Analyzing Gafgyt Malware

This is a writeup of analyzing the Gafgyt malware with radare2.

Bits	32
Arch	x86
OS	Linux
MD5	6668a65e995dd565043421cfdbd48384

Analysis

This binary is an ELF executable downloaded from VirusShare.com. We'll start by dumping the functions detected by radare2 in a table.

```
[0x08052661]> aflt
```

addr	size	name	nbbs	xref	calls	cc
0x080480c0	68	sym.__do_global_dtors_aux	8	5	0	4
0x08048110	87	sym.frame_dummy	6	5	1	4
0x08059af0	49	sym.__do_global_ctors_aux	4	3	0	3
0x08048366	66	sym.printchar	4	7	1	2
0x080483a8	218	sym.prints	20	17	1	9
0x08048482	319	sym.printi	18	16	2	10
0x08048849	722	sym.print	33	29	3	17
0x08053a44	54	sym._charpad	5	7	1	3
0x08053a7a	106	sym._fp_out_narrow	8	5	3	5
0x08054264	41	sym._promoted_size	4	4	0	3
0x08055578	38	sym.__malloc_largebin_index	3	3	0	2
0x08055d40	141	sym.__malloc_trim	7	7	1	5
0x08056c3b	3	sym.__pthread_return_0	1	62	0	1
0x08056c3e	1	sym.__pthread_return_void	1	58	0	1
0x08052e90	87	sym.__libc_fcntl	6	17	2	3
0x08053104	75	sym.__GI_open	5	8	1	3
0x08058d8c	134	sym.inet_pton4	15	14	1	10
0x08058fdc	273	sym.inet_ntop4	11	9	3	5
0x080595b9	724	sym.__read_etc_hosts_r	53	43	8	31
0x08057034	54	sym.__GI_execve	3	2	1	2
0x08053388	6	sym.__errno_location	1	72	0	1
0x08056e7f	218	sym.__libc_sigaction	10	8	2	6
0x08054b28	27	sym.strcpy	3	7	0	2
0x08052ee8	63	sym.__GI_fcntl64	3	2	1	2
0x080494b7	470	sym.recvLine	13	10	3	7
0x0805541c	42	sym.__GI_sigaddset	4	5	1	4
0x08055538	32	sym.__sigaddset	1	5	0	1
0x08056f90	50	sym.__socketcall	3	9	1	2
0x08057d10	35	sym.__GI_memchr	5	3	0	3
0x08054c6c	29	sym.__GI___glibc_strerror_r	1	1	1	1
0x08054c8c	182	sym.__xpg_strerror_r	13	8	4	7
0x08053300	26	sym.waitpid	1	3	1	1
0x080571d8	59	sym.wait4	3	3	1	2

Thankfully this binary is unstripped, which will greatly speed up the reversing process. We'll start by looking at the main function.

We can see there are several system calls in the first basic block. The most important of these is the call to prctl, which is often used to disguise malware processes as benign utilities.

```
0x080526df      e874240000      call sym.strncpy        ;[1] ; char *strncpy(char *dest, const char *src, size_t  n)
0x080526e4      bac4cf0508      mov edx, 0x805cfc4
0x080526e9      8b4604          mov eax, dword [esi + 4]
0x080526ec      8910            mov dword [eax], edx
0x080526ee      8b45b4          mov eax, dword [var_4ch]
0x080526f1      c7442410000000. mov dword [var_10h], 0      ; unsigned long v5
0x080526f9      c744240c000000. mov dword [var_ch], 0       ; unsigned long v4
0x08052701      c7442408000000. mov dword [var_8h], 0       ; unsigned long v3
0x08052709      89442404        mov dword [var_4h], eax     ; unsigned long v2
0x0805270d      c704240f000000. mov dword [esp], 0xf        ; [0xf:4]=-1 ; 15 ; int option
0x08052714      e87f0a0000      call sym.prctl          ;[2] ; int prctl(int option, unsigned long v2, unsigned long v3, un
0x08052719      c704240000000000. mov dword [esp], 0        ; time_t *timer
0x08052720      e8ab0b0000      call sym.__GI_time      ;[3] ; time_t time(time_t *timer)
0x08052725      89c3            mov ebx, eax
0x08052727      e808090000      call sym.getpid         ;[4] ; int getpid(void)
0x0805272c      31d8            xor eax, ebx
0x0805272e      890424          mov dword [esp], eax        ; int seed
0x08052731      e8163c0000      call sym.srand          ;[5] ; void srand(int seed)
0x08052736      c704240000000000. mov dword [esp], 0        ; time_t *timer
0x0805273d      e88e0b0000      call sym.__GI_time      ;[3] ; time_t time(time_t *timer)
0x08052742      89c3            mov ebx, eax
0x08052744      e8eb080000      call sym.getpid         ;[4] ; int getpid(void)
0x08052749      31d8            xor eax, ebx
0x0805274b      890424          mov dword [esp], eax
0x0805274e      e8395affff      call sym.init_rand      ;[6]
0x08052753      e8dcfcffff      call sym.getOurIP       ;[7]
0x08052758      e887080000      call sym.fork           ;[8]
```

We can then see some calls that are used to seed a random number generator, and a call is made to get the IP of the machine.

Next there are two calls made to fork, with the parent process exiting each time. I assume this is supposed to be an anti-debugging trick. I honestly don't know why malware authors think this is effective.

```
[0x08052661]> 0x8052661 # int main (int argc, char **argv, char **envp);
    call sym.fork;[ok]
    mov dword [var_48h], eax
    cmp dword [var_48h], 0
    je 0x805278c

 f t

 0x8052766 [oo]
 ; int options
 mov dword [var_8h], 0
 lea eax, [var_50h]
 ; int *wstatus
 mov dword [var_4h], eax
 mov eax, dword [var_48h]
 ; pid_t pid
 mov dword [esp], eax
 ; pid_t waitpid(pid_t pid, int *wstatus, int options)
 call sym.waitpid;[om]
 ; int status
 mov dword [esp], 0
 ; void exit(int status)
 call sym.__GI_exit;[on]

 0x805278c [op]
 ; CODE XREF from main @ 0x8052764
 cmp dword [var_48h], 0
 jne 0x80527ac

 f t

 0x8052792 [oq]
 call sym.fork;[ok]
 mov dword [var_44h], eax
 cmp dword [var_44h], 0
 je 0x80527ac

 f t
```

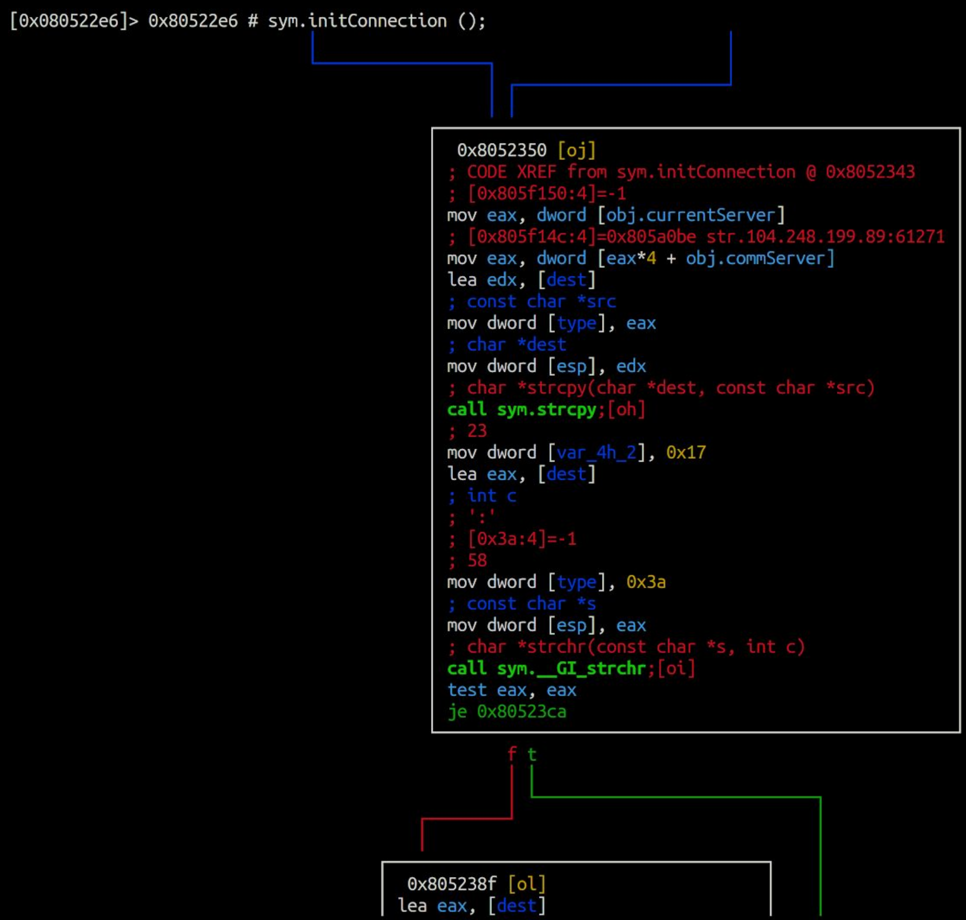
Next a function is called that initializes a connection to what I assume is a command and control server. The binary will return to this call if the connection is lost.

```
[0x08052661]> 0x8052661 # int main (int argc, char **argv, char **envp);

 0x80527d3 [ox]
; CODE XREFS from main @ 0x80527d1, 0x80527e8, 0x8052e72
call sym.initConnection;[ow]
test eax, eax
je 0x80527ea

 f t

 0x80527dc [oz]
; int s
mov dword [esp], 5
; int sleep(int s)
call sym.sleep;[oy]
jmp 0x80527d3

 0x80527ea [oAb]
; CODE XREF from main @ 0x80527da
; [0x806568c:4]=0
mov eax, dword [obj.ourIP]
; void *in
mov dword [esp], eax
; char *inet_ntoa(void *in)
call sym.__GI_inet_ntoa;[oa]
mov ebx, eax
call sym.getBuild;[ob]
; [0x805f400:4]=0
mov edx, dword [obj.mainCommSock]
mov dword [var_ch], ebx
mov dword [var_8h], eax
; [0x805cfde:4]=0x3b305b1b
mov dword [var_4h], str.e_0_32m_CONNECTED____s____s
mov dword [esp], edx
call sym.sockprintf;[oAa]
mov dword [var_40h], 0
mov dword [var_3ch], 0
jmp 0x8052e39
```

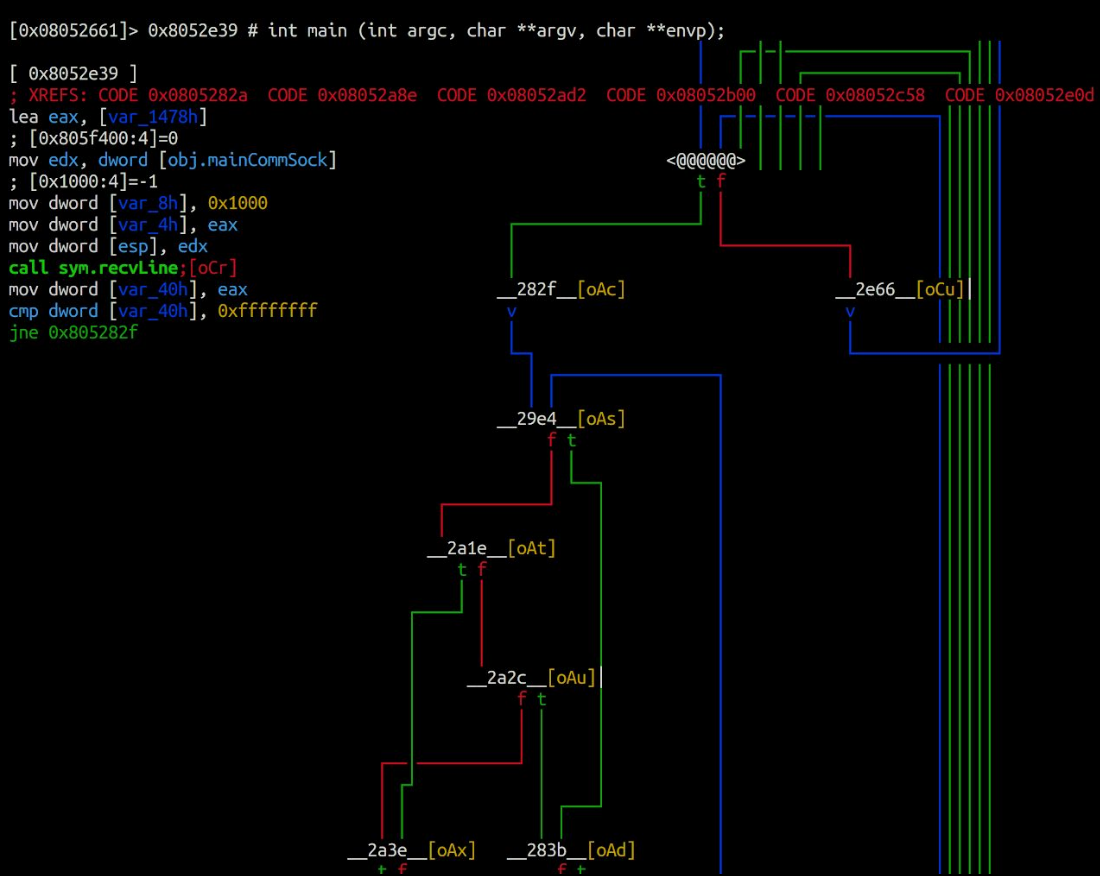
It’s worth checking what ip it attempts to connect to. The usual trick is to just examine the strings.

```
46  0x000125aa 0x0805a5aa 10  11   .rodata ascii   88.5.%d.%d
47  0x000125b5 0x0805a5b5 12  13   .rodata ascii   41.254.%d.%d
48  0x000125c2 0x0805a5c2 12  13   .rodata ascii   103.20.%d.%d
49  0x000125cf 0x0805a5cf 12  13   .rodata ascii   103.47.%d.%d
50  0x000125dc 0x0805a5dc 12  13   .rodata ascii   103.57.%d.%d
51  0x000125e9 0x0805a5e9 12  13   .rodata ascii   45.117.%d.%d
52  0x000125f6 0x0805a5f6 12  13   .rodata ascii   101.51.%d.%d
53  0x00012603 0x0805a603 12  13   .rodata ascii   137.59.%d.%d
54  0x00012610 0x0805a610 12  13   .rodata ascii   14.204.%d.%d
55  0x0001261d 0x0805a61d 11  12   .rodata ascii   27.50.%d.%d
56  0x00012629 0x0805a629 11  12   .rodata ascii   27.54.%d.%d
57  0x00012635 0x0805a635 11  12   .rodata ascii   27.98.%d.%d
58  0x00012641 0x0805a641 11  12   .rodata ascii   36.32.%d.%d
59  0x0001264d 0x0805a64d 12  13   .rodata ascii   36.248.%d.%d
60  0x0001265a 0x0805a65a 11  12   .rodata ascii   39.64.%d.%d
61  0x00012666 0x0805a666 12  13   .rodata ascii   43.253.%d.%d
62  0x00012673 0x0805a673 12  13   .rodata ascii   43.230.%d.%d
63  0x00012680 0x0805a680 12  13   .rodata ascii   163.53.%d.%d
64  0x0001268d 0x0805a68d 12  13   .rodata ascii   43.245.%d.%d
65  0x0001269a 0x0805a69a 12  13   .rodata ascii   123.25.%d.%d
66  0x000126a7 0x0805a6a7 12  13   .rodata ascii   103.54.%d.%d
67  0x000126b4 0x0805a6b4 12  13   .rodata ascii   27.255.%d.%d
68  0x000126c1 0x0805a6c1 13  14   .rodata ascii   103.204.%d.%d
```

However it appears there are a large number of extraneous ip’s (more than shown above). A better strategy is to examine the function directly.



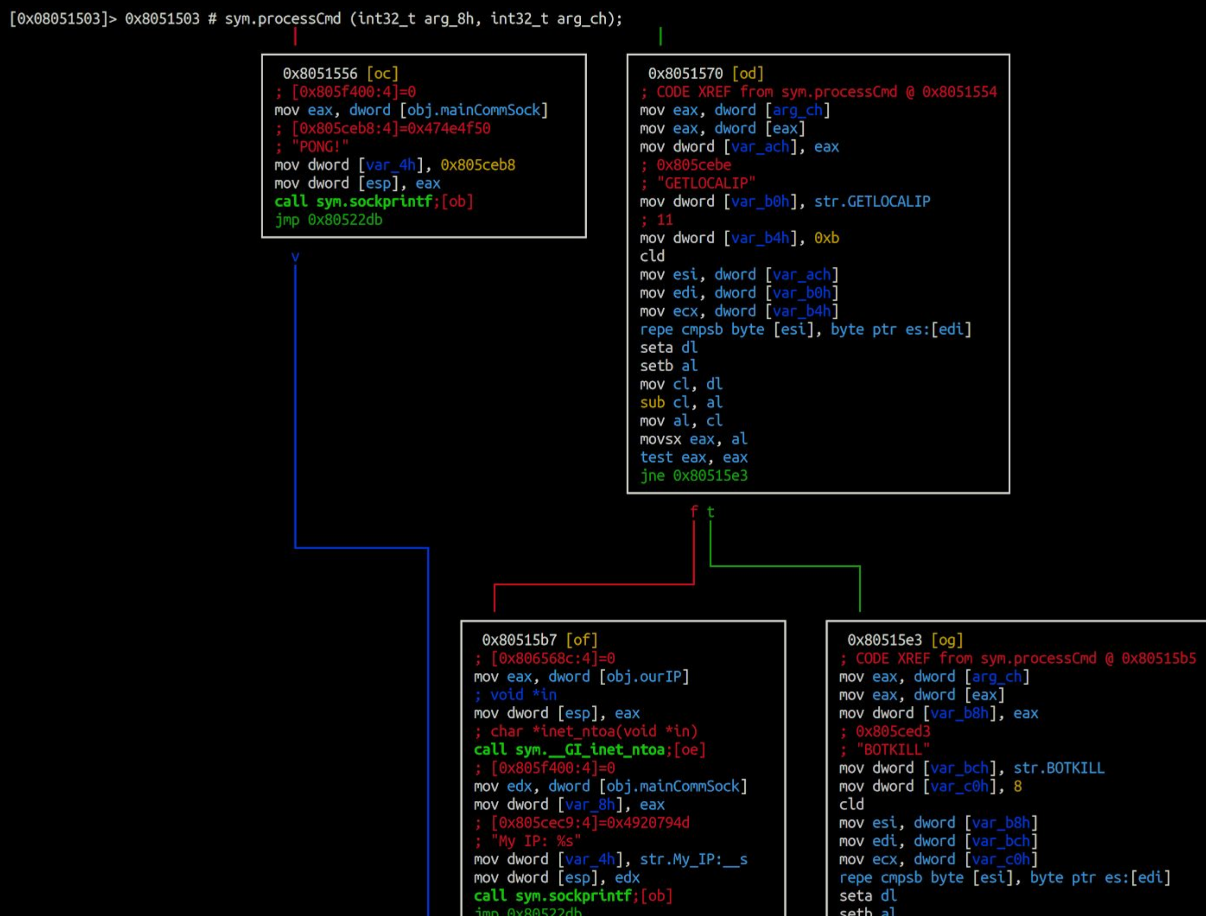
Here we can see that it connects to 104.248.199.89 on port 61271. After this block the binary enters the main loop where it receives commands from the server, parses them, and deploys various payloads.



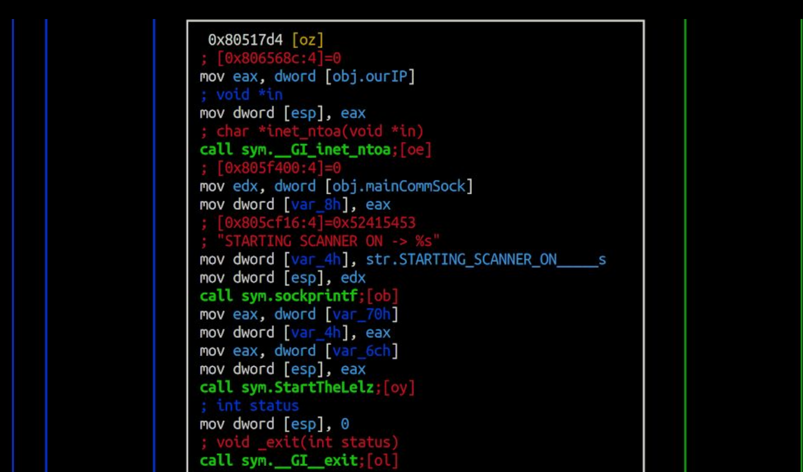
The control graph here is large and relatively complicated so I'll summarize the results of my analysis below. If the recvline() call fails, it will print "BYE MISTER HITTA" and return to the initconnection() call. If recvline() succeeds, it will enter a branching structure that parses the data for various commands.

The malware supports several DDOS style attacks. This includes a SYN flood attack with the "PING DUP" command. The "PING PONG" command causes the malware to return to the recvline().

There is more information in the processCmd() function.

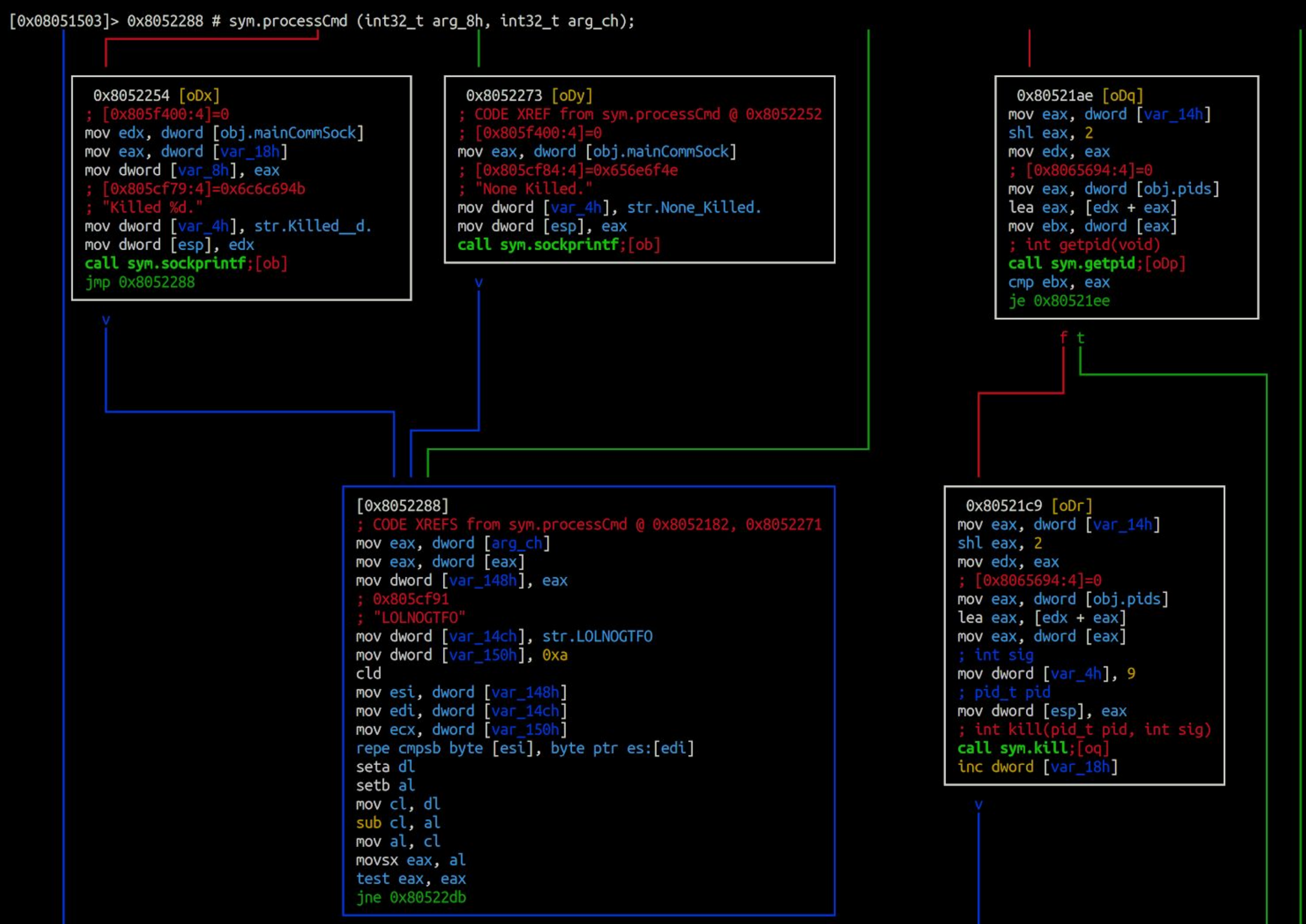


First this function can respond to the command server with the "PONG!" string. It will also respond to the with its IP address if it receives the "GETLOCALIP" command. This function also includes a scanner that can be triggered with the "SCANNER" command, and killed with the "SCANNER OFF" command.



Further analysis of this function showed that there are both TCP and UDP scanning options, and more DDOS options, including HTTP flooding. There is also a phone scanner that can be turned on and off. Attacks can be killed with the “KILLATTK” command.

At the end is a code block that kills the application and all associated processes.



EOF