CARLOS III UNIVERSITY OF MADRID

HIGHER TECHNICAL SCHOOL

MASTER IN CYBERSECURITY

MASTER THESIS

SECURITY ANALYSIS AND EXPLOITATION OF ARDUINO DEVICES IN THE INTERNET OF THINGS

Author: Carlos Alberca Pozo

Mentors: Sergio Pastrana Portillo & Guillermo Suarez-Tangil

September 20th, 2014

Security Analysis and Exploitation of Arduino devices in the Internet of Things

Carlos Alberca Pozo
Student of Master in Cybersecurity
Universidad Carlos III de Madrid
Email: carlos.alberca@alumnos.uc3m.es

Sergio Pastrana Portillo
COSEC Research group
Universidad Carlos III de Madrid
Email: spastran@inf.uc3m.es

Guillermo Suarez-Tangil
COSEC Research group
Universidad Carlos III de Madrid
Email: Guillermo.suarez.tangil@uc3m.es

***Abstract*—The pervasive presence of interconnected objects enables new communication paradigms where devices can easily reach each other while interacting within their environment. The so-called Internet of Things (IoT) represents the integration of several computing and communications systems aiming at facilitating the interaction between these devices. Arduino is one of the most popular platforms used to prototype new IoT devices due to its open, flexible and easy-to-use architecture. Ardunio Yun is a dual board microcontroller that supports a Linux distribution and it is currently one of the most versatile and powerful Arduino systems. This feature positions Arduino Yun as a popular platform for developers, but it also introduces unique infection vectors from the security viewpoint. In this work, we present a security analysis of Arduino Yun. We show that Arduino Yun is vulnerable to a number of attacks and we implement a proof of concept capable of exploiting some of them.**

***Index Terms*—Internet of Things (IoT); Arduino Yun; Security analysis; Heap overflow; Atmel.**

I. Introduction

The widespread adoption of communicating technologies such as smart devices (smartphones, tablets, watches, etc.), wearables (RFID tags, bracelets, e-shirts, etc.), home automation systems (smart alarms, smart fridges, etc), or medical devices (e.g. pacemakers, insulin pumps, neurostimulators) enables users to interconnect their systems world widely. The so-called Internet of Things (IoT) represents the integration of several computing and communications systems aiming at facilitating the interaction between these devices. In such scenario, security and privacy plays an important role as many of these devices incorporate sensors that could leak highly sensitive information (e.g.: locations, behavioral patterns, audio and/or video of their surroundings).

With the emergence of the Internet of Things (IoT), it is expected to reach the figure of 50 billion of devices in 2020. World-leading companies such as Google, Microsoft or Samsung are aware of that, and are developing their own IoT projects (e.g. Brillo Project by Google, W10 IoT by Microsoft and SmartThings by Samsung). Nevertheless, nowadays one of the most extended platform used in the IoT scenario is Arduino. It is expected that Arduino will remain one of the most used platforms in the next years, because it is open source, very cheap, versatile, and provides an easy implementation framework.

The reason of Arduino being used to develop IoT projects is twofold. First, Arduino is a very well-known technology, cheap and open-source. Second, it is very easy to learn and supported by a large open community on the Internet. Most of the developers with a minimal programming skills can buy one Arduino device at a very low price and start developing within it in few days. Then, the developed software can be shared or sold with no quality control at it.

A. Motivation

Arduino business model provides huge opportunities for non experienced users, which may cause security and privacy issues. Arduino is not considered to be a safe system, and among their main characteristics it is a clear lack of security measures. The market idea of Arduino is rather similar to Android (i.e., the idea of multiple developers sharing their projects in public markets), but the latter has been tested and can be considered a trustworthy operative system, at least secure against classical computer-based attacks. Anyway, it should be cleared that there is no 100% reliable operative system.

While the security in other IoT devices such as RFID tags or smartphones has been extensively studied, the security on IoT sensors based on Arduino platforms still lacks of research studies. It is not clear to what extent Arduino can be targeted with attacks initially developed for classical computers, like buffer overflows or other exploits. At a first sight, it seems that Arduino does not provide any memory failure control or other security measures that are incorporated in most of the current computer architectures, like Address Space Layout Randomization (ASLR) or Canary Stack Protection.

Latest improvements and researches conducted in powerful microchips used on PCs, smartphones, etcetera, are providing secure services developed on the firmware. Also, robust operating systems have security controls in Kernel layer. However, tiny chips, created to be used on embedded systems, are quite small and have limited resources, and therefore it is quite complicated to provide secure logic control on these chips. This makes that the security of the implemented applications relies on the developers, rather than in the supporting technology.

In this work, we propose a security analysis on one of the most common Arduino devices, the Arduino Yun. One of the main characteristic is that it is composed by a two-tier architecture. In a lower level, it contains a classical Atmel chip [1] (which is common in other Arduino devices). In a higher level, it has an a Atheros processor supporting Linino, which is a Linux distribution based on OpenWRT [2]. Both chips are connected between a Bridge (we will provide further details in this work). What makes it interesting is that, given that the security in Linux systems has been extensively implemented, it is not clear whether the overall system can be compromised by exploiting vulnerabilities or flaws in the lower level tier of the architecture.

B. *Main contributions*

This work provide the following contributions:

1) An analysis and study of the internal Arduino Yun architecture and its components. We reverse both major components of Arduino Yun, i.e.: the Linux environment and the Arduino environment.
2) A security analysis of Arduino Yun, showing that it is vulnerable to classical exploits. Here, we show that the resource-constrained nature of Arduino Yun limits the implementation of advanced secure mechanisms such as ASLR.
3) A proof of concept attack, exploiting a vulnerability of the lower-layer architecture, based on Arduino, to gain privileges in the higher-layer chip, based on Linux. More precisely, we show how to exfiltrate credentials, perform DoS attacks, perform update attacks, and install a rootkit.

C. *Structure of the document*

The remainder of this document is structured as follows. First, section II presents the related work regarding IoT security. Then, section III provides a description of the Arduino Yun internal architecture, showing which are its most critical components. Section IV presents the security analysis performed and the vulnerabilities and security flaws encountered. The design and implementation details of the proof-of-concept attack is provided in Section V, along with the results. Finally, Section VI presents a series of recommendations and discusses about the necessity of security measures in IoT, and Section VII concludes the work.

Additionally, two appendices are provided. Appendix A shows the source code of the vulnerable program used in the proof-of-concept, and Appendix B shows the research planning and estimated budget of this work.

II. Related Work

To the best of authors knowledge, this is the first paper providing security analysis on Arduino Yun. However, various studies confirm that security has not be a primary concern in the design of IoT systems. For example, Ghena et al. [3] analyzed the security of traffic lights infrastructure in a smart city, and showed how an attacker can turn on green lights permanently due to some vulnerable components in the sensors installed within the infrastructure. Radcliffe [4] discovered a vulnerability in the wireless communication used by a medical device (specifically, an insulin pomp), and carried out a replay attack exploiting the lack of a timestamp protection.

Indeed, the attack surface area in IoT is so extensive (see Figure 1), and research community [5] [6] have discussed about the ongoing challenges in IoT security which require attention. Next, we mention some of these ongoing challenges:

- Object Identification and Locating: The main challenge of object identification is to ensure the integrity of records used in the naming architecture.
- Authentication and Authorization: Although public-key cryptosystems have the advantage of constructing authentication schemes or authorization systems, the lack of a global root certificate authority (global root CA) for IoT applications prevent the deployment of theoretically feasible schemes.
- Privacy: The challenges can be divided into two categories: data collection policy and data anonymization. Data collection policy describes the policy during data collection where it enforces the type of collectable data and the access control of a specific section of the data. To ensure data anonymity, both cryptographic protection and concealment of data relations are desirable.
- Lightweight Cryptosystems and Security Protocols: Public key cryptosystems generally provide more security features than symmetric-key, but it implies a high computational overhead which may be prohibited in resource constrained devices used in many IoT.
- Software Vulnerability and Backdoor Analysis: Dynamic analysis may be inefficient to deploy in IoT. Multi-level examination to reduce software vulnerabilities, discovery of backdoors with reverse engineering, and software auditing are all useful to prevent the usage of backdoors. However, all these disciplines are still very immature when used in devices such as sensors or smartphones.
- Malware: IoT-targeted malware is a serious threat due to the limited resources of IoT devices. Indeed, many of the proposed security mechanisms, such as antivirus or intrusion detection systems, cannot be applied in such devices due to the overhead incurred by real-time scanning functionality.

In addition, the organization OWASP have analyzed which are the top 10 vulnerabilities in the IoT [7] [8]. Among others, there are vulnerabilities that coincided with the above mentioned, like insecure web interface, insufficient authentication/authorization, lack of transport encryption and so on.

Besides, there have been reported some attacks in real life making use of IoT devices. For example, in an Spam Campaign [9] between late December 2013 and early January 2014, researchers found that more than 25 % of the malicious emails (over 750,000 messages) came from devices that were not conventional laptop or desktop computers, but rather systems classified as members of the Internet of Things (IoT). The concept of *IoT Botnet or ThingBot* was emerged.

Given all the studies and reports analyzed, it can be concluded that attacks against IoT are becoming more common, and along with the critical scenarios where they may be applying, makes security in IoT a primary concern. In such

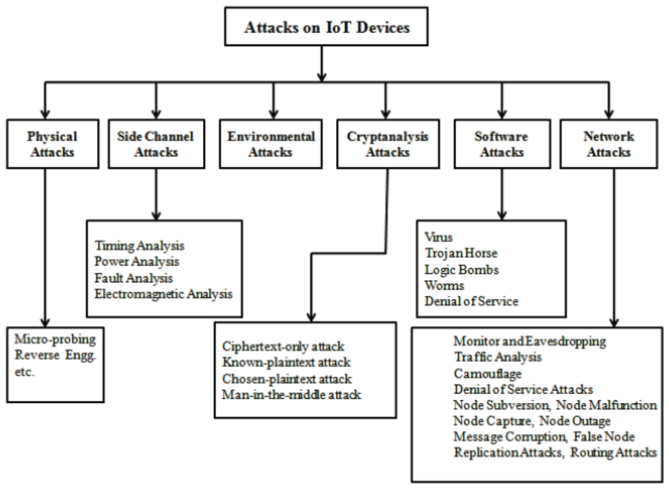


Fig. 1. Attacks on IoT Devices

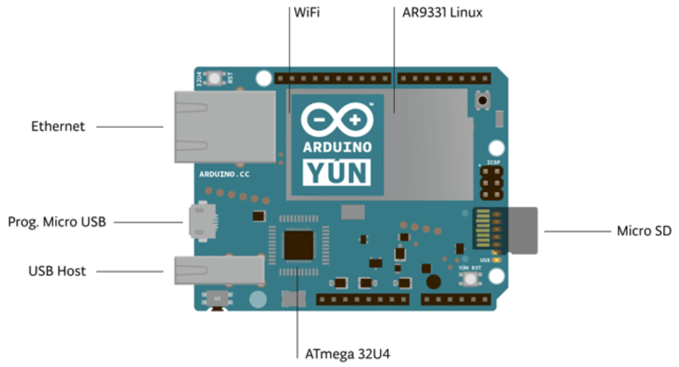


Fig. 2. Arduino Yun

TABLE I
SPECIFICATION OF THE AVR ARDUINO MICROCONTROLLER

Microcontroller	ATmega32u4 8bits
Operating Voltage	5V
Input Voltage	5
Digital I/O Pins	20
PWM Channels	7
Analog Input Pins	12
DC Current per I/O Pin	40 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (4 KB used by bootloader)
SRAM	2.5 KB
EEPROM	1 KB
Clock Speed	16 MHz

research direction, in this work we propose an analysis to demonstrate the insecurity of a common IoT device, i.e., the Arduino Yun.

III. THE ARDUINO YUN PLATFORM

Arduino is an open-source platform providing ”easy-to-use hardware and software intended for anyone making interactive projects” [10]. Originally it was proposed for being used in electronics and micro-controller projects. With the increasing interest of the IoT, the Arduino Yun has been designed specifically to run IoT applications, as stated in the official Arduino website [10], by combining both the low-level electronics originally present in other Arduino devices with higher level architectures running a Linux based Operating system. As such, this section provides a detailed description of the characteristics and architecture of Arduino Yun, with the purpose of analyzing how the minimal security measures can be circumvented by an adversary.

The Arduino Yun is composed of one micro-controller board based on two chips. One is the ATmega32u4 and the other an Atheros AR9331 (see Figure 2). The Atheros processor holds a Linux distribution based on OpenWrt named OpenWrt-Yun. The board has built-in Ethernet and WiFi support, a USB-A port, micro-SD card slot, 20 digital input/output pins (of which 7 can be used as PWM outputs and 12 as analog inputs), a 16 MHz crystal oscillator, a micro USB connection, an ICSP header, and a 3 reset buttons.

Arduino Yun distinguishes itself from other Arduino boards in that it can communicate with the Linux distribution onboard, offering a powerful-networked computer with the ease of Arduino (see Figure 3). In addition to Linux commands like cURL, developers can write their own shell and python scripts for robust interactions. Both parts, Arduino and Linux environment, can be connected each other through a Bridge. This Bridge is a logical component programmed in python that contains different modules (e.g. bridge.py, packet.py, mailbox.py, processes.py, an so on). The main module is bridge.py which, along with the other ones, allows to extend the functionality. This library facilitates communication, giving Arduino sketches the ability to run shell scripts, communicate with network interfaces, and receive information from the AR9331 processor. Consequently, this is a critical component, because by exploiting Arduino environment, as we show bellow, it is possible to bypass inner security mechanisms, therefore compromising the Linux environment.

The USB host, network interfaces and SD card are not connected to the 32U4, but the AR9331, and the Bridge library also enables the Arduino to interface with those peripherals.

Tables I and II provide details of the hardware specifications of both the ATmega32u4 and the Atheros chips respectively. As it can be observed, the characteristics on both chips are resource constrained, compared to classical computer. However, it is noteworthy to observe the difference in processing capabilities of both chips (2.5KB of RAM in ATmega32u4 vs 64 MB in Atheros). The RAM memory of the Atheros chip makes it suitable to run a lightweight operating system like Openwrt, implementing a minimum security capabilities.

In our experimental analysis, we have used the latest stable image OpenWRT-Yun 1.5.3 of November 13th, 2014. By reverse engineering and taking a look in vendors info, the next two sections provide a detailed description on some internal parts of the Linux and Arduino environments, respectively.

TABLE II
SPECIFICATIONS OF ATHEROS MICROPROCESSOR

Processor	Atheros AR9331
Architecture	MIPS @400MHz
Operating Voltage	3.3V
Ethernet	IEEE 802.3 10/100Mbit/s
WiFi	IEEE 802.11b/g/n
USB Type-A	2.0 Host
Card Reader	Micro-SD only
RAM	64 MB DDR2
Flash Memory	16 MB
SRAM	2.5 KB
EEPROM	1 KB
Clock Speed	16 MHz
PoE compatible 802.3af card support	See *Power*

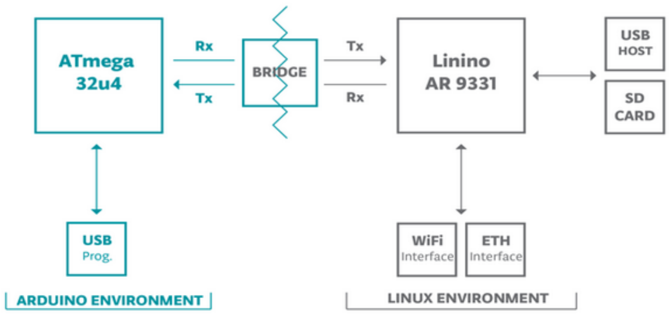


Fig. 3. Arduino Yun Diagram

A. *Linux Environment*

The Linux Environment has an OpenWRT distribution [11] for Arduino Yun composed by Linux Kernel v3.3.8. OpenWRT is a Debian based distribution of Linux, and as such it has some packages installed, like Dropbear sshd v2011.54 which implements an SSH server (required for remote administration). In the next section, we show that some of these packages may contain some vulnerabilities if the system is not properly updated or patched.

Indeed, as the Openwrt is a debian based distribution, the chances of an attacker are huge it is able to get root privileges. For the sake of illustration, the following analysis lists paths of the system that can be abused by an adversary and which could be the consequences:

- /var/run $\rightarrow$ at this path information about processes are stored, therefore some attacker can modify them or include his pidfile.
- /var/run/wpa_supplicant-wlan0.conf $\rightarrow$ WiFi parameters stored in cleartext, so it is possible to steal it for adding it in a wordlist to do a brute force attack against the user.
- /tmp/resolv.conf.auto $\rightarrow$ DNS servers are possible to change them, giving rise to pharming attack.
- /etc/arduino $\rightarrow$ there are GPG keys to steal them for creating and signing own packages.
- /etc/config $\rightarrow$ configurations about Arduino, ssh, firewall, wireless, etcetera, where it is possible to get WiFi key, change REST API security to false to it does not ask about the password, disable ssh authentication, and so on. Note that the Arduino access password is stored with SHA256 hash function.
- /etc/dropbear $\rightarrow$ steal the SSH RSA keys.
- /etc/hosts $\rightarrow$ file to map hostnames to IP addresses giving way pharming attack.
- /etc/avrdude.conf $\rightarrow$ has the pin mapping and other parameters that are used by avrdude for the ATmega32u4, so modify the pins, memory parameters and more, is possible.
- /etc/opkg.conf $\rightarrow$ it is possible to change the official repository by own malicious one, where there could be some vulnerable packages.
- /sys/ $\rightarrow$ contains modules about crypto, kernel, drivers, firmware, etcetera, where is possible to analyze what are in used (e.g. crypto modules, WiFi driver...).
- /rom/ $\rightarrow$ if files are changed here, the changes are persisted against factory reset.
- /usr/bin/kill-bridge $\rightarrow$ is the script to kill the bridge service between Arduino and Linux environment giving rise to Denial of Service, or change the path to execute an own module to open backdoors, spam email... Note that there is another script to run again the bridge (run-bridge) when some hypothetical change is made in bridge.py.
- /usr/bin/wifi-reset-and-reboot $\rightarrow$ to reset wifi parameters and reboot the system giving the option, for instace, to change the default host IP.
- /usr/lib/python2.7/bridge/ $\rightarrow$ contains python bridge scripts that define the functionality about the bridge (e.g. bridge.py, mailbox.py, processes.py). It would be possible to change the scripts doing an own malicious action.

B. *Arduino environment*

The Arduino environment contains an ATmega32u4 chip based on AVR architecture, which is Little Endian. It has 32 registers and the memory is divided in three types: EEPROM (as a HDD), SRAM (is dynamic in the execution time) and Flash (where code is loaded). Figure 4 shows an schematic view of the most relevant memory areas of the SRAM and its base addresses (i.e., the registers, heap and stack). We consider these areas as they are classical targets of software exploitation and other cyber attacks.

32 Registers contain positions from *R0* (0x00) to *R31* (0x1F). The registers *R26...R31* have some added functions to their general-purpose usage. For example, the *Y* register is the Frame Pointer (*FP* is used to reference the local variables and the parameters). These registers are 16-bit address pointers used for indirect addressing of the data space. The three indirect address registers *X*, *Y*, and *Z* are defined as follows:

- *X* register = 0x[*R27* (0x1B) + *R26* (0x1A)]
- *Y* register = 0x[*R29* (0x1D) + *R28* (0x1C)]
- *Z* register = 0x[*R31* (0x1F) + *R30* (0x1E)]

Data area contains global variables used by the program that are not initiated to zero. **BSS** segment contains all global variable that are initiated to zero and constant strings. The *AVR* **Stack Pointer (*SP*)** points to top of the stack or in other words the data SRAM Stack area where the Subroutine and

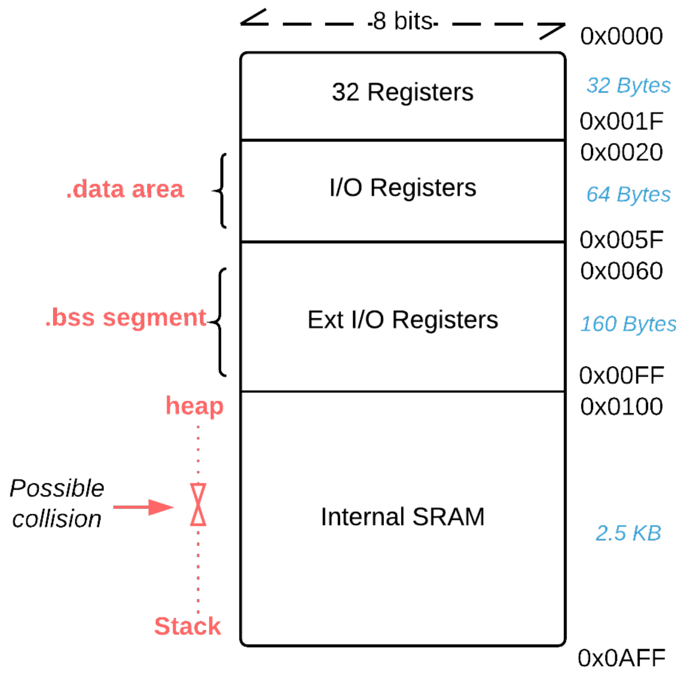


Fig. 4. ATmega32u4 SRAM memory

Interrupt Stacks are located; It is implemented as two 8-bit registers (*SPH*:*SPL*) in the I/O space.

The **stack**, that grows from higher to lower memory locations, contains the return addresses for subroutines, saved registers and local variables, and when it grows, the deallocation process is automatic. Nevertheless, the **heap**, that grows from lower to higher memory locations, is managed by malloc(), realloc() and free(). It is shared by all common libraries and dynamic load modules in a process, and its deallocated must be done manually (e.g. free()).

C. *Programming*

Regarding the Programming, Arduino Yun can be programmed with the Arduino IDE. This software is open-source and makes it easy to write code and upload it to the board to interact with the Arduino environment and Linux environment if it is necessary.

The environment is written in Java and based on Processing (an other open-source software), and GitHub hosts its source code where there is an active development of the Arduino software IDE.

This IDE is a little bit limited because it does not perform any security checks in compilation time, as it will see in next section.

The most people in the Arduino community use this IDE to develop some real life examples where Arduino Yun is used [12], e.g., alarms, boiler controllers, home controllers, and so on, sharing them in websites such as Instructables. In almost all of these cases the developer has not used secure programming as it is going to show in the next sections.

D. *Remote configuration*

Remote configuration enables developers and users to trigger actions and control their devices from anywhere.

By default, Arduino Yun is not previously connected to any WiFi, therefore it will create its own free WiFi hotspot (ARDUINO-YUN90XXXXXX). Once we are connected to this hotspot, the administration website is accessible visiting the IP 192.168.240.1 in a browser, and when it asks for a password, the default one is *arduino* to configure the Yun to connect to a specific wireless network, and then meet it back on that network.

When the WiFi network, where the Yun is connected, goes down or is not up, it creates a new free WiFi hotspot after more or less one minute, therefore anyone can connect to this hotspot and perform actions. So, once the Yun is accessible, either the hotspot or the network where it is connected, the administration website is available to try some website attacks, and also a ssh connection, which is vulnerable as it will see in next section.

IV. Security Analysis

This section describes how an adversary can compromise the device by exploiting vulnerabilities found in its design. It must be cleared that the analysis is performed in the devices with its default configuration, i.e., the factory defect device.

A. *Linux Environment*

The first and one of the most important vulnerabilities found is that *root* user is a unique and default user, with no password assigned, and thus it has full privileges. Therefore, there is no authentication when locally accessing to the device, for example by using the Bridge at the Arduino environment.

We have conducted an external reconnaissance and vulnerability analysis on the Arduino Yun, using two popular open-source tools, i.e., Nmap [13] and Nessus [14], from a remote host. The analysis reports some vulnerabilities in the installed software. In the next, we describe some of these vulnerabilities and list their corresponding CVE identifier [15]:

- Dropbear sshd v2011.54: the remote SSH service is affected by multiple vulnerabilities that allows an attacker to perform three different attacks. First, it is possible to run a Denial of Service (DoS) against the server, provoked by the way the buf_decompress() function handles compressed files (CVE-2013-4421). Second, it is possible to perform a User-enumeration due to a timing error when authenticating users (CVE-2013-4434). Finally, and probably the most critical one, is that it allows adversaries to execute remote code (CVE-2012-0920) when the called *Use-after-free* condition runs on concurrency channels.
- BusyBox v1.19.4. The DHCP client implementation (udhcpc) allows malicious remote DHCP servers to execute arbitrary commands via shell script meta-characters (CVE-2011-2716).
- Dnsmasq v2.62. The DNS server may respond from some prohibited interfaces, and allows attackers to run specific

Denial of Service attacks (CVE-2013-0198 and CVE-2012-3411).
- Linux Kernel v3.3.8. The results of the vulnerability scan reports that there are several vulnerabilities for this kernel, such as an netfilter bug which allows attacker to crash the system, i.e., perform a DoS (CVE-2014-2523) or an implementation error that allows local users to gain privileges (CVE-2013-1763). However, it should be noticed that existing exploits for these vulnerabilities might not be directly applicable to Arduino Yun since its kernel has been compiled specifically to run on embedded devices. Thus, would required further analysis to confirm that these vulnerabilities can be exploited in Arduino Yun.

While these vulnerabilities have related patches and updates, the official Arduino repository [16] has not been updated when this work is written. Among others, this repository contains the last packages belonging to the OpenWRT distribution

Since there is not integrity on critical file checks, like the Bridge (/usr/lib/python2.7/bridge/bridge.py) or the binaries, it is possible for an attacker to replace them with their own files. Regarding the persistence of the modifications, it is possible to recover the original files performing a factory reset. However, we have observed that, if the modified files are saved on /rom/ directory, changes will be persist. To solve this, it would be required to flash a new image into the memory of the device, with the possibility of losing previous configuration files stored on it.

Denial of services attack can be performed on the bridge executing the script /usr/bin/kill-bridge, on the network or /usr/bin/wifi-reset-and-reboot when WiFi is used. This way, the connection between the arduino and the linux environments would be lost.

Additionally, due to the fact that root user is enabled by default, any package can be installed. This way, an adversary could modify the repositories in the *opkg* configuration file (which is the package manager application, similar to the more famous *apt-get*) and install malicious applications, such as SSLSnif, Arpwatch, or SSLcat, among others.

The Arduino administration webpage can be accessed via port number 80 (HTTP) without being redirected to port 443 (HTTPS). Therefore, a normal connection can be sniffed from a node within the same local network. In addition to this, in an application that uses the REST web service, traffic could be modified, adding or modifying it for the adversary purpose.

If it is not possible to reach the Yun over the network where it is connected, deauthentication attack can be executed during 1 or 2 minutes, and the Yun creates a new free WiFi hotspot. This attack sends disassociate packets to the Yun, which are currently associated with a particular access point. Disassociating the Yun can be done for more reasons, e.g., recovering a hidden ESSID, which is not being broadcast, or Capturing WPA/WPA2 handshakes by forcing clients to reauthenticate.

B. Arduino Environment

Devices or sensors connected to Arduino are not authenticated. Thus, these tools could be disconnected from the serial port and then connected to any other device. This would allow, for example, to send a huge amount of data to the Arduino and perform an DoS.

Exploring vulnerabilities in Arduino Yun is not as straightforward as in the case of the Linux environment, as there are not known automated tools for such purpose. We have conducted the vulnerability search by an static analysis of the memory address map and an ”try-and-error” approach. Additionally, unlike in other architectures like Intel x86 or ARM, in the Atmel32u4 there is a lack of debugging tools. Thus, to do debugging, we use the ’Serial.print()’ statement to follow the program behavior, which obviously makes debugging harder.

For this platform, we have found that the Arduino IDE compiler does not perform any security checks in compilation time. Therefore, when a sketch (program) runs out of memory, there is no warning nor segment violation, but just random malfunctions. The next step is to try to take advantage of such out-of-memory errors. Concretely, we check errors in two differentiated parts of the SRAM memory: the Heap and the Stack.

- In the case of the Heap, it has been demonstrated how a Heap Buffer Overflow (see Figure 5) can allocate consecutive memory buffers in the heap (whose corresponding variables are named *command*, and *argument*) using malloc(). Then, the overflow occurs by overwriting the first set of data (i.e. *command*) with data from the second one (i.e. *argument*). In Section V we provide further details of this exploitation technique, and in Appendix A we provide the vulnerable program code.

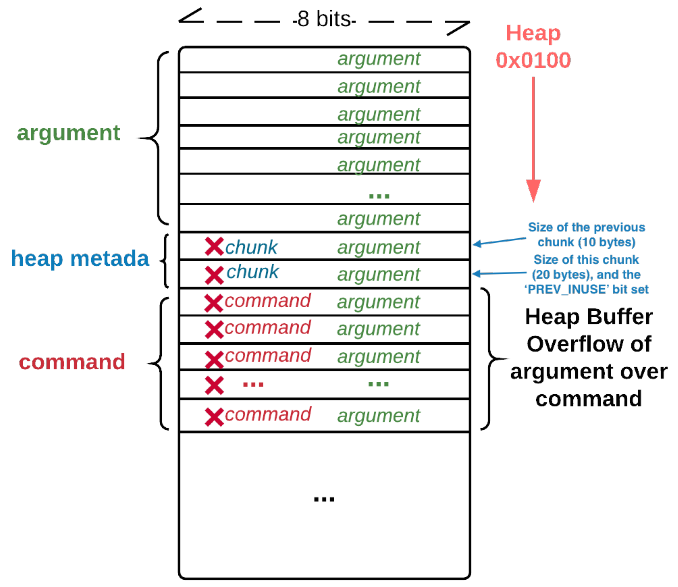

-8 bits-
Heap
0x0100
argument
heap metada
command
chunk
command
Size of the previous chunk (10 bytes)
Size of this chunk (20 bytes), and the 'PREV_INUSE' bit set
Heap Buffer Overflow of argument over command

Fig. 5. ATmega32u4 Heap Buffer Overflow

- In case of the Stack, it is possible to perform a Stack Buffer Overflow attack because no protection has been

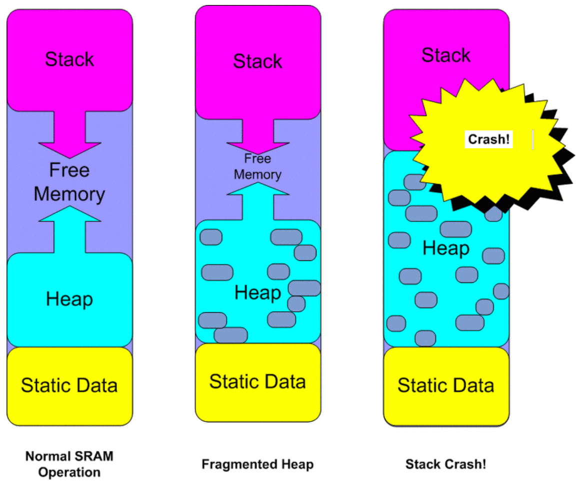


Fig. 6. ATmega32u4 Stack and Heap Collision [17]

implemented to avoid it. Unfortunately, due to the lack of debugging tools, we have not be able to take the control of the program flow and thus we are not able to execute our own shellcode, as it is going to shown in the next section. However, it can be derived that a DoS attack has been performed, because the system crashes. Moreover, it is important to note that the Heap and the Stack can collide when a big amount of subroutine calls with many local variables are done [17] (see Figure 6).

V. Proof of concept attack

In this section, we describe the attack we have performed to demonstrate the vulnerabilities found in the security analysis. This is a proof of concept attack showing that the system can be abused because the security does not exist, carrying on malicious post-exploitation actions through malware or backdoors.

A. ATmega32u4 Heap Buffer Overflow

For our experiments, we have developed a vulnerable program whose aim is to configure the network interfaces of the Arduino Yun via bluetooth (e.g., this would be helpful, for example, if the ethernet or wireless connection is lost, and the only possible way to restore it is using bluetooth). The program, whose source code is provided in Appendix A, uses bluetooth to ask and receive data from users, storing the data in a variable (*argument*). This data corresponds to the arguments required to configure the network interfaces. It also has the static command used to configure the network interfaces hardcoded in it (i.e., *ifconfig*) to execute with that argument. Note that, as a regular developer would do, there is also a little security check that prevents the user to execute more than one command, by checking if there is an '&' character in the *argument* buffer. The two data buffers (*argument*=10 Bytes and *command*=20 Bytes) are assigned on the heap with malloc(). *Argument* is used to store user-supplied input from strcpy() by bluetooth. So the attack requires the following malicious data to be stored as *argument*, as it provokes the overwriting of the buffer *command* (see Figure 5): *10 bytes of data (overwrites argument variable) + 2 bytes more because of heap metadata + wished command*. In this case, the *command* variable is executed in Linux environment through the python-based Bridge. So the goal of the attack is to overwrite with wished command (remember that the user is root by default) to perform a malicious action. Some malicious payloads that have been executed are:

- **Credential exfiltration**. *12 blank data + cp /etc/shadow /www/proof.txt* $\rightarrow$ Copies the content of the file shadow in the folder www, so it can be retrieved by an HTTP request.
- **DoS attacks**. *12 blank data + /usr/bin/kill-bridge* $\rightarrow$ Kills the bridge between the two chips, causing a DoS in the device.
- **Downgrade attacks**. *12 blank data + echo 0 >/proc/sys/kernel/randomize_va_space* $\rightarrow$ Disables address space randomization in the Linux environment.
- **Payload injection**. *12 blank data + opkg sslstrip* $\rightarrow$ Install a malicious package to perform attacks on SSL.
- **Persistence attacks**. *12 blank data + wget http://myserver.com/rootkit.sh && ./rootkit.sh* $\rightarrow$ Download and executes a script, which could be for example a rootkit.
- **Update attacks**. *12 blank data + cd /etc/ && wget http://myserver.com/opkg.conf* $\rightarrow$ Replaces the configuration file where the package repositories are downloaded.

B. ATmega32u4 Stack Buffer Overflow

The program first read an input from Serial in main(), an then passes this input to another buffer in the function f1(). The original input can have a maximum length of 30 bytes, but the buffer in f1() has only 10 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. If this program is running to send data from Arduino environment to the Linux one, Denial of Service can happen. So, if there is a debugger for Arduino IDE, it can execute malicious code.

The attack requires to overwrite the SP register (similar to EIP in x86) with the buffer. In this experiment with 16, 17 and 18 characters the program returns properly to main() but no work any more, and with 19 and more characters the program, and even the Serial connection, dies.

Sometimes, due to the irregular Arduino behavior, instead of just dying, it returns the stack memory printed in Serial port, so if there is some password in the code and in clear text, it would be possible to obtain it.

C. Attacks on the Linux environment

1) DoS against SSH server: As stated before, the SSH client based on Dropbear is vulnerable. We have used the Kali Linux distribution to launch an exploit which successfully performs a Denial of Service because of CVE-2006-1206. Given a parameter X, the exploits provokes that during X seconds the Arduino SSH does not work properly.

2) Google Hacking and Shodan.io: Google Hacking and Shodan.io are search engines that allow to look for devices in the global Internet. Using such engines, we have realized that there are more than one Arduino Yun opened to the Internet, and some of them are vulnerable to attacks, as shown in previous section. Others even are still configured with the default password of the factory defaults. For example, we have found a webcam without password. An example of Google Hacking we have conducted is:

inurl:"/cgi-bin/luci/webpanel" -intext:"/cgi-bin/luci/webpanel" -inurl:"80".

In those ones where the password is the default one, it permits connect by ssh, and perform lateral movements in the network, threatening the privacy.

VI. Discussion

Based on the analysis and findings of this work, this section provides a series of recommendations for the IoT community, concretely to be adopted by Arduino developers. It is clear that Arduino should enhance Security in both sides, the Atmel chip and the Linux OpenWRT distribution, but this is not easy due to the limited resources of the supporting technology. Research efforts are required to provide lightweight memory controls in the Atmel chip, otherwise all the security cautions must be taken by developers, which may not be aware of the security breaches they leave in their programs. Additionally, since many Arduino devices may not have direct connection to Internet (or at least, permanent connection), some measures should be taken to update periodically and more often the Linux OpenWRT distribution.

In order to help developers to design robust and secure programs, it would be desirable to have some guidelines from the Arduino community, similar to the "Best practices for Security and Privacy" published for Android [18]. Such guidelines would help to develop secure programs (e.g. Buffer handling, How to free memory to avoid stack and heap collision, Input/Output control on Serial port, pin layout, etcetera). And to avoid unskilled programmers to write unsecure code would be good to introduce protections against Heap and Stack Buffer Overflow (e.g. Canaries, Bound checking, Tagging) on the Arduino IDE compiler.

At the Linux environment, it is clear that *root* should not be the only and default user in the system. The Openwrt distribution installed is the default one, while it would be interesting to adapt it to the Arduino Yun environment. For example, it should be set up an specific user with limited permissions to handle commands from the non-secure Atmel chip, and another user to handle remote ssh connections. Morevoer, it is important to apply integrity controls to detect changes on critical files, for the system applying secure boot architecture for Embedded devices [19] (see Figure 7).

Also, it is recommended to study the adoption of modern security controls in IoT [8] such as: Object identifications, Authentication and authorization, Privacy, Lightweight Cryptosystems and Security Protocols [20] (e.g. Yoking-Proof), Software Vulnerability and Backdoor Analysis, Malware prevention in IoT, protections against USB HID,etc.

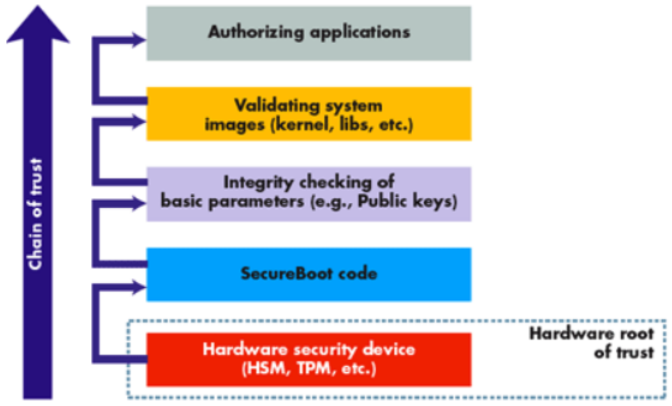

Chain of trust
Authorizing applications
Validating system images (kernel, libs, etc.)
Integrity checking of basic parameters (e.g., Public keys)
SecureBoot code
Hardware security device (HSM, TPM, etc.)
Hardware root of trust

Fig. 7. Generic secure boot architecture [19]

A big challenge in IoT, at least in Arduino, is to provide an exception handling control, like try/catch sentences, to keep the system running when a stack buffer overflow occurs. But this is quite hard to perform on these tiny processors since they have limited resources.

Finally, it is important to make people aware of the need of take security controls when developing applications in critical environments, since the interest of hackers in IoT devices is increasing due to it extend popularity.

VII. Conclusion

Internet of Things offers new opportunities in modern applications, and its use has increased considerably in the last years. However, it comes up with a new computation paradigm which opens new vulnerabilities, given that the attack surface area is so wide. Deficient security capabilities and difficulties for patching vulnerabilities in these devices, as well as a lack of consumer security awareness, provide malicious cyber actors with opportunities to exploit these devices. Criminals can use these opportunities to remotely facilitate attacks on other systems, send malicious and spam e-mails, steal personal information, or interfere with physical safety.

To show up the actual deficiency of security in IoT, in this paper we have analyzed and exploited one of the most used devices used in IoT sensors, the Arduino Yun. Concretely, we have analyzed deeply the internal architecture of one device, and provided a security analysis on it. Then, we have presented a proof-of-concept attack over a vulnerable application, which we have developed for this purpose. By exploiting such vulnerability, we have been able to perform a series of more serious post-exploitation attacks, such as rootkit installation, opening backdoors and the like.

Due to the use of IoT is growing fast and there are vulnerable IoT devices, malware is expected to grow exponentially in the coming years, as it has occurred in smartphones, threatening the privacy, availability, and integrity. Thus, it is essential to be prepared and provide further security checks to the developed applications. Unfortunately, the processors used in the IoT environment mainly consists of tiny devices with limited processing power. As the attackers become sophisticated, it becomes necessary to dedicate entire co-processor with high scalability to offer entire security features that an embedded system may require. This is probably one of the major challenges for the research community in the next few years.

Finally, a standardized security protocol is indispensable for the success of IoT. When every object in our daily life is connected to the Internet, they must speak the same (security) protocol to ensure interoperability. The standardization efforts have to grow with the technology, giving rise to a very important effort to make IoT a reality.

VIII. Future Works

The work presented in this document opens new horizons in the security of IoT devices. Concretely, we envision the following research lines:

- Improve the stack overflow exploitation to hijack the program flow and redirect the execution to my own malicious code, in order to show how extra malicious code could be executed in the Atmel chip.
- Analyze the security of the network applications used for remote configuration, such as the LuCI web application, the wireless implementation of future updates of the Dropbear for the SSH server.
- Provide a document for basic measures for secure development in IoT platforms (e.g. Arduino).
- Design and develop specific malware pieces that run in Arduino chips, in order to provide the corresponding countermeasures.

References

[1] Atmel official page. [Online]. Available: http://www.atmel.com

[2] Linino arduino yunprovi. [Online]. Available: http://www.linino.org/product/arduino-yun/

[3] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman, "Green lights forever: analyzing the security of traffic infrastructure," in *Proceedings of the 8th USENIX conference on Offensive Technologies*. USENIX Association, 2014, pp. 7–7.

[4] J. Radcliffe, "Hacking medical devices for fun and insulin: Breaking the human scada system," in *Black Hat Conference presentation slides*, vol. 2011, 2011.

[5] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "Iot security: Ongoing challenges and research opportunities," in *Service-Oriented Computing and Applications (SOCA), 2014 IEEE 7th International Conference on*. IEEE, 2014, pp. 230–234.

[6] S. Babar, A. Stango, N. Prasad, J. Sen, and R. Prasad, "Proposed embedded security framework for internet of things (iot)," in *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011 2nd International Conference on*. IEEE, 2011, pp. 1–5.

[7] Owasp internet of things project. [Online]. Available: https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project

[8] How to test the security of iot smart devices. [Online]. Available: http://resources.infosecinstitute.com/test-security-iot-smart-devices/

[9] Malware or spam campaign on internet of things. [Online]. Available: https://blog.fortinet.com/post/malware-or-spam-campaign-on-internet-of-things

[10] Arduino official page. [Online]. Available: http://www.Arduino.cc

[11] Openwrt official page. [Online]. Available: https://openwrt.org

[12] Arduino yn: the best hacks you will ever see. [Online]. Available: http://www.open-electronics.org/arduino-yun-the-best-hacks-you-will-ever-see/

[13] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Insecure, 2009.

[14] J. Beale, R. Deraison, H. Meer, R. Temmingh, and C. V. D. Walt, *Nessus network auditing*. Syngress Publishing, 2004.

[15] Common vulnerability exposure details. [Online]. Available: http.//www.cvedetails.com

[16] Arduino yun repository. [Online]. Available: http://downloads.arduino.cc/openwrtyun/1/packages/

[17] Optimizing sram. [Online]. Available: https://learn.adafruit.com/memories-of-an-arduino/optimizing-sram

[18] Best practices for security & privacy. [Online]. Available: http://developer.android.com/training/best-security.html

[19] A. Ukil, J. Sen, and S. Koilakonda, "Embedded security for internet of things," in *Emerging Trends and Applications in Computer Science (NCETACS), 2011 2nd National Conference on*. IEEE, 2011, pp. 1–6.

[20] C. Alberca-Pozo, *Implementacin del protocolo de seguridad Yoking-Proof en dispositivos mviles*. Final Degree Project, UC3M, 2014.

Appendix A
Source code of the vulnerable application used in the Proof of Concept

```
/* ******************** LOOP - WHERE 'HEAP OVERFLOW
    ' IS GOING TO BE EXPLOITED ********************
    */
void loop() {

  Process cmd;
  String btData = "";
  char *argument, *command;

  // Reservating dynamic memory (char = 1 byte).
      These variables are in the Heap.
  argument = (char *) malloc(10*sizeof(char));
  command = (char *) malloc(20*sizeof(char));

  // This is going to be overwritten.
  strcpy(command, "ifconfig␣");

  Serial.print(F("\n\nCommand␣variable␣before␣Heap␣
      Buffer␣Overflow:␣\n"));
  Serial.print(command);
  BTSerial.print(F("\n\nCommand␣variable␣before␣Heap
      ␣Buffer␣Overflow:␣\n"));
  BTSerial.print(command);

  // Asking for a parameter. Here we will do the
      Heap Buffer Overflow.
  // Heap Buffer Overflow example: "eth1 aaaaa cat /
      etc/shadow".
  Serial.print(F("\n\nWhat␣interface␣do␣you␣want␣to␣
      up␣or␣down␣or␣get␣the␣info?\n"));
  BTSerial.print(F("\n\nWhat␣interface␣do␣you␣want␣
      to␣up␣or␣down␣or␣get␣the␣info?\n"));
  while (!BTSerial.available());

  while (BTSerial.available()) {
    char c = BTSerial.read();  //gets one byte from
        serial buffer
    delay(50);
    btData.concat(c);
  }

  // Security check: Checks if the user wants to do
      more than one command
  if(btData.indexOf('&') == -1){ // '&' not found
    strcpy(argument, btData.c_str()); // HEAP
        OVERFLOW
      Serial.print(F("\n\nCommand variable after Heap 
          overflow: \n"));
      Serial.write(command);
      BTSerial.print(F("\n\nCommand despues after Heap
           overflow: \n"));
      BTSerial.write(command);

      //execute the command
      int exitCode = cmd.runShellCommand(command + (
          String)argument);
      showOutputProcess(cmd);
    }else{
      Serial.print(F("\n\nError: You are trying to 
          execute another command.\n"));
      BTSerial.print(F("\n\nError: You are trying to 
          execute another command.\n"));
      return;
    }

    delay(5000); // 5 seconds
}
```

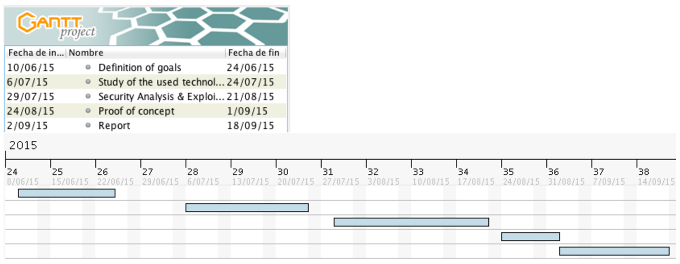
Appendix B
Planning and Budget

First, it is going to be presented a planning about this project defining the different tasks that have been carried out. To do this, it has been used a tool called GanttProject, which generates the typical Gantt chart presenting in a graphical form the duration of each of the activities.

This project has been divided into tasks, which are showed in Figure 8 with its start and end date. Gantt chart with detailed information is shown in Figure 9.

	Duration	Start Date	End Date
Definition of goals	15 days	10/06/2015	24/06/2015
Study of the used technologies	19 days	06/07/2015	24/07/2015
Security Analysis & Exploitation	25 days	29/07/2015	22/08/2015
Proof of concept	9 days	24/08/2015	01/09/2015
Report	19 days	02/09/2015	20/09/2015

Fig. 8. Work planning



Fecha de in...	Nombre	Fecha de fin
10/06/15	Definition of goals	24/06/15
6/07/15	Study of the used technol...	24/07/15
29/07/15	Security Analysis & Exploi...	21/08/15
24/08/15	Proof of concept	1/09/15
2/09/15	Report	18/09/15

Fig. 9. Gantt chart of the work planning

The budget is going to be broken down into various estimations: Personal cost (87 days and 5h/day) (see Figure 10); Hardware cost (see Figure 11); Software cost (see Figure 12); Indirect cost (see Figure 13); Total cost (see Figure 14). Note: VATs are included.

Concept	Time	Fees	RRHH cost
Security Engineer	435 hours	50 €/hour	21750 €
PhD	30 hours	55 €/hour	1650 €
TOTAL			23400 €

Fig. 10. Estimated Personal cost of the project

Concept	Units	Unit price	Estimated life time	Airtime	Cost for the project
Apple MacBook Pro 13"	1	997,50 €	60 months	2,9 months	48,21 €
Google Nexus 5	1	400 €	36 months	2,9 months	32.22 €
Apple Magic Mouse	1	70 €	60 months	2,9 months	3,38 €
Arduino & Components	1	90 €	60 months	2,9 months	4,35 €
TOTAL					88,16 €

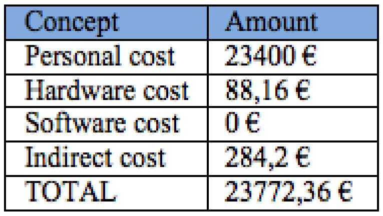
Fig. 11. Estimated Hardware cost of the project

Concept	Units	Unit price	Estimated life time	Airtime	Cost for the project
Apple Mac OS X 10.9.5	1	0 €	-	2,9 months	48,21 €
Sublime Text 2	1	0 €	-	2,9 months	0 €
Mactex & Texmaker	1	0 €	-	2,9 months	0 €
LucidChart	1	0 €	-	2,9 months	0 €
Dropbox	1	0 €	-	2,9 months	0 €
GanttProject	1	0 €	-	2,9 months	0 €
Kali Linux	1	0 €	-	2,9 months	0 €
Nessus	1	0 €	-	2,9 months	0 €
Arduino IDE	1	0 €	-	2,9 months	0 €
TOTAL					0 €

Fig. 12. Estimated Software cost of the project

Concept	Monthly price	Airtime	Cost for the project
Electricity	63 €	2,9 months	182,7 €
Internet	35 €	2,9 months	101,5 €
TOTAL			284,2 €

Fig. 13. Estimated Indirect costs of the project



Concept	Amount
Personal cost	23400 €
Hardware cost	88,16 €
Software cost	0 €
Indirect cost	284,2 €
TOTAL	23772,36 €

Fig. 14. Estimated Total cost of the project