Detect SnappyClient C&C Traffic Using PacketSmith + Yara-X Detection Module

By Netomize Official Blog

Published: 2026-03-23 · Archived: 2026-04-05 17:04:41 UTC

Introduction

Zscaler published a blog post about a new malware called SnappyClient, written in the C++ programming language. The malware communicates with its C&C server using a custom binary protocol. The traffic is encrypted with ChaCha20-Poly1305 using a key and nonce received from the server, which are exchanged and validated before any control commands are sent or received. I've selected this sample because the C&C traffic structure is almost unfilterable, even with a traditional IDS/IPS, without incurring a high rate of false positives or significant performance impact.

The variant with the following information (in the table shown below) matches the C&C traffic dissected by Zscaler threat research blog. The sample is available on VT.

Attribute	Value
MD5	ec8258adfbf4ba5b9e8a06d75c5634cc
SHA-1	feb928a54be40ad4bbf245aaae6968f83b4937f5
SHA-256	eb523f6b0f306ce9fb68adeadac41d2c25b720075f03c75bd3611584dee28cf9
File size	3053296 bytes
File Type	Win32 EXE

VT's sandbox has a full capture of the C&C traffic, and I've made it available for download via Netomize's official repo (RFiles), SnappyClient Traffic VT (48.9 KB).

This blog post is about writing detection logic for the C&C traffic based on PacketSmith + Yara-X detection module, using custom pattern identifiers. For an in-depth analysis of how the malware works, please refer to the Zscaler blog post.

Once the malware successfully connects to the server, it receives a ChaCha20 key, a nonce, and a control session ID. Subsequently, SnappyClient sends an encrypted packet with the message header, followed by another packet containing the encrypted and compressed message. Our focus is on the structure of the second packet sent by the malware to the server, used for reporting back to the C&C server. This packet is transmitted via the TCP protocol over ports 3333/3334 to the C&C server at 151[.]242[.]122[.]227.

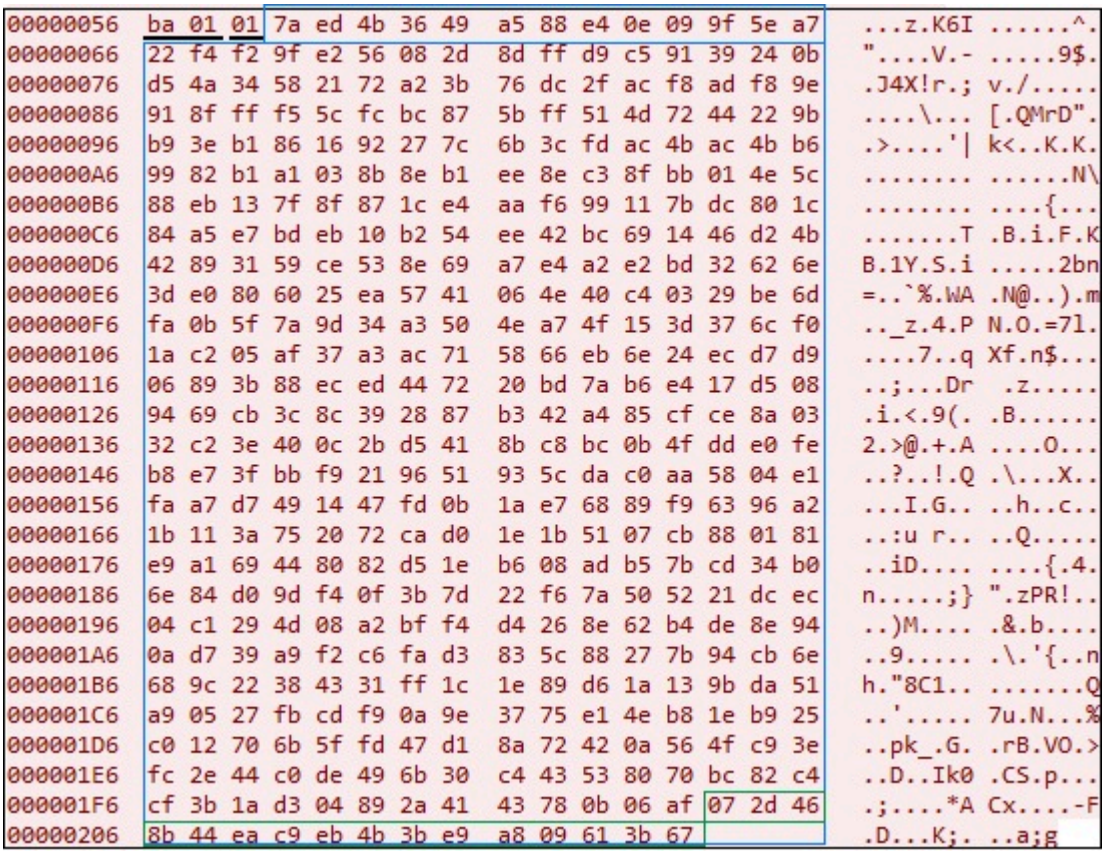

Figure 1 - SnappyClient Egress Cmd Packet

The structure of the packet in Figure 1 is as follows:

Offset	Length	Description
0x00	0x02	Packet length in little-endian, starting from offset 0x03
0x02	0x01	A flag indicating whether the packet contains output tag data or not (the data in the green box). 0x00 or 0x01
0x03	uint16(0x00)	Encrypted message (including output tag, depending on whether the flag at offset 0x02 is set or not)

Detection Logic

With the packet structure documented, we can develop detection logic to avoid false positives. The packet lacks unique content for anchoring, except for the byte at offset 0x02, which can be 0x00 or 0x01. This is insufficient for quick pattern matching in real-time traffic. Additionally, the encrypted message begins at offset 0x03 and continues to the packet's end, minus the 16-byte output tag size if the flag at offset 0x02 is set.

Understanding these limitations and indicators, we can develop a Yara-X + PacketSmith detection rule utilizing custom PaIDs (Pattern Identifiers) such as `tcp`, `ip`, and `flow`, along with the advanced math tools and operations built into Yara-X.

We could derive a rule similar to the following:

```
 import "math"

rule snappyclient_malware_encrypted_pkt
{
  meta:

    description = "TCP encrypted and compressed packet (client->server)"
    reference   = "https://www.zscaler.com/blogs/security-research/technical-analysis-snappyclient"
    filter      = "Frames (frames:)"
    sha1        = "feb928a54be40ad4bbf245aaae6968f83b4937f5"
    author      = "Netomize"
    date        = "22/03/2026"

condition:

   tcp.is_set and ip4.is_set and not ip4.in_ip6
   and
   tcp.data.size > 3
   and
   ip.dst.type == 1 // public
   and
   flow.to_server   // direction
   and
   math.in_range(port.src, 1024, 65535) // ephemeral ports
   and
   with buf_size = uint16(tcp.data.offset), buf_offset = tcp.data.offset:
         (
                  buf_size == (tcp.data.size - 3)
                  and
                  ( uint8(buf_offset + 2) == 0x00 or uint8(buf_offset + 2) == 0x01 )
                  and
                  math.entropy(buf_offset + 2, buf_size) >= 4
                  and
                  math.count(0x00, buf_offset + 2, buf_size) < 8
         )
}
```

The rule checks for the following atomic indicators

- First, we check that we're dealing with a TCP packet over IPv4 using the `tcp.is_set` and `ip4.is_set` PaIDs, respectively, while ensuring that this is not an encapsulated IPv4 in IPv6 packet (`not ip4.in_ip6`)
- The minimum size of the TCP payload is 3 bytes (`tcp.data.size > 3`)

- PacketSmith `ip` PaID can infer the type of the IP address using enum values (PUBLIC = 1, UNSPECIFIED = 2, PRIVATE = 3, CGNAT = 4, LOOPBACK = 5, LINK_LOCAL = 6, IETF_ASSIGNMENTS = 7, DOCUMENTATION = 8 and RELAY = 9, among others)
    - The destination IP address is public (`ip.dst.type == 1`)
- The directionality of the packet is to the server (`flow.to_server`)
- Using the "math" module `in_range` function, we check the range of the ephemeral source port such that it is between 1024 and 65535 (`math.in_range(port.src, 1024, 65535)`)
- We use the `with` statement to alias the TCP packet payload offset and size
    - `with buf_size = uint16(tcp.data.offset), buf_offset = tcp.data.offset:`
- The uint16 value at offset 0x00 in the TCP payload is equal to the payload's size, minus the first 3 bytes (the header)
    - `buf_size == (tcp.data.size - 3)`
- The byte at offset 0x02 could be either 0x00 or 0x01
    - `( uint8(buf_offset + 2) == 0x00 or uint8(buf_offset + 2) == 0x01 )`
- Since the data is encrypted and compressed, the entropy of the data has to be >= 4
    - `math.entropy(buf_offset + 2, buf_size) >= 4`
    - You can safely increase the value if you want to check for large encrypted messages
- Finally, using the "math" module and the function `count()` , we check the encrypted data in the packet for all occurrences of the byte 0x00 such that it is < 8
    - `math.count(0x00, buf_offset + 2, buf_size) < 8`

Running the above Yara-X rule through the linked pcap via PacketSmith and saving the result as XML, we get the file yara_dte_2026_03_23_17_11_08.xml (use MS Excel to view it) with all the detections:

```
PacketSmith.exe -i <infile_pcap> -D yara:xml -F frames: -O .
```

To check for potential false positives, we applied the detection rule to two large pcaps from our collection (511MB and 136MB) and found zero hits.

Conclusion

The PacketSmith + Yara-X approach demonstrates that even highly obfuscated, ChaCha20-Poly1305-encrypted C&C channels like SnappyClient can be detected reliably without deep payload inspection. By focusing on protocol-level fingerprints — the packet header, fixed-length framing, and the byte distribution—the detection module achieves a useful signal with potentially zero false positives and modest performance cost.

Author: Mohamad Mokbel

First release: March 23, 2026
