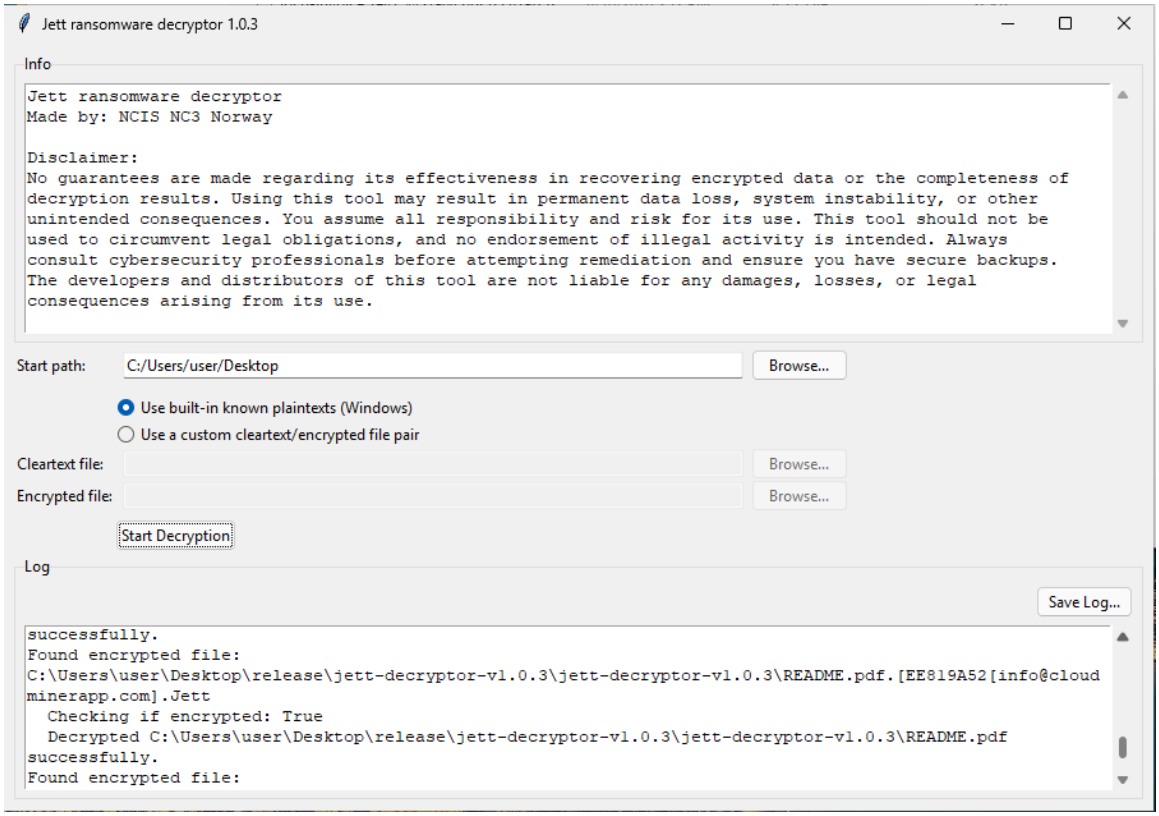
Jett decryptor

Version 1.0.4

Made by: National Criminal Investigation Service NC3 in Norway



Jett is a type of ransomware. When it executes, it generates a random seed that initializes a ISAAC pseudorandom number generator, which produces the keystream used in a XOR encryption scheme. However, the same seed is reused for all files, making the encryption vulnerable to a known-plaintext attack. By obtaining a known plaintext, it is possible to reconstruct the keystream and decrypt every encrypted file on the system. This decryptor tool leverages known plaintext files present in Microsoft Windows* to recover the keystream and decrypt all other files.

Disclaimer

Disclaimer:
No guarantees are made regarding its effectiveness in recovering encrypted data or the completeness of decryption results. Using this tool may result in permanent data loss, system instability, or other unintended consequences. You assume all responsibility and risk for its use. This tool should not be used to circumvent legal obligations, and no endorsement of illegal activity is intended. Always consult cybersecurity professionals

before attempting remediation and ensure you have secure backups. The developers and distributors of this tool are not liable for any damages, losses, or legal consequences arising from its use.

Table of contents

- Background and related research
- Bugs and remarks
- How It Works
- Usage
  - Requirements for python tools
  - Installing Tkinter
    - Windows
    - Linux - Ubuntu/Debian
    - Linux - Fedora/RHEL/CentOS
  - Installing Python Dependencies
  - Project Structure
  - Command-Line Interface (decryptorCLI.py)
  - Graphical User Interface (decryptorGUI.py)
  - Building Windows Executables
    - Prerequisites
    - Building on Windows
      - Method 1: Using the Batch Script (Recommended)
      - Method 2: Manual PyInstaller Build
    - Building on Linux
    - Using the Built Executable
    - Customization
  - Library Usage (jettlib)
  - Testing & CI/CD
  - Troubleshooting
- Jett ransomware Indicators of Compromise (IoCs)
  - Network traffic
- Support

Background and related research

The Jett malware is not much discussed in open sources. The company itm8 have however published a blog post on design flaws found in the Jett ransomeware. See:

- https://itm8.com/articles/decrypting-jett-ransomware
- https://itm8.com/articles/who-hacked-us

Development of this decryptor leverages itm8’s findings of the Jett ransomware strain observed in their incident, together with artifacts, indicators, and operational insights from NCIS Norway’s investigation of a Jett variant tied to a active threat actor.

Bugs and remarks

The Jett ransomeware found in the NCIS case haa a race condition that sometimes causes it to encrypt only half of the files it targets. We have not yet investigated the root cause, but it is likely related to a threading bug that results in duplicate or incomplete encryption operations. The decryptor accounts for such race conditions.

*) **Chosen known plaintext** Some chosen known plaintext files are included in the Python source code (`jettlib/core.py`). You may need a pair of plaintext and ciphertext files from you victim, or change the hex content of each file if the built in ones if they are missing or is different on your Windows victim operating system. Here are two of the plain text files choosen:

- Your pre-infection victim Windows 11 version should at least* have the un-encrypted file `C:\ProgramData\Microsoft\User Account Pictures\user.png` with the MD5-sum of: `e6f6b37815399773a2f7365cf805077f`.
- Your pre-infection victim Windows 10 version should have the un-encrypted file `C:\ProgramData\Microsoft\User Account Pictures\user.png` with the MD5-sum of: `d7ee4543371744836d520e0ce24a9ee6`.

The hard coded plaintext are only able to generate a keystream that can decrypt files smaller than 819200 bytes. If you have a plaintext version of a file that has been encrypted you can use your pair of files directly in the GUI or CLI to recover more key material. The larger the filesize of this pair the better, i.e. more key material is recovered, and you can recover larger files on the victim machine.

Recovering files larger than 0xC8000 (819200) bytes: Jett re-keys every `CHUNK_SIZE` (`0xC8000`) bytes (see How It Works), so a known-plaintext file only unlocks as many chunks as it itself spans – a 50 KB known-plaintext file, for example, only recovers the first `0xC8000` bytes of any other file, no matter how large that other file is. When a file can't be fully recovered this way, the tool logs a warning and saves it with a `.partial-decrypted` suffix instead of its normal decrypted name (e.g. `bigfile.exe.partial-decrypted`) – so a partial recovery is never mistaken for a complete one. To fully recover larger encrypted files, add a **larger** known-plaintext/ciphertext pair – ideally at least as large as the biggest file you need to recover – there are options for specifying this both in the GUI and the CLI version.

How It Works

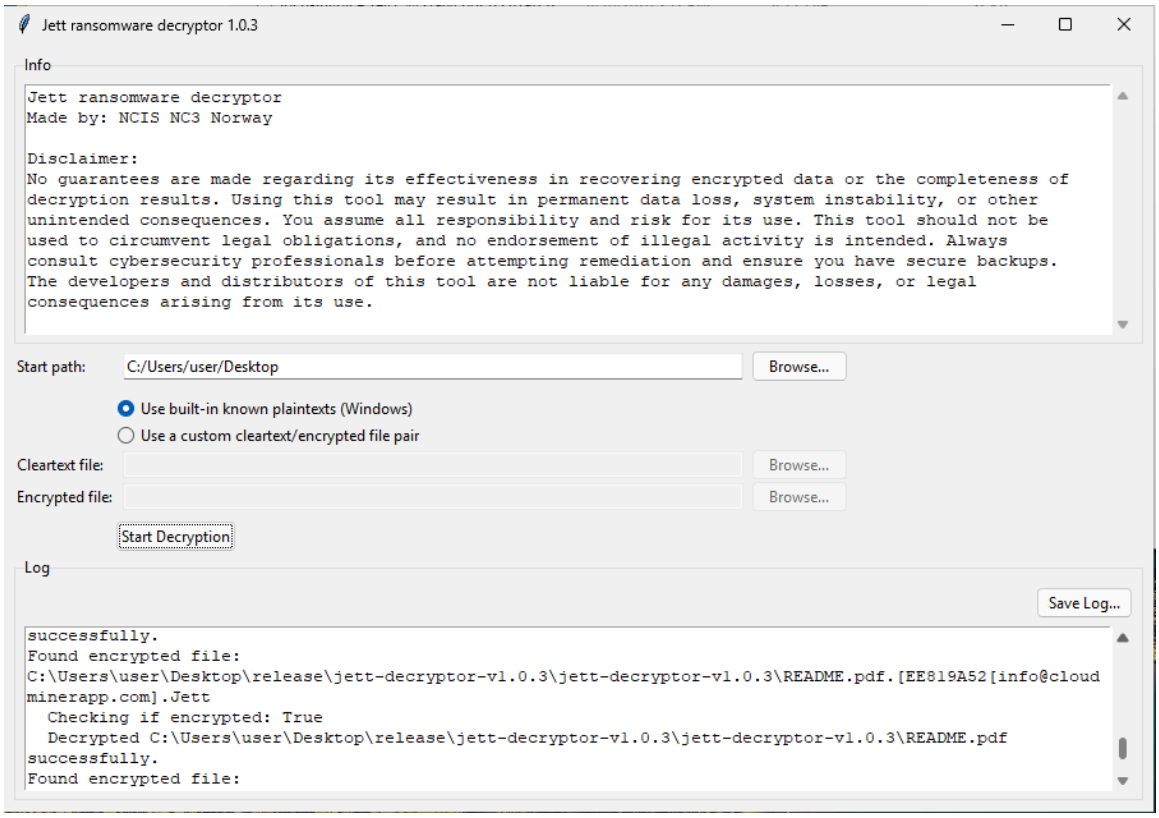
1. **Keystream Derivation**: The tool searches for known plaintext files (Windows system files like `user.png`) that match their encrypted `.Jett/.jett` counterparts (the suffix is matched case-insensitively, and whichever case is found on disk is reused for decryption) – or, if a custom cleartext/encrypted pair was given (CLI `--cleartext`/`--encrypted`, or the GUI's "custom cleartext/encrypted file pair" option), that pair is used directly instead
2. **Keystream Verification**: Before using it for anything, the derived keystream is verified by actually decrypting the chosen encrypted file with it and comparing the result against the known cleartext byte-for-byte (SHA1 hashes). Both the pass and the fail outcome are logged explicitly (`Verification PASSED`/`Verification FAILED`, with the reason) – a keystream that fails verification is discarded and never used to decrypt anything else

3. **Chunked Keystream Recovery**: Jett's encryption (`CryptFile`) regenerates its keystream every 819200 bytes (`0xC8000`, one "chunk"), and *within* a chunk that keystream repeats with a period of 512 bytes. There's no per-file IV, so a given chunk's keystream is identical across every file from the same infection – as soon as a known-plaintext/known-ciphertext pair gives 512 known bytes anywhere in a chunk, XORing them reveals that whole chunk's keystream for *every other file*, not just the part the known sample covered. Practically: a known-cleartext reference of size S fully unlocks `ceil(S / 819200)` chunks of any other file from the same run; bytes beyond that are left untouched (still encrypted) rather than guessed at – see "Recovering files larger than 0xC8000 (819200) bytes" under Usage above for how to fix that, and `jettlib/core.py`'s `recover_keystream_chunks()`/`decrypt()` docstrings for the full technical detail
4. **XOR Decryption**: Once the per-chunk keystream is recovered, it decrypts all `.Jett/.jett` files in the target directory, chunk by chunk
5. **Encryption Check**: Before writing a decrypted file, each `.Jett/.jett` file is checked to see if it's actually encrypted – first via known file-type magic bytes (e.g. a `.png` sibling that already starts with the PNG signature is definitely still plaintext, regardless of entropy), then falling back to a Shannon-entropy heuristic. This matters because Jett has a known race condition (see Bugs) that sometimes leaves a targeted file completely untouched despite the `.Jett` suffix; files that fail this check are copied through unchanged (as `*.already_cleartext.*`) instead of being (incorrectly) decrypted
6. **File Recovery**: Decrypted files are saved with their original names (e.g., `document.docx.Jett` → `document.docx`, or `document.docx.jett` → `document.docx`). If the recovered keystream didn't cover the whole file (step 3), a warning is logged and the output is saved as `document.docx.partial-decrypted` instead, so an incomplete recovery is never mistaken for a complete one

Usage

The Jett decryptor is available as both a command-line tool and a graphical user interface (GUI).

A precompiled exe file is also available (`jettdecryptor.exe`):



Run as administrator if Jett has encrypted files outside the users directories.

Requirements for python tools

- Python 3.7+
- **Tkinter** (for GUI - required for `decryptorGUI.py`)
- Required packages in `requirements.txt`:
  - `puremagic` (runtime dependency – file-signature detection used by `is_encrypted()`, see How It Works)
  - `pyinstaller` (for building Windows executables)
  - `pyinstaller-hooks-contrib` (for GUI packaging support, and bundles `puremagic`'s signature data file into the exe)
  - `Pillow` (for image processing in GUIs)

Installing Tkinter

Tkinter is usually included with Python on Windows, but must be installed separately on Linux.

Windows

Tkinter is typically included with Python on Windows. To verify:

```
python -m tkinter
```

If a small test window appears, Tkinter is already installed. If not, reinstall Python and ensure "tcl/tk and IDLE" is selected during installation.

Linux - Ubuntu/Debian

Install Tkinter using apt:

```
sudo apt update
sudo apt install python3-tk
```

Verify installation:

```
python3 -m tkinter
```

Linux - Fedora/RHEL/CentOS

Install Tkinter using dnf:

```
sudo dnf install python3-tkinter
```

Verify installation:

```
python3 -m tkinter
```

Installing Python Dependencies

Once Tkinter is installed, install the remaining dependencies:

```
pip install -r requirements.txt
```

Project Structure

```
jett-decryptor/
├── jettdecryptorCLI.py          # Command-line interface
├── jettdecryptorGUI.py          # Graphical user interface (Tkinter)
├── jettdecryptor.spec           # PyInstaller spec file for building
exe
├── build_exe.bat                # Windows batch script to build exe
├── jettlib/                     # Jett decryption library
│   ├── __init__.py
│   └── core.py
├── requirements.txt
└── README.md
```

Command-Line Interface (decryptorCLI.py)

The CLI tool scans a directory recursively for `.Jett` files (matched case-insensitively, so `.jett` is also found) and decrypts them. It runs on both Windows and Linux.

Basic Usage (Windows):

```
python decryptorCLI.py --path C:\Users
```

By default the keystream is derived by scanning the built-in, Windows-path-keyed known-cleartext files (`jettlib/core.py`'s `KNOW_CLEARTEXTS`) – this only works on a live Windows system (or when running directly against `C:\...` paths).

On Linux (or anywhere those hardcoded Windows paths don't apply – e.g. a mounted or copied image of the victim system), point directly at a known-cleartext file and its matching `.Jett/.jett` counterpart instead:

```
python3 decryptorCLI.py \
  --cleartext /mnt/victim/ProgramData/Microsoft/User\ Account\
        Pictures/user.png \
  --encrypted "/mnt/victim/ProgramData/Microsoft/User Account
        Pictures/user.png.[ID[attacker@example.com].Jett" \
  --path /mnt/victim
```

Arguments: - `--path PATH`: Root directory to search recursively for `.Jett/.jett` files (required) - `--cleartext PATH` / `--encrypted PATH`: Derive the keystream from this explicit known-cleartext/encrypted pair instead of scanning the built-in `KNOW_CLEARTEXTS` paths. Must be given together. Needed on Linux; optional (but still usable) on Windows. - `--log-file PATH`: Also append all log output to this file, in addition to the console (optional). Each run adds a timestamped header line so multiple runs appended to the same file stay distinguishable. If the file can't be opened, a warning is printed and decryption still proceeds without file logging.

The original encrypted `.Jett/.jett` files are never deleted – there is deliberately no option to remove them. `is_encrypted()`'s detection can false-positive, and a keystream may only cover part of a file (see How It Works), so the only reliably-known-good copy of a file is the one Jett left behind; deleting it based on an unverified decryption result would risk unrecoverable data loss.

Example workflow:

```
# Decrypt all files in C:\Users and save a log
python decryptorCLI.py --path C:\Users --log-file
        C:\Users\decryption.log
```

Graphical User Interface (decryptorGUI.py)

The GUI provides a user-friendly interface with a disclaimer, path selection, and real-time progress logging.

Start the GUI:

```
python decryptorGUI.py
```

Features: * Disclaimer and usage information at startup * Warns on startup if not running elevated, recommending "Run as administrator" (Jett may have encrypted files under other Windows user profiles that require admin rights to read/decrypt) * Browse button to select decryption root directory * Choice of keystream source: the built-in known plaintexts (`KNOW_CLEARTEXTS`), or a custom cleartext/encrypted file pair (Browse buttons for both) – the same choice `--cleartext`/`--encrypted` gives the CLI, see

below * Real-time log view showing decryption progress * Start button to begin the decryption process * “Save Log…” button to export the current log contents to a file at any time (before, during, or after a run) * Responsive UI (decryption runs in background thread)

Building Windows Executables

The project includes a PyInstaller spec file for building `jettdecryptor.exe` on Windows.

Prerequisites

Ensure PyInstaller is installed:

```
pip install -r requirements.txt
```

Or install PyInstaller directly:

```
pip install pyinstaller pyinstaller-hooks-contrib
```

Building on Windows

Method 1: Using the Batch Script (Recommended)

Simply double-click or run from Command Prompt:

```
build_exe.bat
```

This will: - Clean any previous builds - Check that PyInstaller is installed - Compile `jettdecryptorGUI.py` into a standalone executable - Place the executable in `dist/jettdecryptor.exe`

Method 2: Manual PyInstaller Build

From PowerShell or Command Prompt, run:

```
pyinstaller jettdecryptor.spec
```

The executable will be created in `dist/jettdecryptor.exe`.

Building on Linux

You can cross-compile for Windows on Linux using Wine and PyInstaller – this is exactly what GitLab CI does automatically on every tag push, using the Wine build image defined in `docker/wine-build/` (Wine + a real Windows Python installed under it, verified to produce a genuine, runnable Windows PE executable). To build locally the same way, run:

```
./build_wine.sh
```

This builds (or reuses a cached copy of) the Wine image via Podman or Docker and produces `dist/jettdecryptor.exe`, no Windows machine or CI needed. See `docker/wine-build/README.md` for options, the manual image build/push commands, and how to reproduce a CI build to debug it.

To build a full, distributable release instead of just the exe, run:

```
./build_release.sh
```

This runs `build_wine.sh`, renders `README.md` to a PDF (via `pandoc` + headless Chrome/Chromium – both required), and packages `jettdecryptor.exe`, `README.md`, the rendered `README.pdf`, and the Python source (everything except `tests/`) into `release/jett-decryptor-v<version>.zip`, plus a standalone `release/jett-decryptor-README-v<version>.pdf`. The version number is read from `jettlib/__init__.py`'s `__version__`.

Alternatively, build the CLI version which doesn't require a GUI framework:

```
pyinstaller --onefile jettdecryptorCLI.py
```

Using the Built Executable

After building, run the decryptor GUI from the command line:

```
dist\jettdecryptor.exe
```

Or double-click `jettdecryptor.exe` in the `dist/` folder to launch it directly.

Customization

To customize the build (add a custom icon, change the executable name, etc.), edit `jettdecryptor.spec`:

- **Add custom icon**: Change `icon=None,` to `icon='path/to/icon.ico',`
- **Change executable name**: Change `name='jettdecryptor',` to `name='your_name',`
- **Enable console window**: Change `console=False,` to `console=True,` (useful for debugging)

Then rebuild using the spec file:

```
pyinstaller jettdecryptor.spec
```

Library Usage (jettlib)

You can also import the decryption functions directly in your own Python code:

```
from jettlib import KNOW_CLEARTEXTS, get_keystream_and_suffix,
        walk_and_decrypt


# Derive keystream from known cleartexts
```

```
keystream, suffix = get_keystream_and_suffix(KNOW_CLEARTEXTS,
        log_callback=print)

# Walk directory and decrypt all .Jett/.jett files
if keystream:
    processed = walk_and_decrypt(
        startpath="C:\\Users",
        keystream=keystream,
        suffix=suffix,
        log_callback=print,
    )
    print(f"Decrypted {processed} files")
```

Testing & CI/CD

Run the test suite locally with:

```
pip install -r requirements-dev.txt
pytest
```

For a coverage report:

```
pytest --cov=jettlib --cov-report=term-missing
```

GitLab CI/CD (`.gitlab-ci.yml`) runs the full pytest suite with coverage automatically on branch pushes and merge requests (it does not run on tag pushes). To avoid duplicate pipelines, a branch push is skipped in favor of the MR pipeline whenever that branch already has an open merge request. Where the results show up in the GitLab UI:

- **Pipeline coverage badge**: the `pytest` job's total coverage % is parsed from its log and shown on the pipeline/job list.
- **Tests tab**: the JUnit report (`report.xml`) populates the pipeline's Tests tab with per-test pass/fail results.
- **Coverage trend graph**: the same coverage value feeds the project's coverage history under Analytics > Repository.
- **Merge request diff annotations**: the Cobertura report (`coverage.xml`) enables GitLab's per-line green/red coverage highlighting directly in MR diffs.

Troubleshooting

"Keystream generation failed"

- Ensure the known plaintext files exist (Windows system files like `C:\ProgramData\Microsoft\User Account Pictures\user.png`)
- Confirm these files have encrypted `.Jett/.jett` counterparts on the system
- Check that the operating system matches the expected Windows version in `jettlib/core.py`

"File already exists" errors

- The tool saves decrypted files alongside encrypted ones
- If destination filename already exists, check logs for details

Partial decryption results

- Due to the race condition in Jett ransomware, some files may not be encrypted
- The tool will log when files are skipped as “not encrypted”
- If a large file only decrypts correctly for its first `0xC8000` (819200) bytes and stays encrypted beyond that, your known-plaintext reference isn’t large enough to cover it – see “Recovering files larger than 0xC8000 (819200) bytes” above; add a bigger known-plaintext/ciphertext pair to `KNOW_CLEARTEXTS` in `jettlib/core.py`
- Any file the tool couldn’t fully recover is saved as `<name>.partial-decrypted` with a matching warning in the log, rather than silently under its normal name – look for that suffix to find every file affected by the point above

Jett ransomware Indicators of Compromise (IoCs)

Network traffic

Two GET requests over HTTP:

```
[2025-06-26 02:38:47] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
recv: GET / HTTP/1.1
[2025-06-26 02:38:47] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
recv: Host: icanhazip.com
[2025-06-26 02:38:47] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
recv: Connection: Keep-Alive
[2025-06-26 02:38:48] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
info: Request URL: http://icanhazip.com/
[2025-06-26 02:38:48] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
send: HTTP/1.1 200 OK
[2025-06-26 02:38:48] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
send: Content-Type: text/html
[2025-06-26 02:38:48] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
send: Date: Thu, 26 Jun 2025 07:38:48 GMT
[2025-06-26 02:38:48] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
send: Content-Length: 9
[2025-06-26 02:38:48] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
send: Connection: Close
[2025-06-26 02:38:48] [2369] [http_80_tcp 2499] [10.0.0.20:55600]
send: Server: INetSim HTTP Server

[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: GET / HTTP/1.1
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: TargetID: E8846E7C
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
```

```
recv: Attacker: mehrdad
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: TargetName: DESKTOP-6POV1GA
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: TargetOS: Microsoft Windows 10 Pro
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: TargetHard: 25.42 GB
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: TargetCpu: 13th Gen Intel(R) Core(TM) i9-13900H
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: TargetIp: 10.0.0.1
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: Host: recoverydata.mb
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
recv: Connection: Keep-Alive
[2025-06-26 02:38:48] [2369] [http_80_tcp 2500] [10.0.0.20:55601]
info: Request URL: http://recoverydata.mb/
```

TargetID is volume serial number of C: (same as the one you see if you do `dir` on `c:`).

The pseudo-random generated encryption key seem not to be sent to any C2.

Support

This is open source, but you can contact the developer kripos.nc3@politiet.no for questions.