

Tengu: Reverse Engineering a Mirai-Style Linux/IoT Botnet

 app.reverser.space/p/duckie/tengu-reverse-engineering-a-mirai-style-linux-iot

duckie

September 6, 2026

Interactive analysis [Open workspace ↗](#)

▼ Contents 14 min ▼



Figure 01 Sojobo, king of the tengu

Interactive analysis: Every hexadecimal address in this article is a verified address in the accompanying Reverser Space session. Click an address to jump directly to the corresponding instruction, function, or data in the Ghidra workspace.

Detection signatures at the end

Executive summary

The sample examined here, `tengu_sample`, is a stripped, statically linked, position-independent 32-bit x86 Linux ELF. Its behavior is consistent with a Linux bot designed for servers, embedded

systems, and IoT-adjacent environments:

- it hides behind a kernel-worker-style process name;
- protects itself from the Linux out-of-memory killer;
- installs persistence through systemd, SysV init, OpenWrt procd, cron, and rc.local;
- connects to an XOR-obfuscated endpoint at 64[.]89[.]163[.]8:9931;
- implements raw IPv4/UDP and datagram flooding;
- performs SSH banner and key-exchange handshakes;
- generates HTTP requests, including requests with spoofed forwarding headers;
- contains authenticated HTTP CONNECT and SOCKS5 proxy logic; and
- collects host and network configuration.

The binary calls itself **Tengu** in its persistence artifacts. It also contains a table of names and paths associated with Mirai and other Linux botnets. That makes "Mirai-style" a defensible description, but the static evidence in this session does not, by itself, prove direct source-code lineage from Mirai.

Sample profile

Table 01

Property	Value
File	tengu_sample
Format	ELF
Architecture	32-bit x86, little-endian
Size	198,288 bytes (193.6 KiB)
Linking	Static
Symbols	Stripped
Position independence	Enabled
NX	Enabled
RELRO	Partial
Stack canary	Not detected
Entropy	6.4761

Property	Value
Ghidra entry function	0x117A3
Main initialization function	0x1083D
MD5	3a1069cd649e22b87cbccf0c36b69f4b
SHA-1	097522a52986982b9eefc29f95efdd9d3b6032e7
SHA-256	897226af37990fa60f25fea00b0509faa0e78d8bee10875c23b9b6ab0b8faed9

Ghidra identified 464 functions, 389 strings, 3,631 cross-references, and 3,173 calls. Because the binary is static and stripped, most library wrappers retain generated names. During the analysis, useful names were assigned to several important functions, including [main_func](#), [ssh_scanner_flood](#), [ddos_udp_module](#), and [install_persistence](#).

From entry point to bot initialization

The ELF entry function is at [0x117A3](#), while the large initialization routine analyzed as [main_func](#) begins at [0x1083D](#). This routine prepares the process, hides its identity, constructs the command-and-control socket address, and starts the remaining subsystems.

OOM-killer protection

The malware references `/proc/self/oom_score_adj` at data address [0x33570](#). At [0x108EE](#) it loads that path and opens it through the file-open wrapper at [0x108F5](#).

The value written is **not** 5. The instruction at [0x10905](#) pushes a length of five bytes. The next instruction, [0x10907](#), loads the string `-1000` from [0x33589](#). The write occurs at [0x1090F](#) and the descriptor is closed at [0x10917](#).

On Linux, an `oom_score_adj` of `-1000` disables OOM killing for that task. This is therefore a direct attempt to improve the bot's survival during memory pressure.

Discovering its own executable

The path `/proc/self/exe` is stored at [0x334C7](#). The malware loads it at [0x1092C](#) and invokes its read-link wrapper at [0x10939](#). It then searches the resulting path for the suffix `(deleted)`, stored at [0x3358F](#): the suffix is loaded at [0x1094B](#), searched at [0x10953](#), and replaced with a null byte at [0x1095F](#) when present.

This gives the bot a clean path to its own executable even when Linux reports that the on-disk file has already been unlinked.

Daemonization and closed standard streams

The `/dev/null` path used by this branch is at [0x33255](#). It is loaded at [0x109B1](#) and opened at [0x109B8](#). The resulting descriptor is duplicated onto standard input, output, and error by calls at [0x109CC](#), [0x109D6](#), and [0x109E0](#). If the original descriptor is greater than 2, it is closed at [0x109F1](#).

This removes visible terminal output and lets the process continue as a background service.

Kernel-worker masquerading

The fake process-name format `[kworker/%d:%d]` is stored at [0x32000](#). `main_func` loads the template at [0x10B3B](#) and formats it at [0x10B51](#) using pseudo-random numeric components. The generated name is copied over the original argument memory at [0x10B72](#) and terminated at [0x10B7F](#).

The result resembles a Linux kernel worker thread in ordinary process listings, although the bracketed userspace process remains distinguishable through deeper inspection.

Recovering the command-and-control endpoint

The endpoint is assembled at runtime instead of appearing as a normal IP-address string.

At [0x10A4A](#), the program writes address family 2 (`AF_INET`) into the global socket structure. At [0x10A53](#) it places the dword `2A817B62h` on the stack. Because the target is little-endian, the encoded bytes are:

Text

62 7B 81 2A

The decoding loop begins at [0x10A64](#). Each byte is XORed with key `22h` at [0x10A6B](#), shifted into position at [0x10A78](#), and merged into the accumulator at [0x10A7A](#). The loop terminates through the branch at [0x10A80](#), and the completed value is stored in the global IPv4 field at [0x10A82](#).

Text

62h XOR 22h = 40h = 64
7Bh XOR 22h = 59h = 89
81h XOR 22h = A3h = 163
2Ah XOR 22h = 08h = 8

The decoded address is therefore `64.89.163.8`, stored at global address [0x4166C](#).

The port literal 26CBh is pushed at [0x10A8B](#), converted to network byte order by the wrapper called at [0x10A90](#), and written into the global port field at [0x10A95](#). Decimal 26CBh is 9931, producing the endpoint:

Text

64[.]89[.]163[.]8:9931

Later, the initialized socket structure at [0x41668](#) is passed to the connection wrapper at [0x11035](#).

Persistence across Linux distributions

The persistence routine begins at [0x1FFCB](#). Rather than relying on one service manager, it tries several mechanisms associated with desktop/server Linux and embedded distributions.

systemd

At [0x20281](#), the malware constructs `/etc/systemd/system/%s.service` using the format string at [0x32425](#). It opens the resulting path at [0x20294](#), assembles a service body from [0x202B0](#) to [0x202DE](#), writes it at [0x202EE](#), and closes it at [0x202F6](#).

The service template begins at [0x33C0C](#) and identifies itself as System Helper Service. Its restart and install directives are stored at [0x33C6C](#) and include `Restart=on-failure` and `WantedBy=multi-user.target`.

The code constructs `/etc/systemd/system/multi-user.target.wants` at [0x20313](#), formats the `%s/%s.service` link name at [0x20342](#), and reaches the link creation path from [0x2034B](#) to [0x2035B](#).

OpenWrt procd and SysV init

The fallback init-script path `/etc/init.d/%s`, stored at [0x32543](#), is formatted at [0x203CE](#).

The first script body, written through [0x20401](#), begins at [0x33D07](#). It uses `/etc/rc.common`, `USE_PROCD=1`, and `procd_set_param respawn`, identifying it as an **OpenWrt procd** script rather than an OpenRC script.

The alternative SysV-style script is selected at [0x20419](#). Its body at [0x33DB7](#) contains LSB init metadata and start, stop, and restart actions.

The routine also constructs `/etc/rc.d/S99%s` at [0x2047A](#) from the format string at [0x33F07](#). A second startup-link format, `%s/S90%s`, is built at [0x204D6](#) from [0x33F17](#).

Cron

At [0x20514](#) the malware runs `crontab -l 2>/dev/null`, stored at [0x33F20](#). If its executable path is absent from the existing crontab, it builds the following pipeline at [0x20543](#) from the template at

[0x33F37](#):

Shell

```
(crontab -l 2>/dev/null; echo '@reboot %s') | crontab - 2>/dev/null
```

The command is executed at [0x2054B](#).

/etc/rc.local and additional startup files

The /etc/rc.local string resides at [0x33F7B](#). The first existing file check occurs at [0x205C3](#). If the file is missing, the routine can create a minimal shell script from [0x2066B](#) to [0x2069B](#). It reopens the file for rewriting at [0x206AD](#), creates a temporary path at [0x206D4](#), inserts its launch command before `exit 0` from [0x2072B](#) to [0x20779](#), and replaces the original file from [0x20785](#) to [0x207B0](#).

The appended launch-line template, `\n[ -x %s ] && %s &\n`, is formatted at [0x207DB](#).

Finally, helper calls at [0x20852](#) and [0x2085D](#) target /etc/crontab at [0x323B2](#) and /etc/init.d/tengu at [0x33FC2](#). The standalone name tengu appears at [0x33FCE](#), tying the family name directly to its persistence artifacts.

UDP and raw-packet flooding

The analyst-renamed [ddos_udp_module](#) begins at [0x1BC2D](#) and contains two distinct transmission paths.

Raw IPv4/UDP packets

At [0x1BD67](#) the routine requests a raw IPv4 socket with arguments corresponding to `AF_INET` and `SOCK_RAW`. The call at [0x1BD94](#) appears to enable caller-supplied IP headers.

For each worker, it allocates a packet buffer at [0x1BDED](#). It writes the IPv4 version/IHL byte 45h at [0x1BE06](#) and the UDP protocol number 11h at [0x1BE5E](#). Header fields are populated through the conversion calls at [0x1BE1F](#), [0x1BE30](#), [0x1BE83](#), [0x1BE94](#), and [0x1BEAB](#). The destination address is copied into the IP header at [0x1BE77](#).

Pseudo-random values are incorporated from [0x1BF05](#) to [0x1BF29](#). The code then calculates the IP checksum at [0x1C081](#) and the UDP checksum at [0x1C09E](#). The completed packet reaches the transmission wrapper at [0x1C0D2](#), followed by a pacing call at [0x1C0DE](#).

Datagram-socket mode

The second branch converts the target port at [0x1C20D](#) and creates `AF_INET/SOCK_DGRAM` sockets at [0x1C22B](#). It builds a buffer for each worker beginning at [0x1C292](#), adds timing and random material from [0x1C2A2](#) to [0x1C327](#), optionally fills a payload at [0x1C348](#), and transmits it through the wrapper at [0x1C368](#).

The separation between a hand-crafted raw-packet path and a normal datagram path gives the operator control over spoofing, header fields, payload length, and operating environments where raw sockets may not be available.

SSH banner and key-exchange activity

The function renamed [ssh_scanner_flood](#) begins at [0x168F2](#). It constructs an SSH key-exchange packet using recognizable algorithm lists:

- key exchange at [0x169EE](#), backed by the string at [0x32AA5](#);
- host keys at [0x16A37](#), backed by [0x32AF0](#);
- ciphers at [0x16A81](#) and [0x16AC5](#), backed by [0x32B14](#);
- MAC algorithms at [0x16B10](#) and [0x16B54](#), backed by [0x32B4B](#); and
- compression at [0x16B9F](#) and [0x16BE3](#), backed by [0x32B6C](#).

For each target, the routine creates an AF_INET/SOCK_STREAM socket at [0x16D54](#). It fills the destination address and port from [0x16D67](#) to [0x16D80](#) and from [0x16DA0](#) to [0x16DB3](#), then invokes the connection wrapper at [0x16DEA](#).

After waiting for socket readiness at [0x16E74](#), it reads a response at [0x16E93](#) and searches the buffer for SSH- at [0x16EB2](#). A successful banner match leads to sends at [0x16ED0](#) and [0x16F57](#), with readiness waits between them.

This supports describing the feature as an SSH banner scanner and handshake generator. Static analysis alone does not show an authentication exploit or credential attack in this function.

HTTP request generation

The function at [0x1454F](#) constructs three ordinary HTTP/1.1 request forms:

- GET at [0x145DD](#), using the template at [0x32143](#);
- POST at [0x14604](#), using the template at [0x32189](#); and
- HEAD at [0x14627](#), using the template at [0x32227](#).

All three use caller-provided paths, hosts, and user agents and request keep-alive connections. The POST body is a fixed data=random_data form submission.

A second HTTP routine begins at [0x16FDE](#). At [0x17256](#) it formats the larger template stored at [0x32B85](#). That template adds randomized X-Forwarded-For and CF-Connecting-IP addresses plus CDN-Loop: cloudflare, then sends the request at [0x17266](#). Candidate paths stored nearby include /index.php at [0x32C6C](#) and /wp-login.php at [0x32C77](#).

The repeated connection logic, randomized forwarding headers, and burst-oriented send paths are consistent with application-layer request flooding rather than a normal HTTP client.

HTTP CONNECT and SOCKS5 proxy support

The function at [0x15371](#) implements an HTTP proxy path. It searches incoming data for Proxy-Authorization: Basic at [0x1540D](#) using the string at [0x32873](#). It compares the request method with CONNECT at [0x15572](#) using [0x328DD](#).

After establishing the upstream side, it sends HTTP/1.1 200 Connection Established at [0x157AD](#) from [0x328E5](#). The same data region contains explicit 407 Proxy Auth Required and 502 Bad Gateway responses at [0x3288F](#) and [0x3290D](#).

A separate configuration routine begins at [0x15CAB](#). It recognizes the option socks5 at [0x15CD1](#) from data address [0x3292A](#) and creates /tmp/.proxy.pid at [0x15DA5](#) from [0x32936](#). Together, these control-flow and protocol strings establish proxy behavior more strongly than isolated strings would.

Host and network discovery

The function at [0x14936](#) collects basic network configuration. It opens /proc/net/route at [0x14B73](#) using the string at [0x322CE](#). It later opens /etc/resolv.conf at [0x14C4C](#) from [0x322FE](#) and parses nameserver %63s at [0x14CAD](#).

Additional data strings describe CPU, RAM, uptime, process count, and system load, beginning with /proc/cpuinfo at [0x329C1](#) and the report heading at [0x329D7](#). These artifacts indicate host fingerprinting that can inform targeting, reporting, or task selection.

What the sample says about its lineage

The family name has direct support: /etc/init.d/tengu appears at [0x33FC2](#) and is acted on at [0x2085D](#).

The Mirai connection requires more care. The function at [0x20A40](#) iterates a pointer table beginning at [0x40D40](#). The table is loaded at [0x20A54](#) and each entry is compared against an input string at [0x20A62](#). Its entries include:

- tsunami at [0x345D3](#);
- gafgyt at [0x345E0](#);
- hajime at [0x345F3](#);
- /tmp/mirai at [0x34650](#);
- /dev/shm/bot at [0x3465B](#); and
- /tmp/.bot at [0x34671](#).

This looks like a list used to detect competing malware, processes, and paths. It demonstrates awareness of Mirai and related botnets, but it does not prove that Tengu was compiled from Mirai source. The DDoS focus, process masquerading, embedded Linux persistence, and competitor

handling make "Mirai-style" a useful behavioral label. A stronger lineage claim would require a code comparison against known Mirai implementations or a matching command-and-control protocol.

Indicators of compromise

Table 02Scroll to explore →

Type	Indicator	Evidence
IPv4	64[.]89[.]163[.]8	Decoded from 0x10A64 to 0x10A82
TCP port	9931	Constructed from 0x10A8B to 0x10A95
Process name	[kworker/%d:%d]	Format at 0x32000 ; applied at 0x10B51
Init path	/etc/init.d/tengu	String at 0x33FC2 ; used at 0x2085D
PID file	/tmp/.proxy.pid	String at 0x32936 ; used at 0x15DA5
systemd description	System Helper Service	Template at 0x33C0C
SHA-256	897226af37990fa60f25fea00b0509faa0e78d8bee10875c23b9b6ab0b8faed9	Session hash metadata

The network endpoint should be treated as historical analysis data until separately validated. Ownership and behavior of an IP address can change over time.

Defensive observations

Useful detection opportunities include:

- a userspace process presenting a bracketed kworker name while its executable does not map to a kernel thread;
- writes of -1000 to /proc/self/oom_score_adj by an unexpected process;
- creation of System Helper Service units under /etc/systemd/system;
- creation or modification of /etc/init.d/tengu, /etc/rc.local, or unexpected @reboot cron entries;
- /tmp/.proxy.pid combined with a listener implementing HTTP CONNECT or SOCKS5;
- outbound connections to the observed endpoint; and
- bursts of raw UDP packets, SSH handshakes, or HTTP requests with randomized forwarding headers.

These should be combined with process ancestry, executable hashes, file ownership, and network telemetry. Any single string or path may be imitated or reused by unrelated software.

Conclusion

Tengu is a compact but broad Linux bot. Its initialization path prioritizes survival and concealment; its persistence code spans systemd, SysV, OpenWrt, cron, and rc.local; and its network modules cover raw UDP, datagram traffic, SSH handshakes, HTTP requests, and proxying.

The most important recovered configuration is the XOR-decoded endpoint 64[.]89[.]163[.]8:9931. The strongest family-specific artifact is the tengu init path. Its Mirai relationship is plausible based on behavior, but should remain qualified as "Mirai-style" until a code or protocol comparison provides firmer evidence.

Detection

Yara

ASM

```
rule MAL_Linux_Tengu_Botnet_2026 : malware linux botnet tengu
{
    meta:
        description = "Detects the Tengu Linux botnet sample and close variants"
        reference =
            "https://app.reverser.space/v/56fd64025ff28065f10c862ba572bb36675d41993d97f694"
        date = "2026-09-06"
        sample_sha256 = "897226af37990fa60f25fea00b0509faa0e78d8bee10875c23b9b6ab0b8faed9"
        confidence = "high"

    strings:
```

```

/* Family and process identity */
$id_tengu_init = "/etc/init.d/tengu" ascii
$id_kworker = "[kworker/%d:%d]" ascii

/* Persistence and process survival */
$persist_oom_path = "/proc/self/oom_score_adj" ascii
$persist_systemd = "Description=System Helper Service" ascii
$persist_openwrt = "procd_set_param respawn" ascii
2>/dev/null" ascii
$persist_rc_local = "[ -x %s ] && %s &" ascii

/* Proxy, scanning, and flooding capabilities */
$cap_proxy_pid = "/tmp/.proxy.pid" ascii
$cap_proxy_auth = "Proxy-Authorization: Basic " ascii
$cap_proxy_reply = "HTTP/1.1 407 Proxy Auth Required" ascii
$cap_ssh_banner = "SSH-2.0-OpenSSH_8.9p1 Ubuntu-3" ascii
$cap_ssh_kex = "curve25519-sha256,diffie-hellman-group14-sha256,diffie-hellman-
group1-sha1" ascii
$cap_http_spoof = "X-Forwarded-For: %d.%d.%d.%d\r\nCF-Connecting-IP:
%d.%d.%d.%d\r\nCDN-Loop: cloudflare" ascii
$cap_attack_vector = "Unknown attack vector: %d" ascii

/*
 * Encoded C2 IPv4 dword 2A817B62h followed by the bitwise XOR 22h
 * decode loop observed at virtual address 0x10A53.
 */
$code_c2_decode = {
    C7 85 ?? ?? ?? ?? 62 7B 81 2A
    83 C4 10 31 D2 31 FF
    8A 84 15 ?? ?? ?? ??
    83 F0 22 0F B6 C0
    8D 0C D5 00 00 00 00
    D3 E0 09 C7 42 83 FA 04 75 ??
    89 BB 6C 06 00 00
}

$code_c2_port = {
    68 CB 26 00 00
    E8 ?? ?? ?? ??
    66 89 83 6A 06 00 00
}

condition:
/* 32-bit little-endian ELF, capped to reduce broad file scanning. */
uint32(0) == 0x464C457F and
uint8(4) == 1 and
uint8(5) == 1 and
filesize < 1MB and
(
    (
        all of ($code_*) and

```

```

        2 of ($persist_*)
    )
    or
    (
        $id_tengu_init and
        $id_kworker and
        2 of ($persist_*) and
        3 of ($cap_*)
    )
}

```

References

- [Interactive Reverser Space session](#)
- [Reverser Space agent/API documentation](#)
- [Linux kernel documentation for oom_score_adj](#)
- [Wikimedia Commons source for the lead image](#)

Evidence index

Every function, string and address cited in this investigation. Select one to inspect it in the binary.

Functions

4

- Sample profile, From entry point to bot initialization, Kernel-worker masquerading3×
- Sample profile, SSH banner and key-exchange activity2×
- Sample profile, UDP and raw-packet flooding2×
- Sample profile1×

Addresses

183

- Sample profile, From entry point to bot initialization2×
- Sample profile, From entry point to bot initialization2×
- OOM-killer protection1×
- OOM-killer protection1×
- OOM-killer protection1×
- OOM-killer protection1×
- OOM-killer protection1×
- OOM-killer protection1×
- OOM-killer protection1×
- OOM-killer protection1×
- Discovering its own executable1×

- Discovering its own executable1×
- Discovering its own executable1×
- Discovering its own executable1×
- Discovering its own executable1×
- Discovering its own executable1×
- Discovering its own executable1×
- Daemonization and closed standard streams1×
- Daemonization and closed standard streams1×
- Daemonization and closed standard streams1×
- Daemonization and closed standard streams1×
- Daemonization and closed standard streams1×
- Daemonization and closed standard streams1×
- Daemonization and closed standard streams1×
- Kernel-worker masquerading, Indicators of compromise2×
- Kernel-worker masquerading1×
- Kernel-worker masquerading, Indicators of compromise2×
- Kernel-worker masquerading1×
- Kernel-worker masquerading1×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint, Indicators of compromise2×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint, Indicators of compromise2×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint, Indicators of compromise2×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint, Indicators of compromise2×
- Recovering the command-and-control endpoint1×
- Recovering the command-and-control endpoint1×
- Persistence across Linux distributions1×
- systemd1×
- systemd1×
- systemd1×
- systemd1×
- systemd1×
- systemd1×
- systemd1×
- systemd, Indicators of compromise2×

- systemd1×
- systemd1×
- systemd1×
- systemd1×
- systemd1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- OpenWrt procd and SysV init1×
- Cron1×
- Cron1×
- Cron1×
- Cron1×
- Cron1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files, What the sample says about its lineage, Indicators of compromise3×
- /etc/rc.local and additional startup files1×
- /etc/rc.local and additional startup files, What the sample says about its lineage, Indicators of compromise3×
- /etc/rc.local and additional startup files1×
- UDP and raw-packet flooding1×
- Raw IPv4/UDP packets1×
- Raw IPv4/UDP packets1×

- [illegible]

- SSH banner and key-exchange activity1×
- SSH banner and key-exchange activity1×
- SSH banner and key-exchange activity1×
- SSH banner and key-exchange activity1×
- SSH banner and key-exchange activity1×
- SSH banner and key-exchange activity1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP request generation1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support1×
- HTTP CONNECT and SOCKS5 proxy support, Indicators of compromise2×
- HTTP CONNECT and SOCKS5 proxy support, Indicators of compromise2×
- Host and network discovery1×
- Host and network discovery1×
- Host and network discovery1×
- Host and network discovery1×
- Host and network discovery1×
- Host and network discovery1×
- Host and network discovery1×
- Host and network discovery1×

- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×
- What the sample says about its lineage1×

Your workspace

Take the analysis apart yourself

Open a private copy to rename functions, leave comments and put agents to work without changing this frozen writeup. You can also take the original sample.

Your AI

Ask the binary the next question

Connect any MCP client to decompile, cross-reference and search this same analysis with your own model.

MCP endpoint<https://api.reverser.space/mcp/v/b4d3d8910054493011687646329cc60a>

Claude Code
`claude mcp add --transport http tengu-reverse-engineering-a-mirai-style-linux-iot https://api.reverser.space/mcp/v/b4d3d8910054493011687646329cc60a`

More writeups
