Malware Writeup

Research

I knew for this project I wanted to do malware analysis. My initial research consisted of finding a malware sample, which I did through github.com/ytisf/theZoo.

Analysis

This malware is a 32 bit statically linked elf binary meant to run on the linux architecture. I first did some static analysis on the binary, but found this somewhat limiting. Eventually I moved to debug the binary from a bootable disk.

I started my analysis process by looking at the list of functions that radare2 identified. Because the binary is statically linked, there were a lot of extra functions related to system calls, but overall the binary still wasn't too big. The most obvious function to investigate first was the main function.

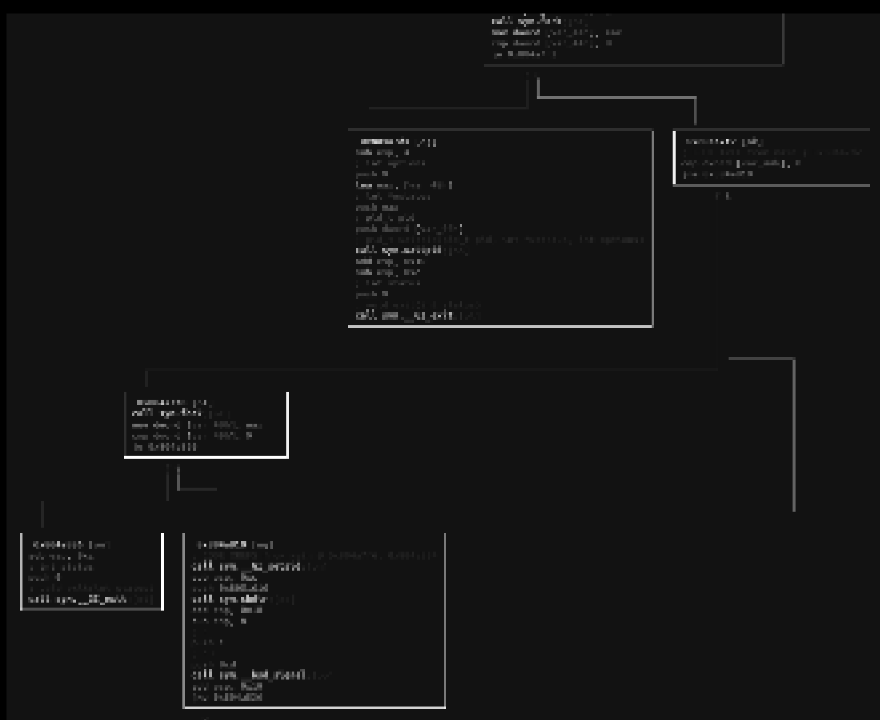
```
[0x0804a769]> afll~!_
address     size  nbbs edges    cc cost  min bound range max bound  calls locals args xref frame name
========== ==== ===== ===== ===== ==== ========== ===== ========== ===== ====== ==== ==== ===== ====
0x08048607   58     4     4     2   30 0x08048607    58 0x08048641     1      0    2    5    28 sym.printchar
0x08048641  215    20    27     9   92 0x08048641   215 0x08048718     3      4    4    4    44 sym.prints
0x08048718  293    18    26    10  115 0x08048718   293 0x0804883d     3      9    7    4    76 sym.printi
0x0804883d  584    33    48    17  248 0x0804883d   584 0x08048a85     7      7    3    1    60 sym.print
0x0804b16c   27     3     3     2   17 0x0804b16c    27 0x0804b187     0      0    2    5    12 sym.strcpy
0x08048be0  396    13    18     7  133 0x08048be0   396 0x08048d6c     3     14    3    1   220 sym.recvLine
0x0804b0a4   26     1     0     1   12 0x0804b0a4    26 0x0804b0be     1      0    1    2    28 sym.waitpid
0x0804d43c   59     3     3     2   29 0x0804d43c    59 0x0804d477     1      0    4    1    16 sym.wait4
0x0804e0f4   19     1     0     1   12 0x0804e0f4    19 0x0804e907     0      0    1    9     4 sym.strlen
0x0804d554   39     5     6     3   22 0x0804d554    39 0x0804d57b     0      0    3   18     8 sym.memcpy
0x0804e5dc   98     5     6     3   52 0x0804e5dc    98 0x0804e63e     4      1    2    2    44 sym.fgets
0x0804d2d8   50     3     3     2   25 0x0804d2d8    50 0x0804d302     1      0    2    1    12 sym.getrlimit
0x0804aefc   63     3     3     2   28 0x0804aefc    63 0x0804af3b     1      1    4    2    32 sym.ioctl
0x0804d294   38     3     3     2   21 0x0804d294    38 0x0804d2ba     1      0    0    1    12 sym.getgid
0x0804ccbc  381    38    36   156  125 0x0804ccbc   325 0x0804ce01     2      1    2    1    28 sym.sysconf
0x0804d21c   37     3     3     2   18 0x0804d21c    37 0x0804d241     1      2    0    1    44 sym.getdtablesize
0x0804c604   72     1     0     1   34 0x0804c604    72 0x0804c64c     4      3    0    1    60 sym.random
0x08049719 1355    36    50    16  504 0x08049719  1355 0x08049c64    27     36    7    2   204 sym.atcp
0x0804b738   51     1     0     1   19 0x0804b738    51 0x0804b763     1      4    4    2    44 sym.recv
0x0804b69c   43     1     0     1   17 0x0804b69c    43 0x0804b6c7     1      3    3    4    44 sym.connect
0x0804a634  309    11    14     5  114 0x0804a634   309 0x0804a769     9      3    0    1   556 sym.initConnection
0x08051158  176    19    29    12   86 0x08051158   176 0x08051208     0      1    3    1    12 sym.memrchr
0x0804d26c   38     3     3     2   21 0x0804d26c    38 0x0804d292     1      0    0    1    12 sym.geteuid
0x0804d57c   39     4     4     2   22 0x0804d57c    39 0x0804d5a3     0      0    3    2     8 sym.memmove
0x0804d32c   50     3     3     2   25 0x0804d32c    50 0x0804d35e     1      0    2    1    12 sym.munmap
0x0804c980   20     1     0     1   11 0x0804c980    20 0x0804c994     1      0    1   31    28 sym.atol
0x0804c994   26     1     0     1   12 0x0804c994    26 0x0804c9ae     1      0    1    1    28 sym.strtol
0x0804d244   38     3     3     2   21 0x0804d244    38 0x0804d26a     1      0    0    1    12 sym.getegid
0x0804aed4   38     3     3     2   21 0x0804aed4    38 0x0804aefa     1      0    0    4    12 sym.getpid
0x08048b3e   59     4     4     2   27 0x08048b3e    59 0x08048b79     1      1    2    3    44 sym.getHost
0x08048b79  103     4     4     2   45 0x08048b79   103 0x08048be0     1      2    2    2    36 sym.makeRandomStr
0x080482b7   48     1     0     1   24 0x080482b7    48 0x080482e7     2      1    1    4    44 sym.getRandomIP
0x0804b2a3    7     1     0     1    5 0x0804b2a3     7 0x0804b2aa     0      0    1    1     0 sym.ntohl
0x0804afbb   25     1     0     1   11 0x0804afbb    25 0x0804afd4     1      0    2    0    28 sym.creat
0x0804b818   43     1     0     1   17 0x0804b818    43 0x0804b843     1      3    3    7    44 sym.socket
0x0804b258   21     1     1     0    6 0x0804b258    21 0x0804b265     0      0    2    1     0 sym.bcopy
0x0804b764   51     1     0     1   19 0x0804b764    51 0x0804b797     1      4    4    3    44 sym.send
0x08048558  175    11    14     5   81 0x08048558   175 0x08048607     2      5    1    1    44 sym.trim
0x0804d304   38     3     3     2   21 0x0804d304    38 0x0804d32a     1      0    0    1    12 sym.getuid
0x0804ea00   29     1     0     1   14 0x0804ea00    29 0x0804ea25     1      1    1    3    92 sym.isatty
0x080584a0   33     1     0     1   18 0x080584a0    33 0x080584c9     1      0    2    1    28 sym.mempcpy
0x0804cb38  393    23    32    11  174 0x0804cb38   393 0x0804ccb9    12      9    1    1   460 sym.sleep
```

Several things happen in the first block of the main function. First, srandom() is called using the current time and the pid of the binary as seeds for it.

Then, init_rand() is called using the xor of the time and the pid of the binary once again. After this, a function is called that gets the IP address of the local computer.

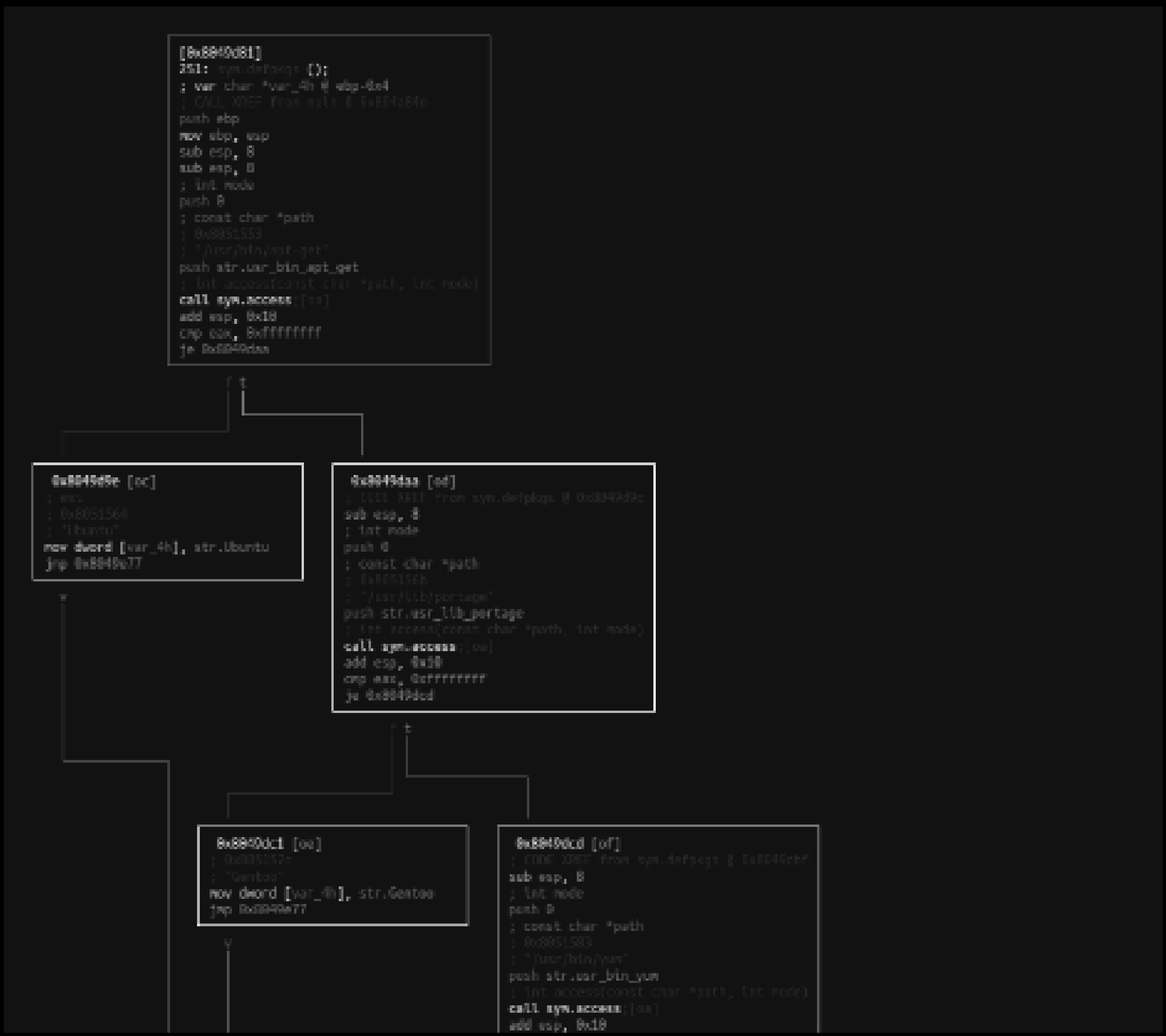
```
call sym.__GI_time;[oa]
add esp, 0x10
mov ebx, eax
; int getpid(void)
call sym.getpid;[ob]
xor eax, ebx
sub esp, 0xc
; int seed
push eax
; void srand(int seed)
call sym.srand;[oc]
add esp, 0x10
sub esp, 0xc
; time_t *timer
push 0
; time_t time(time_t *timer)
call sym.__GI_time;[oa]
add esp, 0x10
mov ebx, eax
; int getpid(void)
call sym.getpid;[ob]
xor eax, ebx
push eax
call sym.init_rand;[od]
add esp, 4
call sym.getOurIP;[oe]
call sym.fork;[of]
```

After this the binary forks twice I think as a bad anti-debugging trick. The parent processes will jump to a block that exits the binary and the final child process will continue on with the rest of the execution.



This child process then enters the main execution loop of the binary. The first step is to initialize some things, including changing to the / directory. Then, the malware checks if there is a server available at 209.141.48.138:666. If this connection fails the binary will sleep for 5 seconds and loop this check again. If it eventually succeeds, it moves on to the next phase of execution.

In the next block it gets some basic information about the system it is running on. In the first function call, it checks whether it has access to various directories, including apt-get, /usr/lib/portage, /usr/bin/yum, /usr/share/YaST2, /usr/etc/pkg and some others. Based on these directories it can guess what distribution is running on the machine because the directories correspond to the default package manager of each distribution.



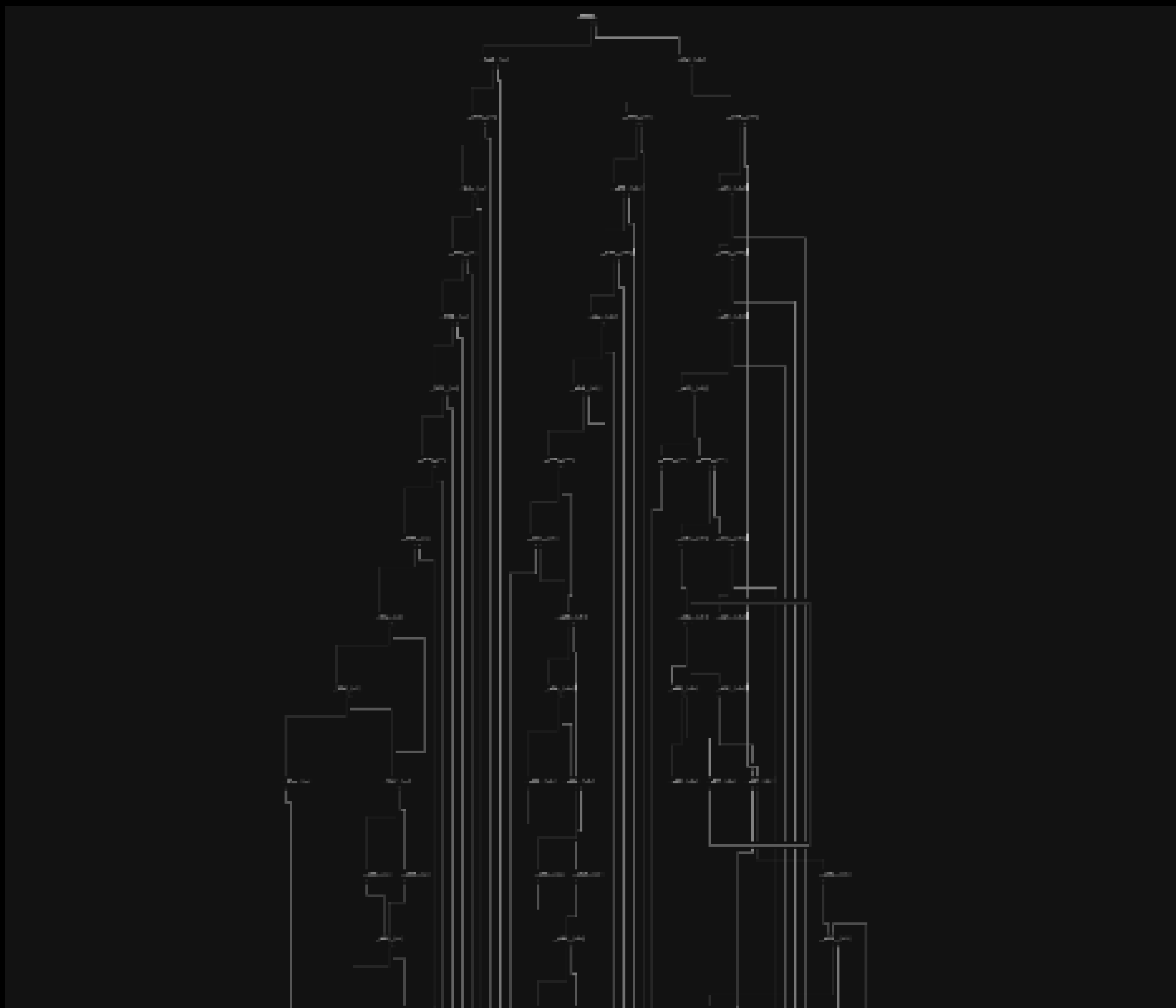
In the next two functions the binary returns that it is running on a linux machine and that it is using the x86_32 architecture. After this there is a call to sockprintf that passes the following string

```
\e[1;31mDemon\e[1;37m[\e[1;31mV5.0\e[1;37m]\e[1;31m-->\e[1;37m[\e[0;36m%s\e[1;37m]\e[1;31m-->\e[1;37m[\e[0;36m%s\e[1;37m]\e[1;31m-->\e[1;37m[\e[0;36m%s\e[1;37m]\e[1;31m-->\e[1;37m[\e[0;36m%s\e[1;37m]
```

What this is doing is passing information about the architecture, distro, etc to the central command server.

After this is the main loop that acts as the payload. At the beginning of this loop it attempts to receive a command from the main control server using a readline() function. This input is split up and converted to uppercase characters. These strings are then passed to the cncinput() function.

This function contains the core payload functionality. It uses the command information coming from the central server to decide which behavior to branch to. The most notable of these behaviors is a DOS attack. This malware does not self propogate. As you can see below, the control flow is complicated.



Analysis revealed the following command that could be used by the malicious server to launch various DDOS attacks.

Command syntax

1. **UDP <ip> <port> <time> <spoofit> <packetsize> <pollinterval>**
    a. UDP DDoS attack with random payload
    b. Time of attack in seconds
    c. If no port specified, random port is generated every <pollinterval> packets
    d. Random data does not change during attack
2. **TCP <ip> <port> <time> <spoofit> <flags> <packetsize> <pollint>**
    a. TCP DDoS attack

    b. If IP is id, TCP seq and TCP source port change randomly every <pollinterval> packets
    c. Payload does not change during attack but changes bot to bot
    d. Flags include all or syn,rst,fin,ack,psh
3. **STD <ip> <port> <time>**
    a. UDP attack with fixed payload
4. **STOMP <ip> <port> <time> <spoofit> <flags> <packetsize> <poolint>**
    a. STD attack followed by UDP attack followed by TCP attack
5. **CNC <ip> <port> <time>**
    a. Make TCP connection to server with <ip> on port <port> and closes after 1 second, repeating continually
6. STOP
    a. Stop all attacks. Each attack gets it’s own fork/process. Main process keeps track of all the forks and kills them all if this command is called.