FIELD NOTE / OSINT

MediaFire File Info: OSINT from a Share Link

Petruknisme · 20 May, 2026 · 6 min to read

MediaFire links are everywhere. They show up in malware reports, phishing emails, forum threads, and the kind of leaked document dump that someone forwards at 2 a.m. with the message “worth a look?”

Most of the time I don’t need the file. I need to know what it is before I download it: name, size, type, upload date, whether the link is still public. The share page shows that, but opening a browser for every link gets old fast, especially when the links arrive in a spreadsheet.

MediaFire has an API endpoint for this. It returns XML, needs no key, and has probably been there for a decade. I wrote a small Python wrapper after an agent handed me a folder of links from a research run and I realized I was about to open thirty tabs by hand. Now I can run a link from the terminal, or let the agent call the script as a skill. This post is the endpoint, the tool, and the parts I’d tell you to be careful about.

The quick key

Every MediaFire share URL carries a file ID. MediaFire calls it a quick key. In this URL:

```
https://www.mediafire.com/file/7fpuwrsb2oelinf/MediaFire_-_Getting_Started
```

the key is `7fpuwrsb2oelinf`. The same ID shows up in `/download/` links. A regex is enough to pull it out:

```
match = re.search(r"mediafire\.com/(?:file|download)/([a-zA-Z0-9]+)", url)
```

With the key, one GET request returns the metadata:

```
https://mediafire.com/api/1.5/file/get_info.php?quick_key=<key>
```

The response is XML. No JSON option, at least not on this endpoint, which tells you roughly how old it is. The root has a `<file_info>` element with the file fields, plus `<permissions>`, `<links>`, and a `<result>` status.

Worth saying plainly: this is not an auth bypass. It returns what the public share page already shows. The API just gives you the same data in a format you can parse.

What the tool does

The script is stdlib only. That's not a moral stance about dependencies; I wanted to run it on a clean analysis VM without installing anything. Python 3, a URL, done.

It does three things: pulls the key out of the URL, fetches the XML with a 15-second timeout, and flattens the response into key-value pairs. The request sends a normal browser User-Agent. No token, no cookie.

The parser is deliberately simple. It walks `<file_info>`, handles `<links>` and `<permissions>` as nested dictionaries, and prints whatever it finds. If MediaFire adds a field, the script shows it. If MediaFire removes one, the script prints a dash.

The full script is at the end of the post. The short version is three functions and a printer.

A real run

I tested it on a public getting-started PDF. I removed the owner name from the output because there's no reason to publish some random account handle.

```
[*] Quick Key: 7fpuwrsb2oelinf
[*] Fetching file info...

============================================================
  □ MEDIAFIRE FILE INFO
============================================================

  ✓ Status: yes
  Quick Key           : 7fpuwrsb2oelinf
  Filename            : MediaFire - Getting Started.pdf
  Description         : (none)
  File Size           : 372.37 KB (381311 bytes)
  File Type           : document
  MIME Type           : application/pdf
  Uploaded            : 2026-05-19 19:21:05
  Uploaded (UTC)      : 2026-05-20T00:21:05Z
```

```
   Downloads            : 0
   Views                : 0
   Revision             : 1
   Privacy              : public
   Password Protected   : no
   SHA256               : 77e00e1c063bec92eb903414398f01dd889639ed4412203c5
   
    Download Links:
     Normal Download      : https://mediafire.com/file/7fpuwrsb2oelinf/Medi
   
    Permissions:
     Value           : ✅ (1)
     Explicit        : ❌ (0)
     Read            : ✅ (1)
     Write           : ❌ (0)
```

A couple of notes. The download count is zero, which is normal for a fresh upload and a good reminder that this field is not a popularity contest. The `created` and `created_utc` values are five hours apart. I assume the local field follows MediaFire's server timezone; I don't know for sure, and I don't care enough to find out. Use `created_utc`.

What the fields are actually good for

Field	Use	Caveat
`filename`	Identify the file without downloading	Uploaders can name it anything
`size`	Spot unusually large or tiny files	Not evidence by itself
`filetype`, `mimetype`	Quick classification	Based on the stored file, not a deep inspection
`created_utc`	Timeline correlation	Use this over `created`
`hash`	Pivot to VirusTotal, MalwareBazaar, or your own corpus	Not present on every file, especially old ones
`owner_name`	Public uploader handle	Often pseudonymous; not an identity
`downloads`, `view`	Rough distribution signal	Can be inflated or gamed
`revision`	Detect that the file changed after upload	A bump alone does not mean malice
`privacy`, `password_protected`	Check whether the link is still open	Public metadata does not mean safe content
`permissions.read`	Confirm the link is readable	Write access is rare on public shares

The hash is the field I care about most. When it's there, it's a SHA-256 you can pivot on without downloading the file. When it's not there, you fall back to the boring fields: upload date, size, owner handle.

Download counts are close to worthless. A link can be mirrored, refreshed, or inflated. I look at them, but I don't draw conclusions.

`owner_name` is a handle, not an identity. People reuse handles, and MediaFire lets you pick whatever. Treat it as a lead, not attribution.

`revision` is more interesting than it looks. A document uploaded once and never touched stays at revision 1. If it jumped to revision 3, someone changed the file. That doesn't mean malware. It means the file you download today may not be the file someone analyzed last month.

Where I use it

Triage is the main one. A link lands in a report, I run the script before downloading. A 4 GB archive uploaded yesterday under a handle I don't recognize gets handled differently from a 300 KB PDF from 2024. The metadata doesn't tell me the file is malicious. It tells me how much attention to pay.

Timeline work is the other reason I keep it around. `created_utc` is a fixed point I can compare with email headers, forum posts, or other indicators. It's not proof of anything, but it's cheap to collect and it doesn't require opening the file.

When the API gives me a hash, I check it against VirusTotal, MalwareBazaar, and my own samples. A hit is useful. A miss means nothing.

Monitoring is the one I use least. A cron job can watch a link and log changes, but most links don't change. When one does, the revision or hash move. That's worth knowing if you're tracking a document that keeps getting updated:

```
0 * * * * python3 /opt/tools/mediafire-info.py \
  "https://www.mediafire.com/file/abcd12345/target/file" \
  >> /var/log/mf-monitor.log
```

The same metadata is useful on the other side of the fence. Public shares linked from a target's site or an employee's post can reveal document names, upload dates, and handles. It doesn't get you into anything private. It just saves you from clicking through pages by hand.

What it doesn't do

The endpoint is undocumented. MediaFire can change it, and this will break. I kept the parser simple so a schema change shows up as missing fields instead of quietly wrong data.

There is no documented rate limit. I send one request per link and don't loop over thousands of URLs. If you need volume, add a delay and cache the results.

Some files hide the owner or skip the hash. Older uploads are worse. The script doesn't guess.

The script needs a full URL. If you only have the key, call the endpoint yourself or rebuild the URL. Adding a bare-key mode is a five-line change; I just haven't needed it.

And the obvious one: metadata is not malware analysis. A clean hash and a plausible date don't make a file safe. If you need to inspect the contents, use a sandbox.

The script

Here's the whole thing. No imports beyond the standard library.

mediafire-info.py

```python
#!/usr/bin/env python3
"""
MediaFire File Info — OSINT Tool
Extract metadata from MediaFire file URLs via the public API.
No external dependencies — uses Python stdlib only.
"""

import re
import sys
import xml.etree.ElementTree as ET
from urllib.request import urlopen, Request
from urllib.error import URLError, HTTPError

# Color codes
GREEN = "\033[92m"
CYAN = "\033[96m"
YELLOW = "\033[93m"
MAGENTA = "\033[95m"
RED = "\033[91m"
BOLD = "\033[1m"
RESET = "\033[0m"


def extract_quick_key(url: str) -> str | None:
    """Extract file quick_key from any MediaFire URL format."""
    match = re.search(
        r"mediafire\.com/(?:file|download)/([a-zA-Z0-9]+)",
        url,
    )
    if match:
        return match.group(1)
    return None


def fetch_file_info(quick_key: str) -> str:
    """Fetch XML response from MediaFire API."""
```

```
    api_url = f"https://mediafire.com/api/1.5/file/get_info.php?quick_key=
    req = Request(api_url, headers={"User-Agent": "Mozilla/5.0"})

    try:
        with urlopen(req, timeout=15) as resp:
            return resp.read().decode("utf-8", errors="replace")
    except HTTPError as e:
        print(f"{RED}[!] HTTP {e.code}: {e.reason}{RESET}")
        sys.exit(1)
    except URLError as e:
        print(f"{RED}[!] Network error: {e.reason}{RESET}")
        sys.exit(1)


def parse_xml_response(xml_data: str) -> dict:
    """Parse XML response into a flat dict."""
    root = ET.fromstring(xml_data)

    # Check for API errors
    error = root.find("error")
    if error is not None and error.text and error.text.strip():
        print(f"{RED}[!] API Error: {error.text.strip()}{RESET}")
        sys.exit(1)

    result: dict = {}

    # Result status
    result_node = root.find("result")
    if result_node is not None:
        result["api_result"] = (result_node.text or "").strip()

    file_info = root.find("file_info")
    if file_info is None:
        print(f"{RED}[!] No <file_info> node in response.{RESET}")
        print(f"{YELLOW}Raw XML (first 500 chars):{RESET}")
        print(xml_data[:500])
        sys.exit(1)

    # Direct text children of file_info
    for child in file_info:
        tag = child.tag
        text = (child.text or "").strip()
```

```
79
80          if tag == "links":
81              links = {}
82              for link in child:
83                  links[link.tag] = (link.text or "").strip()
84              result["links"] = links
85          elif tag == "permissions":
86              perms = {}
87              for perm in child:
88                  perms[perm.tag] = (perm.text or "").strip()
89              result["permissions"] = perms
90          else:
91              result[tag] = text
92
93      return result
94
95
96  def format_bytes(size_str: str) -> str:
97      """Format bytes to human-readable."""
98      try:
99          size = int(size_str)
100     except (ValueError, TypeError):
101         return size_str
102
103     for unit in ("B", "KB", "MB", "GB", "TB"):
104         if size < 1024:
105             return f"{size:.2f} {unit}"
106         size /= 1024
107     return f"{size:.2f} PB"
108
109
110 def display_info(info: dict) -> None:
111     """Print parsed file info to terminal."""
112     print()
113     print(f"{BOLD}{'=' * 60}{RESET}")
114     print(f"{BOLD}  MEDIAFIRE FILE INFO{RESET}")
115     print(f"{BOLD}{'=' * 60}{RESET}")
116     print()
117
118     # File ready status
119     ready = info.get("ready", "no")
120     ready_icon = "✅" if ready == "yes" else "❌"
```

```
121         print(f"  {ready_icon} Status: {ready}")
122 
123         fields = [
124             ("Quick Key", quick_key_colored(info.get("quickkey", "-"))),
125             ("Filename", info.get("filename", "-")),
126             ("Description", info.get("description", "-") or "(none)"),
127             ("File Size", f"{format_bytes(info.get('size', '0'))} ({info.get(
128             ("File Type", info.get("filetype", "-")),
129             ("MIME Type", info.get("mimetype", "-")),
130             ("Uploaded", info.get("created", "-")),
131             ("Uploaded (UTC)", info.get("created_utc", "-")),
132             ("Downloads", info.get("downloads", "0")),
133             ("Views", info.get("view", "0")),
134             ("Revision", info.get("revision", "-")),
135             ("Privacy", info.get("privacy", "-")),
136             ("Password Protected", info.get("password_protected", "no")),
137         ]
138 
139         for label, value in fields:
140             print(f"  {CYAN}{label:20}{RESET} : {GREEN}{value}{RESET}")
141 
142         if info.get("owner_name"):
143             print(f"  {CYAN}{'Owner':20}{RESET} : {YELLOW}{info['owner_name']
144 
145         # SHA256 hash
146         if info.get("hash"):
147             print(f"  {CYAN}{'SHA256':20}{RESET} : {GREEN}{info['hash']}{RESE
148 
149         # Download links
150         links = info.get("links", {})
151         if links:
152             print(f"\n  {BOLD} Download Links:{RESET}")
153             for kind, url in links.items():
154                 label = kind.replace("_", " ").title()
155                 print(f"    {CYAN}{label:20}{RESET} : {url}")
156 
157         # Permissions
158         perms = info.get("permissions", {})
159         if perms:
160             print(f"\n  {BOLD} Permissions:{RESET}")
161             for perm, val in perms.items():
162                 icon = "✅" if val == "1" else "❌"
```

```
            print(f"    {perm.title():15} : {icon} ({val})")

    print()
    print(f"{BOLD}{'=' * 60}{RESET}")
    print()


def quick_key_colored(k: str) -> str:
    """Return quick key with color per char group."""
    return k


def main():
    if len(sys.argv) < 2:
        print(f"\n  {BOLD}Usage:{RESET} python3 {sys.argv[0]} <mediafire_
        print(f"  {BOLD}Example:{RESET} python3 {sys.argv[0]} \"https://w
        sys.exit(1)

    url = sys.argv[1]

    quick_key = extract_quick_key(url)
    if not quick_key:
        print(f"{RED}[!] Could not extract file ID from URL.{RESET}")
        print(f"    Expected: mediafire.com/file/<ID>/... or mediafire.co
        sys.exit(1)

    print(f"{YELLOW}[*] Quick Key: {quick_key}{RESET}")
    print(f"{YELLOW}[*] Fetching file info...{RESET}")

    xml_data = fetch_file_info(quick_key)
    info = parse_xml_response(xml_data)
    display_info(info)


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

The skill file

I also packaged it as an agent skill. The `SKILL.md` is mostly there so an agent knows when to reach for the script and what the output means. Drop the folder into your skills directory and the agent can call it without being told how the API works.

SKILL.md

```
---
name: mediafire-file-info
description: Extract file metadata from MediaFire URLs using the official A
---

# MediaFire File Info — OSINT

Extract metadata from MediaFire file URLs using MediaFire's public API end
No API key required.

---

## Description

Parses a MediaFire share URL (e.g. `mediafire.com/file/...` or `mediafire.

**Data returned:**
- File name & description
- File size (bytes)
- Upload date & last update
- Download count
- File type / MIME type
- Sharing status (public/private)
- Owner information (if public)
- Direct download link (if available)

---

## Local Usage

```bash
python3 scripts/mediafire-info.py "https://www.mediafire.com/file/7fpuwrsb
```

```

````
Or with the download variant:

```bash
python3 scripts/mediafire-info.py "https://www.mediafire.com/download/7fpu
```

Parameters

Argument
- **url** (string, required)
  - Full MediaFire share URL
  - Supports both `/file/` and `/download/` URL formats

Output

Parsed XML fields displayed as key-value pairs, including:
- `filename` — Original file name
- `size` — File size in bytes
- `downloads` — Total download count
- `created` — Upload timestamp
- `description` — Optional file description
- `mimetype` — File MIME type
- `revision` — Current file revision number
- `hash` — MD5/SHA256 hash if available

Rate Limiting

No documented rate limit. Polite usage recommended (1 request per query).

Known Issues

- Some files may have owner info hidden (returns `<owner name=""/>`)
- Very old files may not return hash data
- The API returns XML — the script parses it with Python's `xml.etree.Elem
````

```
77
78  ---
79
80  ## References
81
82  - MediaFire API (undocumented): `https://mediafire.com/api/1.5/file/get_in
83
84  ## Dependencies
85
86  - Python 3 stdlib only (no pip packages required)
87    - `xml.etree.ElementTree` — built-in XML parser
88    - `urllib.request` — built-in HTTP client
```

Running it

```
1  python3 scripts/mediafire-info.py \
2    "https://www.mediafire.com/file/7fpuwrsb2oelinf/MediaFire_-_Getting_Star
```

That's it. One request, one XML response, and a parser small enough to read before you trust it. If MediaFire changes the endpoint, I'll fix it. If not, it'll keep doing the one thing I need.

OSINT # Tooling # Python # Security
