2025 BERLIN

24 - 26 September, 2025 / Berlin, Germany

VO1D RISING: INSIDE THE BOTNET CONTROLLING 1.68 M+ ANDROID TVS WORLDWIDE

Alex Turing

QI-ANXIN, China

hex.asmer@gmail.com

ABSTRACT

Vo1d is a massive *Android* TV botnet with millions of infected devices. Attackers exploit compromised *Android* TVs to carry out various illegal activities, including click fraud, programmatic ad fraud and residential proxy services. This paper will examine Vo1d's technical foundations, including its Bigpanzi-like string encryption, ASR-variant XXTEA algorithm, DGA mechanism, and dual-layer (RSA + XXTEA) payload protection. It also details the *XLab* toolkit employed during reverse engineering of the Vo1d botnet. These technical insights aim to assist the research community in better understanding both the Vo1d botnet infrastructure and its operators.

INTRODUCTION

The Vo1d botnet first came to light in September 2024 [1]. Far from fading, the botnet has showcased impressive resilience through continuous technical evolution. About three months after its first appearance, *QAX XLab* captured 'jddx', a new downloader deployed by the evolved Vo1d [2]. Leveraging this as an entry point, we conducted a thorough investigation, gathering 149 samples and mapping out extensive infrastructure: one reporter, four downloaders, 21 C2 domains, 258 DGA seeds, and over 100,000 DGA-generated domains.

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...

Figure 1: C2s.

We registered 258 of these DGA domains to study Vo1d's network scale and geographic distribution.

Domain Name	Expiration	Status
iqflb08f579d4756.top	9 Jan 2026	Active
iykxab825924a6cb.top	9 Jan 2026	Active
npnxu1a7f035ebc7.top	9 Jan 2026	Active
ntgup3e65f197570.top	9 Jan 2026	Active

Figure 2: DGA domains.

Our data indicates that the Vo1d botnet currently sustains approximately 800,000 daily active IPs, peaking at 1,590,299, as shown in Figure 3.

These IP addresses are distributed across more than 200 countries and regions worldwide. According to cumulative infection data collected between 1 January and 30 April, the top 10 most affected countries are as follows: Brazil (17.64%), India (12.04%), South Africa (8.3%), China (7.28%), Indonesia (5.33%), Argentina (4.3%), Morocco (3.65%), Thailand (2.99%), Mexico (2.70%) and Pakistan (2.37%).

Another perspective on the massive scale of the Vo1d botnet lies in its C2 domain rankings. The Tranco ranking system, a comprehensive metric for gauging website popularity, provides more accurate and reliable global website ranking data. Within Tranco's top one million rankings, a significant portion of Vo1d's C2 domains have broken into the global top 500,000, with some even reaching the 50,000 range, as shown in Figure 5.

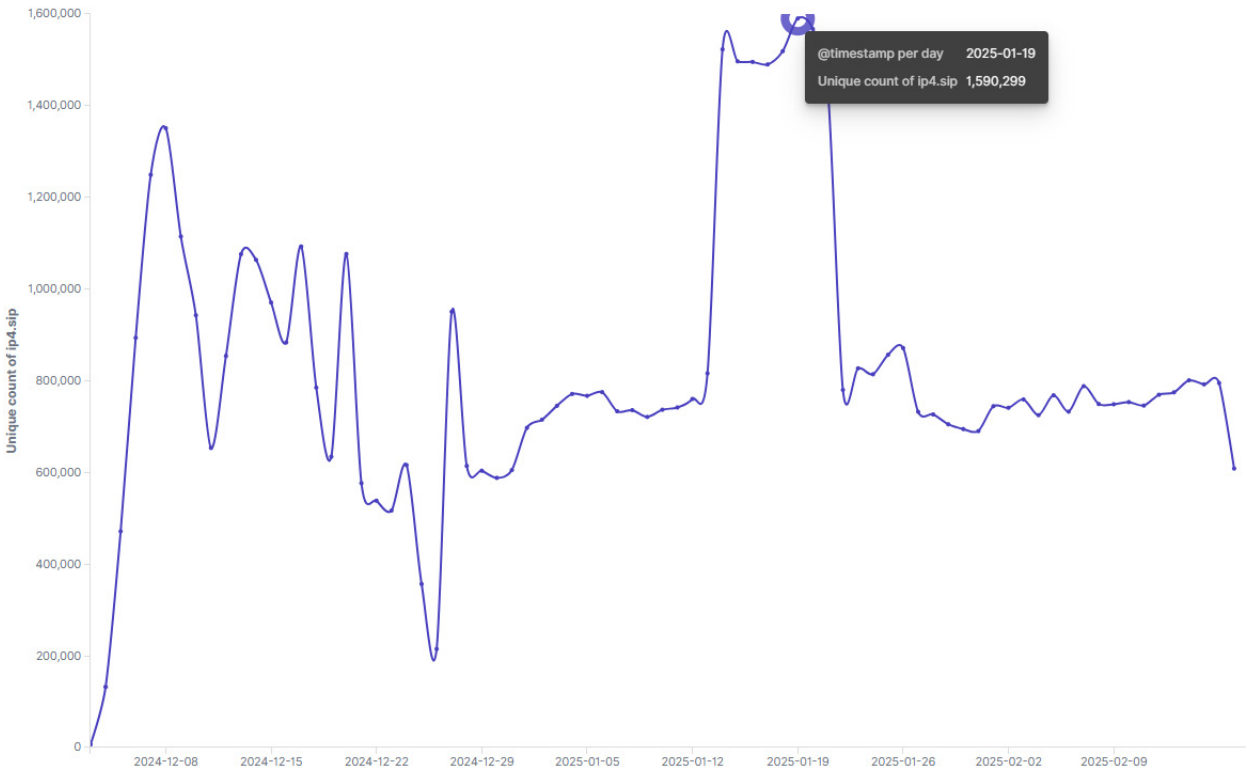

Figure 3: Daily active IPs.

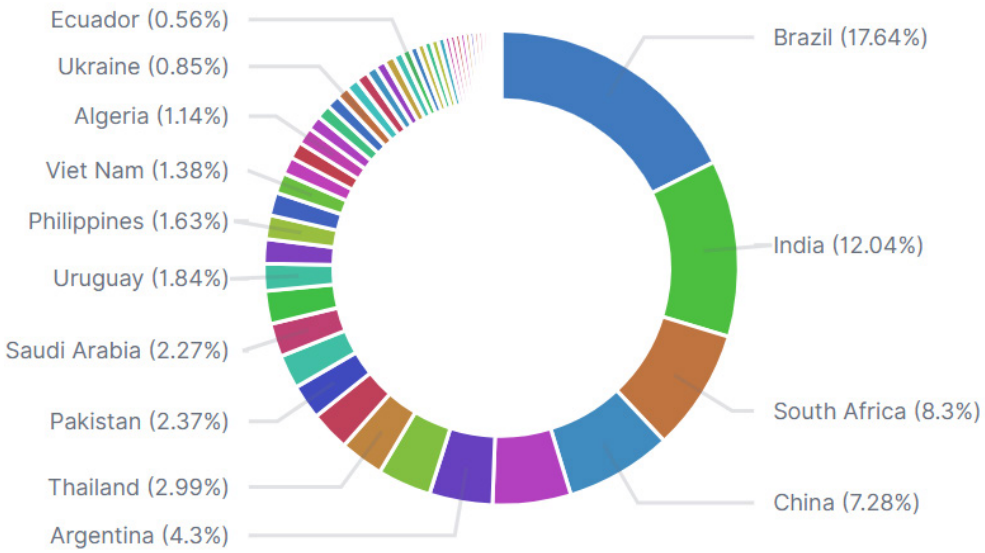

Figure 4: Affected countries.

```
└─$ 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
```

Figure 5: C2 domain rankings.

Particularly noteworthy is the domain ttss442, which was registered on 3 November 2024, yet surged into the global top 55,000 within just a few months. This phenomenon underscores the botnet's vast scale and staggering activity level from a different angle.

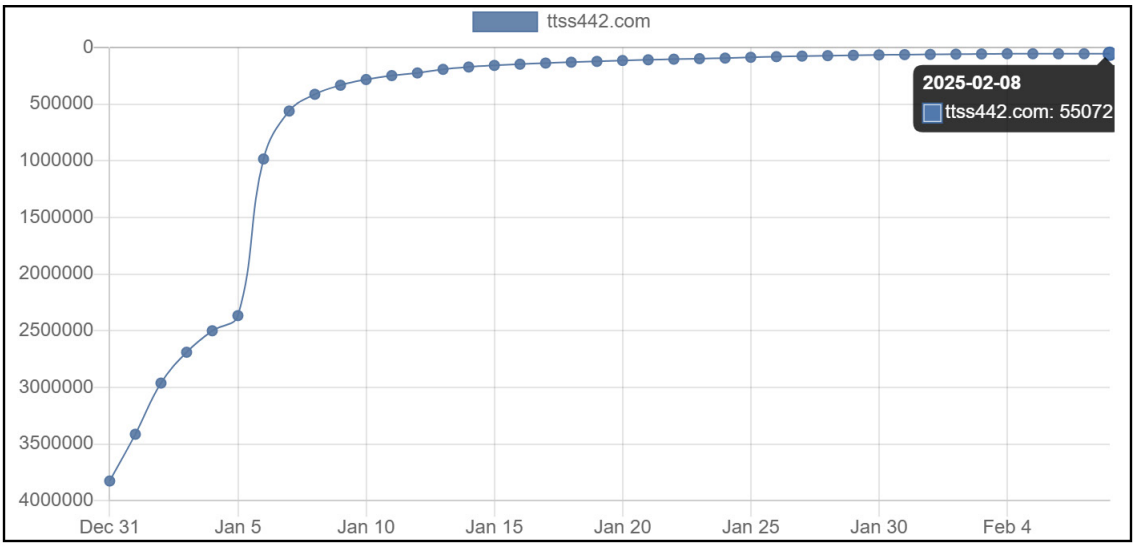

Figure 6: Ranking of domain ttss442.

MITIGATION

Starting 14 January 2025, Vo1d's daily active IPs surpassed 1.5 million for an entire week. In February 2025, *QAX XLab*, in collaboration with the Shadowserver Foundation, successfully took over 12 Vo1d C2 servers. This time, we are able to assess the scale with greater precision. Updated metrics reveal an average of 1.5 million daily active IPs, with a notable peak of 1,685,140.

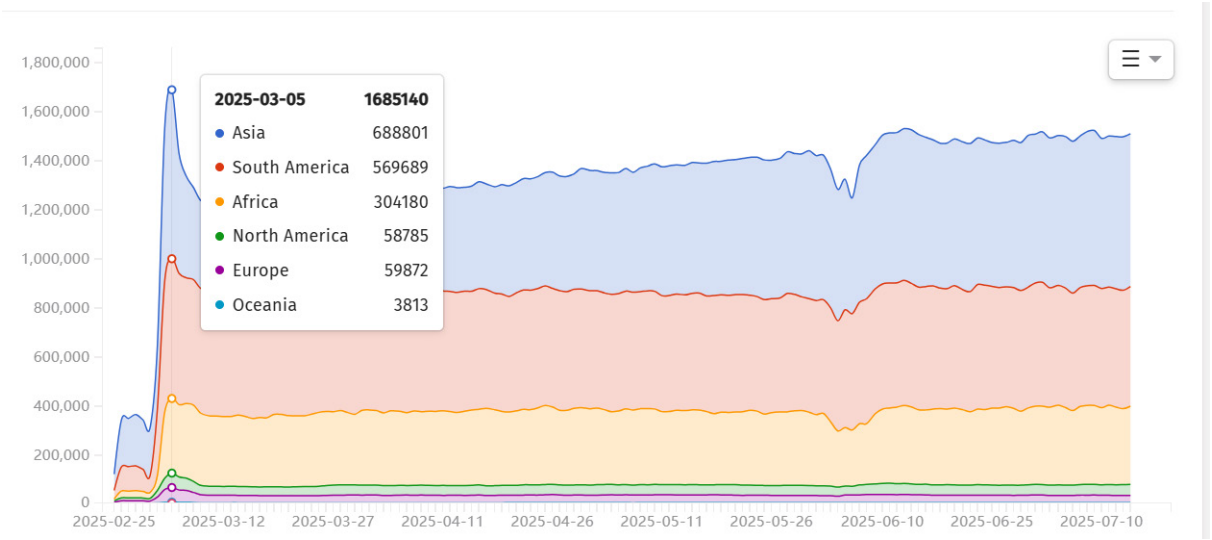

Figure 7: Daily active IPs updated metrics.

JOURNEY OF ANALYSIS

Undoubtedly, the Vo1d botnet – with its vast scale of over one million devices and relentless technological advancements – represents a persistent and significant threat to global cybersecurity.

To support the security community's research on Vo1d, we will explore Vo1d's technical underpinnings, including its strings encryption, DGA mechanism, payload protection, and so on.

0x1: Bigpanzi-like string encryption

Vo1d uses a string encryption method identical to that used by the Bigpanzi group [3] to safeguard sensitive config data.

The ciphertext is structured as follows:

1. **First two bytes**: XOR key (used to decrypt subsequent data).
2. **Third byte**: XOR-encrypted length field (after decryption, this byte specifies the size of the following ciphertext).
3. **Remaining bytes**: XOR-encrypted payload (length defined by the decrypted third byte).

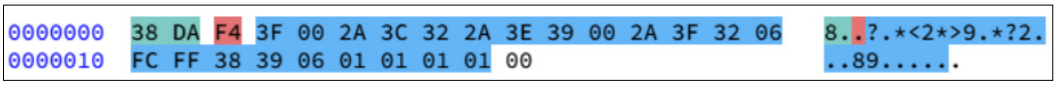

Figure 8: Encrypted string.

Here's the Python code for decryption:

```
def decode_str(buf):
        length = buf[0] ^ buf[1] ^ buf[2]
        out = bytearray(buf)
        out[2] = length
        out[3:length+3] = bytearray(
                (((buf[i] ^ buf[1]) - buf[1]) & 0xff) ^ buf[0]
                for i in range(3, length + 3)
)
return bytes(out)
```

In the example above, the decrypted ciphertext reveals the downloader, payload, and port.

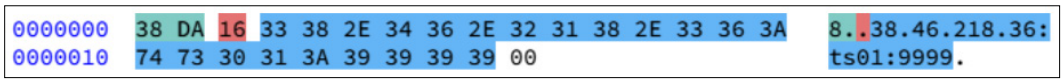

Figure 9: Decrypted string.

To decrypt all strings, *IDAPython* could be used – but the assembly code shows that the ciphertext address (passed as the first argument) requires multiple processing steps.

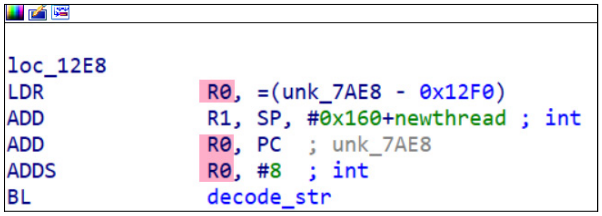

Figure 10: Parameters for decode_str.

For a more streamlined approach, flare_emu [4] simplifies the decryption workflow.

```
1  import flare_emu
2
3  def decbuf(buf):
4      leng=buf[0]^buf[1]^buf[2]
5      out=''
6      for i in range(3,leng+3):
7          tmp=((buf[i]^buf[1])-buf[1])&0xff
8          out+=chr((tmp^buf[0]))
9      return out
10
11 def iterateCallback(eh, address, argv, userData):
12     tmp=ida_bytes.get_bytes(argv[0],3)
13     leng=tmp[0]^tmp[1]^tmp[2]
14
15     if leng!=255:
16         buf=ida_bytes.get_bytes(argv[0],leng+3)
17         print(f'address:{hex(argv[0])}, length:{leng} -->  {decbuf(buf)}')
18
19 eh=flare_emu.EmuHelper()
20 #decode_proc=0x00001BD8
21 decode_proc=eh.analysisHelper.getNameAddr('decode_str')
22 eh.iterate(decode_proc,iterateCallback)
```

Figure 11: flare_emu simplifies the decryption workflow.

The decrypted configuration contains two key elements: XXTEA key (0x7b15) and download URL (0x7ace).

```
address:0x7b15, length:32 -->  b6d5c945d61a73641e710f357214f3e3
address:0x7af9, length:8 -->  su -c id
address:0x7ae8, length:4 -->  root
address:0x7b05, length:12 -->  /data/system
address:0x7af0, length:5 -->  %s/.v
address:0x7ace, length:22 -->  38.46.218.36:ts01:9999
address:0x7aa0, length:13 -->  %s/install.sh
address:0x7ab1, length:25 -->  u:object_r:system_file:s0
```

Figure 12: Decrypted configuration.

0x2: ASR-variant XXTEA algorithm

Vo1d uses a specific algorithm to encrypt the payload which, based on the constants and code patterns, can be easily identified as the XXTEA algorithm.

```
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;
```

Figure 13: XXTEA algorithm.

Does it use standard XXTEA? No, it's modified. A distinctive tweak in Vo1d replaces XXTEA's LSR (logical right shift) with ASR (arithmetic right shift) – a subtle change that's invisible in *IDA*'s decompiled pseudocode.

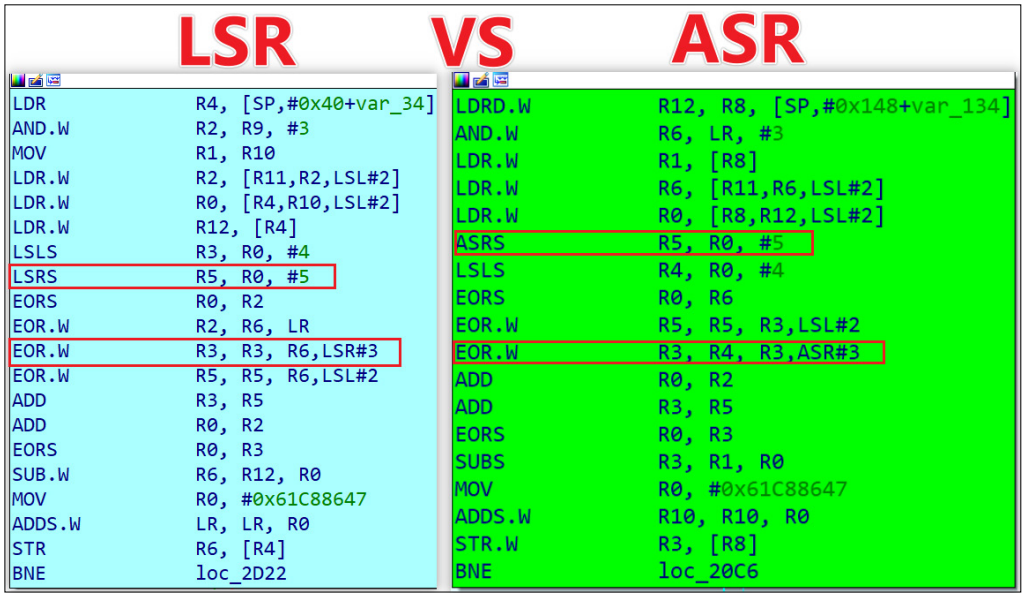

Figure 14: LSR vs ASR.

Therefore, to correctly implement decryption in Python, you need to:

1. Implement an ASR function.
2. Replace LSR operations in the standard XXTEA algorithm with ASR.

```python
def asr(value, shift):
    if value & 0x80000000:  # Check if MSB is set (negative number)
        return (value >> shift) | (0xFFFFFFFF << (32 - shift)) & 0xFFFFFFFF
    else:
        return value >> shift
```

```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
```

Figure 15: Patched XXTEA with LSR operations replaced with ASR.

Using ciphertext `\xf5\x2b\x75\xf0\x47\xc8\xb5\x60\xbd\x86\x70\x36\x7c\xf8\xf0\x76\x82\xc2\x3b\xc2` and key `d99202373076ee9e` as an example, comparative analysis demonstrates that, while standard XXTEA fails, the ASR-XXTEA variant successfully decrypts the C2 address 52.14.24.94.

```
asr_xxtea decryption
00000000: 35 32 2E 31 34 2E 32 34  2E 39 34 3A 38 31 4B 94  52.14.24.94:81K.
00000010: 0E 00 00 00                                       ....
standard xxtea decryption
00000000: 07 FB 71 D5 1C 8B EA 1A  98 0C 76 98 CB 56 DE D1  ..q.......v..V..
00000010: 14 4A 25 BD                                       .J%.
```

Figure 16: Decryption using standard XXTEA vs the ASR-XXTEA variant.

Deviations from using standard algorithms are uncommon in malware in the wild, reflecting the Vo1d group's substantial technical expertise and development resources.

0x3: DGA mechanism

The DGA algorithm of Vo1d employs a Caesar cipher-like method to process the seed. The samples hard code four top-level domains (TLDs) – xyz, top, com, and net – though xyz is not actually utilized in practice. The seed values and the number of domains generated per seed vary across different samples.

```
  switch ( i )
  {
    case 1u:
      seed[1] = sub_B954(*seed + 2 * seed[1]);
      break;
    case 2u:
      seed[2] = sub_B954(*seed + seed[2] - 1);
      break;
    case 3u:
      seed[3] = sub_B954(seed[1] + seed[2] + seed[3]);
      break;
    case 4u:
      seed[4] = sub_B954(seed[1] + 2 * seed[3] + seed[4]);
      break;
  }
}
else
{
  *seed = sub_B954(*seed + seed[3]);
}
```

```
int __fastcall sub_B954(int a1)
{
  return (a1 - 97) % 26 + 'a';
}
```

Figure 17: DGA of Vo1d.

Since the algorithm only processes the first five bytes of the seed, the domain sequences produced by a specific seed show a clear pattern – only the first five bytes change, with the rest remaining constant.

```
DNS     Standard query 0xd689 A dvylx49c6ed34236.top
DNS     Standard query 0xc4cc A hjxdr49c6ed34236.top
DNS     Standard query 0x8fdd A dhsof49c6ed34236.top
DNS     Standard query 0x89c5 A kkuec49c6ed34236.top
```

Figure 18: Only the first five bytes of the domain sequences change, with the rest remaining constant.

Figure 19 shows the functionally equivalent Python code to generate the seed sequence:

```python
def normalize_char(char_code: int) -> int:
    return (char_code - ord('a')) % 26 + ord('a')
def transform_seed(seed: str, iterations: int = 1):
    chars = [ord(c) for c in seed]
    transformations = {
        0: lambda: normalize_char(chars[0] + chars[3]),
        1: lambda: normalize_char(chars[0] + 2 * chars[1]),
        2: lambda: normalize_char(chars[0] + chars[2] - 1),
        3: lambda: normalize_char(chars[1] + chars[2] + chars[3]),
        4: lambda: normalize_char(chars[1] + 2 * chars[3] + chars[4])
    }
    for _ in range(iterations):
        for pos in transformations:
            if pos < len(chars):
                chars[pos] = transformations[pos]()
        yield ''.join(chr(c) for c in chars)
```

Figure 19: Python code to generate the seed sequence.

Taking the seed 'edd3b49c6ed34236' as an example, the first four domains generated by the algorithm perfectly match the domains observed in the Pcap file.

```python
for i in transform_seed("edd3b49c6ed34236", 4):
    print(i+'.top')
```

```
dvylx49c6ed34236.top
hjxdr49c6ed34236.top
dhsof49c6ed34236.top
kkuec49c6ed34236.top
```

Figure 20: First four domains generated by the algorithm.

DGA seeds play a critical role in generating DGA-based C2 domains. Our analysis uncovered 258 seeds that help measure the botnet's scale – 256 obtained through reverse engineering, with the remaining two detected via a novel approach: DNS co-occurrence analysis using the Codomain system.

Codomain is an innovative tool based on DNS co-occurrence technology, designed to monitor and analyse domain relationships by identifying shared query patterns. Its core principle is straightforward: if a group of domains is frequently queried by the same set of hosts within a short timeframe, they are likely associated.

For example, the Vo1d botnet accesses hard-coded C2 domains, DGA-generated C2 domains, and reporter domains during operation. When these queries meet specific temporal conditions, Codomain can establish correlations between them, enabling researchers to track attacker infrastructure. Taking the C2 domain works883.com as an example, it was found to be associated with a cluster of domains following the pattern `{5-bytes}2940637fafa.com`. These domains clearly match Vo1d's DGA naming convention.

Rank	Anchor FQDN	Linked FQDN	Together Count	Ratio	Anchor Count
1	works883.com	fjsjf2940637fafa.com	3	0.1765	17
2	works883.com	tumune.com	3	0.1765	17
3	works883.com	xfgun2940637fafa.com	3	0.1765	17
4	works883.com	kgium2940637fafa.com	3	0.1765	17
5	works883.com	xvxyi2940637fafa.com	3	0.1765	17

Figure 21: The C2 domain works883.com was associated with a cluster of domains following the pattern {5-bytes}2940637fafa.com.

The DGA algorithm verification shows that fjsjf2940637fafa.com produces the identical domain sequence detected by Codomain. This marks the identification of a new seed, `{5-bytes}2940637fafa`, through the Codomain system – achieved without requiring sample capture.

```
for i in transform_seed("fjsjf2940637fafa.com", 4):
    print(i)
    hlrxp2940637fafa.com
    xfgun2940637fafa.com
    kgium2940637fafa.com
    xvxyi2940637fafa.com
```

Figure 22: DGA algorithm verification.

Remarkably, this seed has extended China's infection horizon, with daily bot activity jumping from tens to roughly 20,000. Globally, it has contributed to roughly 150,000 daily infected IPs.

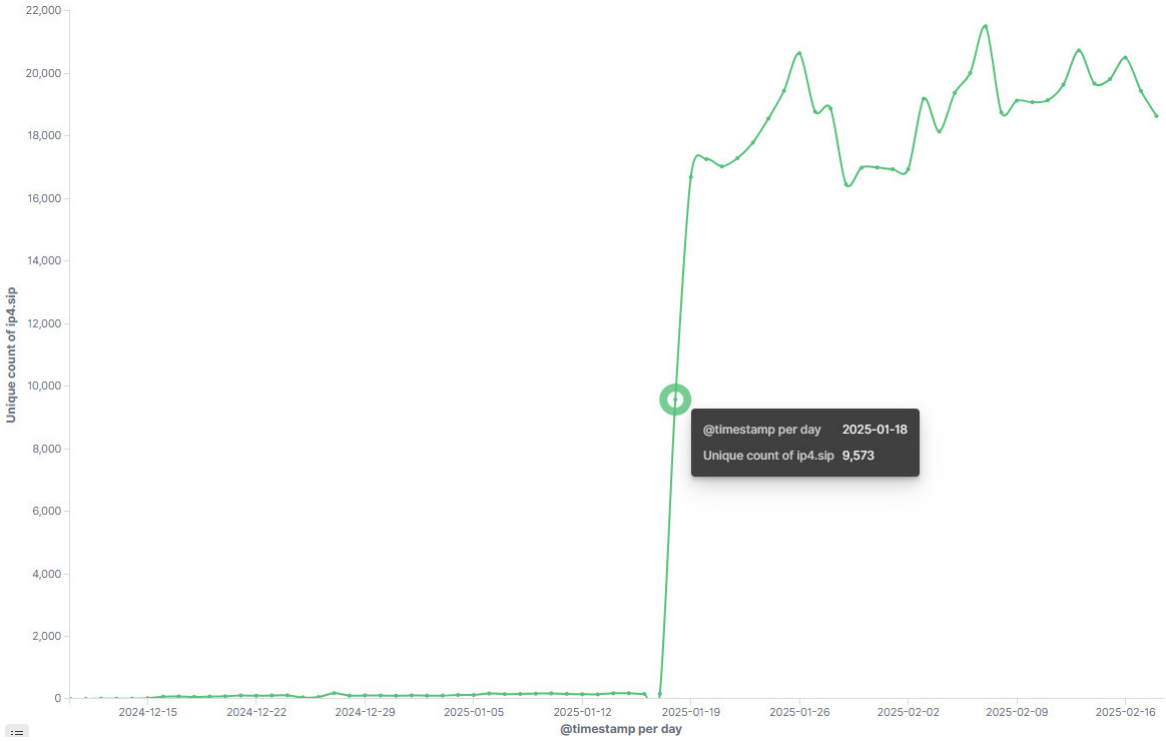

Figure 23: Daily active bots in China.

0x4: Dual-layer payload protection

Vo1d's dual-layer payload protection works by first securing the XXTEA key with RSA, then encrypting the payload via ASR-XXTEA. The Vo1d sample hard codes an RSA public key in the N (modulus) - E (public exponent) format, where the N value is 256 bytes (little-endian), and the E value is a fixed constant of 65537.

```
000026F0  B9 34 C4 68 EA 7C C3 84  29 51 82 D5 36 C6 83 D1
00002700  E6 41 F7 12 47 AB D4 66  5C 09 7F F9 8A D4 0D 8A
00002710  98 8B 62 3A 59 5C 03 F8  6E 2B 82 33 71 D7 7F 9D
00002720  CE D8 28 1D 8A 37 21 EF  59 A9 8A FE 00 7F 22 AE
00002730  88 B4 EA B3 D0 3B AD CF  F5 4A 56 CA FD CB D3 8A
00002740  55 1A F9 B7 1B 1E 6F 05  1F 4D 95 6F AE 92 4F 57
00002750  0D 4A D4 E9 94 6D B8 78  63 37 8A 97 24 C2 77 C2
00002760  05 5B DA 82 94 51 68 A6  CC 68 03 4E EB 8A 1A 1A
00002770  E2 C1 10 DD C6 D4 48 52  C8 09 21 DE D1 25 E4 2B
00002780  F0 9D 9A 22 B6 C8 D3 24  66 2E 75 9B 70 C4 33 B8
00002790  82 1B 05 0B 0F 8A BD 86  11 05 65 CC 33 BC C7 0A
000027A0  43 96 44 7E 25 FB BD D3  E0 B0 B3 62 19 B6 EF DF
000027B0  60 98 E2 F9 8B F3 FE C1  33 1E F1 FF 6C CD 45 65
000027C0  9F CD 49 67 CC 86 9F 95  32 F6 4C 98 73 EC EA CE
000027D0  B1 1B A7 68 5F C5 38 A6  6C 64 8E 65 04 E2 DD 1F
000027E0  0E EC B9 AD 76 03 0B 78  97 13 63 DC 32 43 B0 C8
```

Figure 24: The RSA modulus N used to decrypt the XXTEA key.

The RSA ciphertext can be decrypted simply using Python's `pow` function with the known parameter.

```
n_txt = bytes.fromhex('B9 34 C4 68 EA 7C C3 84 29 51 82 D5 36 C6 83 D1 E6

cipher_txt = bytes.fromhex('7b f2 6d 30 ee a1 2c 92 ba 23 db 24 9d 5e 8d

N = int.from_bytes(n_txt, "little")
cipher = int.from_bytes(cipher_txt, "big")
plain = pow(cipher, 65537, N)
plain_txt = int.to_bytes(plain, 256, "big")
hexdump.hexdump(plain_txt)
```

Figure 25: Decryption using Python pow function.

Taking payload ts01 as an example, after RSA decryption, the last 32 bytes of the decrypted data serve as the ASR_XXTEA key, with only the first 16 bytes ("07edb9cc7a134318") being used in practice.

```
0000000  00 17 84 D5 11 7B F2 6D 30 EE A1 2C 92 BA 23 DB   .....{.m0..,..#.
0000010  24 9D 5E 8D F6 F0 74 12 6F 06 2B B9 E1 01 6F AD   $.^...t.o.+...o.
0000020  9D 3D D8 59 45 83 47 6D 33 FB 0C 37 76 6A 97 37   .=.YE.Gm3..7vj.7
0000030  32 AB 9D 97 C2 69 D8 02 ED 58 B3 44 19 76 A5 8F   2....i...X.D.v..
0000040  65 08 1F 32 1E C0 B3 C3 88 75 A0 F5 11 9A 4E E9   e..2.....u....N.
0000050  20 7E 44 71 E4 4D 57 DE 94 27 19 6E 56 4A 0A 3E    .~Dq.MW..'.nVJ.>
0000060  20 40 90 4C 75 1B 5A 33 EC 43 B7 59 7A E2 63 9D    .@.Lu.Z3.C.Yz.c.
0000070  2E BC 26 9C 32 B3 2E FF 73 1A 31 F9 E9 A2 B4 A0   ..&.2...s.1.....
0000080  FA BA 40 43 24 10 89 7D EF 9C 4C 62 3C A0 6B AD   ..@C$..}..Lb<.k.
0000090  04 2F F3 7F 39 0B EA 84 80 42 9C 20 51 3F A6 AA   ./.9....B..Q?..
00000A0  76 19 F6 DB AA CD AA 22 17 F1 CB 2A 2D 02 68 97   v......"...*-.h.
00000B0  09 09 35 A0 46 4A CE 27 78 AB D9 EE 72 82 50 A2   ..5.FJ.'x...r.P.
00000C0  D1 CF 14 41 64 DE 24 0A A8 2D 44 1E D3 61 05 33   ...Ad.$..-D..a.3
00000D0  D1 A3 F2 06 C3 CE 28 BE 5A 62 A0 8A 7D A8 34 FD   ......(.Zb..}.4.
00000E0  D5 C1 A0 B9 24 BE 22 59 79 BD 04 25 7B 4C 2C 71   ....$."Yy..%{L,q
00000F0  27 1F CC 42 71 73 D1 D9 D5 B2 15 19 4E 89 95 50   '..Bqs......N..P
0000100  8C 52 87 DD 3D 5E 47 96 8E 37 FD 39 2A A1 F5 BC   .R..=^G..7.9*...

0000000  00 02 61 DA DD 4C CF A9 9F 67 B3 D3 D9 65 C2 8D   ..a..L...g...e..
0000010  C3 63 59 E0 DC DB 0C 0E 81 BE 87 B5 08 18 4C E9   .cY...........L.
0000020  F7 AD C3 D4 78 13 FC 97 79 AF E9 52 15 AC 5F 57   ....x...y..R._W
0000030  0F B7 37 6A 92 C1 77 92 FE 7D 47 86 15 92 6F 0C   ..7j..w..}G...o.
0000040  BE 33 DF B6 C4 DB 4D 3D 0A 36 0F 9D 61 6D F3 6F   .3....M=.6..am.o
0000050  A3 2A D8 B4 6A CF 46 69 4C 8C 6E DF 9D DC EA 5C   .*..j.FiL.n....\
0000060  8E 49 12 52 A3 DD 8F AD 92 1D 4A F3 89 BD 62 2C   .I.R......J...b,
0000070  66 BA DF D0 89 A4 B8 54 30 26 B3 4C 02 9D A7 90   f......T0&.L....
0000080  66 38 61 09 94 EF 35 27 8B FE 1A 14 BB 7B BE 21   f8a...5'.....{.!
0000090  B4 1D F0 BC C0 28 11 6F 4D C3 BB CE DF E1 DD C4   .....(.oM.......
00000A0  99 3E CD 2D 2E 81 53 38 7F 6C 4C 3A 67 89 DA 1B   .>.-..S8 lL:g...
00000B0  A6 CA D7 66 72 67 55 BE A9 10 8C 08 70 69 CC 09   ...frgU.....pi..
00000C0  A7 19 36 54 19 88 8B 98 74 56 51 5A DF 2B F3 85   ..6T....tVQZ.+..
00000D0  75 4A 6A E6 30 BE A4 58 4D B0 60 BD 98 AB 45 00   uJj.0..XM.`...E.
00000E0  30 37 65 64 62 39 63 63 37 61 31 33 34 33 31 38   07edb9cc7a134318
00000F0  62 38 65 66 61 32 63 34 62 38 36 36 61 31 36 62   b8efa2c4b866a16b
```

Figure 26: The last 32 bytes of the decrypted data serve as the ASR_XXTEA key.

Decrypting the XXTEA ciphertext in the ts01 payload with this key yields the next malware stage.

D:\dev_turing\python\void\vb2025\payload\ts01.dat.decrypt\

名称	大小	修改时间
z.apk	1 851 421	2025-04-24 10:20
vo1d	149 956	2025-06-06 17:14
install.sh	1 134	2025-04-23 17:18
cv	26 924	2025-06-06 17:20

Figure 27: The next malware stage.

It's worth emphasizing that algorithm identification rarely happens immediately during reverse engineering. For Vo1d, we initially leveraged flare_emu's emulation capabilities to perform batch black-box decryption before ultimately identifying the RSA and ASR_XXTEA cryptographic implementations.

The most significant drawback of this method is its computational inefficiency – decrypting ts01's RSA ciphertext yielded the same result ('07edb9cc7a134318'), but required approximately 1,500 seconds to complete.

```
import flare_emu

eh=flare_emu.EmuHelper()
eh.apiHooks['__memset_chk']=eh.apiHooks['memset']
eh.apiHooks['j_memcpy']=eh.apiHooks['memcpy']
eh.apiHooks['mypatch']=eh.apiHooks['bzero']
import hexdump
import time
start_time=time.time()
out=eh.allocEmuMem(0x100)
with open("ts01.dat", "rb") as f:
    buf=f.read()[0x5:0x105]

hexdump.hexdump(buf)
rsa_ciphertxt=eh.loadBytes(buf)
bufsize=eh.loadBytes(b'\x00\x01')
eh.emulateRange(startAddr=0x000021C0,endAddr=0x000026DC,registers={'R0':rsa_ciphertxt,'R1':bufsize,'R2':out},skipCalls=False)
print(eh.getEmuState())
addr=eh.getRegVal('R0')
rsa_plaintxt=eh.getEmuBytes(addr,0x100)

end_time=time.time()
print(f"time {end_time-start_time} seconds")
hexdump.hexdump(rsa_plaintxt)
```

```
time 1480.8894879817963 seconds
00000000: 30 37 65 64 62 39 63 63  37 61 31 33 34 33 31 38  07edb9cc7a134318
00000010: 62 38 65 66 61 32 63 34  62 38 36 36 61 31 36 62  b8efa2c4b866a16b
```

Figure 28: Black-box decryption using flare_emu.

0x5: Downloader network protocol

Downloader samples support two commands, 0x10 and 0x11, which correspond to the functions of requesting redundant downloader servers and requesting the payload, respectively. The network packet format is `length:cmd:body`, where the length field is four bytes long and represents the combined length of the cmd and body fields; the cmd field is one byte, and the body field's is length - 1.

- 0x10 command: the body is encrypted using standard XXTEA.

```
00000000  00 00 00 01 10                                     .....
    00000000  00 00 00 19 10 f6 2d 18  29 f6 6d a3 81 23 e3 69   ......-. ).m..#.i
    00000010  47 63 5d e7 94 4a f0 81  22 6c 17 d1 b2           Gc]..J.. "l...
```

Figure 29: 0x10 command.

- 0x11 command: the body employs dual-layer payload protection.

```
00000000  00 00 00 10 11 74 73 30  31 00 00 00 00 00 00 00   .....ts0 1.......
00000010  00 00 00 00                                        ....
    00000000  00 17 84 d5 11 7b f2 6d  30 ee a1 2c 92 ba 23 db   .....{.m 0..,..#.
    00000010  24 9d 5e 8d f6 f0 74 12  6f 06 2b b9 e1 01 6f ad   $.^...t. o.+...o.
    00000020  9d 3d d8 59 45 83 47 6d  33 fb 0c 37 76 6a 97 37   .=.YE.Gm 3..7vj.7
```

Figure 30: 0x11 command.

Different downloader variants request distinct payloads, enabling brute-force attacks against downloader servers to harvest additional unknown payloads. We attempted alphanumeric combinations with a maximum length of five characters, successfully capturing 136 different payloads.

Some of the payloads exposed Vo1d's true motives: ad fraud, click fraud, and building residential proxy networks.

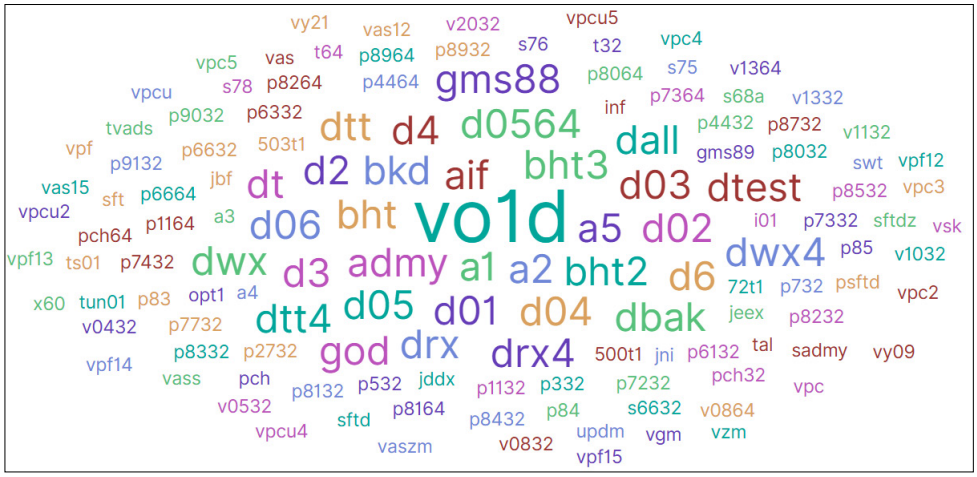

Figure 31: Vo1d payloads.

0x6: C2 network protocol

The process for the bot to obtain the real C2 address is straightforward: it first connects to the `Redirector C2` and sends a fixed four-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;                                  DCB 0xDD, 0xCC, 0xBB, 0xAA
bzero((int)&v14, 0x100u);
if ( (unsigned int)rsa_dec(v13, &v12, &v14) < 0xD )
  return 0;
if ( v14 != 'yakO' )
  return 0;
```

Figure 32: Obtaining the real C2 address.

The RSA key used here matches the one described in the 'Dual-layer payload protection' section. As demonstrated by actual traffic analysis, the decrypted true C2 server was revealed to be 52.14.24.94:80.

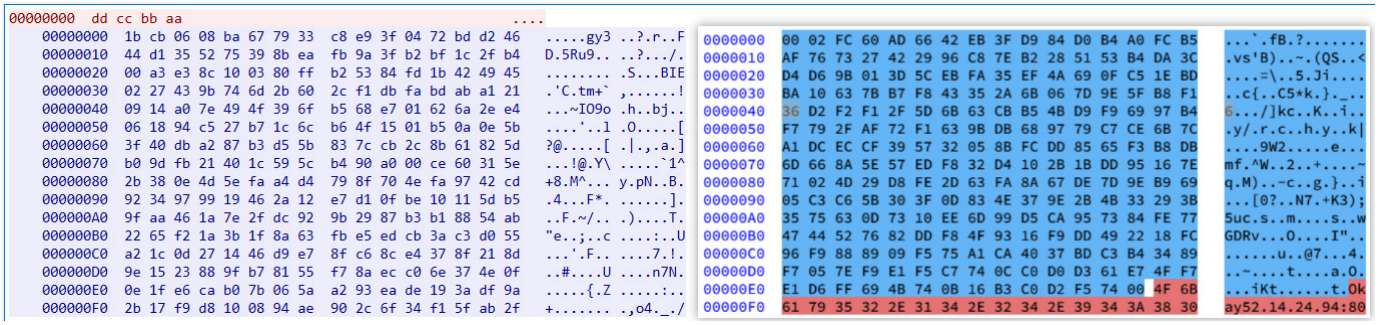

Figure 33: The decrypted true C2 server is revealed as 52.14.24.94:80.

Next, the bot reports device information to the real C2 server and awaits commands, with all communication encrypted via RSA. This time, Vo1d employed a different RSA key, still in the N-e format, where the N value is 256 bytes (little-endian) , and the e value is a fixed constant of 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.

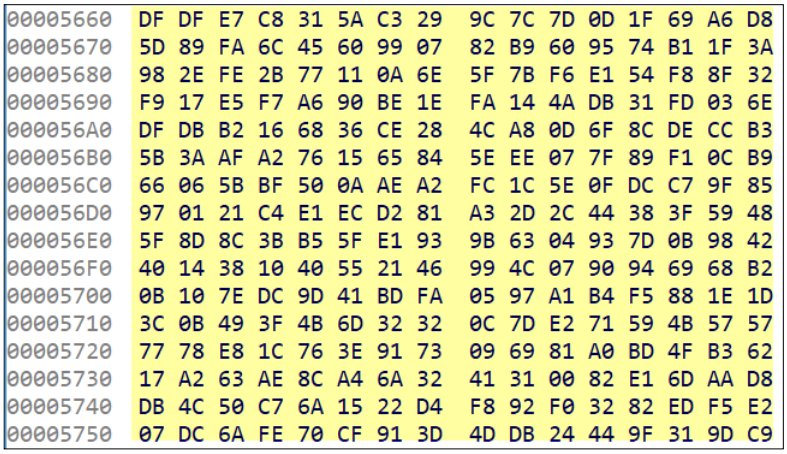

Figure 34: The RSA modulus N used for secure network communication.

BIGPANZI-VO1D-BADBOX THREAT NEXUS

- **Vo1d and Bigpanzi**

  Vo1d and Bigpanzi – both million-scale botnets targeting *Android* TV devices – share a highly unique string decryption algorithm. This consistency is unlikely to be coincidental. We suspect a deeper connection between them – potentially shared codebases, development resources, or even different branches of the same threat actor group.

- **Vo1d and BadBox**

  BadBox, a million-device botnet, targets both mobile and TV devices – unlike Vo1d, which specializes in TVs. *Human Security* found that BadBox 2.0 and Vo1d both leverage catmore88.com as part of their infrastructure.

  > 'Satori researchers believe BB2DOOR is associated with vo1d, a malware strain disclosed by Russian cybersecurity firm Dr. WEB in 2024. While vo1d and BB2DOOR share some similarities—key among them the use of libanl.so—vo1d's reach appears to be limited to CTV boxes, while BB2DOOR targeted several additional types of devices.' [5]

  Evidence collected by *XLab* further supports a connection between the two. The APK (MD5: 6e7c6f0feccd2e388af35975d1968e41) contains both Vo1d and BadBox components.

Figure 35: Vo1d and BadBox components.

However, their operational patterns differ significantly. For instance, Vo1d employs ASR_XXTEA encryption for payloads and uses custom algorithms to protect sensitive strings, while BadBox relies on RC4 for payload encryption and Base32 for string obfuscation. The former's security measures are noticeably more robust. If these were indeed from the same threat actor, we would theoretically expect consistent use of stronger encryption methods.

In fact, our findings show that Vo1d has distributed different payloads for various proxy providers. This leads us to hypothesize that Vo1d might simply be a delivery platform, with BadBox being just one of its clients.

CONCLUSION

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 behaviour.

- **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 behaviour perspective**: Many users harbour 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.

This is only the beginning – the battle against Vo1d, BadBox, Bigpanzi and similar threats will be protracted. This paper synthesizes known Vo1d botnet intelligence, offering a technical baseline for further security research. Stay patient, stay alert!

ACKNOWLEDGEMENTS

Credit is due to *Dr.Web* for their pioneering analysis of the Vo1d botnet, which significantly contributed to the security community's understanding.

REFERENCES

[1] Dr.Web. Void captures over a million Android TV boxes. 12 September 2024. https://news.drweb.com/show/?i=14900.

[2] Turing, A.; Acey9; Hao, W.; heziqian. Long Live The Vo1d Botnet: New Variant Hits 1.6 Million TV Globally. QI-ANXIN. 27 February 2025. https://blog.xlab.qianxin.com/long-live-the-vo1d_botnet/.

[3] Turing, A. Unveiling the dark side of set-top boxes: the Bigpanzi cybercrime syndicate. Virus Bulletin Conference 2024. https://www.virusbulletin.com/conference/vb2024/abstracts/unveiling-dark-side-set-top-boxes-bigpanzi-cybercrime-syndicate/.

[4] mandiant / flare-emu. https://github.com/mandiant/flare-emu.

[5] Satori Threat Intelligence and Research Team. Disruption: BADBOX 2.0 Targets Consumer Devices with Multiple Fraud Schemes. Human Security. 5 March 2025. https://www.humansecurity.com/learn/blog/satori-threat-intelligence-disruption-badbox-2-0/.