Active C2 Discovery Using Protocol Emulation Part1 (HYDSEVEN NetWire)

By Takahiro Haruyama

Published: 2019-11-20 · Archived: 2026-04-05 17:43:21 UTC

Malware C2 addresses can be an important IOC to detect known threats. In order to obtain C2 information, we first need malware samples which are then analyzed dynamically or statically. However the analysis task is often times not straightforward. Increasingly anti-analysis methods are implemented in malware or C2 information is extracted from secondary or tertiary websites.

VMware Carbon Black Threat Analysis Unit (TAU) analyzed HYDSEVEN NetWire samples then implemented a scanner to discover active C2 servers on the Internet by emulating the customized C2 protocol. In this blog post, the latest protocol and scanner implementation are detailed for researchers and practitioners.

For background context, NetWire is a cross-platform remote access tool or trojan (RAT) for Windows/Linux/Mac and sold by World Wired Labs. In the past different threat actor groups have utilized this tool for targeted attacks, Palo Alto published a C2 protocol decoder five years ago.

The NetWire C2 protocol is TCP-based. The RAT and C2 server initially exchange an "authentication packet" including information required for the packet payload encryption. By emulating the exchange, we are able to detect active NetWire C2 servers on the Internet.

HYDSEVEN, that was named by LAC, is a threat actor targeting virtual currency exchange companies for financial motive. This actor utilizes NetWire. However, as LAC pointed out in their report, HYDSEVEN customized the C2 protocol with different data format. Additionally TAU found a completely-different algorithm for the encryption key generation which was implemented in HYDSEVEN's NetWire. Therefore, if we obtain any correct response after sending the customized "authentication packet", the address of the respondent is a C2 server utilized by HYDSEVEN.

The following image is the flow of the key exchange by the customized “authentication packet”.

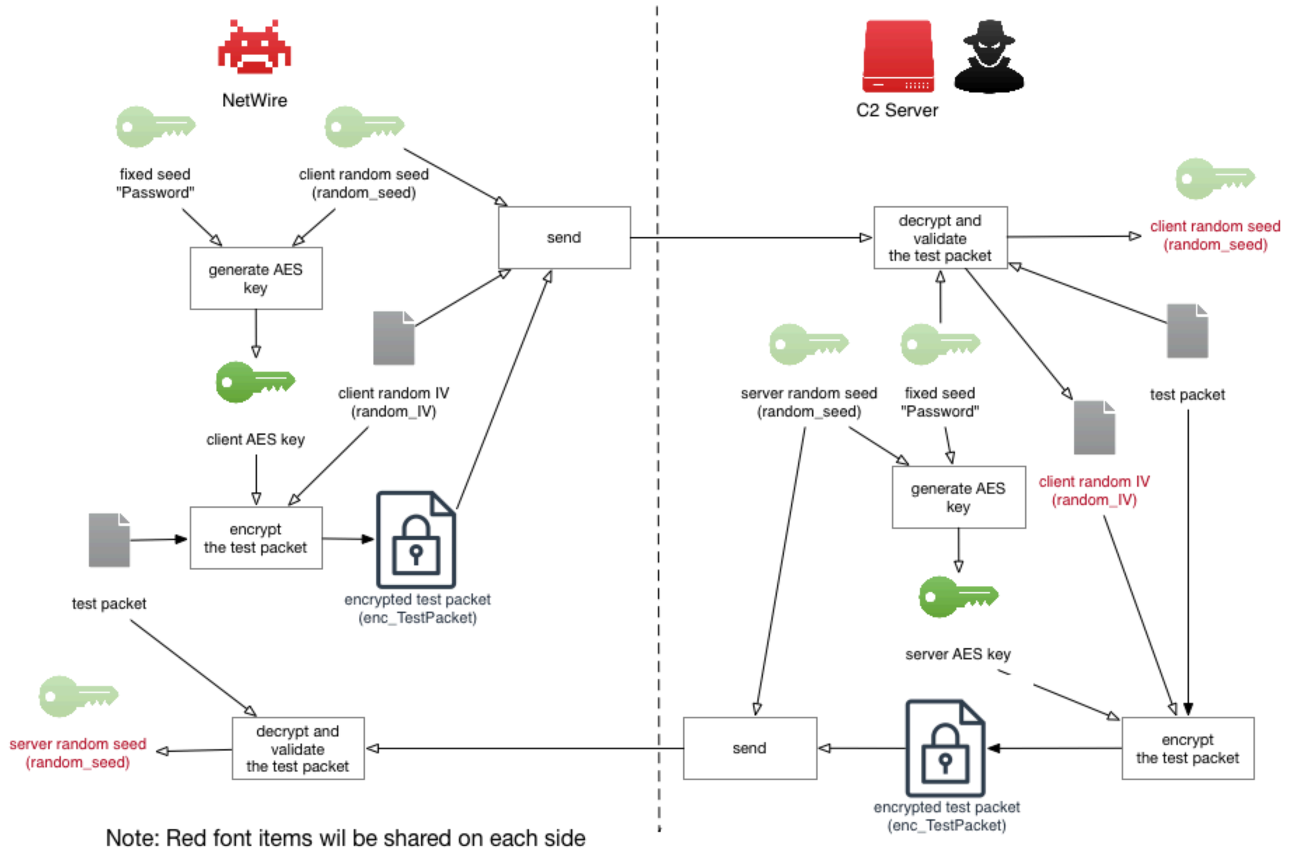

Figure 1: HYDSEVEN NetWire key exchange

The protocol format and encryption are detailed in the following sections.

Protocol Format

The customized “authentication packet” format is below. The packet is separated into the header and payload.

```
struct struc_pkt_header
{
char cmd_id; // client = 0x7F (0x03 ^ 0x7C), server = 0xE6 (0x5 ^ 0xE3)
int payload_len; // 0x40
char os_id; // client = 0xD8(win)/0xDB(linux)/0xDD(mac), server = 0
};
```

The header values are well-documented in the previously referenced LAC report. The command ID (cmd_id) 0x3 and 0x5 specify the packets are used for an initial key exchange. The ID is encoded/decoded by one byte xor on each side. The platform ID (os_id) shows the client OS environment.

```
struct struc_pkt_auth_payload
{
char random_seed[32]; // combined with fixed seed from config (HYDSEVEN uses “Password”)
char random_IV[16]; // only client IV used
```

char enc_TestPacket[16]; // encrypted from binary data “3b 0f db 9a 94 08 cd 6b 06 8c 93 c6 12 d8 b4 17”
};

There are three values in the payload of the “authentication packet.” The random_seed is used for the AES key generation, combined with the hard-coded seed (“Password”). The random_IV is IV data for the AES encryption, generated and shared by the client side. The pcap data on VirusTotal (SHA256: a2e449364b1bc148a19824984010485e2770a2f2e3098a7b59b557a59f735691) showed the random_IV from a C2 server was NULL.

```
00000000  7f 40 00 00 00 dd ee bc  b0 f1 d4 80 be ca 70 a2   .@...... ......p.
00000010  68 fb ac 7a 4e 68 f8 80  55 60 26 14 de e2 f4 a9   h..zNh.. U`&.....
00000020  99 c8 e4 8f f4 ad fc e4  fa f8 40 c8 8f 78 20 76
00000030  f4 74 78 85 b0 61 30 66  1f 60 59 79 d5 e7 b6 b0
00000040  82 35 44 9a 39 ff                                  .5D.9.
    00000000  e6 40 00 00 00 00 16 dc  62 17 32 33 e7 34 93 14   .@...... b.23.4..
    00000010  40 95 02 d2 13 45 de d6  06 8e 77 7a f6 c3 e7 de
    00000020  5f db 3a a9 da 05 00 00  00 00 00 00 00 00 00 00
    00000030  00 00 00 00 00 00 a3 61  08 67 08 7f 33 10 8e df   .......a .g..3...
    00000040  3c a7 d8 36 c5 95                                  <..6..
```

client packet

server packet

random_IV from server

Figure 2: NULL random_IV value from a C2 server

The enc_TestPacket data is utilized in the validation of the exchanged keys and encrypted from the following hard-coded binary data called TestPacket.

```
_TestPacket     db 3Bh
                db  0Fh
                db 0DBh ; Û
                db  9Ah
                db  94h
                db    8
                db 0CDh ; Í
                db  6Bh ; k
                db    6
                db  8Ch
                db  93h
                db 0C6h ; Æ
                db  12h
                db 0D8h ; Ø
                db 0B4h ; ´
                db  17h
```

Figure 3: TestPacket data to be encrypted

Payload Encryption

The HYDSEVEN NetWire's payload encryption algorithm is AES in OFB mode, that is the same as normal NetWire. However, the key generation algorithm is totally different. The SHA1 algorithm is utilized in the customized key generation.

```
def create_key ( password, seed ):
    ## seed is assumed to be hex
    flip = ''
    result = ''

    # This will flip the lower and upper order nibbles
    for i in password:
        i_bin = binascii.hexlify(i)
        tmp = i_bin[1] + i_bin[0]
        flip += tmp

    result += binascii.unhexlify(flip)

    for i in xrange(8, 32):
        tmp = i >> 5 | i * 8
        tmp = tmp & i
        result += chr(tmp)

    a1 = ord(result[len(password) >> 2]) ^ len(password)

    for i in xrange(0, 32):
        v4 = ord(result[i]) ^ ord(seed[i])
        v10 = a1 ^ v4
        v10 = v10 & 0xFF

        v11 = 4 * v10

        #only the low byte gets changed below
        v11 = ord( seed[i] ) ^ ( 4 * v10 )

        a1 = ~v11 & 0xFFF

        v4 = (i ^ (i + len(password))) | (v10 >> 5) | (8 * v10)
        v4 = v4 & 0xFF
        v4 = hex(v4)[2:].zfill(2)
        v4 = binascii.unhexlify(v4)

        pieces = list(result)
        pieces[i] = str(v4)
        result = "".join(pieces)

    return result
```

Figure 4: AES key generation algorithm used by normal NetWire (from Palo Alto protocol decoder code)

```
def create_key_custom(password, seed):
    off_pass = 0
    off_seed = 0
    mixed = []
    for i in range(0x20):
        mixed.append(password[off_pass])
        mixed.append(seed[off_seed])
        off_pass = (off_pass + 1) % len(password)
        off_seed = (off_seed + 1) & 0x1f

    off_mixed = 0
    for i in range(0x20):
        sha1_mixed = hashlib.sha1(''.join(mixed)).digest()
        for j in range(0x14):
            mixed[off_mixed] = sha1_mixed[j]
            off_mixed = (off_mixed + 1) & 0x3f

    return ''.join(mixed)[:0x20]
```

Figure 5: AES key generation algorithm used by HYDSEVEN-customized NetWire

Scanner Implementation

VMware Carbon Black Threat Analysis Unit (TAU) initially implemented the POC scanner then continued improving it for Internet scale.

POC Scanner

While it is possible to implement the scanner based on the static code analysis result, the implementation should be validated in another way. TAU utilized the pcap data described earlier for the validation.

The pcap data contains packets generated by both the client and server sides. The POC code took the client seed and IV from the client packet (auth_send.bin) and the server seed from the server packet (auth_recv.bin) as they are random data, then it generated the entire packets including the encrypted TestPacket (gen_send.bin and gen_recv.bin). As a result, the generated packets exactly matched the data in the pcap, so TAU concluded the POC scanner worked correctly.

```
$ hexdump -C auth_send.bin
00000000  7f 40 00 00 00 dd ee bc  b0 f1 d4 80 be ca 70 a2 |.@………..p.|
00000010  68 fb ac 7a 4e 68 f8 80  55 60 26 14 de e2 f4 a9 |h..zNh..U`&…..|
00000020  99 c8 e4 8f f4 ad fc e4  fa f8 40 c8 8f 78 20 76 |……….@..x v|
00000030  f4 74 78 85 b0 61 30 66  1f 60 59 79 d5 e7 b6 b0 |.tx..a0f.`Yy….|
00000040  82 35 44 9a 39 ff                               |.5D.9.|
00000046
$ hexdump -C auth_recv.bin
00000000  e6 40 00 00 00 00 16 dc  62 17 32 33 e7 34 93 14 |.@……b.23.4..|
00000010  40 95 02 d2 13 45 de d6  06 8e 77 7a f6 c3 e7 de |@….E….wz….|
00000020  5f db 3a a9 da 05 00 00  00 00 00 00 00 00 00 00 |_.:…………|
```

```
00000030  00 00 00 00 00 00 a3 61  08 67 08 7f 33 10 8e df |…….a.g..3…|
00000040  3c a7 d8 36 c5 95                               |<..6..|
00000046
```

```
$ python client.py
[*] gen_send.bin created
[*] 127.0.0.1:443,suspicious
[DEBUG] server key (0x20 bytes):
'XJ\xad+\xb2\xea\xbf\xc9\xa5\xfe\xd1\x81\xdb\x9f\xbd3\xe1\x12\xc4\xf8\x81\xc1\xd6,\xca\x00\xe2\x82&\x98z\x9e'
[+] 127.0.0.1:443,active
```

```
$ sudo python server.py
[*] got custom NetWire packet from ('127.0.0.1', 51155)
[*] client key: 'gA6%\x8f\xfe\x15#\n\x12\xb5\x1fg\xe2;\x086"Y\x7foe\x04\xcb\x89b/\xc6\xbb\xc6d\xb3' (32 bytes)
[+] the key and IV are valid. sending response…
[*] gen_recv.bin created
```

```
$ md5 *_send.bin
MD5 (auth_send.bin) = 50778a98ca957cf1ddb3d96f0b623133
MD5 (gen_send.bin) = 50778a98ca957cf1ddb3d96f0b623133
$ md5 *_recv.bin
MD5 (auth_recv.bin) = 19493425e15c770d971be676bce14aa2
MD5 (gen_recv.bin) = 19493425e15c770d971be676bce14aa2
```

Scanning the Internet

TAU initially focused on port 443 for the scanning as all analyzed samples utilized the port.

The initial version of the scanner sent the customized "authentication packet" after checking if port 443 was open on the host. However port scanning for all hosts on the Internet was too expensive for the scanner to finish in a realistic window of time. Thus the second version would only send the authentication packets to hosts discovered by the MASSCAN port scanner.

While the second version scanner had run for a few months, it was not effective as the port scanning practically didn't work. Specifically, the number of open hosts discovered by MASSCAN was much smaller than actual ones.

specified packet rate (packets per seconds)	discovered open hosts	completion time (hours)
180,000	8,283	6
100,000	13,435	11
60,000	24,916	18

Table 1: MASSCAN experiment result

As shown in the above table, the discovered open hosts changed according to the packet rates. Additionally, the search results by Censys and SHODAN were about 40 and 50 million. It was unknown why the majority of open host were missed with the previous tool.

The external security researcher Tadashi Kobayashi, who published the OllyDumpEx/OllyMigrate IDA plugins, advised the bottleneck was the broadband router performing NAT. Based off of his advice, TAU prepared a Linux VM directly-connecting to the Internet through PPPoE then run ZMap (MASSCAN didn't support the scanning option against non-Ethernet interfaces). Using this method, TAU obtained 52,128,684 hosts, with port 443 open, from the Internet.

Result

TAU scanned against the 52 million open hosts in mid-Nov this year by sending the customized authentication packet but at the time of the scan there was no responsive C2 servers. This initial scanning result indicates a series of possibilities:

- HYDSEVEN was not active at that time or stopped utilizing the customized NetWire
- The actor have changed the hard-coded data in the NetWire samples (e.g., fixed seed and TestPacket), as new variants are used in campaigns.
- The C2 is inaccessible to IP addresses not targeted by the actor

Wrap-up

TAU analyzed the HYDSEVEN NetWire C2 protocol then implemented the scanner to discover the active C2 servers on the Internet. No active servers were found by the scanning, indicating their inactivity or change of the TTPs.

TAU also learned several takeaways for a large-scaled scanning initiative as part of this research. This knowledge will be utilized during ongoing and future C2 research projects.

Acknowledgement

For this research, TAU appreciates Tadashi Kobayashi's insight and advice. Kobayashi provided tactics for different port scanning strategies and the implementation details to identify the issues in MASSCAN and ZMap.

Indicators of Compromise (IOCs)

Indicator	Type	Context
0499aa5c68c59d2d3a484d52d7f1afcc189722ae96dfdde2afd9e12c95085af4 b8b776ebe5cf30c6dc1547ed35a79f42	SHA256 MD5	HYDSEVEN NetWire for Windows shared by LAC

c7c3d70337336fc183135038ce5d0a4bb83ab6d9f4cc1ad5cf600295e6a41e1b 12def981952667740eb06ee91168e643	SHA256 MD5	HYDSEVEN NetWire for Windows shared by LAC
a981a5fbeff782330871fb8a106466cbe61280536c162b3e3c3cbf441265b437 cb75044f5941530d963df9a626c813ae	SHA256 MD5	HYDSEVEN NetWire for Linux shared by LAC
07a4e04ee8b4c8dc0f7507f56dc24db00537d4637afee43dbb9357d4d54f6ff4 de3a8b1e149312dac5b8584a33c3f3c6	SHA256 MD5	HYDSEVEN NetWire for Mac OS shared by LAC
41dfab4ade85a7ea2df6f726ea711b60ddac7aa29d77a6bc5654564dec46cef7 50d4f0da2e38874e417bd13b59f4c067	SHA256 MD5	HYDSEVEN NetWire for Mac OS
a2e449364b1bc148a19824984010485e2770a2f2e3098a7b59b557a59f735691 944b9c731cf3821f1392b40f82ea0947	SHA256 MD5	HYDSEVEN NetWire for Mac OS

Source: https://blogs.vmware.com/security/2019/11/active-c2-discovery-using-protocol-emulation-part1-hydseven-netwire.html