename);
2118 if (strcmp(&filename[end_of_string - 4], ".sol") == 0) {
2119     sol_format = REF_TRUE;
2120     found_keyword = REF_FALSE;
2121     dim = REF_EMPTY;
2122     while (!feof(file)) {
2123     ... status = fscanf(file, "%s", line);
2124     if (EOF == status) break;
2125     REIS(1, status, "line read failed");
2126
```

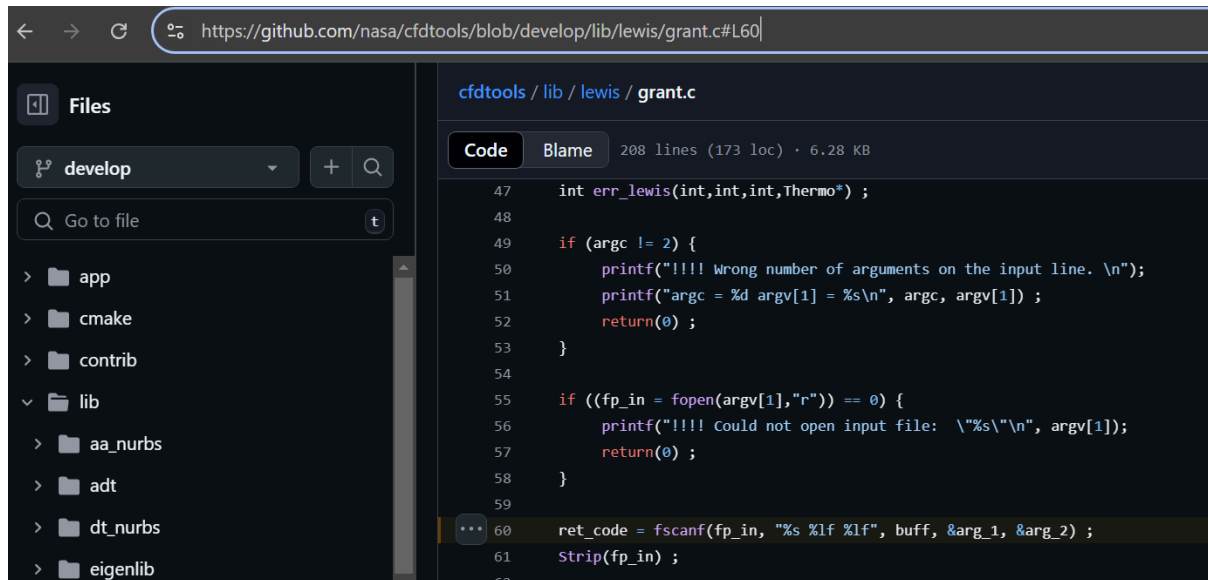
Software: CFDTools Stack Based Buffer Overflow

Description: N/A

Repository: <https://github.com/nasa/cfdtools/>

Vulnerable file and line:

<https://github.com/nasa/cfdtools/blob/develop/lib/lewis/grant.c#L60>



```
47 int err_lewis(int,int,int,Thermo*) ;
48
49 if (argc != 2) {
50     printf("!!!! Wrong number of arguments on the input line. \n");
51     printf("argc = %d argv[1] = %s\n", argc, argv[1]) ;
52     return(0) ;
53 }
54
55 if ((fp_in = fopen(argv[1],"r")) == 0) {
56     printf("!!!! Could not open input file:  \"%s\"\\n", argv[1]);
57     return(0) ;
58 }
59
60 *** ret_code = fscanf(fp_in, "%s %lf %lf", buff, &arg_1, &arg_2) ;
61 Strip(fp_in) ;
62
```

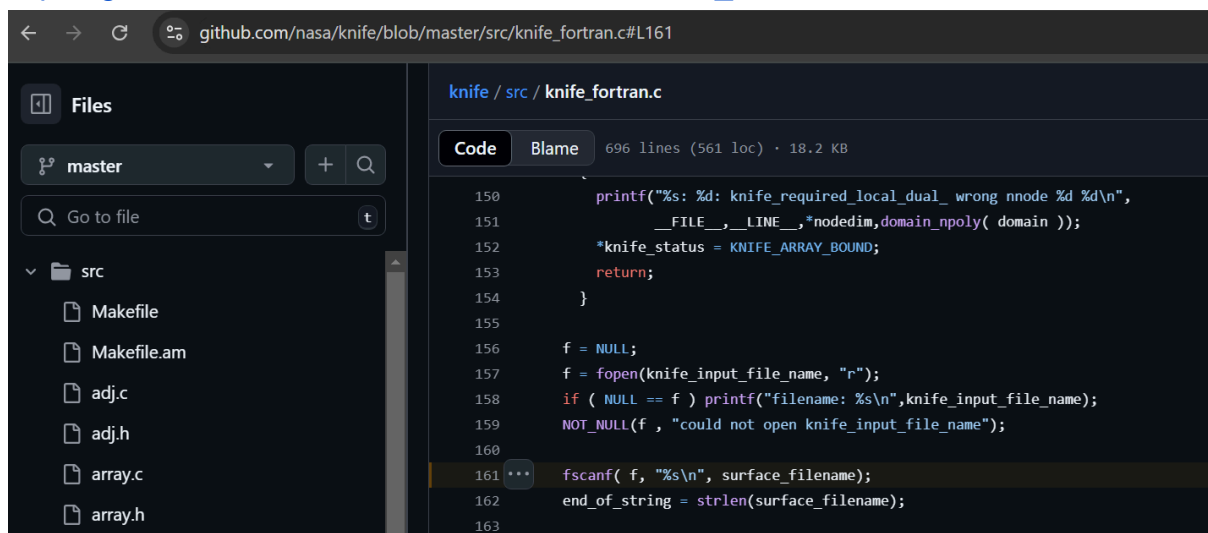
Software: Knife Stack Based Buffer Overflow

Description: N/A

Repository: <https://github.com/nasa/knife/>

Vulnerable file and line:

https://github.com/nasa/knife/blob/master/src/knife_fortran.c#L161



```
150 printf("%s: %d: knife_required_local_dual_wrong nnode %d %d\n",
151     __FILE__, __LINE__, *nodedim, domain_npoly( domain ));
152 *knife_status = KNIFE_ARRAY_BOUND;
153 return;
154 }
155
156 f = NULL;
157 f = fopen(knife_input_file_name, "r");
158 if ( NULL == f ) printf("filename: %s\n",knife_input_file_name);
159 NOT_NULL(f, "could not open knife_input_file_name");
160
161 *** fscanf( f, "%s\n", surface_filename);
162 end_of_string = strlen(surface_filename);
163
```

There are two more similar buffer overflows in the same file:

https://github.com/nasa/knife/blob/master/src/knife_fortran.c#L179

https://github.com/nasa/knife/blob/master/src/knife_fortran.c#L216

Just a quick note after this list, I'm pretty sure that during the analysis I've also seen one heap based buffer overflow, but I apparently lost it in my notes :)

Conclusion

I've spent 2 hours in total doing a brief manual audit of 10 randomly picked NASA's software applications and later 2 additional hours in 20 more applications (from quick Github code patterns search, grepping and manual review). All of them taken from NASA's official Github account and repositories, 4 hours during which I've discovered numerous (around 15) interesting findings which I (to be honest) didn't expect to discover in such a short time period.

To my surprise, if my little NASA's manual source code security audit exercise spanning only over 4 hours timespan resulted in numerous potentially serious security vulnerabilities, imagine how many more security issues can be found in the same software or other NASA's software by malicious adversaries with virtually unlimited budgets.

Some like foreign nation's sponsored threat actors looking for their nefarious way into NASA's systems or other Government/Tech/Research/Academics systems using vulnerable software available for anyone from NASA's GitHub account.

Just as precaution, as I did all of the security research over multiple codebases in a total of a strict ~4 hours, forgive me if I made some mistake, which I would attribute to a maximum of 5% of overall research content presented in this paper.

As of the beginning of a last year (2024), there was a great initiative coming from a USA WhiteHouse which urges tech companies to switch to memory safe programming languages:

<https://www.bleepingcomputer.com/news/security/white-house-urges-devs-to-switch-to-memory-safe-programming-languages/>

According to Microsoft, ~70% of software security vulnerabilities stem from software written in memory unsafe languages like C/C++ which usually involves some sort of memory corruption vulnerabilities and the WhiteHouse initiative is a step in the right direction. Switching to memory safe languages like Rust, Go, Java, C#, etc.

Nowadays in the days of modern computing, that can be achieved with a minimum performance tradeoff, and less and less as each year and technological advances come by.

Since I always report security vulnerabilities which I discover in a responsible disclosure manner to the vendors, I've contacted NASA dozen times over an e-mail (all which I could find related to the software), along with an official security contact - soc@nasa.gov email and telephone to report these issues, but I received no feedback over an email, only a short sentence over the phone.

What I got over NASA's SOC phone number was only that I was asked if I was their employee (and I said - no), and the conclusion was that their policy is that they don't reply to security vulnerability reports - at all.

I don't think or believe that NASA doesn't care about their own cyber security, quite the contrary, I'm more under the impression from what I was told from their side that NASA's security policy strictly forbids them to reply to security vulnerability reports reported directly to NASA from the outside of the organization. But somehow also got the impression that they are overwhelmed by bug bounty researchers looking for easy money that is simply causing too much noise on the wire.

Apparently, NASA's official software Github account (<https://github.com/NASA/>) referenced from <https://code.nasa.gov/> and <https://software.nasa.gov/> where they publish NASA's developed software is not under NASA's bug bounty program, so it's complicated to report any vulnerability / security issue discovered in NASA's software published on GitHub over the public bug bounty platforms.

It's not 2009 any more, back then there was no money involved when you directly contacted vendors (even NASA) about the security issues in their code, it was all much easier and sometimes (who would guess?) even more friendly in a favour of a mutual benefit. :)

Thank You!

Thanks to all of my friends who proofread the paper (you know who you are) and especially thanks to my beloved wife Tanja for the article wordsmith and the style checking, but also for the support during the lonesome nights while I was working on this paper/research.

Disclosure Timeline:

11/27/2024 - Discovery of vulnerabilities in NASA's GeoRef, CMR-CSW and CMR-OpenSearch

11/29/2024 - Reported GeoRef vulnerability to arc-sra-team@mail.nasa.gov and to Agency-DL-VAMP-VDP@mail.nasa.gov

11/29/2024 - Reported CMR-CSW data leak to Agency-DL-VAMP-VDP@mail.nasa.gov

11/29/2024 - Reported CMR-OpenSearch data leak to stephen.w.berrick@nasa.gov

11/29/2024 - Postmaster automatic reply that Agency-DL-VAMP-VDP@mail.nasa.gov is an email for internal communications

12/02/2024 - Reported GeoRef vulnerability over an email to soc@nasa.gov

12/02/2024 - Reported CMR-CSW vulnerability over an email to soc@nasa.gov

12/02/2024 - Reported CMR-OpenSearch vulnerability over an email to soc@nasa.gov

12/03/2024 - Sent an update email to soc@nasa.gov

12/04/2024 - Called NASA's SOC telephone number (877-627-2732), they said that they can't tell me much

12/17/2024 - Sent an email to soc@nasa.gov

01/13/2025 - Called again NASA's SOC phone number (877-627-2732), they asked me if I'm NASA's employee and only got a response that they don't respond to reported security vulnerabilities/issues.

01/13/2025 - Sent an update email to soc@nasa.gov

01/17/2025 - Reported Buffer Overflows in multiple NASA's software to soc@nasa.gov -

05/27/2025 - Public disclosure