Wrocław University of Science and Technology

Faculty of Information and Communication Technology

List 4 Report

Hakan Yildizhan 270056
Gabriel Wechta 250111

Supervisor:
Wojciech Wodo, PhD

January 31, 2023

Report
Embedded Security

Contents

1 | List 4

1.1 Task formulation

"Embedded systems credentials breaking and time side-channel analysis. You will be equipped with the program code for Arduino Uno with implementation of following features:

- Door keypad - PIN (for getting access)
- Login/Password (for getting privileges)
- Red herrings (for misleading adversary)

Upload the code to the Arduino and launch the device. Your task is to overcome each security means by using different analysis methods and prepare scripts/toolkits facilitating attacks. You should be able to establish connection between Arduino and your computer via serial port in order to perform the communication or use another Arduino. Your ultimate goal is changing the blinking frequency of the device."

1.2 Uploading the code to Arduino

We downloaded the binary from `https://wwodo.mokop.co/uploads/docs/ess19/code.hex` using `wget` Unix command.

Then we uploaded it to Arduino using the following command.

```
avrdude -v -p atmega328p -c arduino -P /dev/ttyACM0 -b 115200 -D -U flash
    ↪ :w:/home/gabriel/white_rose/embedded_security/lab_4/code.hex:i
```

Listing 1.1: Command used to upload binary to the Arduino.

Where

- `avrdude` – is a program for downloading and uploading programs from and to AVR microcontrollers [1].
- `atmega328p` – namely **AVR ATmega328P-U DIP** is a microcontroller used in Arduino Uno.
- `/dev/ttyACM0` – USB interface for connected Arduino.
- `115200` – we found baudrate by trying the most popular ones.

By examining the blinking diode on the Arduino, we concluded that the program has successfully started running.

1.3 Talking with Arduino

The Arduino and the program running on it at this point were a black box. All we knew was that we can send and receive messages via serial port. We started by examining some random inputs, simple strings, one-byte characters, and so on. Most of the commands, with some small delay, returned an empty string. Those who returned something interesting returned noticeably faster and provided some enigmatic messages, like `b'500000\r\n'` or `b'Denied.\r\n'`. All messages are byte-encoded ASCII characters. Sending the same message always returned the same result.

One more important thing. After connecting to the Arduino and catching its output, at the beginning there is a brief moment in which the Arduino sends a message `b'CameraKey Driver - 2019\r\n'`.

1.3.1 Open source intelligence

As prudent security researchers, this was the first time we heard about the infamous `CameraKey Driver` - so we decided to put it on our trusty search engine.

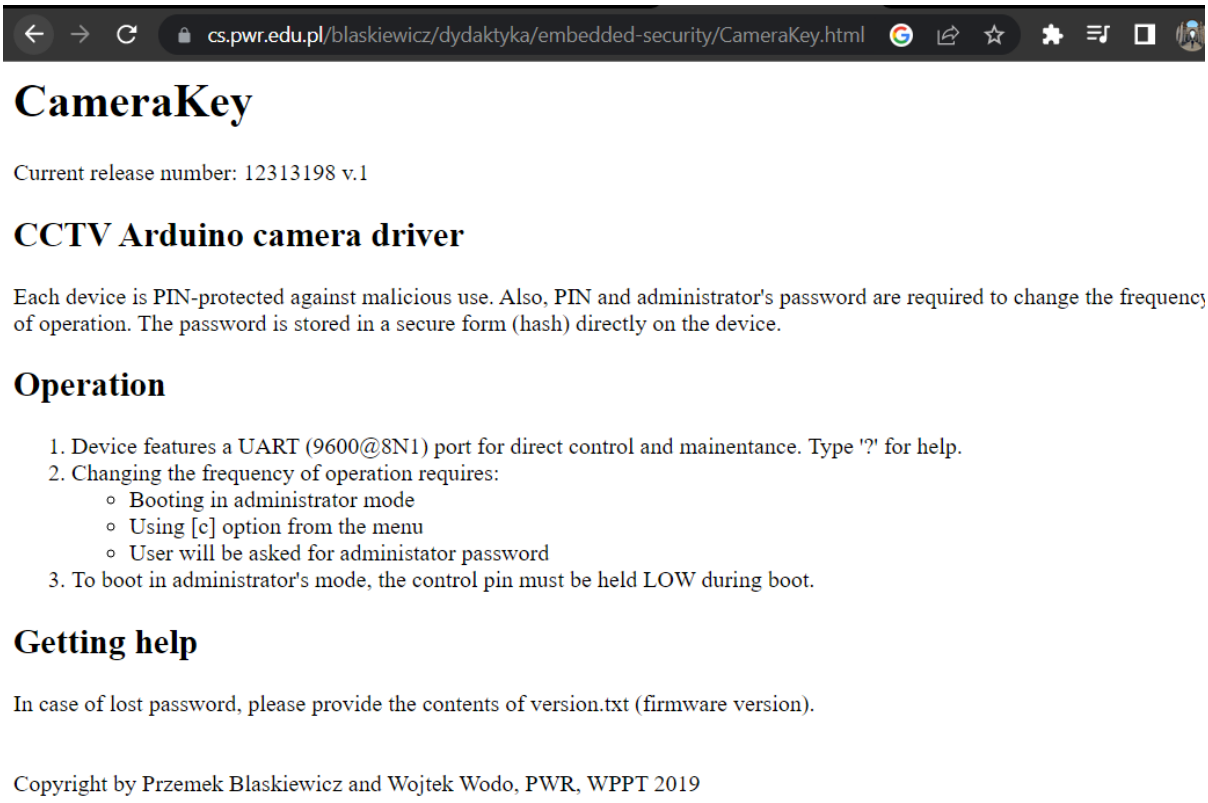

Figure 1.1: Usual suspects

As we still didn't know what the device is supposed to be doing and how should we communicate with it, we thought that the best place to start will be mapping messages to responses, in a hope to discover some undocumented commands.

1.3.2 Mapping dictionary

We wrote a (not so) simple python script for communicating with Arduino programmatically. The idea was to iterate over some dictionary (for starters we used a dictionary of all possible one-byte values) send them, receive a response, measure response time, and save it to the file. As a result, we got the file with inputs to outputs mapping, see picture 1.2. We wrote that script in such a way that providing some arbitrary list of commands would be easy.

```
57  2.017850399017334:6->b''
58  2.021930456161499:7->b''
59  2.014704704284668:8->b''
60  2.0118753910064697:9->b''
61  2.011176109313965::->b''
62  2.019519805908203:;->b''
63  2.0141749382019043:<->b''
64  2.022223949432373:=->b''
65  2.0197510719299316:>->b''
66  0.03825116157531738:?->b'[c]hange frequency <number>\n'
67  0.0466001033782959:@->b'[l]ist files\n'
68  0.031415700912475586:A->b'[?] - help\r\n'
69  2.0134918689727783:B->b''
70  2.022761821746826:C->b''
71  2.021865129470825:D->b''
72  2.021080255508423:E->b''
73  2.0175585746765137:F->b''
74  2.0197269916534424:G->b''
75  2.01838755607605:H->b''
76  2.0108470916748047:I->b''
```

Figure 1.2: A fragment from mapping dictionary.

We planned to add here a complete implementation of the python script that we used for programmatically talking to the Arduino, but after many iterations of new functionalities, it turned out that the code itself is pretty long. Because of that we only add crucial functions with concise descriptions.

The class constructors define the USB interface and baudrate for communication. The main loop of the script on one thread sends commands to the Arduino while the second threat listens for responses, see 1.2.

```
...
def __init__(self):
    self.ser = MySerialWorker('/dev/ttyACM0', baudrate=9600, timeout=0.1)
    self.filename = datetime.now().strftime("%H:%M:%S") + ".csv"
    self.pending_command = False

def main_loop(self):
```

```
    while self.ser.isOpen():
        while self.pending_queries:
            start_time, fd, result_dict, cmd, callback = self.pending_queries.pop(0)
            callback(start_time, fd, result_dict, cmd, self._do_query(cmd))

        if self.ser.inWaiting():
            _ = self.ser.readlines()
        while self.pending_commands:
            self.ser.write(self.pending_commands.pop(0))
        sleep(0.01)
```

Listing 1.2: Python code for getting the longest execution time digits.

The general idea of communication is to provide a list of to-be-send commands to the `send_recv_procedure()`. The commands generation procedure can be anything, for example, the generation of all one-byte commands may look very neat like `generate_all_one_byte_commands()` function, see 1.3.

```
def send_recv_procedure(self, cmds, result_dict=None, fd=None):
    i = 0
    self.pending_command = True
    self.ser.query_command(start_time=0.0, fd=fd, result_dict=result_dict, cmd=" \n",
        ↪ callback=self.on_result)
    sleep(1) # getting init message from arduino
    while True:
        if self.pending_command is False:
            if i >= len(cmds):
                self.ser.terminate()
                break
            self.pending_command = True
            cmd = cmds[i]
            self.ser.query_command(start_time=time(), fd=fd, result_dict=result_dict, cmd
                ↪ =cmd + "\n", callback=self.on_result)
            i += 1
    return result_dict

def generate_all_one_byte_commands():
    cmds = [chr(i) for i in range(256)]
    return cmds
```

Listing 1.3: Python code for getting the longest execution time digits.

In 1.4 we present the list of messages that got not empty responses with their response time.

```
0.03825116157531738:?->b'[c]hange frequency <number>\n'
0.0466001033782959:@->b'[l]ist files\n'
0.031415700912475586:A->b'[?] - help\r\n'
0.03992772102355957:c->b'Denied.\r\n'
0.01966261863708496:d->b'500000\r\n'
0.021085262298583984:l->b'1: version.txt\r\n'
1.0203211307525635:s->b'Denied.\r\n'
```

Listing 1.4: List of messages with not empty responses.

What can we gather from that? (*Information below is partially incorrect, but we will find out about it a bit later, we decided to keep it in the report, in order to show an incremental gain of our knowledge.*)

- `A` – prints information about help, `[?]`.
- `?` – prints information about our target, which is changing the blinking frequency, `[c]`.
- `@` – prints information about listing the files via `[l]`.
- `c` – is used to change frequency and most probably takes some additional parameter `<number>`. But at this point returns `b'Denied.\r\n'`, and we don't have access to it yet.
- `d` – prints number `b'500000\r\n'`, it as a wild guess at that point but maybe it is the frequency of the **d**iode.
- `l` – lists a file name `b'1: version.txt\r\n'`. Maybe `l` defaults to `1`?
- `s` – the most enigmatic command. At this point, we do not know what it does but is the only command that returned in over one second.

We moved to try to get something from the `s` command. We began by appending at the end of the commnand. What all yield `Denied` but interestingly yield shorter times for combinations with digits and "-" character. Then we moved to appending at the beginning of the command.

```
0.03262686729431152:?s->b'[c]hange frequency <number>\n'
0.02085399627685547:@s->b'[d]ump current frequency\r\n'
0.027386903762817383:As->b'[s]how file <number>\r\n'
```

Listing 1.5: List of messages with not empty responses.

That made us realize that we do not parse output correctly. We had to adjust the code so that it doesn't end reading from the input on the first newline character.

1.4 Booting Arduino in admin mode

Thanks to the documentation website, we learned that: *"To boot in administrator's mode, the control pin must be held LOW during boot."*

We didn't know which pin is referenced by "control pin" but during googling we found that some of the microcontrollers use **GPIO0**, **GPIO1**, or **GPIO12** as the pin controlling the boot sequence. It seemed strange to use **GPIO0** or **GPIO1** because they are respectively **RX** and **TX**. SO using cable we connected **GND** pin to **GPIO12** and after boot, as a hello message from Arduino we received a new message, see 1.6. *First try.*

```
b'Enter administration PIN:\n'
```

Listing 1.6: Admin mode hello message.

1.5 Timing attack on PIN

With a manual check, we found out that a PIN has to be at least 3 digits long. What somehow hindered the timing attack. We began by changing the format in which our script saves to the file, to CSV format. We needed it to find out which 3-digit inputs take the longest to return.

The assumption about string comparison We correctly assumed that PIN checking in the Arduino is performed in a naive and insecure way that allows for performing timing attacks. Typically such an equality check function is written to find the first character that strings differ and return. Such implementation is not time constant and allows, by comparing execution time, to find correct input, see listing 1.7.

```
function check_string_equality(string a, string b) {
  for (i = 0; i < b.length; i++)
    if a[i] != c[i]
      return false;
  return true;
}
```

Listing 1.7: Insecure way of checking string equality that allows timing attack.

Using our IDE we ordered the received CSV file by execution time, see picture 1.3. Thanks to that we can see that the most promising PINs begin `012`. We prepended `012` to our already written PIN generation function achieving code that produces a combination that allowed us to find the correct PIN, see picture 1.4. The command appended two bytes at the end, thus we end up with 5 digit PIN `01240`, but the correct one is `0124`, apparently, the check function does not even check the length of the PIN.

Although we managed to find the PIN the other way if the PIN was bigger it would be very impractical. So we implemented a proper timing attack using our `talker.py` script, see listings 1.8, 1.9, 1.10. We provide complete code because the task is not hard in theory, but can be rather inconvenient in managing data. We have used dictionaries to store return time in the `execution_time` field. Later we use `max()` function to grab the command that took the longest to return. It's also important to notice that attack is divided into two parts, the first one is for finding the first 3 digits (because Arduino waits for 3 characters), and the second one is for finding the following digits. We wrote this function in such a way that would work for arbitrary PIN lengths.

```
@staticmethod
def get_longest_digits(result_dict):
    longest_evaluation_digits = max(result_dict, key=lambda x: result_dict[x]["exec_time"
        ↪ ])
    return longest_evaluation_digits
```

Listing 1.8: Python code for getting the longest execution time digits.

```
def perform_timing_attack(self, pin_length: int):
    # grabbing the "Enter administration PIN"."
    self.pending_command = True
    self.ser.query_command(start_time=0.0, fd=None, result_dict=None, cmd=" \n",
        ↪ callback=self.on_result)
```

	time ▾ 1	cmd	result
1	0.20710515975952148	000	[b'Enter administration PIN: \r\n']
2	0.19073271751403809	012	[b'Enter administration PIN: \r\n']
3	0.18768000602722168	015	[b'Enter administration PIN: \r\n']
4	0.18750524520874023	018	[b'Enter administration PIN: \r\n']
5	0.18501663208007812	011	[b'Enter administration PIN: \r\n']
6	0.18425583839416504	013	[b'Enter administration PIN: \r\n']
7	0.18405652046203613	014	[b'Enter administration PIN: \r\n']
8	0.18310785293579102	017	[b'Enter administration PIN: \r\n']
9	0.18208646774291992	010	[b'Enter administration PIN: \r\n']
10	0.18008041381835938	016	[b'Enter administration PIN: \r\n']
11	0.1777803897857666	019	[b'Enter administration PIN: \r\n']
12	0.17235183715820312	031	[b'Enter administration PIN: \r\n']
13	0.17168354988098145	076	[b'Enter administration PIN: \r\n']
14	0.17167186737060547	022	[b'Enter administration PIN: \r\n']
15	0.17140793800354004	096	[b'Enter administration PIN: \r\n']
16	0.17139577865600586	004	[b'Enter administration PIN: \r\n']
17	0.1712799072265625	090	[b'Enter administration PIN: \r\n']
18	0.17125582695007324	053	[b'Enter administration PIN: \r\n']
19	0.17109346389770508	037	[b'Enter administration PIN: \r\n']
20	0.17108368873596191	046	[b'Enter administration PIN: \r\n']

Figure 1.3: A fragment from mapping dictionary.

```
sleep(1) # getting init message from arduino

# first phase (3 digits PINs)
timing_result_dict = {}
for combination in list(product(range(0, 10), repeat=3)):
    cmd = "".join(str(x) for x in combination)
    self.pending_command = True
    self.ser.query_command(start_time=time(), fd=None, result_dict=
        ↪ timing_result_dict, cmd=f"{cmd}\n",
                               callback=self.on_result)
    sleep(0.2) # reading thread has to wait for arduino to process input
first_3_digits = self.get_longest_digits(result_dict=timing_result_dict)
...
```

Listing 1.9: Python code for performing the timing attack – phase 1.

```
# second phase (following digits)
known_digits = first_3_digits
timing_result_dict = {}
for i in range(pin_length):
    for digit in range(0, 10):
        self.pending_command = True
        self.ser.query_command(start_time=time(), fd=None, result_dict=None, cmd=f
            ↪ "{known_digits}{digit}\n",
                                   callback=self.on_result)
        sleep(0.2)
```

```
236 0.21925830841064453,01234,[b'Enter administration PIN: \r\n']
237 0.21548795700073242,01235,[b'Enter administration PIN: \r\n']
238 0.22265267372131348,01236,[b'Enter administration PIN: \r\n']
239 0.218977689743042,01237,[b'Enter administration PIN: \r\n']
240 0.21665692329406738,01238,[b'Enter administration PIN: \r\n']
241 0.21602559089660645,01239,[b'Enter administration PIN: \r\n']
242 0.2560276985168457,01240,[b'CameraKey Driver - 2019\r\n', b'*** ADMIN MODE ***\r\n']
243 0.11504483222961426,01241,[]
244 0.11458468437194824,01242,[]
245 0.11399602890014648,01243,[]
246 0.11354660987854004,01244,[]
```

Figure 1.4: A fragment from mapping dictionary.

```
        next_digit = self.get_longest_digits(result_dict=timing_result_dict)
        known_digits = known_digits + next_digit
    print(f"{known_digits=}")
```

Listing 1.10: Python code for performing the timing attack – phase 2.

1.6 One step further - Automating the timing attack

Following a similar approach, we were able to fully automatize the timing attack on the PIN.

Once the user boots the device in Admin mode, the C# program below will ask the user the port number, then proceed to trying 3-digit PINs to login, increasing the number of digits as they are exhausted.

```
private static SerialPort port;
private static TimingManager timingAttacker = new TimingManager();

static void Main(string[] args)
{
    Console.Write("Enter port number: ");
    var portNumber = Convert.ToInt32(Console.ReadLine());
    port = new SerialPort($"COM{portNumber}",
        9600, Parity.None, 8, StopBits.One);
    port.DataReceived += new SerialDataReceivedEventHandler(
        ↪ port_DataReceived);
    port.Open();

    while (port.IsOpen)
    {
        var message = timingAttacker.GetNext();
        Console.WriteLine($"Trying PIN {message}");
        timingAttacker.StartTiming();
        port.WriteLine(message);
        Thread.Sleep(TimeSpan.FromSeconds(1));
    }
```

```
    Console.WriteLine("Enter any key to exit..");
    Console.ReadLine();
    Environment.Exit(0);
}

private static void port_DataReceived(object sender,
    ↪ SerialDataReceivedEventArgs e)
{
    var receivedMessage = port.ReadExisting();
    if (!receivedMessage.StartsWith("Enter administration PIN"))
    {
        Console.WriteLine($"Found PIN: {timingAttacker.GetCurrentPIN()}")
            ↪ ;
        port.Close();
        return;
    }
    timingAttacker.Received(receivedMessage);
}
```

Listing 1.11: Automated timing attack - contents of Program.cs

The contents of the actual class that counts the time and handles what PIN to try next is given below.

```
private readonly Stopwatch _stopwatch = new Stopwatch();
private int _nextDigit = 0;
private string _correctDigits;
private static int PIN_DIGITS = 3;
private readonly Dictionary<int, long> _digitTimings;

public TimingManager()
{
    _correctDigits = string.Empty;
    _digitTimings = new Dictionary<int, long>();
}

public void Received(string message)
{
    StopTiming();
    _digitTimings.Add(_nextDigit, _stopwatch.ElapsedMilliseconds);
    ResetTiming();
    _nextDigit++;

    if (_nextDigit == 10)
    {
        int correctIndex = FindCorrectIndex(_digitTimings.Values.ToList()
            ↪ );
        _correctDigits += correctIndex;
        _digitTimings.Clear();
```

```
        _nextDigit = 0;

        if (_correctDigits.Length == PIN_DIGITS)
        {
            PIN_DIGITS++;
        }
    }
}

public string GetNext()
    => _correctDigits
       + (_nextDigit)
       + new string('0', PIN_DIGITS - _correctDigits.Length - 1);

public string GetCurrentPIN() => _correctDigits + _nextDigit;

public void StartTiming() => _stopwatch.Start();

private static int FindCorrectIndex(IList<long> timings)
    => timings.IndexOf(timings.MaxBy(t => t));

private void StopTiming() => _stopwatch.Stop();

private void ResetTiming() => _stopwatch.Reset();
```

Listing 1.12: Automated timing attack - contents of TimingManager.cs

When this console application is run, it gives the output below:

```
Enter port number: 6
Trying PIN 000
Trying PIN 000
Trying PIN 100
Trying PIN 200
Trying PIN 300
Trying PIN 400
Trying PIN 500
Trying PIN 600
Trying PIN 700
Trying PIN 800
Trying PIN 900
Trying PIN 000
Trying PIN 010
Trying PIN 020
Trying PIN 030
Trying PIN 040
Trying PIN 050
Trying PIN 060
Trying PIN 070
Trying PIN 080
```

```
Trying PIN 090
Trying PIN 010
Trying PIN 011
Trying PIN 012
Trying PIN 013
Trying PIN 014
Trying PIN 015
Trying PIN 016
Trying PIN 017
Trying PIN 018
Trying PIN 019
Trying PIN 0120
Trying PIN 0121
Trying PIN 0122
Trying PIN 0123
Trying PIN 0124
Found PIN: 0124
Enter any key to exit..
```

Listing 1.13: Output of the console application

The application successfully finds the correct PIN by inspecting the response times for each attempt, and then checking the response from the Arduino.

1.7 Breaking the admin password

After finding the correct pin. We invoked `c <number>` command, just to find that we require an admin password. But we could print the content of two files present in the device.

```
s 1 -> 1.234e //version.txt
s 2 -> 87078152d2b9449738e69e0afec0c03b //passwd
```

Listing 1.14: Repsonses for s command.

- **1.234e** – is the most enigmatic line in the whole task.
- **87078152d2b9449738e69e0afec0c03b** – most probably is the MD5 hash of the password. Most likely password is being checked by being hashed to MD5 and checked against the content of the `passwd` file. Unfortunately, checking rainbow tables for a match yielded no results.

We started by checking how the device responds to different lengths of passwords. This time Arduino accepted all lengths.

1.7.1 Many dead-ends

Breaking the password caused the biggest problems.

What is the answer to everything? We began by running the brute-force attack on the admin password. But sadly we found that the device bricks after 42 incorrect attempts. Therefore already difficult setup procedure would become even harder and what's more important – more time-consuming, if we wanted to brute-force the password in such circumstances.

Quick command injection We had an idea that sending two `c` commands (so that the second is sent before the device responds with the admin password question) would somehow leverage the admin password check. Sadly it didn't work.

1.7.2 Trying out Hashcat

`Hashcat` is an open source password cracker, suitable for deducing the correct password from a given hash, and/or a salt value.[2] Once downloaded, using the `--help` switch gives us a very detailed set of instructions. Here are some switches of interest:

```
hashcat (v6.2.6) starting in help mode

Usage: hashcat [options]... hash|hashfile|hccapxfile [dictionary|mask|
   ↪ directory]...

- [ Options ] -

 Options Short / Long | Type | Description | Example
==============================+======+============================
 -m, --hash-type | Num | Hash-type, references below (otherwise
    ↪ autodetect) | -m 1000
 -a, --attack-mode | Num | Attack-mode, see references below | -a 3
 -i, --increment | | Enable mask increment mode |
```

Listing 1.15: Hashcat help prompt.

Another instruction of interest is regarding the available attack modes and hash types:

```
- [ Hash modes ] -

      # | Name | Category
  ======+============================================================
    900 | MD4 | Raw Hash
      0 | MD5 | Raw Hash
     70 | md5(utf16le($pass)) | Raw Hash
    170 | sha1(utf16le($pass)) | Raw Hash
   1470 | sha256(utf16le($pass)) | Raw Hash
  10870 | sha384(utf16le($pass)) | Raw Hash
   1770 | sha512(utf16le($pass)) | Raw Hash
    610 | BLAKE2b-512($pass.$salt) | Raw Hash salted and/or iterated
    620 | BLAKE2b-512($salt.$pass) | Raw Hash salted and/or iterated
     10 | md5($pass.$salt) | Raw Hash salted and/or iterated
     20 | md5($salt.$pass) | Raw Hash salted and/or iterated
   3800 | md5($salt.$pass.$salt) | Raw Hash salted and/or iterated
```

```
   3710 | md5($salt.md5($pass)) | Raw Hash salted and/or iterated
   4110 | md5($salt.md5($pass.$salt)) | Raw Hash salted and/or iterated
   4010 | md5($salt.md5($salt.$pass)) | Raw Hash salted and/or iterated
  21300 | md5($salt.sha1($salt.$pass)) | Raw Hash salted and/or iterated
     40 | md5($salt.utf16le($pass)) | Raw Hash salted and/or iterated
   2600 | md5(md5($pass)) | Raw Hash salted and/or iterated
   3910 | md5(md5($pass).md5($salt)) | Raw Hash salted and/or iterated
   3500 | md5(md5(md5($pass))) | Raw Hash salted and/or iterated
   4400 | md5(sha1($pass)) | Raw Hash salted and/or iterated
   4410 | md5(sha1($pass).$salt) | Raw Hash salted and/or iterated
  20900 | md5(sha1($pass).md5($pass).sha1($pass)) | Raw Hash salted and/
     ↪ or iterated
  21200 | md5(sha1($salt).md5($pass)) | Raw Hash salted and/or iterated
   4300 | md5(strtoupper(md5($pass))) | Raw Hash salted and/or iterated
     30 | md5(utf16le($pass).$salt) | Raw Hash salted and/or iterated


- [ Attack Modes ] -

  # | Mode
 ===+======
  0 | Straight
  1 | Combination
  3 | Brute-force
  6 | Hybrid Wordlist + Mask
  7 | Hybrid Mask + Wordlist
  9 | Association
```

Listing 1.16: Hash types and attack modes

1.7.3 Dictionary attack

We have found extensive common password lists such as the one compromised after the company RockYou was hacked, namely rockyou [3] as well as CrackStation Password Cracking Dictionary [4]. Then we tried the commands below, hoping that one of the words in these dictionaries contains the password corresponding to our hash.

```
hashcat -a 1 -m 0 87078152d2b9449738e69e0afec0c03b [dictionaryFileName]
hashcat -a 3 -m 0 87078152d2b9449738e69e0afec0c03b [dictionaryFileName]
```

Unfortunately this didn't work either.

1.7.4 Firmware version as salt?

Here we had a flashback to the time when we googled the `CameraKey Driver`, specifically to this sentence:

```
In case of lost password, please provide the contents of version.txt
(firmware version).
```

This made us think it's probably part of the password, or perhaps the salt of the password hash.

Fortunately, `HashCat` has couple of hash modes we could try:

```
   10 | md5($pass.$salt) | Raw Hash salted and/or iterated
   20 | md5($salt.$pass) | Raw Hash salted and/or iterated
 3800 | md5($salt.$pass.$salt) | Raw Hash salted and/or iterated
```

But unfortunately these too did not yield any result.

1.7.5 HashCat - mask increment mode

We were sure the firmware version is part of the password, so we utilized a Regex-like feature of HashCat where we can enter a fixed part of a password, and specify unknown character locations and their types, and the utility does the rest.

After many tries, this was the command that yielded a surprise:

```
hashcat -m 0 -a 3 87078152d2b9449738e69e0afec0c03b -i 1.234e?u?u?u
```

The password turned out to be `1.234ePWR`:

```
87078152d2b9449738e69e0afec0c03b:1.234ePWR

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 0 (MD5)
Hash.Target......: 87078152d2b9449738e69e0afec0c03b
Time.Started.....: Mon Jan 30 23:00:17 2023 (0 secs)
Time.Estimated...: Mon Jan 30 23:00:17 2023 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Mask.......: 1.234e?u?u?u [9]
Guess.Queue......: 9/9 (100.00%)
Speed.#1.........: 243.5 kH/s (0.01ms) @ Accel:2048 Loops:1 Thr:32 Vec:1
Speed.#2.........: 467.6 kH/s (0.03ms) @ Accel:1024 Loops:1 Thr:32 Vec:1
Speed.#*.........: 711.1 kH/s
Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (
   ↪ new)
Progress.........: 8096/17576 (46.06%)
Rejected.........: 0/8096 (0.00%)
Restore.Point....: 6848/17576 (38.96%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Restore.Sub.#2...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidate.Engine.: Device Generator
Candidates.#1....: 1.234eCQP -> 1.234eYLK
Candidates.#2....: 1.234eKTU -> 1.234eAUG
Hardware.Mon.#1..: Temp: 0c Util: 0% Core:1136MHz Mem:2505MHz Bus:16
Hardware.Mon.#2..: N/A

Started: Mon Jan 30 22:59:50 2023
Stopped: Mon Jan 30 23:00:19 2023
```

2 | Final remarks

2.1 List 4

In this list we were not able to inspect or decompile the source code of the application uploaded to the embedded environment, which surely limited our possibilities.

First, we implemented a brute force technique as well as open source intelligence to understand the target software we are dealing with and how we can interact with it.

Then, we implemented a timing attack which easily cracked the administrator PIN. This was a security flaw deliberately left on the application, which might have been prevented in real life by comparing all characters of entered PIN and actual PIN before returning a response.

The hardest part of the problem was figuring out the password, given only a hash and a firmware version string. Luckily there were powerful password cracking applications around such as `HashCat`. One takeaway for us was that generic password dictionaries were useless - we could have benefited more from building a context-based custom dictionary specifically for this attack. While the unknown part of the password turned out to be **PWR** for this problem, it could have been the initials of the company which produced the application, or some other keyword that revolves around their specific line of work or products.

Bibliography

[1] *Avrdude*, https://github.com/avrdudes/avrdude, Accessed: 15-01, 2023.

[2] *Hashcat*, https://hashcat.net/hashcat/, Accessed: 30-01, 2023.

[3] *Common Password List (rockyou.txt)*, https://www.kaggle.com/datasets/wjburns/common-password-list-rockyoutxt, Accessed: 30-01, 2023.

[4] *CrackStation's Password Cracking Dictionary*, https://crackstation.net/crackstation-wordlist-password-cracking-dictionary.htm, Accessed: 30-01, 2023.

[5] *Atmega 328p*, https://botland.com.pl/avr-w-obudowie-tht/1264-mikrokontroler-avr-atmega328p-u-dip-5903351249928.html, Accessed: 15-01, 2023.