THREATLEAP

Finding Critical Security Vulnerabilities In Widely Used Research And Scientific Software For Fun Not Profit, HDF5 Story

How to discover critical security vulnerabilities in one of the most popular research, scientific, data storage and processing file format designed to store and manage massive amounts of complex, heterogeneous data.

Leon Juranic (leon.juranic@threatleap.com)

Published: 04/12/2026

Not so long ago in the galaxy, not so far, far away…

Since A.D. 2009 I have been discovering and reporting critical security vulnerabilities in high profile scientific software, developed and used mostly by NASA for Earth and Space scientific missions and observation, but also used by thousands of other academic, defensive, aerospace, government, research and commercial companies and government agencies mostly in the USA, but also across the world.

These days I don't do much security research any more, I did everything that I wanted in the area of cybersecurity since I entered it as a curious kid and much, much more than I could have ever imagined when I was a teenager. Now I am leading my latest cybersecurity company with 25+ years of experience in the cybersecurity industry and just trying to enjoy life.

But now and then brief and sudden intellectual craving emerges to do something new, and to prove myself that I still have that hacker's spirit and to shake off that dust off of my hacking talent.
Stay with me if you want to find out about a vulnerabilities in HDF5 software from HDFGroup which is according to Google Gemini being used in research and data processing by high-profile organizations and companies like CERN, NASA, ESA, SpaceX, Boeing, Lockheed Martin, Google, Meta, Shell, Wall Street, AI/ML community and many more…

How I stumbled across HDF5?

Almost a year ago I was reading an article about various software for storing massive, unimaginable amounts of data and processing them for various purposes, used by the largest tech and scientific giants in the world.

I've stumbled across **HDF5** software (https://www.hdfgroup.org/), abbreviation for **Hierarchical Data Format version 5**, something that I, to be honest, never heard of before.
As it sparked my curiosity, I've asked a quick question to Google Gemini about **HDF5** software and here is what Gemini said.

HDF5 is the "industrial-strength" backbone for many of the world's most data-intensive organizations. Because it can handle massive datasets (petabytes) and is highly portable, it is used by both government giants and private innovators.

Here are the most prominent agencies and companies that rely on HDF5:

1. Public & Government Agencies

These organizations use HDF5 primarily for long-term archiving and sharing massive scientific observations.

- **NASA** (National Aeronautics and Space Administration): Perhaps the biggest proponent. HDF5 is the standard format for the Earth Observing System (EOS). Missions like the *Orbiting Carbon Observatory (OCO-2)* and the *Joint Polar Satellite System (JPSS)* store tens of terabytes of data in HDF5.
- **NOAA** (National Oceanic and Atmospheric Administration): Uses it for weather satellite data, climate modeling, and oceanographic records.
- **CERN** (European Organization for Nuclear Research): While they have their own formats (like ROOT), HDF5 is often used in the high-energy physics community for secondary analysis and sharing experiment results from the Large Hadron Collider.
- **U.S. Department of Energy (DOE) Labs**: * Lawrence Livermore National Laboratory (LLNL)
    - Los Alamos National Laboratory (LANL)
    - Sandia National Laboratories
    - These labs use HDF5 for massive supercomputing simulations (weapons research, fusion energy, etc.), as it was originally co-developed to handle "Big Data" on DOE supercomputers.
- **ESA (European Space Agency)**: Like NASA, the ESA uses HDF5 for various Earth observation and planetary missions to ensure interoperability with international partners.

---

2. Private Companies & Industries

In the private sector, HDF5 is the "quiet workhorse" behind engineering and financial simulations.

- Aerospace & Defense:
  - **Boeing**: Uses it for flight test data and complex structural engineering simulations.
  - **SpaceX**: Generates massive amounts of telemetry and sensor data during rocket launches and tests, much of which is processed using HDF5-compatible stacks.
  - **Northrop Grumman & Lockheed Martin**: For radar signal processing and advanced simulation modeling.
- Energy & Oil/Gas:
  - **Shell, BP, and ExxonMobi**l: Seismic imaging (looking for oil underground) generates some of the largest datasets in the world. HDF5 is used to store 3D and 4D seismic survey data.
- Finance:
  - **High-Frequency Trading Firms**: Many quantitative hedge funds use HDF5 to store tick-by-tick historical market data because it allows for lightning-fast "slicing" of data without loading entire files into memory.
- Technology & Software:
  - **MathWorks (MATLAB)**: Since version 7.3, the standard .mat file format is actually an HDF5 file under the hood.
  - **Google & Meta**: While they use many formats, HDF5 is a primary container for Deep Learning weights and datasets (often used via Keras or TensorFlow).

---

3. Summary Table: Why they choose it

Sector	Key Use Case	Why HDF5?
Space/Science	Satellite Imagery	Long-term archiving & portability
Engineering	Flight Testing	Handles heterogeneous data (video + sensors)

Finance	Market History	Extremely fast random access (slicing)
AI/ML	Training Models	Efficient storage of multidimensional tensors

What is actually the **HDF5** data format? Without bothering you with too many boring details, **HDF5** is according to Gemini:

In short: HDF5 is a high-tech suitcase that lets scientists pack massive amounts of different types of data into one organized, fast, and easy-to-carry file.

BINGO!

Since all of these high profile companies and organizations are using **HDF5** software, mostly in high profile scientific research and data processing, it definitely looks like an interesting target to the hackers in order to compromise their systems for stealing highly classified research data, industrial espionage or a foothold into the internal network.

Moreover, the core of **HDF5** software is written in **C**, programming language which implies that there could potentially lurk all sorts of memory related security issues and vulnerabilities in the core of the **HDF5** software itself.

I've decided to give it a shot, 8 hours of my spare time.

HDF5 source code can be downloaded from the HDFGroup vendor - https://www.hdfgroup.org/ .

HDF5 Source Code Analysis And The Vulnerabilities

When I downloaded **HDF5** source code, I quickly started to do manual analysis of most common pitfalls in software applications written in C while processing data from various file formats.
I have spent roughly an hour getting more familiar with source code architecture and structure, then I started to look into parts which actually does data processing of file formats.
I've quickly stumbled across the **/tools/src/h5import/h5import.c** command line tool used to read raw data from text or binary input files and convert them to the **HDF5** format.
It actually defines how data should be stored in **HDF5** format, which sizes are required, data types and metadata information.
Quick analysis of the **h5import.c** guided me to this interesting code snippet - **fscanf()** without boundary checking on the code line **597**.

```
591
592         case 64:
593             in64 = (H5DT_INT64 *)in->data;
594             switch (in->inputClass) {
595                 case 0: /* TEXTIN */
596                     for (i = 0; i < len; i++, in64++) {
597                         if (fscanf(strm, "%s", buffer) < 1) {
598                             (void)fprintf(stderr, "%s", err1);
599                             return (-1);
600                         }
601                         *in64 = (H5DT_INT64)strtoll(buffer, NULL, 10);
602                     }
603                     break;
```

These sorts of things are very common programming pitfalls in C. **fscanf()** is an inherently insecure C function if not used with proper format string boundaries.
Previous code snippet is part of function `readIntegerData(FILE *strm, struct Input *in)`

Pretty much the similar vulnerability can be also found in the same file on the line 764 in the following code snippet from `readUIntegerData(FILE *strm, struct Input *in)`

```
759         case 64:
760             in64 = (H5DT_UINT64 *)in->data;
761             switch (in->inputClass) {
762                 case 6: /* TEXTUIN */
763                     for (i = 0; i < len; i++, in64++) {
764                         if (fscanf(strm, "%s", buffer) < 1) {
765                             (void)fprintf(stderr, "%s", err1);
766                             return (-1);
767                         }
768                         *in64 = (H5DT_UINT64)strtoll(buffer, NULL, 10);
769                     }
770                     break;
```

The main difference between these two functions is that one is processing signed and the other unsigned integer data from the data file provided by the user, or maybe a malicious threat actor.

Since both functions and vulnerabilities are pretty much the same and have similar code path for exploitation, for the sake of length of this publication, we will examine first one - `readIntegerData(FILE *strm, struct Input *in).`
They are both called from the same parent function - `processDataFile(char *infile, struct Input *in, hid_t file_id)`

Function **readIntegerData()** takes data from the **FILE** stream of the file name provided to **h5import.c** at the command line.

Vulnerability Analysis - Digging further Into The code

Let's get back to the `readIntegerData(FILE *strm, struct Input *in)` and first vulnerability.

```
591
592         case 64:
593             in64 = (H5DT_INT64 *)in->data;
594             switch (in->inputClass) {
595                 case 0: /* TEXTIN */
596                     for (i = 0; i < len; i++, in64++) {
597                         if (fscanf(strm, "%s", buffer) < 1) {
598                             (void)fprintf(stderr, "%s", err1);
599                             return (-1);
600                         }
601                         *in64 = (H5DT_INT64)strtoll(buffer, NULL, 10);
602                     }
603                     break;
```

As stated before, vulnerability itself is located on line **597**, where input data from file is being read into the **buffer** variable using insecure **fscanf()** function into the variable located on stack without boundary checking. Shortly after that on line **601** data from input file is being converted to 64 bit signed integer using **strtoll()** function, and stored to the location where **in64**, 64 bit pointer to the heap is directed towards.

In the good old days, **fscanf()** buffer overflow would usually be all that we would need with little bit of trickery to overwrite return address and exploit buffer overflow to take control of the Instruction Pointer hence **EIP/RIP** register.

But simple good ol' days are long gone, so let's check how variables are organized on stack.

```
464 static int
465 readIntegerData(FILE *strm, struct Input *in)
466 {
467     H5DT_INT8  *in08;
468     H5DT_INT16 *in16;
469     H5DT_INT16  temp16;
470     H5DT_INT32 *in32;
471     H5DT_INT32  temp32;
472     H5DT_INT64 *in64;
473     H5DT_INT64  temp64;
474     char        buffer[256];
475     hsize_t     len = 1;
476     hsize_t     i;
477     int         j;
```

As we can see on the line **474**, our **buffer** variable which can be overflowed is **256** bytes, located on the stack memory right above the **temp64,** 64 bit signed integer, which is not important for exploitation and above the **in64**, 64 bit pointer towards the memory allocated on heap, **which is very important for the exploitation**.

That is usually the case unless the compiler does some optimization or realignment, but in normal circumstances that should be the stack layout.

In good ol' days, we would with a little trickery overwrite the Instruction Pointer or **in64** pointer on the stack (or **SEH** handler on the Windows) and point it towards the return address on stack to overwrite it or directly towards our shellcode.
As I said, the good ol' days are long gone.

To exploit this vulnerability and take full control over the process, we now need to defeat various memory protection mechanisms and mitigations, even different ones since **HDF5** is being distributed for Linux, Windows and MacOS which all have somewhat different security mechanisms for protection from memory corruption vulnerabilities.

One thing that is common and first most important line of defense is **ASLR** (Address Space Layout Randomization) which prevents us from blindly setting hardcoded memory address to overwrite, since upon every start of the process or system boot, base addresses for stack, heap, libraries, etc. are being randomized over and over again.

We either need some memory information leak to defeat **ASLR** which is unlikely in this case of data file processing security vulnerability in the command line tool, or we would have to try to think of something else.

So far what we know is that we can overwrite **in64** 64 bit pointer located on stack below the **buffer** variable, **in64** pointer pointing towards the heap, but since stack and heap base address is randomized, overwriting complete **in64** with some hardcoded address would lead us to memory violation / segmentation fault and crashing the program.

```
601:  *in64 = (H5DT_INT64)strtoll(buffer, NULL, 10);
```

The previous line is the key to the potential evasive solution for **ASLR** randomization.
After the **buffer** is being overflowed with user/malicious data, it is possible to also overwrite the **in64** pointer which is on the stack below the **buffer** and points to the heap.

If complete **in64** is overwritten, the process will 99,99% crash, but what came across my mind is one rather old technique used when buffer overflow is somehow limited or there is no way of memory leak, like in this case with **ASLR** - **partial overflow** even without memory information leak.
Partial overflow is a well known technique for defeating **ASLR** in specific cases.

ASLR usually randomizes base address, **but lower 12 bits** of memory address or **4Kb** memory page usually always have predictable offsets.
So instead of completely overwriting an **in64** pointer, it is possible to overwrite the **buffer** on stack and overwrite only 2 bytes or single **LSB** (Least Significant Byte) of **in64** pointer to target some **other interesting data or structure** in the memory within the same predictable memory page offset.

```
601:  *in64 = (H5DT_INT64)strtoll(buffer, NULL, 10);
```

So first part of exploitation would be to exploit that **fscanf()** stack buffer overflow to overwrite **buffer**, **temp64** and just **1-2** bytes of **in64** 64 bit pointer which will make it in the previous code line, in the second part, possible to overwrite data on some predictable memory address other than what it is meant to write to.
Data which is being written to **in64** is also coming from user input - **buffer** filled with data from the user/hacker controlled input file read with vulnerable **fscanf()** and later written to some memory location as it can be seen on the previous code line **601**.

From the hackers point of view, since **HDF5** input file format can contain various data types, various sizes and formats, it is possible to play around with a memory and alignment in order of how it is processed and allocated which increases the chances of grooming, finding and hitting something useful to overwrite.

Basically this vulnerability could be exploited by a specially crafted malicious input file processed by **h5import.c** command line tool for converting data to **HDF5** format.

I'm now leaving further exploitation analysis and research for an interesting data to overwrite and take this vulnerability to its full potential for a curious reader.

Disclosure Timeline:

07/18/2025 - Initial contact with the vendor - HDFGroup
07/18/2025 - Vulnerability details sent to the vendor
10/02/2025 - Vulnerabilities fixed and PR request closed
10/02/2025 - Vulnerability case closed, new fixed version available from HDFGroup
04/13/2026 - Public disclosure

You might be interested why I waited months for public disclosure?

Since all that information from Google Gemini that **HDF5** software is being used in research and data processing by high-profile organizations and companies like CERN, NASA, ESA, SpaceX, Boeing, Lockheed Martin, Google, Meta, Shell, Wall Street, AI/ML community and many more… and the potential high or even critical severity of the vulnerabilities, I've decided to postpone public disclosure a little bit longer than usual before going public with it.

Thanks!

I would like to thank for the support and encouragement in my business and research endeavours especially to my family, colleagues in ThreatLeap and RNTrust - and of course to all of my friends, you know who you are 🙂