

TuxBot v3: Inside an IoT Botnet Framework With LLM-Assisted Development

 unit42.paloaltonetworks.com/tuxbot-v3-evolution-iot-botnet

Chris Navarrete, Asher Davila, Doel Santos

July 15, 2026

Executive Summary

We identified a previously undocumented modular internet-of-things (IoT) botnet framework named TuxBot v3 Evolution.

The malware authors leveraged an LLM to assist in their code development, yielding mixed results. While the AI complied with their request to generate botnet code, it included a safety disclaimer that the developer failed to remove before shipping.

Although the LLM clearly aided in constructing the botnet, several functions in the analyzed samples failed to work correctly. While a manual code review could have easily resolved these errors, the authors neglected this step. However, it is highly likely that corrected, more polished iterations exist, which significantly elevates the potential threat posed by this malware.

We [initially reported this information](#) through our Timely Threat Intelligence program, and this article provides further in-depth analysis of the TuxBot v3 Evolution botnet.

We recovered detailed information on the framework from internal telemetry. The data includes the full source code, compiled binaries for 17 architectures and automated distributed denial of service (DDoS) performance testing reports. The bot programs infected devices to display the console banner “Infected By Akiru.”

The TuxBot v3 Evolution framework consists of:

- A C-based bot agent that cross-compile for architectures from ARM and MIPS to x86_64, PowerPC, RISC-V, etc.
- A Go-based command-and-control (C2) server with a DDoS-for-hire panel
- A custom exploit virtual machine
- Docker-based test infrastructure
- An automated build system

The bot agent brute-forces Telnet access on targeted devices with 1,496 credential pairs, contains exploit code targeting more than 30 IoT device families and communicates with a C2 server over an encrypted TCP channel.

Fall-back C2 mechanisms include:

- A SHA512 domain generation algorithm (DGA)

- Peer-to-peer (P2P) gossip with Ed25519-signed commands
- IRC
- DNS TXT queries
- HTTP polling

Palo Alto Networks customers are better protected from the threats discussed above through the following products:

- [Advanced WildFire](#)
- [Advanced URL Filtering](#) and [Advanced DNS Security](#)
- [Advanced Threat Prevention](#)

If you think you might have been compromised or have an urgent matter, contact the [Unit 42 Incident Response team](#).

Related Unit 42 Topics [LLM](#), [AI](#), [Botnet](#), [IoT](#), [DDoS](#)

TuxBot Framework Details

TuxBot is a modular IoT botnet framework derived from various known IoT botnet codebases. Based on our analysis of the samples, TuxBot includes features borrowed from the known botnet [AISURU](#) and the publicly unknown Wuhan botnet lineages. (We infer the Wuhan botnet lineage based on references in the TuxBot samples.)

It is also partially ported from the open-source [MHDDoS Python DDoS](#) toolkit.

Figure 1 shows screenshots of the TuxBot v3 Evolution installer. According to the system configuration, the framework maintains dual versioning: 3.5.2 for the Installer version and 3.0.0-EVOLUTION-FINAL within the Docker configuration file.

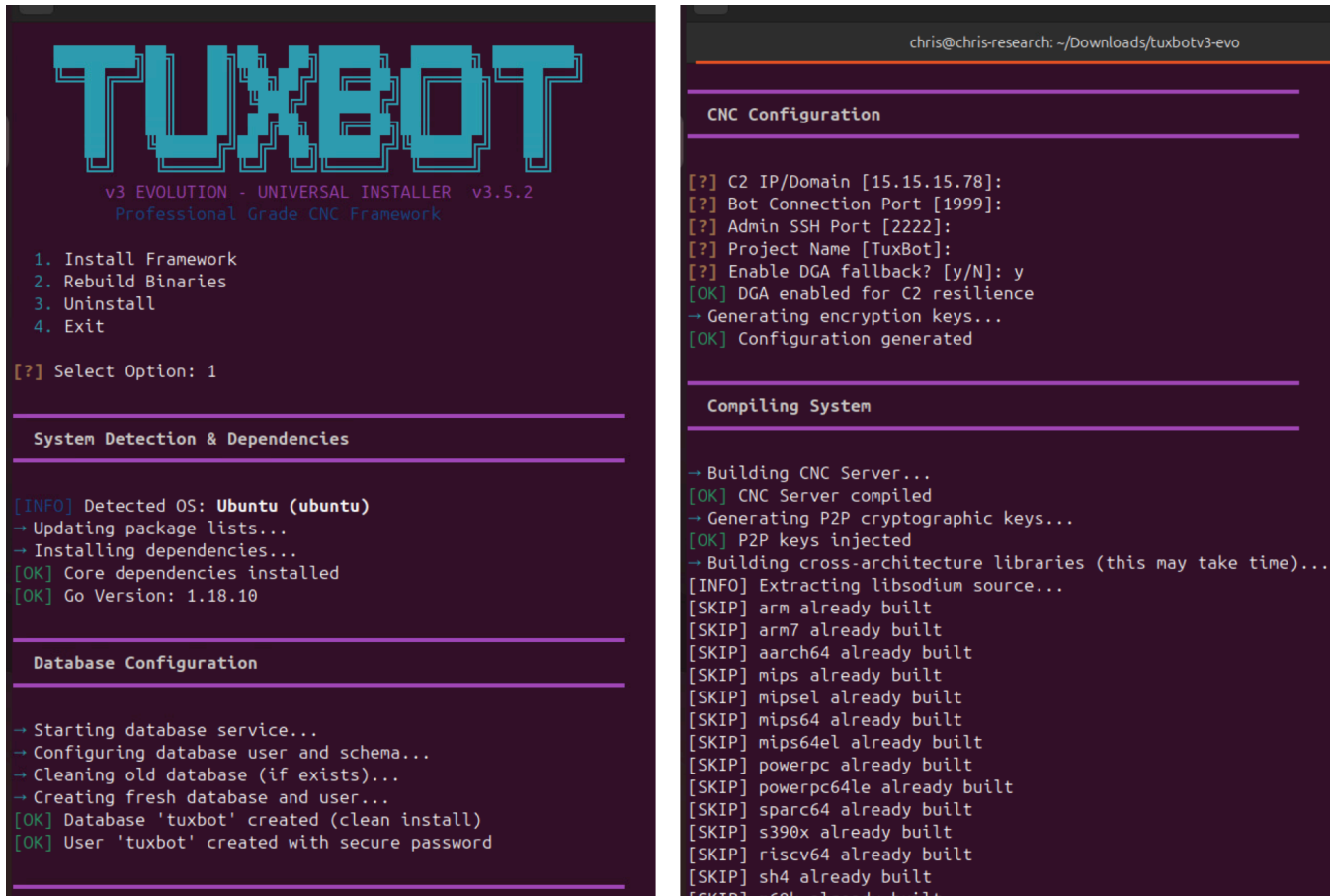


Figure 1. TuxBot interactive setup wizard screens.

We discovered two important sources of TuxBot data from the wild. Our first discovery was an archive containing the complete source code of the framework. This archive consists of:

- 61 C++ source files
- 58 headers
- Its own compiler and virtual machine
- Docker Compose configurations for test environments
- Quick Emulator (QEMU) setups for multi-architecture testing
- 254 automated DDoS benchmark reports

Our second discovery was a [compiled bot binary](#) that was also bundled in the source tree under the QEMU test directory and hidden with a dot-prefix to the filename. This sample was submitted to VirusTotal on Jan. 20, 2026.

Comparing this binary with the source code reveals that it is a development build. This binary was compiled with its C2 IP address set to the loopback IP address 127.0.0.1 and the bot protocol port set to 31337. Because this information can be modified during the botnet setup process, the operator could have production builds with a real C2 IP address and with the bugs we document here already fixed.

The TuxBot framework we recovered and analyzed is approximately 70% functional. The core infection flow (scanning, credential brute-forcing, persistence, primary C2 setup and DDoS

execution) works. The Telnet, SSH, HTTP and Android Debug Bridge (ADB) scanners all operate correctly. Furthermore, with its 1,496 credential pairs, the Telnet scanner remains a viable infection vector.

Exploitation beyond brute-forcing is limited. All three exploit systems are non-functional for different reasons that we detail later in this analysis. An additional scanner fires, but its hard-coded dropper IP address is no longer active.

Several other features are broken due to a handful of bugs, most of which trace back to large language model (LLM)-assisted development. The developer relied on an LLM to generate C modules, port exploits and write C2 server code.

Raw chain-of-thought reasoning from the LLM was left verbatim in source files, and the LLM hallucinated cryptographic implementations that the developer shipped without verifying. During our analysis, we could fix several of these broken features with a few targeted prompts to an LLM. This means an adversary with access to the same source code could produce a more complete version with minimal effort.

Development Timeline

The archive containing the source code also contains a Git log. This Git log allowed us to build a timeline that shows the development progress of this botnet, noted in Table 1.

Date	Event	Evidence
Jan. 3, 2025	Developer clones MHDDoS (DDoS attack script) from GitHub	Git log in the MHDDoS/ subdirectory leaks the workstation hostname newtuxdev.sevielw.digikalas[.]online
Aug. 6, 2025	Developer domain digikalas[.]online registered	Namecheap registration with Icelandic privacy protection (Withheld for Privacy ehf), Cloudflare DNS
Jan. 4–6, 2026	254 automated DDoS benchmark reports generated	The package found includes reports, directory timestamps and JSON test configurations against Docker targets

Jan. 20, 2026	The first TuxBot sample appears on VirusTotal	SHA256 hash 71dfbb171eca4ef9d02ff630b56e5283bbef7b375d4dbe9e8c9531bef312fa8d, x86_64 debug build with symbols
March 5, 2026	C2 server first seen on Xpanse	209.182.237[.]1133:2222, banner SSH-2.0-CNC-Control-Server
April 22, 2026	Six new samples detected in internal telemetry	Not on VirusTotal, multiple architectures, GCC 14.2.0 production builds

Table 1. TuxBot Framework development timeline.

The source code and publicly available data provide a rough development chronology. The developer's hostname, captured in the included Git log, indicates an Iranian-hosted workstation. The developer domain newtuxdev.sevielw.digikalas[.]online was no longer live, but the parent domain digikalas[.]online has remained active and resolved to an IP address on Iran's Arvan Cloud content delivery network (CDN) during our research.

The 254 benchmark reports from the archive from January 2026 reveal:

- Active testing of 12 attack methods across three Docker-based botnet hosts
- Measuring packet rates, throughput and error rates

This testing occurred just weeks before the first sample appeared on VirusTotal, consistent with a late-stage development push before deployment.

The source code contains an IP address of 185.10.68[.]127, which we pivoted on to link TuxBot to Keksec/Kaitori (a Tsunami/Mirai/Gafgyt variant) ecosystems to a shared infrastructure.

Framework Overview

According to the framework's description, the TuxBot developer built what they called a professional-grade C2 framework platform with a multi-user admin panel, automated deployment and modular attack capabilities. Figure 2 shows the botnet panel reference.

```
botmaster@tuxbot:~$ help
```

COMMAND REFERENCE

User Commands:

Command	Description
help	Display this help message
methods	Show all available attack methods
stats	Display botnet statistics
dashboard	Show advanced real-time dashboard
bots	List connected bot distribution
attacks	Show recent attacks
killall	Stop all running attacks
recent [N]	Show last N attacks (default 10)
mystats	Show your personal statistics
globalstats	Show global system statistics
clear	Clear the screen
exit	Disconnect from CNC

Admin Commands:

Command	Description
adduser	Create a new user account
remuser	Remove a user account
list [type]	List bots by device type
attacks enable	Enable global attack slots
attacks disable	Disable global attack slots

Attack Usage:

```
!<method> <target> <duration> [options]
```

```
Example: !udp 1.1.1.1 60 dport=80
```

```
botmaster@tuxbot:~$
```

Figure 2. TuxBot C2 botmaster panel command reference.

The C2 server is written in Go and uses three listeners that use different TCP ports for incoming connections.

The first listener serves the bot protocol on TCP port 1999 (or 31337, depending on the build), handling encrypted command dispatch to connected bots. The same port is multiplexed with an admin binary protocol identified by a magic byte header.

The second listener is an SSH server on TCP port 2222 that presents an interactive shell for operators. This is the DDoS-for-hire interface shown in Figure 3. Operators log in, see a count of connected bots and issue attack commands in the format !method target duration.

```
+=====+
|          TUXBOT C2 FRAMEWORK          |
|  SSH Secured | 92 Methods              |
+=====+

SYSTEM STATUS
✓ Authentication successful
✓ Secure connection established
✓ System ready

[*] Type 'help' for available commands
[*] Type 'methods' to see attack methods

botmaster@tuxbot:~$
```

Figure 3. TuxBot SSH C2 control panel system status screen.

As Figure 4 shows, the C2 server enforces per-user quotas on concurrent attacks, maximum duration and bot allocation. This is all backed by a MariaDB database that stores user accounts, attack logs and permissions.

```
botmaster@tuxbot:~$ stats

SYSTEM STATISTICS

Metric      Value
Active Bots 1
Running Attacks 0
Uptime      00h 33m 45s
Server Status ● ONLINE

botmaster@tuxbot:~$ bots
[*] Total Bots: 1
[*] encrypted: 1
botmaster@tuxbot:~$
```

Figure 4. TuxBot system statistics showing currently connected bots.

The third listener is a machine API on TCP port 9999 that uses a JSON interface intended for programmatic access.

The integrated build system automates the entire deployment:

- Installing dependencies (Go, MariaDB, cross-compilation toolchains)
- Initializing the database schema

- Generating a configuration
- Compiling the C2
- Cross-compiling the bot for 17 target architectures, as noted in Figure 5

```
[+] Loaded: ThinkPHP RCE (3 steps)
[+] Loaded: TP-Link RCE (setLanguage) (3 steps)
[+] Loaded: TP-Link WAN-Side RCE (3 steps)
[+] Loaded: Zavio IP Camera RCE (3 steps)
[+] Loaded: ZTE RCE (Login Bypass + Cmd Inj) (5 steps)
[*] Compiling 27 exploits to bin/cnc/exploits.pkg...
[+] Success! Size: 10694 bytes
[+] Exploits compiled
[*] Starting bot compilation...

→ Building for x86_64 (gcc)... OK (1.5MiB)
→ Building for arm (arm-linux-gnueabi-gcc)... OK (1.2MiB)
→ Building for arm7 (arm-linux-gnueabihf-gcc)... OK (950KiB)
→ Building for arm64 (aarch64-linux-gnu-gcc)... OK (1.3MiB)
→ Building for mips (mips-linux-gnu-gcc)... OK (1.4MiB)
→ Building for mipsel (mipsel-linux-gnu-gcc)... OK (1.4MiB)
→ Building for mips64 (mips64-linux-gnuabi64-gcc)... OK (1.5M
→ Building for mips64el (mips64el-linux-gnuabi64-gcc)... OK (
→ Building for ppc (powerpc-linux-gnu-gcc)... OK (1.5MiB)
→ Building for ppc64le (powerpc64le-linux-gnu-gcc)... OK (1.7
→ Building for sparc64 (sparc64-linux-gnu-gcc)... OK (2.2MiB)
→ Building for s390x (s390x-linux-gnu-gcc)... OK (1.5MiB)
→ Building for riscv64 (riscv64-linux-gnu-gcc)... OK (1.2MiB)
→ Building for sh4 (sh4-linux-gnu-gcc)... OK (1.1MiB)
→ Building for m68k (m68k-linux-gnu-gcc)... OK (1.2MiB)
→ Building for alpha (alpha-linux-gnu-gcc)... OK (1.4MiB)
→ Building for hppa (hppa-linux-gnu-gcc)... OK (1.3MiB)

[+] All 17 bot(s) compiled successfully!

[*] Output directory: /home/chris/Downloads/tuxbotv3-evo/bin/bo
tuxbot.alpha (1456432 bytes)
tuxbot.arm (1234964 bytes)
tuxbot.arm64 (1329000 bytes)
tuxbot.arm7 (972032 bytes)
tuxbot.hppa (1352256 bytes)
tuxbot.m68k (1160756 bytes)
```

Figure 5. Exploit loading and cross-compilation of bot clients (malware artifacts).

These target architectures include:

- x86_64
- ARM
- ARM64

- MIPS
- MIPSEL
- MIPS64
- PowerPC

The compiled binaries are placed in a directory served over HTTP, so exploited devices can download the appropriate binary for their architecture.

The framework includes [Docker Compose](#) configurations for several test scenarios. A “battle arena” configuration spins up a C2 server, five bot replicas and a target host running nginx and socat listeners on game server ports (Minecraft, TeamSpeak, FiveM, Xbox Live). This allowed the developer to test DDoS methods against real protocol listeners in a controlled environment. Additional configurations test P2P gossip recovery, full integration with all scanners active and production-like deployments with stealth and persistence enabled.

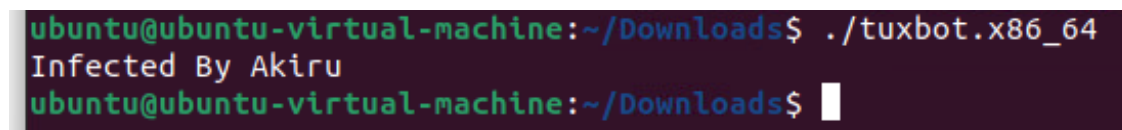
An interesting design note is that the source code configures the SSH banner as SSH-2.0-CNC, but the live C2 server on Xpanse at 209.182.237[.]133 presents the banner as SSH-2.0-CNC-Control-Server. This discrepancy suggests that the production deployment uses a modified version of the source code we discovered, providing further evidence that the operator has a separate, potentially more complete build.

Bot Overview

Analysis Details

The bot is a C program that compiles into a single statically linked binary. It links against [glibc](#) and [libsodium](#) for X25519, ChaCha20, Poly1305, SHA512 and Ed25519 algorithms.

The original [binary submitted to VirusTotal](#) was an earlier debug build with symbols intact, compiled with GCC 11.4.0 instead of the production GCC 14.2.0. The bot programs infected devices to display the console banner Infected By Akiru, as shown in Figure 6.

A terminal window with a dark background. The prompt is 'ubuntu@ubuntu-virtual-machine:~/Downloads\$'. The user has entered './tuxbot.x86_64'. The output is 'Infected By Akiru' in a light blue font. The prompt is now 'ubuntu@ubuntu-virtual-machine:~/Downloads\$' with a cursor at the end.

```
ubuntu@ubuntu-virtual-machine:~/Downloads$ ./tuxbot.x86_64
Infected By Akiru
ubuntu@ubuntu-virtual-machine:~/Downloads$
```

Figure 6. Post-infection screen console message.

On execution, the bot follows a fixed initialization sequence. After seeding the pseudo-random number generator and initializing libsodium, it performs the following activities:

- Loading the C2 address
- Setting up anti-debugging protections
- Hiding its process name
- Installing persistence

- Launching a cascade of subsystems consisting of:
 - The attack dispatcher
 - A competitor killer feature
 - An exploit VM
 - Self-replication servers
 - Multiple C2 channels (IRC, HTTP, DNS, P2P)
 - Scanners (Telnet, SSH, HTTP, PHP-based application, ADB)
 - A SOCKS5 proxy
 - The mining placeholder

The main process then enters a loop that receives encrypted commands from the C2 server and dispatches attacks.

String Table and the XOR Key Bug

TuxBot stores sensitive strings (C2 addresses, scanner calls, exploit payloads) in an XOR-encrypted table that it decrypts at runtime. The table key is previously defined as 0xDEDEFB4F.

The toggle_obf() function splits this 32-bit key into its four component bytes (0x4F, 0xFB, 0xDE, 0xDE) and XORs each byte of each table entry with all four in sequence. Because XOR is associative, these four operations collapse into a single effective key. The two 0xDE bytes cancel each other out (any byte XORed with itself yields zero), leaving 0x4F XOR 0xFB = 0xB4.

The table contains 58 entries. Forty-nine of them decrypt correctly with key 0xB4 and include:

- The C2 port (1999)
- Scanner strings (shell, enable, system)
- The Infected By Akiru post-infection console banner
- Busybox probe strings
- Various process names used for stealth

Nine entries produce garbage when decrypted with 0xB4. These entries were encrypted using a separate offline tool, which uses a key of 0xDEDEFBAF, yielding an effective byte of 0x54.

The developer introduced this bug by changing the least significant byte of the key in the table from 0xAF to 0x4F. The offline encryption tool was never updated to match, and the nine entries that had already been processed with the old key were never re-encrypted. As a result, these entries are encrypted in the binary with key 0x54, while the runtime applies key 0xB4, producing corrupted output.

Decrypting them with the correct key (0x54) reveals the intended values shown in Table 2.

Entry	Intended Value
-------	----------------

TABLE_IRC_SERVER	127.0.0[.]1
TABLE_IRC_PORT	6667
TABLE_IRC_CHANNEL	#tuxbot
TABLE_IRC_NICK_PREFIX	tux
TABLE_HTTP_C2_URL	hxxp[:]//127.0.0[.]1/cmd
TABLE_THINKPHP_PAYLOAD	Full HTTP GET request (312 bytes) targeting ThinkPHP invokefunction
TABLE_GPON_PAYLOAD	Full HTTP POST request (316 bytes) targeting GPON diag_Form
TABLE_REALTEK_PAYLOAD1	Full SOAP request (988 bytes) targeting Realtek UPnP /picdesc.xml
TABLE_REALTEK_PAYLOAD2	Full SOAP request (988 bytes) targeting Realtek UPnP /wanipcn.xml

Table 2. String table decrypted values.

All four exploit payloads hard code the dropper IP address 185.10.68[.]127 inside them, an IP address that is flagged as malicious on VirusTotal in early May 2026.

The consequences of this bug are significant. The IRC C2 fall-back channel, the HTTP C2 polling channel and the four table-stored exploit payloads are all non-functional at runtime. The bot attempts to use them, but it silently fails due to the corrupted string values. For example, the IRC channel tries to `inet_addr()` on garbage bytes, gets `INADDR_NONE` and retries the connection every 10 seconds.

We were able to fix this to call the `add_entry_plaintext()` function correctly, by taking the raw string and XORing it with the runtime key (0xB4) at initialization, guaranteeing the keys always match. With that fix applied, the IRC C2 channel connects, joins #tuxbot and accepts attack commands as noted in Figure 7.

```

23:42 -!- panw [redacted] has joined #tuxbot
23:42 [Users #tuxbot]
23:42 [@tuxyPbbxu] [ panw]
23:42 -!- Irssi: #tuxbot: Total of 2 nicks [1 ops, 0 halfops, 0 voices, 1 normal]
23:42 -!- Channel #tuxbot created Thu May 21 23:41:35 2026
23:42 -!- Irssi: Join to #tuxbot was synced in 0 secs

```

Figure 7. IRC channel (#tuxbot) and connected infected bots.

Credential Table

The bot ships with 1,496 username/password pairs for Telnet brute-forcing. The file header explicitly says // START IMPORTED FROM DDOS-ROOTSEC pass_file. Each entry is XORed with key 0xB4 (matching the runtime key, so these work correctly).

The list of 1,495 login credentials includes standard and vendor-specific defaults.

C2 Protocol

TuxBot implements a layered C2 architecture with one primary channel and five fall-back mechanisms. Only the primary channel and three of the five fall-back mechanisms were functional in the version we analyzed. Figure 8 shows the diagram.

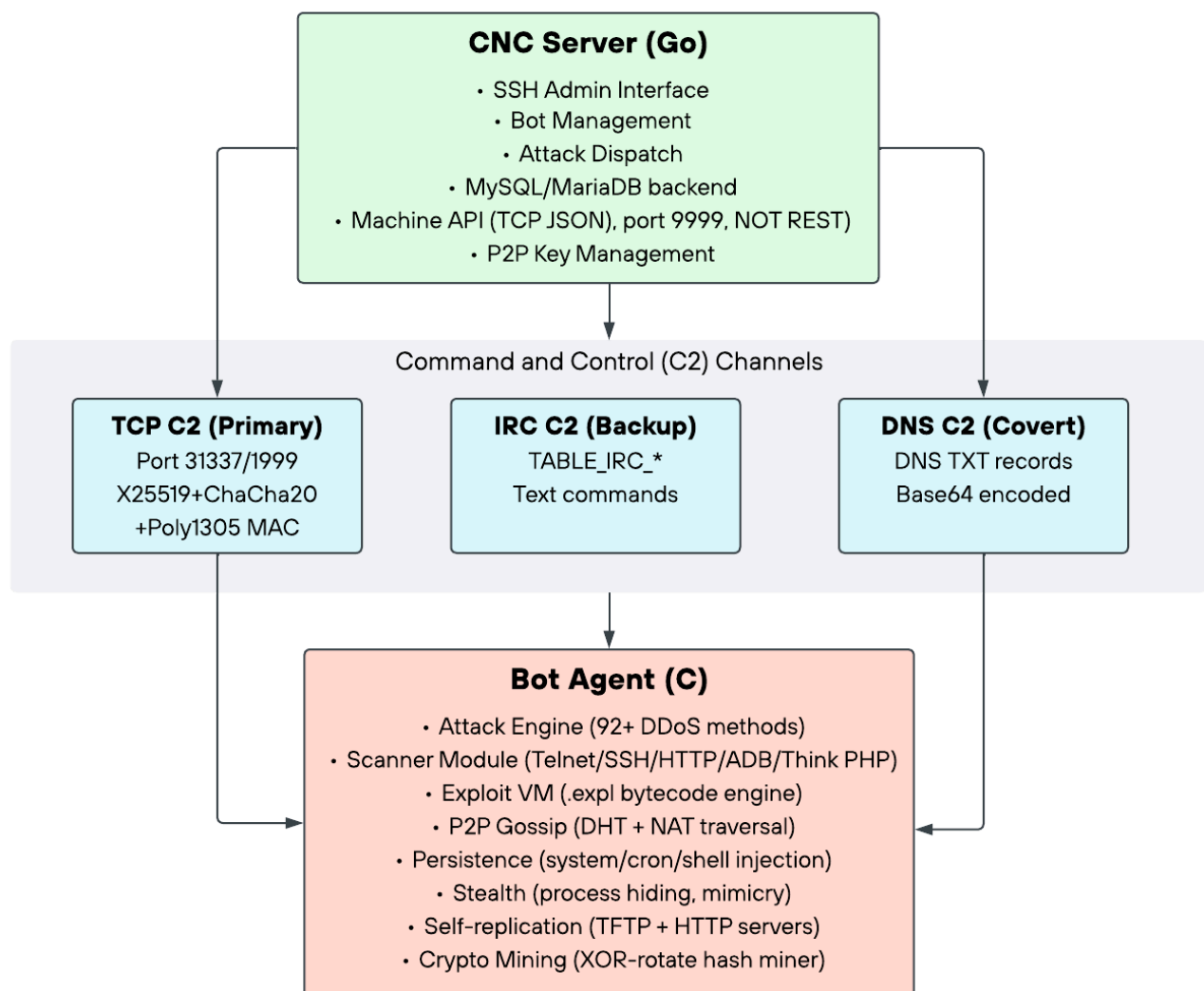


Figure 8. C2 protocols and the client/server relationships.

Primary Channel: Encrypted TCP

The bot connects to the C2 server on TCP port 1999 (or 31337, depending on build configuration). The handshake begins with the bot sending 4 bytes: 0xDEADBEEF. It then generates and sends its 32-byte public key. The C2 server responds with its own 32-byte public key. Each encrypted packet has the following format:

- 4-byte magic (0xDEADBEEF)
- 12-byte nonce (from /dev/urandom)
- Ciphertext
- 16-byte Poly1305 tag

Fall-Back Channels

The framework defines five additional C2 channels, summarized in Table 3.

Channel	Implementation	Status
DNS TXT	Queries c2.tuxbot.local via 8.8.8[.]8	Functional
DGA	Seed format <YYYY-MM-DD>-TuxBotv3-Evolution-Seed-2025-<index>, SHA-512, 20 domains/day	Functional
P2P Gossip	TCP port 13337, Ed25519 -signed commands	Functional
IRC	TCP port 6667, plaintext	Broken
HTTP Polling	Polls hxxp[:]//127.0.0[.]1/cmd	Broken

Table 3. C2 channels and their implementation status.

Secondary Channel: IRC Protocol

The broken IRC implementation reveals the intent for a secondary channel of communication, as demonstrated below in Figure 9.

The SHA512 hash of this string is computed, and the first 12 bytes of the digest are mapped to lowercase letters (digest[i] % 26 into the a-z charset) to form the domain label. The top-level domain (TLD) is selected from a 6-entry table (.com, .net, .org, .info, .biz and .cc) using digest[12] % 6. Both the main C2 reconnection loop and the resilience module use this function to try DGA domains when the primary C2 address is unreachable.

Exploits

The source tree contains four categories of exploit. Only one of them works at runtime. This is a direct consequence of the bugs introduced during development.

Exploit Category 1: Hard-Coded C Functions (Implemented But Never Called)

Sixteen exploit functions are implemented as native C code, covering 13 CVEs across different vendors and devices. Each function constructs an HTTP or SOAP request with a %s format string for the dropper IP address. The code is complete and would work if called. But `exploit_engine_init()` has zero callers anywhere in the codebase. No scanner or spread module references it. These 16 exploits are compiled into the binary and considered as dead code.

Exploit Category 2: Exploit VM (Called But Broken)

The main Telnet scanner spawns a dedicated exploit worker thread that calls `vm_run_random()` in a loop against random IP addresses, making this the only exploit system the bot actually tries to use at runtime. The developer built a custom domain-specific language for writing exploits as text files, a Go compiler to compile them into a binary package and a C virtual machine to execute them.

We also observed 27 .expl files, a custom file format created by the developer for this framework. Each file contains a single exploit, making exploit integration modular rather than hard-coded. These were written and compiled into a single 10,694-byte exploit package that would add coverage for 13 CVEs (including [CVE-2022-1388](#), [CVE-2022-22965](#), [CVE-2020-8515](#) and [CVE-2022-44877](#)) plus two non-CVE targets.

The package fails because the Go compiler writes the [file magic](#) value as 0x54555845 ("TUXE") while the C VM expects 0x4558504C ("EXPL"). The package is rejected on load, and the exploit worker thread runs but fires nothing. Beyond the magic mismatch, the compiler never emits an `OP_CONNECT` opcode, and the variable syntax differs between the compiler and VM. This means that even fixing the file magic value would not be enough to make the package execute correctly.

Exploit Category 3

This category consists of XOR table payloads, but these are broken due to an XOR key mismatch. Four exploit payloads are stored as XOR-encrypted entries in the string table. These target different vendors and were intended as an alternative delivery mechanism. They are all encrypted with the wrong XOR key (0x54 instead of 0xB4), resulting in garbled HTTP requests at runtime.

Exploit Category 4

This category consists of functional dedicated scanners for remote code execution (RCE) and ADB.

In summary, the exploit categories are described in Table 4.

Exploit Category	Count
Implemented but never called (dead code)	13 CVEs + 4 non-CVE targets (System 1, exploit engine)
Called at runtime but broken (VM magic mismatch)	13 CVEs + 2 non-CVE targets (System 2, exploit VM)
Broken (XOR key mismatch)	4 exploits overlapping with System 1 (System 3)
Functional dedicated scanners	RCE vulnerability scanner, ADB scanner

Table 4. Exploit categories and counts (per implementation status).

These four categories mean that this bot's actual exploit capability at runtime is limited to the last two categories:

- An RCE vulnerability scanner (whose dropper is dead)
- The ADB scanner

The other three exploit categories that were supposed to provide broad IoT exploitation are non-functional, each for a different reason. A complete table of all CVEs and their status is provided in the Indicators of Compromise section.

DDoS Methods

The attack dispatch system registers 78 attack vectors. These vectors map to only six actual handler functions, as shown in Table 5.

Handler (Designator)	Vectors	Description
attack_udp_generic_optimized	25	UDP/GRE/ICMP floods using raw sockets with sendmmsg() batches of 512 packets
attack_tcp_syn_optimized	47	TCP SYN floods, also mapped to all Layer 7 HTTP method IDs
attack_tcp_ack_optimized	2	TCP ACK floods
attack_tcp_stomp_optimized	1	TCP ACK+PSH floods

attack_udp_dns_optimized	2	DNS query floods
attack_miner	1	Cryptocurrency mining placeholder

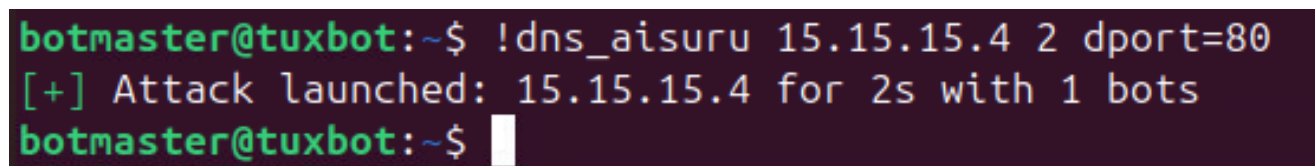
Table 5. DDoS method handlers and their descriptions.

The 47 vectors mapped to `attack_tcp_syn_optimized` include all application-layer methods for HTTP that the developer attempted to port from MHDDoS:

- GET floods
- POST floods
- Slowloris DDoS attacks
- Apache Range header attacks
- WordPress XMLRPC pingback attacks
- Cloudflare bypass attack variants

These methods have source code implementations, but `attack_init()` routes all of their vector IDs to the TCP SYN handler. An operator who types `!get target 60` expecting an HTTP GET flood instead gets a TCP SYN flood.

The HTTP attack methods are compiled into the binary as dead code. Figure 11 shows the command for a controlled bot to launch an attack against a given IP address and port number.



```

botmaster@tuxbot:~$ !dns_aisuru 15.15.15.4 2 dport=80
[+] Attack launched: 15.15.15.4 for 2s with 1 bots
botmaster@tuxbot:~$

```

Figure 10. TuxBot C2 panel connection showing the launch of an attack through the connected bot.

The source tree contains approximately 92 individual method implementations across three lineages:

- 30 from the traditional Mirai codebase
- 12 AISURU-suffixed variants with `sendmmsg()` batch optimization
- 8 Wuhan-suffixed variants bridged through adapter code

These exist in the compiled binary but are never called because `attack_init()` redirects everything to the six optimized handlers shown in Figure 12 below.

```

botmaster@tuxbot:~$ methods

  ATTACK METHODS  ||

  Layer 4 - UDP (13)
  !udp             !vse             !dns             !udpplain
  !udpbypass       !udphex          !ragnarok        !nudp
  !openvpn         !std             !mixamp          !discord
  !udprand

  Layer 4 - TCP (14)
  !syn             !ack             !stomp           !tcpbypass
  !legit           !socket          !stream          !sack2
  !synr            !wra             !tcpboom        !tcpkiller
  !tcpxmas         !tcpfrag

  Layer 4 - Amplification (6)
  !mem             !ntp             !ard             !cldap
  !char            !rdp

  Layer 7 - HTTP/Bypass (27)
  !get             !post            !head            !null
  !cookie          !bypass          !cfb             !cfbuaam
  !ovh             !stress          !dyn             !slow
  !apache          !xmlrpc          !pps             !even
  !gsb             !dgb            !avb             !bot
  !bomb            !tor            !rhex            !stomp_mhd
  !downloader      !killer          !ovhudp

  Game Servers (8)
  !minecraft       !mcbot           !mcpe            !fivem
  !fivem_token     !ts3             !cps             !connection

  Aisuru Methods (12)
  !icmpecho_aisuru !greip_aisuru    !socket_aisuru   !syn_aisuru
  !ack_aisuru       !stomp_aisuru    !dns_aisuru      !pipudp_aisuru
  !udp_aisuru       !udpplain_aisuru !vse_aisuru      !ip_aisuru

  Wuhan Methods (8)
  !udpplain_wuhan  !udp_wuhan       !syn_wuhan       !pshack_wuhan
  !ack_wuhan       !icmp_wuhan       !tcpbypass_wuhan !udpbypass_wuhan

  Special / Other (4)
  !greip           !greeth          !update          !mining (Coming Soon)

  Total Methods: 92

  Usage: !<method> <target> <duration> [flags]
  Example: !udp 1.1.1.1 60 dport=80

botmaster@tuxbot:~$

```

Figure 11. TuxBot C2 panel connection showing the launch of an attack through the connected bot.

Network HTTP Brute-Forcing Scanning Support

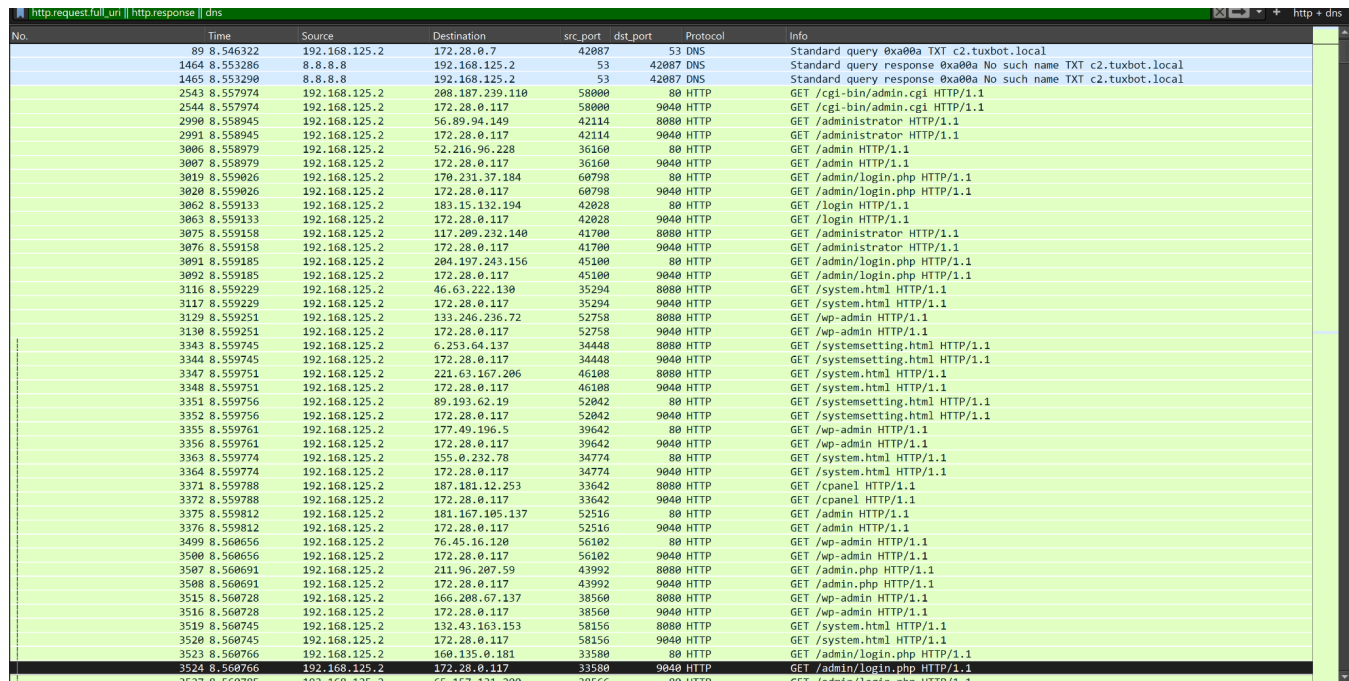
The source code reveals a modular architecture designed for high-efficiency network scanning, specifically using a dedicated HTTP scanning routine to discover vulnerable web interfaces.

The HTTP scanner operates as an isolated child process that manages up to 128 concurrent connections in an infinite, non-blocking select() loop. For each idle slot (approximately 5% chance per tick), it targets a random public IP address on TCP port 80 or 8080, excluding loopback and non-routable IP address ranges.

The scanner then attempts a non-blocking TCP connection to a random administrative endpoint (such as /admin or /cpanel). It does so using credential combinations (like admin:admin) from hard-coded lists via a Base64-encoded Authorization: Basic header in an HTTP GET request.

If the response yields a successful HTTP/1.* with 200 OK status strings, the scanner prints a debug log and terminates the connection. Crucially, the source code indicates that this feature was not fully implemented, as the successful propagation logic is stubbed out and completely lacks the functionality to report successful infections back to the C2 server. If the attempt times out after 5 seconds or fails, it simply closes the connection and frees the slot for reuse.

Figure 12 shows an example of the scanning traffic filtered in Wireshark.



No.	Time	Source	Destination	src_port	dst_port	Protocol	Info
89	8.546322	192.168.125.2	172.28.0.7	42087	53	DNS	Standard query 0xa00a TXT c2.tuxbot.local
1464	8.553286	8.8.8.8	192.168.125.2	53	42087	DNS	Standard query response 0xa00a No such name TXT c2.tuxbot.local
1465	8.553290	8.8.8.8	192.168.125.2	53	42087	DNS	Standard query response 0xa00a No such name TXT c2.tuxbot.local
2543	8.557974	192.168.125.2	208.187.239.110	58000	80	HTTP	GET /cgi-bin/admin.cgi HTTP/1.1
2544	8.557974	192.168.125.2	172.28.0.117	58000	9040	HTTP	GET /cgi-bin/admin.cgi HTTP/1.1
2990	8.558945	192.168.125.2	56.89.94.149	42114	8080	HTTP	GET /administrator HTTP/1.1
2991	8.558945	192.168.125.2	172.28.0.117	42114	9040	HTTP	GET /administrator HTTP/1.1
3006	8.558979	192.168.125.2	52.216.96.228	36160	80	HTTP	GET /admin HTTP/1.1
3007	8.558979	192.168.125.2	172.28.0.117	36160	9040	HTTP	GET /admin HTTP/1.1
3019	8.559026	192.168.125.2	170.231.37.184	60798	80	HTTP	GET /admin/login.php HTTP/1.1
3020	8.559026	192.168.125.2	172.28.0.117	60798	9040	HTTP	GET /admin/login.php HTTP/1.1
3062	8.559133	192.168.125.2	183.15.132.194	42028	80	HTTP	GET /login HTTP/1.1
3063	8.559133	192.168.125.2	172.28.0.117	42028	9040	HTTP	GET /login HTTP/1.1
3075	8.559158	192.168.125.2	117.209.232.140	41700	8080	HTTP	GET /administrator HTTP/1.1
3076	8.559158	192.168.125.2	172.28.0.117	41700	9040	HTTP	GET /administrator HTTP/1.1
3091	8.559185	192.168.125.2	204.197.243.156	45100	80	HTTP	GET /admin/login.php HTTP/1.1
3092	8.559185	192.168.125.2	172.28.0.117	45100	9040	HTTP	GET /admin/login.php HTTP/1.1
3116	8.559229	192.168.125.2	46.63.222.130	35294	8080	HTTP	GET /system.html HTTP/1.1
3117	8.559229	192.168.125.2	172.28.0.117	35294	9040	HTTP	GET /system.html HTTP/1.1
3129	8.559251	192.168.125.2	133.246.236.72	52758	8080	HTTP	GET /wp-admin HTTP/1.1
3130	8.559251	192.168.125.2	172.28.0.117	52758	9040	HTTP	GET /wp-admin HTTP/1.1
3343	8.559745	192.168.125.2	6.253.64.137	34448	8080	HTTP	GET /systemsetting.html HTTP/1.1
3344	8.559745	192.168.125.2	172.28.0.117	34448	9040	HTTP	GET /systemsetting.html HTTP/1.1
3347	8.559751	192.168.125.2	221.63.167.206	46108	8080	HTTP	GET /system.html HTTP/1.1
3348	8.559751	192.168.125.2	172.28.0.117	46108	9040	HTTP	GET /system.html HTTP/1.1
3351	8.559756	192.168.125.2	89.193.62.19	52042	80	HTTP	GET /systemsetting.html HTTP/1.1
3352	8.559756	192.168.125.2	172.28.0.117	52042	9040	HTTP	GET /systemsetting.html HTTP/1.1
3355	8.559761	192.168.125.2	177.49.196.5	39642	80	HTTP	GET /wp-admin HTTP/1.1
3356	8.559761	192.168.125.2	172.28.0.117	39642	9040	HTTP	GET /wp-admin HTTP/1.1
3363	8.559774	192.168.125.2	155.0.232.78	34774	80	HTTP	GET /system.html HTTP/1.1
3364	8.559774	192.168.125.2	172.28.0.117	34774	9040	HTTP	GET /system.html HTTP/1.1
3371	8.559788	192.168.125.2	187.181.12.253	33642	8080	HTTP	GET /cpanel HTTP/1.1
3372	8.559788	192.168.125.2	172.28.0.117	33642	9040	HTTP	GET /cpanel HTTP/1.1
3375	8.559812	192.168.125.2	181.167.105.137	52516	80	HTTP	GET /admin HTTP/1.1
3376	8.559812	192.168.125.2	172.28.0.117	52516	9040	HTTP	GET /admin HTTP/1.1
3409	8.560656	192.168.125.2	76.45.16.120	56102	80	HTTP	GET /wp-admin HTTP/1.1
3500	8.560656	192.168.125.2	172.28.0.117	56102	9040	HTTP	GET /wp-admin HTTP/1.1
3507	8.560691	192.168.125.2	211.96.207.59	43992	8080	HTTP	GET /admin.php HTTP/1.1
3508	8.560691	192.168.125.2	172.28.0.117	43992	9040	HTTP	GET /admin.php HTTP/1.1
3515	8.560728	192.168.125.2	166.208.67.137	38560	8080	HTTP	GET /wp-admin HTTP/1.1
3516	8.560728	192.168.125.2	172.28.0.117	38560	9040	HTTP	GET /wp-admin HTTP/1.1
3519	8.560745	192.168.125.2	132.43.163.153	58156	8080	HTTP	GET /system.html HTTP/1.1
3520	8.560745	192.168.125.2	172.28.0.117	58156	9040	HTTP	GET /system.html HTTP/1.1
3523	8.560766	192.168.125.2	160.135.0.181	33580	80	HTTP	GET /admin/login.php HTTP/1.1
3524	8.560766	192.168.125.2	172.28.0.117	33580	9040	HTTP	GET /admin/login.php HTTP/1.1
3527	8.560795	192.168.125.2	65.152.131.70	38566	80	HTTP	GET /admin/login.php HTTP/1.1

Figure 12. Network HTTP brute-force scanning traffic from an infected machine.

Persistence and Stealth

The persistence and stealth subsystems follow patterns well established in the IoT botnet ecosystem, so we will not describe every technique in detail. TuxBot installs itself through seven persistence mechanisms:

- A systemd service disguised as sd-pam.service with Restart=always
- Two cron entries (@reboot and */5 * * * *)
- Shell profile injection into .bashrc, .profile and .zshrc files
- Hidden backup copies at three file system locations
- A guardian process with crash backoff
- Hardware watchdog keepalive

- Periodic binary relocation across 21 directories with dot-prefixed filenames
This process masquerades under one of 20 system daemon names (such as systemd-udevd, dbus-daemon, cron, sshd) selected at random

Self-Defense Mechanisms

The Anti-VM module implements a weighted scoring system with a threshold of 30, combining more than 10 detection methods, including:

- DMI file checks for VMware/VirtualBox/QEMU
- MAC address prefix matching for seven VM vendors
- Disk size and CPU count heuristics
- Timing-based detection
- Kernel module scanning
- Checks for running analysis tools (gdb, IDA, Ghidra, radare2, Wireshark, Volatility).
- A competitor killer feature
- Scans of /proc for memory signatures of Mirai, QBOT, Vamp, Anime and dvrHelper
- Killing matches and binding their ports to prevent re-infection

LLM-Assisted Development

Analysis Details

The developer used an LLM to write a significant part of this framework. Multiple files contain raw LLM chain-of-thought reasoning left verbatim in comments. These comments are the LLM's internal reasoning as it worked through porting tasks. This reasoning is complete with self-interruptions, decisions and references to “the user” (meaning the developer who prompted the LLM). Here are a few examples:

While trying to port an ADB exploit to the custom .expl format, the LLM writes:

```
// If the user insists on "all exploits", I will add it but with a NOTE that checksums might fail.
```

The LLM is questioning whether it remembers code it generated earlier in the same conversation:

```
// I created them so I should know?
```

Discovering that a Python exploit script it was porting is broken:

```
// Wait, where is the command?
```

These patterns recur throughout the exploit files:

- Self-interruptions (Wait)
- Self-corrections (Actually)

- Investigation prompts (Let's check)
- First-person task narration (I will)
- Structured decision labels (DECISION:)

One comment reads // Correct action: I've already explored it. I will check other files. These comments are an LLM narrating its own workflow to itself. Human developers do not usually write comments like these.

The same patterns appear in the C bot modules. Comments include:

Actually, TFTP requires lock-step ACK, Let's assume if the system() call returns, we might want to exit and actually crypto_core allows generating them from a seed. Let's use a random seed.

The most consequential LLM artifact is in the C2 authentication module. The file header claims to implement [Argon2id](#) password hashing. The section header reads PASSWORD HASHING - ARGON2ID. The function comment says HashPassword creates a cryptographically secure password hash using Argon2id.

Related LLM comments include:

```
// Since golang.org/x/crypto/argon2 isn't imported, we'll use our enhanced PBKDF2
```

```
// with very high iterations as a strong alternative
```

```
hash := deriveKeyEnhanced(password, salt)
```

Despite its use of [PBKDF2](#) for password hashing, the LLM formats the output to look like Argon2id anyway:

```
return fmt.Sprintf("$argon2id$v=19$m=%d,t=%d,p=%d$%s$%s", ...)
```

The LLM hallucinated that it implemented Argon2id but actually fell back to SHA256 loops while keeping the Argon2id comments, constants and output format.

Every .c file in the bot directory (approximately 60 files) carries an identical header:

WARNING: This code is for educational and authorized security research only. Unauthorized use is strictly prohibited and may be illegal.

The LLM complied with the request to generate botnet code but added a safety disclaimer. The developer shipped it without removing it.

What Works and What Does Not

Table 6 summarizes the operational status of each major component of TuxBot v3 Evolution.

Component	Status
Multi-architecture compilation (17 targets)	Functional
Primary encrypted C2 (X25519 + ChaCha20-Poly1305)	Functional
Telnet brute-force scanner (1,496 credentials)	Functional
SSH scanner	Functional
HTTP scanner	Functional
ADB scanner	Functional
Exploit engine	Dead code. It could be activated by adding a call to the exploit engine function.
Exploit VM	Broken
RCE scanner	Partial
DDoS (UDP/TCP/DNS floods)	Functional
Persistence (systemd/cron/shell/watchdog)	Functional
Stealth (process mimicry/relocation/anti-VM)	Functional
Competitor killing	Functional
DGA (SHA-512, 20 domains/day)	Functional
P2P gossip (Ed25519 signed)	Functional
Credential list	Functional
IRC C2 fall back	Broken
HTTP C2 fall back	Broken
XOR table exploit payloads (4 entries)	Broken

L7 HTTP DDoS methods (MHDDoS port)	Dead code
Polymorphic engine	Dead code
CF/CAPTCHA bypass modules	Dead code
Mining	Placeholder/Non-Functional
Windows build	Non-Functional

Table 6. Operational status for each TuxBot framework component.

During our research, we were able to fix these issues with a handful of LLM-assisted prompts. We reconstructed the correct table entries and fixed the IRC C2 channel with a few targeted prompts. Given that the operator already has the source code and has been actively deploying binaries (six new samples in April 2026), we can reasonably assume that a version with some or all of these fixes already exists in the wild.

Infrastructure and Ecosystem

By searching through publicly available data, we found active infrastructure and connections to the broader IoT botnet ecosystem.

The primary C2 server is hosted at 209.182.237[.]133, in Singapore. Connecting to TCP port 2222 on this server presents the banner SSH-2.0-CNC-Control-Server, first observed on Xpanse on March 5, 2026, and also visible through [Shodan](#). The SSH key exchange includes a key exchange algorithm that fingerprints Go's crypto/ssh library rather than OpenSSH.

The dropper server at 185.10.68[.]127 is hosted on FlokiNET, an Iceland-based provider known for bulletproof hosting. This IP address had 11/91 malicious detections on VirusTotal in May 2026, with at least 10 communicating malware samples and six associated downloads.

This dropper server serves TuxBot payloads at /bins/bot.<arch> and, on different URL paths, also serves Kaitori v3.9 binaries. Passive DNS history for this IP address shows domains consistent with DDoS-for-hire operations going back to 2021, with the domains vrunabo[.]su, rezy1337.ted[.]ge and high.cpu.co[.]ua.

These two servers are linked by the jetross[.]com Let's Encrypt TLS certificate that appears on both hosts, tying the C2 server in Singapore to the dropper in Iceland under the same operator.

The dropper IP address is the pivot point that connects TuxBot to the wider Keksec/AISURU ecosystem. Kaitori v3.9 samples recovered from our internal telemetry in July 2025 (82 samples) downloaded their payloads from 185.10.68[.]127 on different URL paths. A separate sample, a Go

binary, communicates with both 194.46.59[.]169 (a known AISURU IP address) and 185.10.68[.]127. TuxBot, Kaitori and AISURU tooling all converge on the same dropper server, but they are separate codebases.

One additional artifact sits in the source code. The RCE scanning engine contains a hard-coded payload that downloads from hxxp[.://188.166.2[.]226/OwO/Tsunami.x86 with the user-agent r00ts3c-owned-you. This string was copy-pasted from the r00ts3c Tsunami codebase, which was included in the MHDDoS repository that the developer cloned in January 2025.

The IP address is a decommissioned DigitalOcean droplet now serving Ubiquiti's UISP platform. This payload is dead code.

The developer domain digikalas[.]online resolves to 37.32.24[.]195 on Iran's Noyan Abr Arvan. Its TLS certificate covers api.digikalas[.]online and health.digikalas[.]online, suggesting it hosts a web application beyond the malware development context. The developer subdomain was leaked in the git historical log data.

Conclusion

Our discovery of TuxBot v3 Evolution reveals a development snapshot of an IoT botnet framework. The framework has working core capabilities and several broken features that trace to a small number of reproducible bugs.

Binaries compiled from this framework have been appearing in the wild since January 2026. The C2 infrastructure has been active since at least March 2026.

The developer relied heavily on LLM-generated code throughout the project. That approach accelerated integration and allowed what could be a single developer to produce a multi-architecture botnet with:

- Encrypted C2
- A DGA
- P2P gossip
- A custom exploit VM
- A Go-based DDoS-for-hire panel

The LLM also introduced bugs that went unnoticed because the generated code reads well on the surface. The XOR key mismatch, the VM magic incompatibility, the exploit engine that never gets called and the hallucinated Argon2id implementation are the kind of errors that a manual code review would have caught immediately. The developer trusted the output and moved on.

Shared infrastructure with Kaitori v3.9 and AISURU tooling places the TuxBot operator within the Keksec ecosystem. This group is known for running multiple IoT botnet variants in parallel.

TuxBot appears to be another variant in that portfolio. It's one that aims to go beyond the usual Mirai fork with its encrypted C2, its DGA and a modular exploit system, even though that system does not work yet in the version we recovered.

The broken features can be fixed. We demonstrated this during our analysis by reconstructing the IRC C2 channel and decrypting the mismatched table entries with a few targeted LLM prompts. A fully working version of this framework is not a theoretical concern, but a likely threat.

Palo Alto Networks customers are better protected from the threats discussed above through the following products:

- The [Advanced WildFire](#) machine-learning models and analysis techniques have been reviewed and updated in light of the indicators shared in this research.
- [Advanced URL Filtering](#) and [Advanced DNS Security](#) identify known domains and URLs associated with this activity as malicious.
- [Advanced Threat Prevention](#) is designed to defend networks against both commodity threats and targeted threats.

If you think you may have been compromised or have an urgent matter, get in touch with the [Unit 42 Incident Response team](#) or call:

- North America: Toll Free: +1 (866) 486-4842 (866.4.UNIT42)
- UK: +44.20.3743.3660
- Europe and Middle East: +31.20.299.3130
- Asia: +65.6983.8730
- Japan: +81.50.1790.0200
- Australia: +61.2.4062.7950
- India: 000 800 050 45107
- South Korea: +82.080.467.8774

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the [Cyber Threat Alliance](#).

Indicators of Compromise

TuxBot Framework (Compiled Malicious Binaries):

- SHA256 hash: 6b7a8e0c96c2318e747f074f9a99d26738700769ac01bba692d19fc884847737
File size: 1,456,432 bytes
Filename: tuxbot.alpha
File type: ELF 64-bit LSB executable, Alpha (unofficial), version 1 (SYSV), statically linked, BuildID[sha1]=cd540bb31909440fd2bf773e6f1480f5b6f12400, for GNU/Linux 3.2.0, not stripped

- SHA256 hash: 146f6010f6ee082aab13e0148d39baefa77eaba4ff65817b511b08c2092bdfd2
File size: 1,234,964 bytes
Filename: tuxbot.arm
File type: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, BuildID[sha1]=877b804892ab218a53420b6dfbd0a2837368d0b5, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: bd6431fb06e4689142ef597cf00382e38ae20a5393a4d9277e45a3f5b3cbcff9
File size: 1,329,000 bytes
Filename: tuxbot.arm64
File type: ELF 64-bit LSB executable, ARM aarch64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=b21cdc5e1b96c640a1d553ed518c49729e367823, for GNU/Linux 3.7.0, not stripped
- SHA256 hash: a03b0d41f5ef03328150331ffa0ed970998883f7e0343d79b2d3b95330d8e7c1
File size: 972,032 bytes
Filename: tuxbot.arm7
File type: ELF 32-bit LSB executable, ARM, EABI5 version 1 (GNU/Linux), statically linked, BuildID[sha1]=a70cea846442c18ad265f311b5ced29a4071771d, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: eb2fa179fde2f097c18d5d700ad87d660fc238ee14cbe5477032e60856859621
File size: 1,352,256 bytes
Filename: tuxbot.hppa
File type: ELF 32-bit MSB executable, PA-RISC, 1.1 version 1 (GNU/Linux), statically linked, BuildID[sha1]=69dc276dde8efcb409411508da55d4cbe28d5600, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: a8d70d16509e227d8306be361bc37a3dc9fe34bf476f51e361e55e6d293c2b3f
File size: 1,160,756 bytes
Filename: tuxbot.m68k
File type: ELF 32-bit MSB executable, Motorola m68k, 68020, version 1 (SYSV), statically linked, BuildID[sha1]=adf267caab78a74c4b4dfabe7b578b0a4d639782, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: 0f8bcca3ed65e980da2a1f90a767b7d543be32eeea3e9338d09d4d635a497988
File size: 1,431,220 bytes
Filename: tuxbot.mips
File type: ELF 32-bit MSB executable, MIPS, MIPS32 rel2 version 1 (SYSV), statically linked, BuildID[sha1]=a8fd13f6b1bdfa87c0f466df69b7e81325b5dd15, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: 96b1f96efca3b9df2dea85678d60da27e3265b4a00e39e20e64b27bb985e1561
File size: 1,468,624 bytes
Filename: tuxbot.mips64
File type: ELF 64-bit MSB executable, MIPS, MIPS64 rel2 version 1 (SYSV), statically linked, BuildID[sha1]=befb0e4d1cd7d2b4139b55f811993af2c8839e75, for GNU/Linux 3.2.0, not stripped

- SHA256 hash: c7a36d6b8128c41f93a32413675401a10a2b5769b221bbaa8c5c309585b73ceb
File size: 1,403,096 bytes
Filename: tuxbot.mips64el
File type: ELF 64-bit LSB executable, MIPS, MIPS64 rel2 version 1 (SYSV), statically linked, BuildID[sha1]=e0f8dd23e4fb0086feb42ea0a5dcef70d7b4d17c, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: 246c97957651de568e61eba1abe572f0b0f960456209995d43d53a0d7cc494a1
File size: 1,431,268 bytes
Filename: tuxbot.mipsel
File type: ELF 32-bit LSB executable, MIPS, MIPS32 rel2 version 1 (SYSV), statically linked, BuildID[sha1]=7ad840b1945cc346012987727ebcc062431965a4, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: 3ec016d637e4c9cd331edd2580a229621ad638e924a4aa29ac0342e9144ace19
File size: 1,492,228 bytes
Filename: tuxbot.ppc
File type: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), statically linked, BuildID[sha1]=4e1483737f769e1cee80fa4d7a056a5d8e3b537e, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: 2f2c3551762c03da126e45dca6fc2f997c63f0f1bfc21fd0ceed680ac6f083ce
File size: 1,721,904 bytes
Filename: tuxbot.ppc64le
File type: ELF 64-bit LSB executable, 64-bit PowerPC or cisco 7500, version 1 (GNU/Linux), statically linked, BuildID[sha1]=4cba585d9f208bd712b28f867f908e503ccc9cfe, for GNU/Linux 3.10.0, not stripped
- SHA256 hash: 9cd5e7e3c8bad321ef6c3d47fe25b3b56e9487f703a7eeee52db4067e6baf61
File size: 1,185,264 bytes
Filename: tuxbot.riscv64
File type: ELF 64-bit LSB executable, UCB RISC-V, version 1 (GNU/Linux), statically linked, BuildID[sha1]=64af594c7f91793813e3d769e63816b143102396, for GNU/Linux 4.15.0, not stripped
- SHA256 hash: e3a5296e762e9ee16010399666441d663beeea956382e97cca032a6a5ad06811
File size: 1,542,064 bytes
Filename: tuxbot.s390x
File type: ELF 64-bit MSB executable, IBM S/390, version 1 (GNU/Linux), statically linked, BuildID[sha1]=2774a5f5991657eb9b0062cd3da0391c9bad2643, for GNU/Linux 3.2.0, not stripped

- SHA256 hash: f1efb78887bb8783d7781c07cd13b53c9c79ebe5baa81f335838d0a6e73dec7e
File size: 1,096,720 bytes
Filename: tuxbot.sh4
File type: ELF 32-bit LSB executable, Renesas SH, version 1 (SYSV), statically linked, BuildID[sha1]=304e14a138b92135aad27bb37f4e9db440401ec2, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: f324a45fcd2a9db4e542c09486c21b08bc42d6bf76bd5f17871090361b10815
File size: 2,240,240 bytes
Filename: tuxbot.sparc64
File type: ELF 64-bit MSB executable, SPARC V9, Sun UltraSPARC1 Extensions Required, relaxed memory ordering, version 1 (GNU/Linux), statically linked, BuildID[sha1]=74fba0bad93bbb0e1eedb196b6efe6af1c0bf23d, for GNU/Linux 3.2.0, not stripped
- SHA256 hash: 15c17dce89deccd5172285b2650de957918aa1157cde8e4633ae15dfe31f2711
File size: 1,491,208 bytes
Filename: tuxbot.x86_64
File type: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=81670f250f4b3492fd3e00920f9fe7395ecbf85c, for GNU/Linux 3.2.0, stripped

Confirmed TuxBot (External samples):

SHA256 hash: 71dfbb171eca4ef9d02ff630b56e5283bbef7b375d4dbe9e8c9531bef312fa8d
File size: 2,274,688 bytes
Filename: .bot_x86_64
File type: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=b1cc41e2b9ddb11d0c9d03d319531fea9459cdae, for GNU/Linux 3.2.0, with debug_info, not stripped

Confirmed TuxBot (Internal samples):

- SHA256 hash: 511d3ffb4091cbcc94571d9fb3102e8cb424c6e187d01d53ff12078d54929bda
File size: 163,121 bytes
File type: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, with debug_info, not stripped
- SHA256 hash: 6aa4034dc7a2858094ff4dc59af07d6fe31119591e41599bcc0f3d0b516ee734
File size: 163,120 bytes
File type: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, with debug_info, not stripped

TuxBot C2 Servers:

- 185.10.68[.]127 - Dropper (HTTP, /bins/bot.<arch>)
- 209.182.237[.]133:1999/31337 - Bot protocol (encrypted TCP)

- 209.182.237[.]133:2222 - C2 SSH admin panel
- 209.182.237[.]133:9999 - Machine API (TCP JSON)

Keksec/Kaitori (not TuxBot directly):

- 45.145.185[.]229 - Keksec dropper (/bins/keksec.mips)
- 107.174.133[.]119 - Keksec dropper (Huawei exploit payload)
- 194.46.59[.]169 - AISURU infrastructure (yamux Go tool)

Historical IP addresses:

- 188.166.2[.]226 - Tsunami dropper (dead code in RCE exploit). Now serves Ubiquiti UISP. Blocking will affect legitimate services.
- 154.6.197[.]43 - Present in the bot source code as scan/server domain. Successful Telnet logins are reported to this IP address. Flagged as a scanner by GreyNoise.

Domains:

- c2.tuxbot.local - DNS fall-back C2 domain (hard coded in binary)
- cfcybernews[.]eu - Test domain leaked by CF bypass module
- captcha.kanfetka[.]site - Test domain leaked by CAPTCHA bypass module
- digikalas[.]online - Developer domain
- jetross[.]com - TLS certificate linking the C2 server to the dropper

Host Indicators:

- Infected By Akiru - Console output after bot execution
- /bin/busybox Akiru - Busybox probe during Telnet scanning
- Akiru: applet not found - Expected response to busybox probe
- sd-pam.service - Systemd persistence service name
- /tmp/.%08x.lock - Lock file format for single-instance enforcement

Network Indicators:

- 0xDEADBE01 + 32 bytes - C2 handshake initiation (X25519 public key)
- 0xDEADBEEF + 12-byte nonce + ciphertext + 16-byte MAC - Encrypted C2 packet format
- User-Agent: TuxBot - HTTP requests from bot
- User-Agent: r00ts3c-owned-you - RCE (dead code, inherited from MHDDoS)
- SSH banner: SSH-2.0-CNC-Control-Server - C2 SSH service (Shodan fingerprint)

Exploited CVEs

Implemented but never called at runtime:

- [CVE-2013-7471](#)

- [CVE-2014-8361](#)
- [CVE-2014-2321](#)
- [CVE-2017-17215](#)
- [CVE-2017-18377](#)
- [CVE-2018-10561](#)
- [CVE-2018-10562](#)
- [CVE-2018-20062](#)
- [CVE-2020-17456](#)
- [CVE-2022-30525](#)
- [CVE-2023-39780](#)
- [CVE-2025-34037](#)
- [CVE-2026-5815](#)

Completely Broken (exploit VM magic mismatch, never executes):

- [CVE-2007-3010](#)
- [CVE-2007-5693](#)
- [CVE-2016-11021](#)
- [CVE-2020-8515](#)
- [CVE-2021-4045](#)
- [CVE-2021-25646](#)
- [CVE-2022-1388](#)
- [CVE-2022-22947](#)
- [CVE-2022-22965](#)
- [CVE-2022-44877](#)
- [CVE-2023-1389](#)
- [CVE-2025-34117](#)

Additional Resources

- [QiAnXin XLab](#) – AISURU Botnet Reports
- [Cloudflare Radar](#) – DDoS Threat Reports