[Part 1] - Analysing the new Linux/AES.DDoS IoT malware

Nov 19, 2017

As the title suggests this is a bot which is spread by brute forcing SSH daemons and exploiting IoT devices using an array of exploits — this malware is mainly distributed by a Chinese actor who is familiar with C++ and C constructs, however the knowledge of C++ by this threat actor only extends to using the `std::string` class in C++. This bot was being distributed a few years ago just for x86_64 targets, this has changed along with some key fundementals of the bot. It's started to target embedded systems, which is why I thought I would cover it again. Linux/AES.DDoS is programmed in C++, we can see this due to the fact that all of the symbols are exported and C++ constructs are used.

We are going to be using:

- gdb-peda: https://github.com/longld/peda
- BinaryNinja: https://binary.ninja/
- ltrace: https://linux.die.net/man/1/ltrace
- radare2: http://rada.re/r/

A look at the file..

If we run 'file' on the executable we get:

```
ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
statically linked, for GNU/Linux 2.6.14, not stripped.
```

So, we're working with an ELF (which is the COFF for UNIX systems) and it's 32 bit. It's architecture is of type ARM and all of the libraries its using are statically linked — this is normal behaviour of an IoT bot to have its libc linked as many systems will have incomplete or sometimes even broken libraries. So, instead of dynamically linking the executable, they are statically linked. Since its not stripped, this means that the analysis will be a lot easier as we have

meaningful names in relation to objects in the executable. For some strange reason, the executable was compiled on a 12 year old version of the Linux kernel — this could indicate to us that the malware was compiled on an IoT device or just has an extremely old computer.

MD5: `125679536fd4dd82106482ea1b4c1726`

SHA1: `6caf6a6cf1bc03a038e878431db54c2127bfc2c1`

A quick rundown on ARM

ARM is 32 bit by design, so all of our registers are 32 bit wide. In ARM, the standard calling convention is to place the arguments into `r0`-`r3`. That's only four registers though, if there are more than three arguments then we place the rest of the arguments on the stack. We have 15 registers to play with though, they all serve a special purpose.

- r0, r1, r2, r3 are used for passing arguments, r0 usually also holds the return value of routines.
- r4, r5, r6, r7, r8, r9 are used internally within routines to store variables.
- r10 holds the limit for the current stack.
- r11 holds the stack frame pointer.
- r12 can also be used as a variable within routines, however there is no guarantee that this register will remain unchanged by the caller.
- r13 holds the stack pointer (SP).
- r14 is the link register, which points back to the caller.
- r15 holds the program counter.

Nice, so now you're an ARM expert we can continue.. ;)

Dive into main(..)..

As soon as the malware boots from the original entry-point `main` which is at `0x13DEC`. We use `rabin2` to find the original entry point by doing `rabin2 -s kfts | grep "type=FUNC name=main"`. Which gives us the following output:

```
vaddr=0x00013dec paddr=0x0000bdec ord=5366 fwd=NONE sz=688
bind=GLOBAL type=FUNC name=main
```

Nice! The executable isn't packed. The main method then branches to function named `get_executable_name` which reads the symlink `/proc/self/exe` via `readlink(..)`. When reading this symlink internally from our process it will return the location from that our executable is running from. We can see from the disassembly that we create a type of `std::string` and copy from the char array containing the path of the current executable. This is then used and passed into the function used for persistence.

We then either check if we're running or add to startup. In the `check_running` procedure we do a call to `ps -e` then sleep for two seconds; we then get the output from the command and check to see if the current executable name exists in the output. If it does, we continue to branch to `exit` with an exit code of 0 (which is placed into r0) which will effectively shut-down the process. If not, we then go onto persistence. So, if we're already running, we effectively exit the process.

Persistence

Persistence is achieved by the malware by adding to `/etc/rc.local` and the `/etc/init.d/boot.local` files (in the `auto_boot` function); however before it overwrites this file it first checks to see if it has already done so. The `/etc/rc.local` will execute certain commands after all of the systems' services have started. The way that this is achieved is somewhat amateur as the malware constructs a shell command and then uses the `system` function

to execute them (which in theory just calls the exec and hangs for a return code from the callee).

A string is formatted and the `sed` program is called which writes to the file in question (there are several string operations, such as `sed -i -e '2 i%s/%s'` `/etc/rc.local` is formatted for example). This is then passed to `system`, as described above. The buffer used in all of these formatting operations is at the virtual address `0x9F48` and has a size of 300 bytes. Technically, since the input is un-sanitized we could manipulate the path that the malware resides in and utilise a format string exploit. We could therefore manipulate the stack; read local variables, overwrite addresses etc. This is the only persistence method used by the bot.

Information harvesting

The process then forks itself and breaks away from its parent by calling `setsid(..)`. All of the file descriptors are also closed which are inherited from the parent (0-3). A thread is then created to call the `SendInfo` function which collects information such as the number of CPUs in the system; the network speed; the amount of load on the system CPUs; the local address of the network adapter.

This routine then calls the subroutine `get_occupy`. We can see that we calculate the load average by iterating over all the CPUs in the system. We can see that `r3` is being used for the counter for this loop, then the `blt` instruction is executed which branches if the first operand is less than the second. In x86, this is the equivalent of `jle`. Please note in earlier versions of this malware a thread used to be created to `backdoorA`, however not anymore.

The way the malware gets information regarding the network adapter is reading the `/proc/net/dev` file. It then seeks to the start of the file; and parses it to get the local IP address from the default adapter.

What I found strange about this sample of malware is that it created statistics which had no real meaning, for example it would create a random value and use it as the network speed. In the subroutine `fake_net_speed` we can see `srandom` is seeded with `time` — we pass the first parameter ‘0’

into `time` as we have no structure to fill (usually, a pointer to `time_t` would be passed into this function). We then move the result of `time` into `r3`; then back into `r0` to be a parameter for `srandom` — this is most likely the compiler being strange for some reason or another.

The value generated by this call is then used in a `sprintf` statement to create a string which represents the network speed in mega bytes per second (apparently). This is super weird, the malware is creating a fake network speed for some reason.. the only conclusion I can come to regarding this is that someone has hired a programmer and they have failed to implement this feature — so they're faking it to their client/boss etc.

These values are then sent within this subroutine to the main C2..

Communication Initialisation

After all of these operations we finally come to the main core of the bot; the part where it connects to the C2 and receives commands. The procedure `ConnectServer` (which is at `0xCA1C`) is called from the main body of the function, this then branches to `ServerConnectCli` (`0xB5BC`) which returns a socket to the C2 server. The global variable `MainSocket` is then set to this value. Diving into `ServerConnectCli`..

`ServerConnectCli` starts by creating a socket of protocol type TCP, if this operation is not successful then the assembly branches off to a subroutine at `0xB654` which displays the error by calling the `perror(..)` function to display a human readable error.

For those of you familiar with C, this is like doing:

```
r0 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

The port that we are going to connect to is partly obfuscated by the author. The original number is loaded into the register `r3`. We then shift this value by 16 bytes (`0x10`) to the left (we can see this being done in the `lsl` instruction. We then shift is again to the right by 16 bytes. Then again, this may of have been put in there by the compiler. I've put the assembly down below so you can see clearly whats happening.

```
mov r3, r0

strh r3, [r3, #0x104]

lsl r3, r3, #0x10 // shift r3 RIGHT by 0x10

lrl r3, r3, #0x10 // shift r3 LEFT by 0x10

mov r0, r3

bl htons
```

Could look at it this way in pseudocode..

```
r3 = ((r3 << 0x10) >> 0x10)
```

The bot has a symbol named `AnalysisAddress` at at `0xB1B8`, at first sight one may think this is to divert the attention of researchers — but all this subroutine does is setup a `hostent` struct (http://man7.org/linux/man-pages/man3/gethostbyname.3.html) which we'll use later on for the connection. We pass in the first parameter in the register 'r0' from the location `0xC1FC8` which has the value of `61.147.91.53`. We put the return value os `gethostbyname` into register r3 then preform some arithmetic on it.

The malware does two calls to `setsockopt`, passing the `IP_DROP_SOURCE_MEMBERSHIP` flag and the After the above has completed the the malware does a call to `connect` to make an initial connection to the C2 server and then does a mixture of `select` and `getsockopt` calls on the socket to ensure that the non-blocking socket has successfully connected. If the connection is not successful the malware will close the socket and exit.

Once we have a working socket returned from ServerConnectCli we then set the value of the global variable MainSocket to the return value. The malware then goes on to getting more statistics from the infected box. First of all, it grabs the username from the user running the executable which is placed into 'r11'. We can see that if the 'uname' call fails then "Unknown" will be copied into the destination buffer which originally resided in r11, else if it is successful, will branch to `0xCA8C`.

More information is gathered about the infected host via the `GetCpuInfo` function. Although this is self explanatory I'll explain it

anyways. The virtual file `/proc/cpuinfo` is opened and read in WORDs until an EOF (-1) is hit whilst reading the file chunk by chunk — then `fclose` is called to free up the opened file. The number of CPUs is then passed back out from this method and the clock speed in MHz too.

The malware then calls `sysinfo(..)` and reads into the struct (also named `sysinfo`) which is originally located in r3. We then read several members of this struct, such as the total amount of swap, total amount of RAM etc.. this is all then formatted and output into a string.

We can see here that the threat actor seems to be joking about by using the string 'Hacker'.. the string to be format is: `VERSONEX:Linux-%s|%d|%d MHz|%dMB|%dMB|%s` — you've spelt 'version' wrong Mr Threat Actor. It's very strange here that `sprintf` was used before rather than `snprintf` (which helps mitigate buffer overflows/format string exploits to an extent as the function knows the length of the buffer). Programmers will usually be inclined to use one naturally over the other, this indicates to myself there may be more than one programmer whom is contributing to this malware.

In the control flow we can then see that we try to then send this information on 'MainSocket', if the send is not successful (so, if we send 0) we branch to another subroutine and close the socket.

For those not familiar with ARM, the `beq` instruction basically says if the flag is set that they are equal then jump to address `0xCBD4`. This subroutine simply closes the socket, as said before. We then move onto `select`, if this call is successful thence move onto reading data from the C2. We then read data from the C2, but first, we zero out the buffer that we are using and have a maximum size of `0x1380` that we want to receive. Again, if this is not successful it then prints an error message and branches to a subroutine which closes the socket and cleans up.

Conclusion, so far..

It is obvious so far that this malware has been programmed by more than one author. We can also see so far that the author has experience in socket programming. The author is also using pascal case (LikeThis) and names

certain functions such as `GetCpuInfo` -- this could in turn indicate to us that the author is used to programming on Windows.

This is the end of this section, we will now move onto the details about the attack methods and what else the bot can be commanded to do.