SuperProxy

The Unhealthy Marriage of SuperBox and Residential Proxies

Part 1

Plume Security Labs

Introduction

Media player devices were already a "thing" two decades ago when Netflix was still a DVD rental service. These small black boxes were targeting a market favoring file based playback – usually from internally or externally attached storage. Less than a handful of vendors were trying to conquer this market. The devices were purpose built with custom operating systems and limited application store capabilities.

The rise of Android as a modern mobile and media platform had a significant impact on this market segment. Nowadays almost all of the pioneers are already gone, except for Dune. One key reason behind the change is a much more crowded market with dozens of Android-based devices ranging from very inexpensive to expensive. The other reason is even more obvious: the skyrocketing popularity of video streaming services.

In this research paper we detail the investigation of a popular streaming device and its unhealthy ties to malicious activities in the form of residential proxies.

Superboxes

The SuperBox brand isn't new. They have been selling devices for years and their popularity is on the rise. Currently, SuperBox is advertising and offering its 6th and 7th generation of streaming devices for sale. Based solely on appearances, these boxes appear identical to similar devices on the market. The price of the current line ranges from $300 to $500 per box – a price that cannot be considered inexpensive in today's market given that similar products are sold for as low as $50. The differentiating factor lies elsewhere. It is openly stated that if you invest in buying a SuperBox, you will get access to thousands of TV channels and the latest movies – without the need to pay for any subscription services.

Figure 1 - The latest SuperBox devices (Source: https://mysuperboxtv.com/)

Residential Proxies

Before diving deeper into our research, we need to make a quick detour and briefly talk about residential proxies. Residential proxies are network connections routed through common devices – such as desktop computers, laptops, mobile phones and even IoT equipment – which, in addition to their primary use, serve a secondary purpose enabled by a software package running on them. That 'mostly hideous' piece of code is responsible for organizing the devices into a portion of a different network, where it acts as proxies for internet-facing queries. There are dozens of companies who operate these networks and provide access to the nodes to whomever is willing to pay. Occasionally, individuals deploy these software components themselves, in exchange for *de minimis* payments or other benefits in return. However in the majority of cases, they have no idea that their home network is being rented out – nor are they aware of all of the associated risks.

Network Overload

Plume's NOC (Network Operation Center) recently flagged certain IoT devices generating an unhealthy amount of outbound traffic, occasionally destabilizing residential networks in certain areas. Verification through Plume telemetry quickly revealed that the majority of the identified devices belong to the SuperBox family, they are from the S4, S5 and S6 series. It would be completely normal if a video streaming device contributed to a high amount of inbound traffic, but experiencing the opposite raises concerns. Initially we could not be sure whether the activity belonged to some faulty, unwanted operation of SuperBox devices or if there was malicious intent. After further investigation of the network telemetry of certain of the top bandwidth-consuming devices, we were quickly able to determine that the activity stemmed from malicious intent. Our investigation uncovered tens of thousands of connection requests to thousands of different destinations on any given day. As this extraordinary amount of connection requests was highly suspicious – and we have close to 10,000 SuperBox devices within Plume's user base – combined with the fact that it could also contribute to serious issues on an ISP level – we secured a SuperBox S6MAX to continue with a deep dive.

Infection Vectors

There are different ways for residential proxies and other unwanted applications to be installed on Android devices. The SuperBox device in question had two core properties, which if combined, can make deployment of software rather trivial.

First, the manufacturer enabled ADB (Android Debug Bridge) over TCP on port 5858. The daemon was listening, accepting connections from anywhere on the local network, instead of enforcing the pairing flow that modern Android OS uses to require on-screen approval before a new ADB client is allowed to connect. Just about any device on the same network could connect via ADB and receive a working shell immediately. Second, the 'su' binary was present and configured to grant root level access without authentication.

```
blueline:/ $ su
blueline:/ # whoami
root
```

Figure 2 - Example of root level access

With the combination of these properties, no exploitation is required. Attaining root level access means the attacker successfully achieved full device compromise and will be capable of performing any action.

This is the foundation on which every further infection path rests throughout our research.

Local ADB Shell

Once ADB is reachable and root is acquired, installing additional software can be done via a single command utilizing the package manager:

```
pm install /sdcard/Download/payload.apk
```

In the above example, the package manager will accept and install APKs (Android Package) from any location once invoked by a root shell. Having root-level access will render useless all of the default Android installation protection features such as signature verification, prohibited use of unknown sources, permission review dialog and Play Protect scanning. An attacker can push an APK to the device and install it without the user noticing. Scanning for exposed ADB ports is a well-known technique exploited by cybercriminals for years. In 2018 threat actors were already deploying cryptominers and recent campaigns like the Kimwolf botnet showcase it as well.

Pre-Installation at the Factory

One of the simplest methods to deploy malware on a streaming device is pre-installation by the manufacturer. When they initially build the factory image, it is up to the vendor to decide what applications come pre-installed on the device. This method is both convenient and less suspicious from the user's perspective; the apps are included when the user unboxes the device, and there is no further installation event, no consent required, no notifications displayed. Making things worse, pre-installed applications can even survive a factory reset in case further suspicious activities – such as the device becoming slow or unresponsive – would convince the user to do so. On this class of devices, the pre-installed set typically includes a custom launcher, a custom application store, and various streaming and other "value-added" applications.

Application Vulnerabilities

Some applications might unintentionally open up the local network to whomever is willing and capable of listening. For example, an application might implement remote control features, letting the user control a streaming box from a mobile device on the same network. Such listeners are often implemented without authentication in mind, or with authentication that can be easily bypassed. These applications effectively create their

own version of the open-ADB problem and implement a proxy-tunnel-to-localhost mechanism similar to that seen in previous Kimwolf campaigns.

Custom Application Stores

When someone is powering up a SuperBox for the first time, there might be an unexpected surprise as the pre-installed applications do not reflect the functionality that the brand advertises. No one will find thousands of TV channels or the latest movies or series; instead a user will be presented with very few basic applications, such as Music, Gallery or a Chrome browser, all of which are common on any Android device.

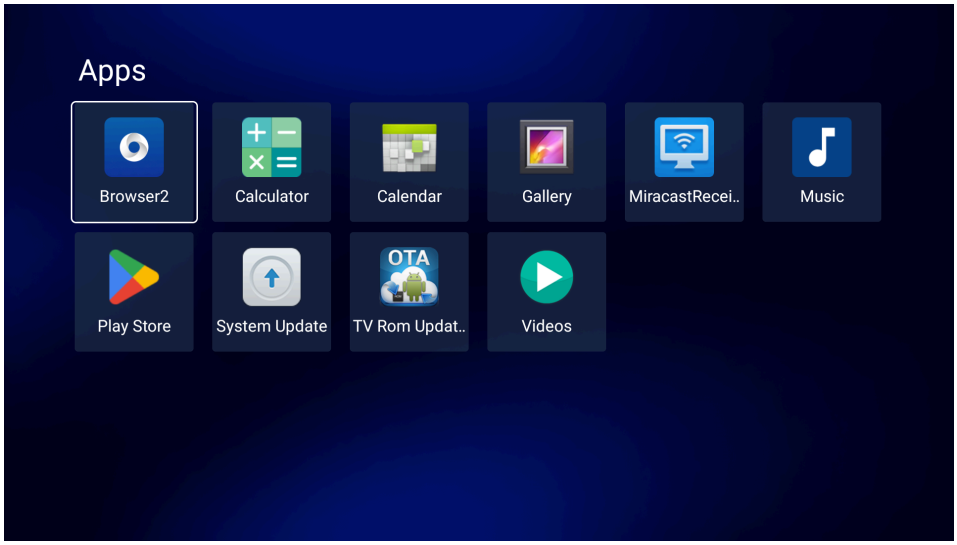


Figure 3 - Default applications on SuperBox S6MAX

Google Play Store is accessible by default, however to get the "real deal" an additional application store must be installed. People familiar with the Android ecosystem would find this step concerning. One main advantage of the official Play Store is that Google is constantly monitoring its catalog, and is dedicated to keeping it free of malware and other grey zone applications. If a vendor is unable or unwilling to have its application deployed in the official store, that should be an immediate red flag. When a user opts in for the alternate store, the box will download and install an APK called "appstore3".

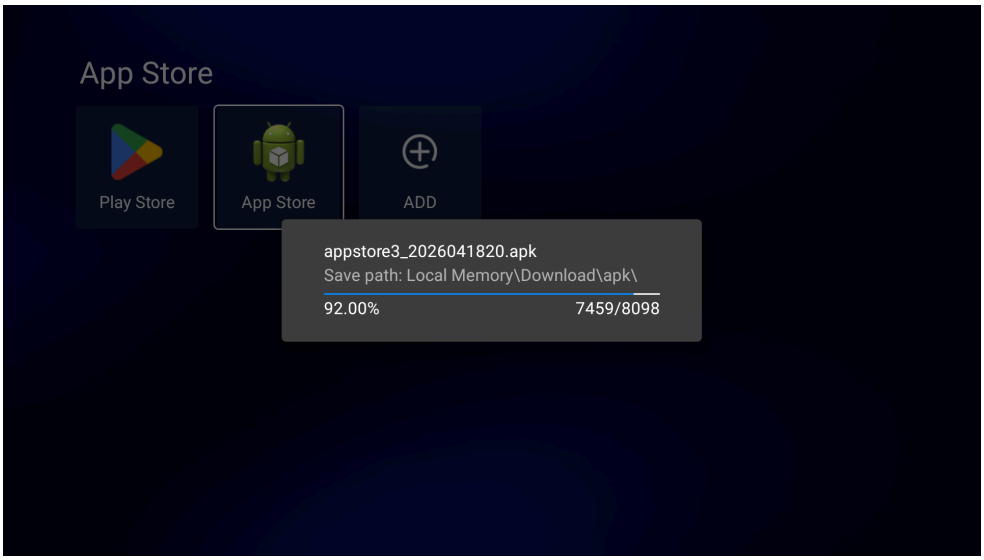


Figure 4 - Installation of the custom app store on SuperBox S6MAX

The newly deployed store will run as a system-privileged application, meaning whenever the user hits "install" on any of the apps, the same silent installation mechanism will be invoked via "pm install" like an attacker would use over ADB. The "Unknown Sources" dialog won't appear either, as the store is not an unknown source from the system's perspective. The store's catalog decisions are made by whomever operates the store, not by the user, and promoted apps tend to be whatever the operator of the store is willing – or being paid – to distribute.

Below is an example of the installation code in the DeviceUtil class of "appstore3", which executes a shell command to silently install an APK using root permissions.

```
process = Runtime.getRuntime().exec("su");
dos = new DataOutputStream(process.getOutputStream());
dos.write(("pm install -r " + inputMsg + "\n").getBytes(Charset.forName("utf-8")));
dos.flush();
dos.writeBytes("exit\n");
dos.flush();
process.waitFor();
```

Figure 5 - Example of silent APK installation

Starting clean, we made some baseline monitoring of network activity – first, prior to deploying the custom store, second after the store was ready for use, and finally when a set of applications were also installed. There was no suspicious activity in the first two cases, but once a fair number of the newly available applications were ready for use, things started to look vastly different.

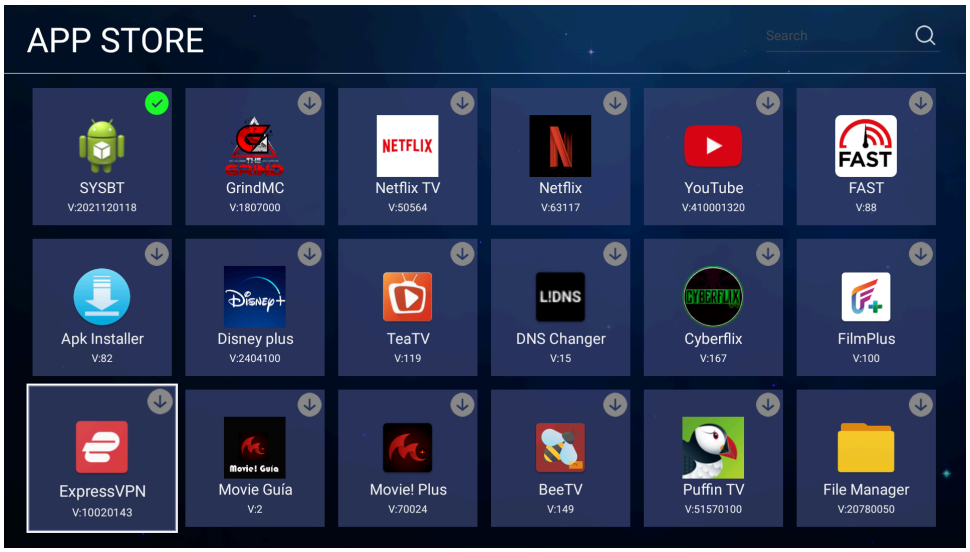


Figure 6 - Applications in the custom store on SuperBox S6MAX

Popanet in Cyberflix

When we started to review the new additions on the software side, it did not take long to find something which justified further investigation. Among the few dozen applications which can be found in SuperBox's custom App Store, Cyberflix TV seemed to come bundled with certain extra functionality. Upon launching the Cyberflix TV application, it immediately raised suspicion by initiating high-volume traffic to a remote machine on port 6000.

```
tcp6       0      0 ::ffff:192.168.13:39998 ::ffff:99.84.91.91:443  ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:40002 ::ffff:99.84.91.91:443  ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6    1199      0 ::ffff:192.168.13:41138 ::ffff:192.178.25.1:443 CLOSE_WAIT  994/com.google.android.gms.pe
tcp6       0      0 ::ffff:192.168.13:36588 ::ffff:51.89.11.19:6000 ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:45360 ::ffff:99.84.91.52:443  ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:40010 ::ffff:99.84.91.91:443  ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:49736 ::ffff:74.125.71.1:5228 ESTABLISHED 994/com.google.android.gms.pe
tcp6       0      0 ::ffff:192.168.13:43456 ::ffff:142.251.153.:443 ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:58456 ::ffff:162.159.138.:443 ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:40000 ::ffff:99.84.91.91:443  ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:39988 ::ffff:99.84.91.91:443  ESTABLISHED 7166/com.cybermedia.cyberflx
tcp6       0      0 ::ffff:192.168.13:39068 ::ffff:148.251.20.7:443 ESTABLISHED 7166/com.cybermedia.cyberflx
```

Figure 7 - Example of Cyberflix communication on port 6000

Analysis of the traffic revealed that the remote machine's address was retrieved from the "lb.gmslb.net" server. This address was utilized within the Popanet package found in the "classes3.dex" file.

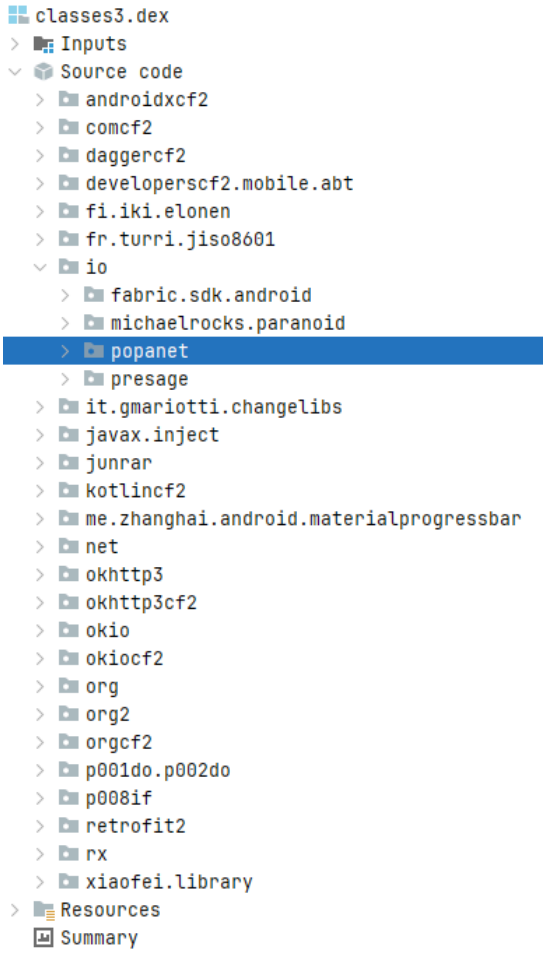


Figure 8 - Popanet package inside classes3.dex

Initial registration

At this point, the nature of the communication was unclear, however it warranted a closer look. Before the device actually could speak to a remote / tunnel server, it had to register itself and request the list of current tunnel servers. This discovery happens over a single HTTPS GET request sent to a registration server with the device's identity and status encoded directly in the URL. The URL of the registration server is hardcoded into the Popa.Builder class.

```
public static class Builder {
    private boolean enablePopaclientLogging;
    private boolean loggable;
    private boolean mobileForeground;
    private String publisher;
    private String userId;
    private String category = "888";
    private String regUrl = "https://lb.gmslb.net:5002/devicereg";
    private String regEndpoint = Popa.u;
```

Figure 9 - Hardcoded tunnel server address

Although the registration URL was fixed, we suspected that multiple versions of the SDK might exist with different hardcoded servers. Based on the registration URL and the server response we were able to find five additional addresses via Shodan.

Figure 10 - Popanet servers found on Shodan (Source: https://www.shodan.io/)

During testing, all the above addresses except “135.181.137.6” returned the following URLs in their response:

["s1832.viki-play.com:6000","s269.viki-play.com:6000","s6.viki-play.com:6000","s4.viki-play.com:6000","s5.viki-play.com:6000","s7.viki-play.com:6000","s1766.viki-play.com:6000","s1752.viki-play.com:6000","s1746.viki-play.com:6000","s1760.viki-play.com:6000","s1758.viki-play.com:6000","s1764.viki-play.com:6000","s1750.viki-play.com:6000","s1730.viki-play.com:6000","s1754.viki-play.com:6000","s1748.viki-play.com:6000","s1708.viki-play.com:6000","s1756.viki-play.com:6000","s1726.viki-play.com:6000","s1762.viki-play.com:6000","s1744.viki-play.com:6000","s1732.viki-play.com:6000","s1408.viki-play.com:6000","s18.viki-play.com:6000","s1694.viki-play.com:6000","s1842.viki-play.com:6000","s1834.viki-play.com:6000","s1830.viki-play.com:6000","s1844.viki-play.com:6000","s1824.viki-play.com:6000","s1836.viki-play.com:6000","s1838.viki-play.com:6000","s1814.viki-play.com:6000","s1851.viki-play.com:6000","s1852.viki-play.com:6000","s1880.viki-play.com:6000","s1862.viki-play.com:6000","s01688.viki-play.com:6000","s1828.viki-play.com:6000","s1822.viki-play.com:6000","s01687.viki-play.com:6000","s1854.viki-play.com:6000","s1807.viki-play.com:6000","s1826.viki-play.com:6000","s1810.viki-play.com:6000","s1812.viki-play.com:6000","s1856.viki-play.com:6000","s1816.viki-play.com:6000","s1820.viki-play.com:6000","s1848.viki-play.com:6000","s1850.viki-play.com:6000","s1846.viki-play.com:6000","s1840.viki-play.com:6000"],"extra":""}

The response from “135.181.137.6” contained different domains:

["s1832.gmslb.net:6000","s269.gmslb.net:6000","s6.gmslb.net:6000","s4.gmslb.net:6000","s5.gmslb.net:6000","s7.gmslb.net:6000","s1766.gmslb.net:6000","s1752.gmslb.net:6000","s1746.gmslb.net:6000","s1760.gmslb.net:6000","s1758.gmslb.net:6000","s1764.gmslb.net:6000","s1750.gmslb.net:6000","s1730.gmslb.net:6000","s1754.gmslb.net:6000","s1748.gmslb.net:6000","s1708.gmslb.net:6000","s1756.gmslb.net:6000","s1726.gmslb.net:6000","s1762.gmslb.net:6000","s1744.gmslb.net:6000","s1732.gmslb.net:6000","s1408.gmslb.net:6000","s18.gmslb.net:6000","s1694.gmslb.net:6000","s1842.gmslb.net:6000","s1834.gmslb.net:6000","s1830.gmslb.net:6000","s1844.gmslb.net:6000","s1824.gmslb.net:6000","s1836.gmslb.net:6000","s1838.gmslb.net:6000","s1814.gmslb.net:6000","s1851.gmslb.net:6000","s1852.gmslb.net:6000","s1880.gmslb.net:6000","s1862.gmslb.net:6000","s01688.gmslb.net:6000","s1828.gmslb.net:6000","s1822.gmslb.net:6000","s01687.gmslb.net:6000","s1854.gmslb.net:6000","s1807.gmslb.net:6000","s1826.gmslb.net:6000","s1810.gmslb.net:6000","s1812.gmslb.net:6000","s1856.gmslb.net:6000","s1816.gmslb.net:6000","s1820.gmslb.net:6000","s1848.gmslb.net:6000","s1850.gmslb.net:6000","s1846.gmslb.net:6000","s1840.gmslb.net:6000"],"extra":""}

Extending the search further by utilizing Plume telemetry revealed another large pool of domains and servers:

s117.adfuse-ssp.com
s117.bcc-ssp.com
s118.adfuse-ssp.com
s120.bcc-ssp.com
s1244.net-echo.com
s1244.pixellog.io
s1248.gmslb.net
s1248.house-spirit.com
s1256.net-echo.com
s126.bcc-ssp.com
s127.bcc-ssp.com

With a quick enumeration, we were able to identify and verify 255 IP addresses behind these domains.

s1807.viki-play.com (57.128.231.179)
s1808.viki-play.com (148.113.217.28)
s1810.viki-play.com (57.128.231.172)
s1812.viki-play.com (57.128.231.178)
s1814.viki-play.com (57.128.231.166)

s1816.viki-play.com (57.128.231.191)
s1818.viki-play.com (57.128.189.112)
s1820.viki-play.com (57.128.231.196)
s1822.viki-play.com (57.128.231.167)
s1824.viki-play.com (51.195.24.115)
s1826.viki-play.com (57.129.39.241)
s1828.viki-play.com (51.195.24.72)
s1830.viki-play.com (51.89.11.246)
s1832.viki-play.com (51.195.24.3)
s1834.viki-play.com (51.195.24.6)
s1836.viki-play.com (51.195.24.58)
…

The control channel

Every interaction between the device and the server runs over a single, long-lived TCP connection wrapped in TLS using SSLSocketFactory. The connection target is taken from the server list of the registration server's response. Alternatively there is a hardcoded fallback address ("s1.gmslb.net:6000") in case registration does not succeed.

Once the handshake is complete, the first action by the device is to send a "REGISTER" message. Then it waits up to five seconds expecting to receive a "REG_REPLY" response. If the reply arrives and successfully validates the registration within the given time interval, the connection enters a steady state and goes into an infinite loop reading and dispatching messages. If the reply doesn't arrive or doesn't validate, the socket will be closed and communication with the next server from the list will be attempted after an additional ten second delay.

Message structure

Messages in both directions have the same structure: an 8-byte header followed by a payload. The header is two 32-bit big-endian integers, a "type code" and a "total length". The total length includes the 8-byte header itself, so the actual payload is total length - 8 bytes.

Inside the payload resides a type-length-value (TLV) container which the code refers to as TlvBox. TlvBox is a sequence of entries, each structured as [tag:int32][entry_length:int32][bytes] where entry_length counts the 8-byte preamble. Entries are flexible, they can contain raw bytes, UTF-8 strings, fixed-width integers, or recursively nested TlvBoxes.

Every application message uses the same framing pattern: the outer TlvBox contains exactly one entry, where the tag equals the message type code, and the value is an inner TlvBox.

There are eight message types in total, identified by their type-code as follows:

Code	Name	Direction	Purpose
1	REGISTER	client → server	Initial handshake
2	REG_REPLY	server → client	Acknowledge REGISTER
3	PING	either	Keepalive with timestamp
4	PONG	either	Keepalive reply
5	OPEN_TUNNEL	server → client	Dial a new destination
6	TUNNEL_STATUS	client → server	Report tunnel lifecycle events
7	TUNNEL_MESSAGE	both	Carry tunneled bytes
8	CLOSE_TUNNEL	either	Tear down a tunnel

Field tags inside each message's inner TlvBox use type-specific ranges: REGISTER uses 0x1000–0x100C, REG_REPLY uses 0x2000–0x2001, PING and PONG share 0x3000, OPEN_TUNNEL uses 0x5000–0x5002, TUNNEL_STATUS uses 0x6000–0x6001, TUNNEL_MESSAGE uses 0x7000–0x7001, and CLOSE_TUNNEL uses 0x8000. The numbering scheme is deliberate but incomplete — some messages like PONG reuse PING's tag, which works because the outer type code disambiguates them.

Message types

REGISTER carries the device's full identity in a single payload. The fields in order are:

- UID (string, tag 0x1000) — the persistent UUID
- Publisher ID (string, 0x1001) — the integrator's identifier
- State (string, 0x1002)
- City (string, 0x1003) — however the code also writes carrier/operator to tag 0x1003 earlier in the same box, which is either an obfuscator artifact or an actual collision
- Version (string, 0x1004) — "2.0.24" or "2.0.24fg" depending on foreground mode
- ASN (string, 0x1005)
- Country (string, 0x1006)

- Init type (int32, 0x1008) — value 1 for new device, 2 for update
- Device model (string, 0x1009) — Build.MANUFACTURER + "_" + Build.MODEL
- Extra info (string, 0x100A) — only written if init type is not 1
- OS version (string, 0x100B) — SDK_INT + "(Android " + RELEASE + ")"
- Network type (string, 0x100C) — "WIFI", "4G", etc.

REG_REPLY is minimalistic; it contains a status byte at tag 0x2000 and an optional message string at tag 0x2001. Client side validation involves checking that the message type is 2, the status byte equals to 1, and if the message string also presents it will get logged. Any other status value means failure and the handshake gets aborted.

PING and PONG

The keepalive protocol is bidirectional and gets initiated by the client. At every scheduler tick - meaning every 15 seconds - the client checks the timestamp of the last received message. If more than 60 seconds have passed, the client creates a PING message containing the current wall-clock time at tag 0x3000 and sends it over. After that, it records the timestamp locally for later validation.

When a PONG message arrives, the client parses the timestamp field and compares it against the previously recorded value. If they match, the exchange is considered valid and the code logs the round-trip time. If they don't, the PONG is silently discarded.

The server can also send PING messages in the opposite direction. When the client receives one, it extracts the timestamp and echoes it back in a PONG to the same tag. This provides the server an ability to measure round-trip time to the device and detect unresponsive clients.

OPEN_TUNNEL is the only message type that the protocol actually exists to deliver. It carries three fields:

- Tunnel ID (int64, tag 0x5000) — server-generated unique identifier for the new tunnel
- Host (string, 0x5001) — destination hostname or IP literal
- Port (int32, 0x5002) — destination TCP port

The tunnel ID is deliberately 64-bit due to the server using it as the correspondence token for all subsequent traffic. Every byte flowing through the tunnel in either direction carries this ID in its TUNNEL_MESSAGE header, meaning the tunnel ID namespace must be unique across the operator's entire customer base, not just per device.

The host field can be either a DNS name or a literal IP address. The device resolves it with InetAddress.getByName() prior to connecting. The resolution is implemented in the tunnel worker thread rather than the control channel thread, preventing the blocking of other tunnels or control messages by a slow DNS lookup.

TUNNEL_MESSAGE is the carrier for tunneled traffic in both directions. There are two fields used:

- Tunnel ID (int64, tag 0x7000) — which tunnel the message belongs to
- Payload (byte array, tag 0x7001) — the raw bytes

No framing is added on top of the payload, the bytes are whatever the two endpoints exchange with each other. If the tunnel is carrying HTTPS, the payload will consist of TLS records. If it's carrying raw HTTP, the payload will be ASCII request and response bytes.

TUNNEL_STATUS reports lifecycle events from the device to the server. There are two fields:

- Tunnel ID (int64, tag 0x6000)
- Status code (byte, tag 0x6001)

There are three status values defined:

- 1 (OK) — tunnel connected successfully, ready to carry traffic
- 2 (ERR) — connection attempt failed
- 3 (REMOTE_CLOSE) — destination closed the socket

The client sends exactly one OK after connection was successfully established or one ERR after failure. The server is utilizing these messages to update its internal state; OK means the tunnel is billable, ERR means pick a different device, and REMOTE_CLOSE means the exchange is done.

CLOSE_TUNNEL carries a single field (the tunnel ID at tag 0x8000) which tells the receiving side to tear down the tunnel immediately. Either side can initiate the message. When the server sends CLOSE_TUNNEL, the client will look up the tunnel worker by its ID, call its stop method, close the peer socket, and remove the entry from the tunnel map. When the client sends CLOSE_TUNNEL - which happens if the tunnel is idle for more than an hour - the server presumably marks the tunnel “done” on its side and releases the customer-side allocation.

Questionable internal network protection

The Popanet proxy implements some basic checks in “io.popanet.task.a” in an attempt to protect the internal network, however it won’t prevent a special way of bypass. After the hostname was received in the tunnel and got resolved by InetAddress.getByName, the two most obvious abuse scenarios will be blocked. It will reject the local address ranges (10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16) via isSiteLocalAddress(), and it will also reject 127.0.0.0/8 via isLoopbackAddress().

```
io.popanet.d.a.a(str2, "TCP Client Connecting...");
InetAddress byName = InetAddress.getByName(str);
a(byName);
if (!a(byName)) {
    io.popanet.d.a.b(str2, "Hacking? The Host Resolved Ip is " + byName + " on tunnel id:"
    throw new IllegalArgumentException("Hacking? The tunnel host resolved ip is internal");
}
```

```
private boolean a(InetAddress inetAddress) {
    try {
        InetAddress byAddress = InetAddress.getByAddress(inetAddress.getAddress());
        return !(byAddress.isSiteLocalAddress() || byAddress.isLoopbackAddress());
    } catch (UnknownHostException e) {
        e.printStackTrace();
        return false;
    }
}
```

Figure 11 - Internal network check in Popanet

What it won't cover is pretty much everything else that the InetAddress classification API exposes. Most of the missing categories are irrelevant on an Android device, except for 0.0.0.0 being classified as "any-local". According to RFC 1122 it is technically a "non-routable" meta-address that should only be used as source and never as destination. The Linux and macOS kernels historically handled a destination of 0.0.0.0 by automatically routing the request to the loopback interface. This issue was already highlighted in 2024 when multiple browsers and security filters assumed 0.0.0.0 was "invalid" or "non-routable" thus allowing it to pass through. In our case something similar is happening. The verification function allows the 0.0.0.0 address, as it is neither a site local nor a loopback address, but the underlying kernel routes it to 127.0.0.1 and allows the remote side to access the local machine. You can witness this exact behavior below with a SuperBox device, where a connection to 0.0.0.0 is routed to a local listener.

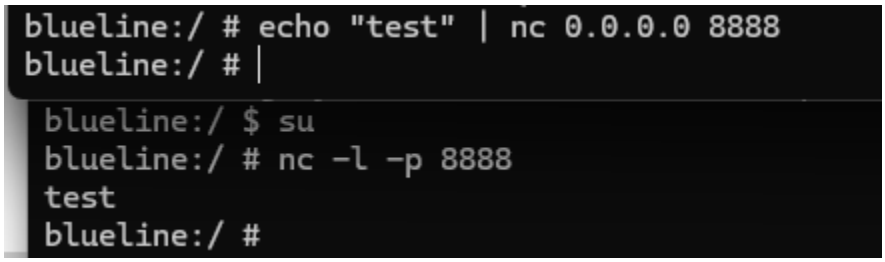


Figure 12 - Local connection reroute

The seriousness of the threat is better understood once we have a look at the services available on localhost. The device listens on 4 TCP and 3 UDP ports, which makes TV remote and ADB connections possible. Out of these, the most dangerous is the ADB connection highlighted previously.

```
blueline:/ $ su
blueline:/ # netstat -nap
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program Name
tcp        0      0 127.0.0.1:5005          0.0.0.0:*               LISTEN      270/pqd
tcp      130      0 192.168.137.191:45398   216.58.205.131:443      CLOSE_WAIT  1089/com.google.android.katniss:interactor
tcp        0      0 192.168.137.191:38046   64.233.167.188:5228     ESTABLISHED 1089/com.google.android.katniss:interactor
tcp       73      0 192.168.137.191:55062   216.239.32.223:443      ESTABLISHED 1089/com.google.android.katniss:interactor
tcp6       0      0 [::]:6466               [::]:*                  LISTEN      906/com.google.android.tv.remote.service
tcp6       0      0 [::]:5858               [::]:*                  LISTEN      366/adbd
tcp6       0      0 [::]:6467               [::]:*                  LISTEN      906/com.google.android.tv.remote.service
tcp6       0     25 ::ffff:192.168.13:57984 ::ffff:18.172.242.2:443 FIN_WAIT1   -
tcp6       0    182 ::ffff:192.168.137:5858 ::ffff:192.168.13:10392 ESTABLISHED 366/adbd
tcp6       1      0 ::ffff:192.168.13:38326 ::ffff:144.217.77.23:80 CLOSE_WAIT  1154/com.sb.launch6
tcp6       0     25 ::ffff:192.168.13:59806 ::ffff:18.172.242.9:443 FIN_WAIT1   -
tcp6       0     25 ::ffff:192.168.13:57982 ::ffff:18.172.242.2:443 FIN_WAIT1   -
tcp6      32      0 ::ffff:192.168.13:59274 ::ffff:14.22.7.140:443  CLOSE_WAIT  1154/com.sb.launch6
tcp6       0     25 ::ffff:192.168.13:46624 ::ffff:18.172.242.2:443 FIN_WAIT1   -
udp        0      0 0.0.0.0:5353            0.0.0.0:*                           381/mdnsd
udp        0      0 127.0.0.1:30001         0.0.0.0:*                           246/android.hardware.bluetooth@1.0-service
udp        0      0 0.0.0.0:51050           0.0.0.0:*                           381/mdnsd
udp        0      0 192.168.137.191:68      192.168.137.1:67        ESTABLISHED 772/com.android.networkstack.process
udp6       0      0 [::]:5353               [::]:*                              381/mdnsd
udp6       0      0 [::]:5353               [::]:*                              381/mdnsd
udp6       0      0 [::]:40856              [::]:*                              381/mdnsd
```

Figure 13 - Example of running services

Inside the Popanet proxy

Residential proxies are often viewed as simple tools providing anonymity, but if we change our perspective and look at it through the lens of a host node, they essentially appear as tunnels carrying streams of unknown data. It is specifically for this reason that we wanted to find out if under the hood the traffic passing through the device is being used for harmless web scraping or if there is malicious intent as well. By auditing the actual usage patterns, we can also see if the "security" of these connections is a technical reality or if sensitive data is being exposed as it flows through an unverified domestic relay. To capture the data moving through the relay, we re-routed the proxied traffic and enabled logging. This setup allowed us to intercept and examine the communication in real-time, providing an unfiltered look at the intent behind the connections passing through our node. It is worth noting that Popanet is not exclusively used by Cyberflix TV; there are well-documented campaigns such as the Vo1d botnet where it was included in the form of a plugin. This also backed up our earlier suspicion about the existence of different versions of the Popanet SDK.

While the vast majority of web traffic today uses HTTPS, our testing revealed that the "security" provided by the protocol is often superficial. A surprising number of automated scripts and applications routed through our proxy failed to perform proper SSL certificate verification or were configured to accept a custom root certificate. In cases involving interactive browsing, the fact that we were able to decrypt the traffic suggests that users likely bypassed security warnings and accepted our intercepting certificate to be able to proceed further.

This oversight turns the residential proxy into a transparent MitM (Man-in-the-Middle) as the encryption was effectively stripped away, allowing us to see the original content of the requests, including headers, cookies, and even login credentials. The URLs captured during this session provided a realistic look at how these networks are being (ab)used. The

traffic we've seen was a mixture of automated maintenance, search engine scraping, and sensitive personal data.

Fingerprinting and IP verification

A significant amount of the traffic is purely operational. Repeated calls to various IP-lookup services suggest that the proxy users are constantly verifying that the residential IP is active and not yet blacklisted.

http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip
http://98.89.132.151	GET	/ip
https://pro.ip-api.com	GET	/json?key=R6jMCDblUGui2ha
https://pro.ip-api.com	GET	/json?key=R6jMCDblUGui2ha
http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip
https://pro.ip-api.com	GET	/json?key=R6jMCDblUGui2ha
http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip
http://44.231.131.214	GET	/api/v1/get_client_ip

Figure 14 - Example of IP lookups

Sensitive authentication

We captured requests to the account management system of EA (Electronic Arts) and verification codes for WhatsApp. In a real-world attack scenario, a proxy operator having access to these codes could take over the user account in real-time.

https://signin.ea.com	POST	/p/juno/login?execution=e1275032913s2&initref=https%3A%...
https://accounts.ea.com	GET	/connect/auth?response_type=code&redirect_uri=https%3A%...
https://myaccount.ea.com	GET	/am/callback/authorization?code=QUOxAKFIN9ZoK8XkONKux..
https://myaccount.ea.com	GET	/am/ui/account-information
https://myaccount.ea.com	GET	/am/data/1/security-privacy
https://myaccount.ea.com	POST	/am/data/1/targeted-adv-settings
https://myaccount.ea.com	POST	/am/data/1/targeted-adv-settings
https://myaccount.ea.com	POST	/am/data/1/send-code

Figure 15 - Example of authentication requests

Intimate search history

The logs also included search queries across Google and Bing for diverse topics, ranging from specific Chinese-language educational PDFs through medical queries to local service lookups like "air conditioner service" or "klaviertransport".

https://www.google.com	/complete/search?q&cp=0&client=mobile-gws-wiz-serp&xssi=t...
https://www.google.com	/complete/search?q&cp=0&client=mobile-gws-wiz-serp&xssi=t...
https://www.google.com	/complete/search?q&cp=0&client=mobile-gws-wiz-serp&xssi=t...
https://www.google.com	/complete/search?q=klaviertransport%20berlin&cp=0&client=...
https://www.google.com	/complete/search?q=klaviertransport%20berlin&cp=0&client=...
https://www.google.com	/maps/vt?pb=!1m4!1m3!1i9!2i275!3i168!1m4!1m3!1i9!2i274!3i1...
https://www.google.com	/search?q=klaviertransport+berlin&oq=klaviertransport+berlin...
https://www.google.com	/search?q=klaviertransport+berlin&oq=klaviertransport+berlin...

Figure 16 - Example of search engine queries

Bot mitigation bypass

While not significant in quantity, we found specific instances of interaction with high-end WAFs (Web Application Firewall). There were requests to the Cloudflare Challenge Platform and to AWS WAF SDK endpoints. These individual requests are of significant importance; they showcase a user attempting to solve or bypass JavaScript challenges. By using residential IPs, they can avoid the "hard block" typically applied to data centers, allowing a bot to appear as a legitimate home user.

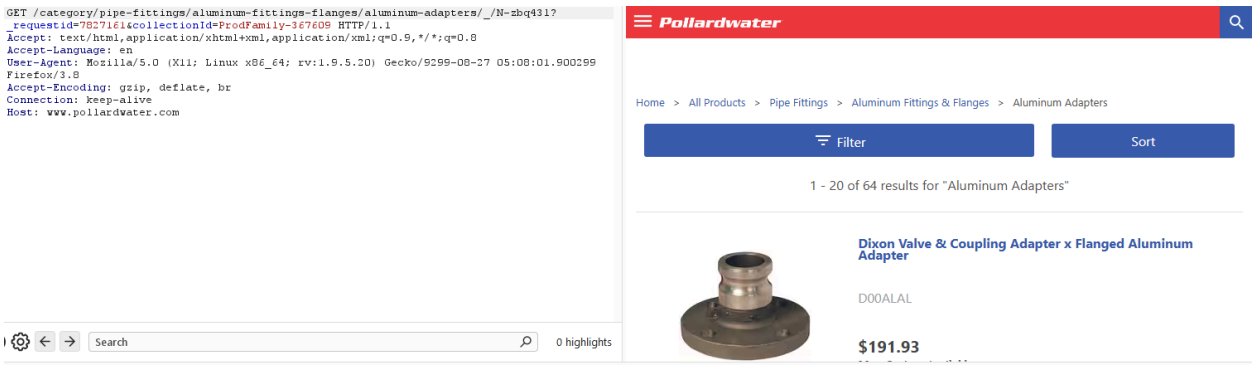
https://b2037b2ab8ee.edge.sdk.awswaf.com	/b2037b2ab8ee/e50decfea67c/inputs?client=ios
https://b2037b2ab8ee.edge.sdk.awswaf.com	/b2037b2ab8ee/e50decfea67c/inputs?client=ios
https://b2037b2ab8ee.edge.sdk.awswaf.com	/b2037b2ab8ee/e50decfea67c/verify
https://b2037b2ab8ee.edge.sdk.awswaf.com	/b2037b2ab8ee/e50decfea67c/verify

Figure 17 - Example of bypass queries

Additional requests

To illustrate the broad nature of the activity, here are some notable examples of activity passing through the proxy.

Someone navigated into specialized industrial hardware catalogs.



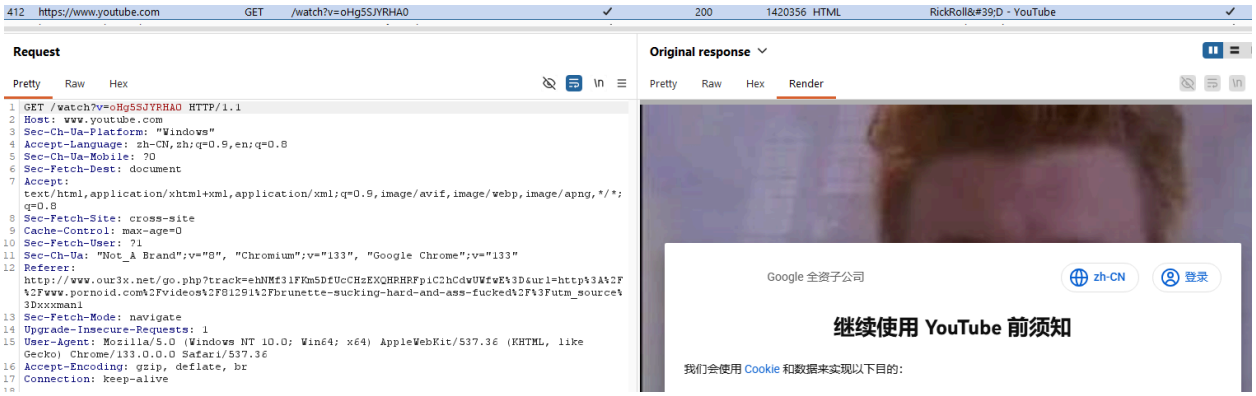
Someone was looking up a recipe for banana cake while hiding behind a domestic IP.

```
GET /banana-bundt-cake-cream-cheese-frosting/?
query-9b76970e-page=10 HTTP/2
Host: www.tastymealstocook.com
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Upgrade-Insecure-Requests: 1
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Cache-Control: max-age=0
Sec-Fetch-User: ?1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0
Safari/537.36
Referer:
https://www.tastymealstocook.com/banana-bundt-cake-cream-che
ese-frosting/?query-9b76970e-page=7
Sec-Fetch-Site: same-origin
Sec-Ch-Ua: "Not_A Brand";v="8", "Chromium";v="136", "Google
Chrome";v="136"
Sec-Ch-Ua-Mobile: ?0
Sec-Ch-Ua-Platform: "macOS"
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/
avif,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

Jump to Recipe · Print Recipe

Cozy Banana Bundt Cake

And finally we bumped into an example of humor dating back as far as the early 2000s. We observed a user session that transitioned from an adult portal directly into the inescapable trap of a YouTube Rickroll.



End of Part One

With this, we are concluding the first part of our journey into the deeper realms of SuperBox devices and residential proxies. In part two, we will continue our investigation and focus on how different kinds of malware are exploiting the presence of already deployed proxies. Stay tuned!

IOCs

All the corresponding IOCs will be shared after each part 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.
