

kaspersky

QR code SQL injection and other vulnerabilities in a popular biometric terminal

Georgy Kiguradze

Contents

A brief overview of biometric terminals	3
A brief overview of the device in question.....	5
Black box analysis.....	9
Circuit analysis	9
Network analysis	12
Camera and QR code scanner analysis	14
Getting and unpacking the firmware	15
Searching the web for the firmware	15
Getting the firmware from the flash memory	21
Analyzing the protocol on port 4370/TCP	25
Protocol authentication and its issues.....	25
Vulnerability analysis of command handlers	27
pushcomm analysis.....	31
QR code handler analysis	33
Conclusion	34

QR code SQL injection and other vulnerabilities in a popular biometric terminal

Biometric scanners offer a unique way to resolve the conflict between security and usability. They help to identify a person by their unique biological characteristics – a fairly reliable process that does not require the user to exert any extra effort. Yet, biometric scanners, as any other tech, have their weaknesses. This article touches on biometric scanner security from the red team's perspective and uses the example of a popular hybrid terminal model to demonstrate approaches to scanner analysis. These approaches are admittedly fairly well known and applied to analysis of any type of device.

We also talk about the benefits of biometric scanners for access control systems and their role in ensuring a due standard of security given today's realities. Furthermore, we discuss vulnerabilities in a biometric scanner from a major global vendor that we found while analyzing its level of security. The article will prove useful for both security researchers and architects.

We have notified the vendor about all the vulnerabilities and security issues we found. A CVE entry has been registered for each of the vulnerability types: [CVE-2023-3938](#), [CVE-2023-3939](#), [CVE-2023-3940](#), [CVE-2023-3941](#), [CVE-2023-3942](#).

A brief overview of biometric terminals

In a security context, biometric terminals are used for personal identification. They rely on the analysis of unique human physical characteristics, such as fingerprints, voice, facial features, or the iris.

Importantly, though, a biometric terminal is somewhat different from a regular scanner. First, it can both acquire biometric data and validate it. Second, terminals can be connected to other scanners, such as electronic pass readers, or support other authentication methods using built-in hardware.

Their main purpose is to control access to an area or site. As such, they can be used for restricting access to premises that house confidential data, such as a server room or executive office, or to control access to hazardous facilities, such as a nuclear power or chemical plant.

Another application is recording employees' work hours to improve productivity and reduce the likelihood of successful fraud.

In terms of security, biometric terminals can be said to offer the following benefits:

- **Highly accurate identification:** biometric data is unique to each human being, which makes it a reliable way of identity verification.
- **Secure:** biometric data is difficult to forge or copy, which increases system security.
- **User-friendly:** biometric identification does not require subjects to remember passwords or carry access cards.
- **Efficiency:** biometric terminals can process large amounts of data fast to reduce wait times.

These devices are not without their downsides, though.

- **Cost:** biometric terminals are typically more expensive than traditional access control systems.
- **Risk of error:** although biometric data is unique, in some cases, systems have misidentified individuals who had damaged fingertips, etc.
- **Privacy:** some may have concerns about their biometric data being stored and used without their consent.
- **Technological limitations:** some biometric identification methods (such as face recognition) can be less efficient under low light conditions, when the subject is wearing a mask, etc.

Biometric terminals are quite an intriguing target for a pentester. Vulnerabilities in these devices, positioned at the nexus of the physical and network perimeters, pose risks that can be considered when analyzing the security of both these perimeters.

Some of the goals that can be achieved in terms of offensive security are:

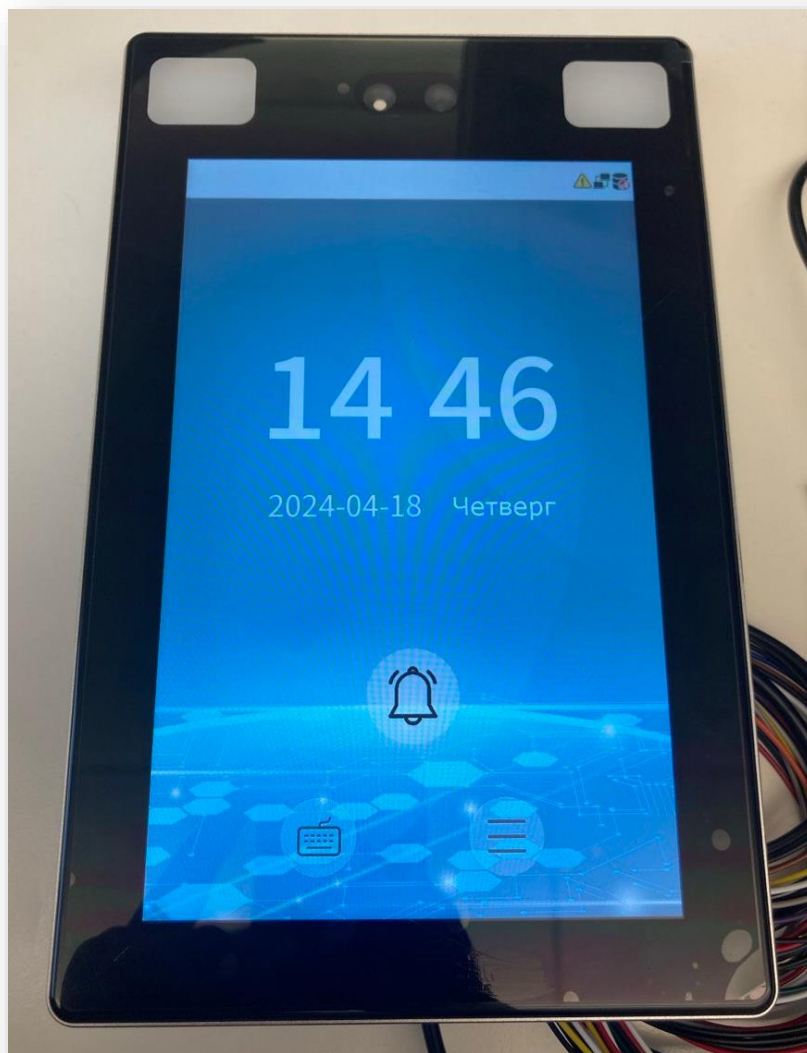
- Authentication bypass and physical access violation
- Biometric data leak

- Gaining network access to a device and exploiting that to further develop the attack

Now that we have defined the biometric terminal, its applications, benefits and downsides, and security analysis objectives associated with it, we can move on to analyzing a specific device.

A brief overview of the device in question

The device under review is a hybrid biometric terminal made by ZkTeco. It may come under various names depending on the distributor. You can see its external appearance in the photograph below.



External appearance of the device

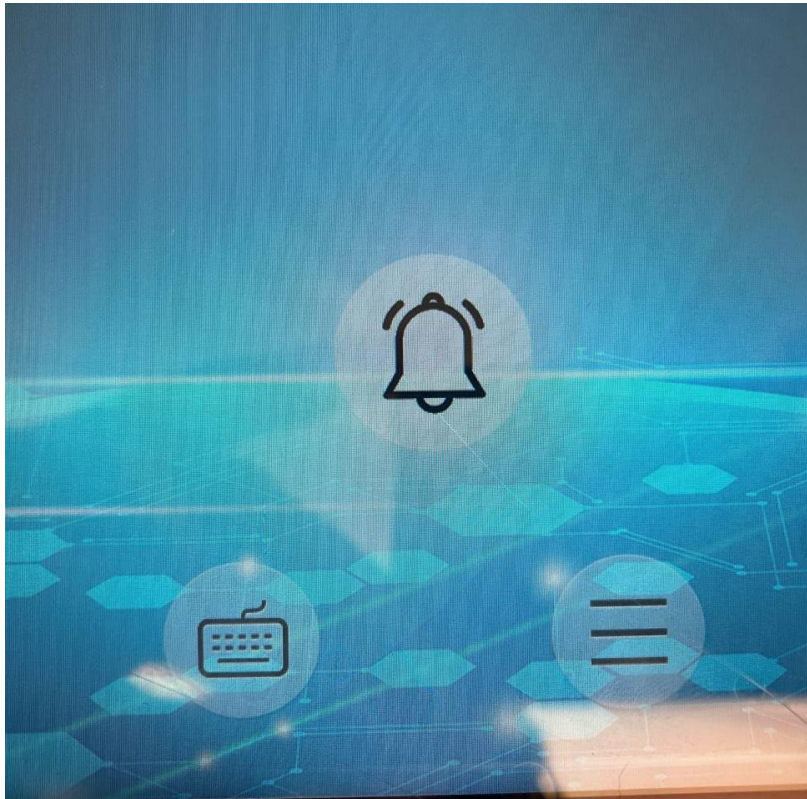
The device has several physical interfaces, supporting four authentication methods: biometric (facial recognition), password, electronic pass, and QR code.

The following physical interfaces are present:

- RJ45

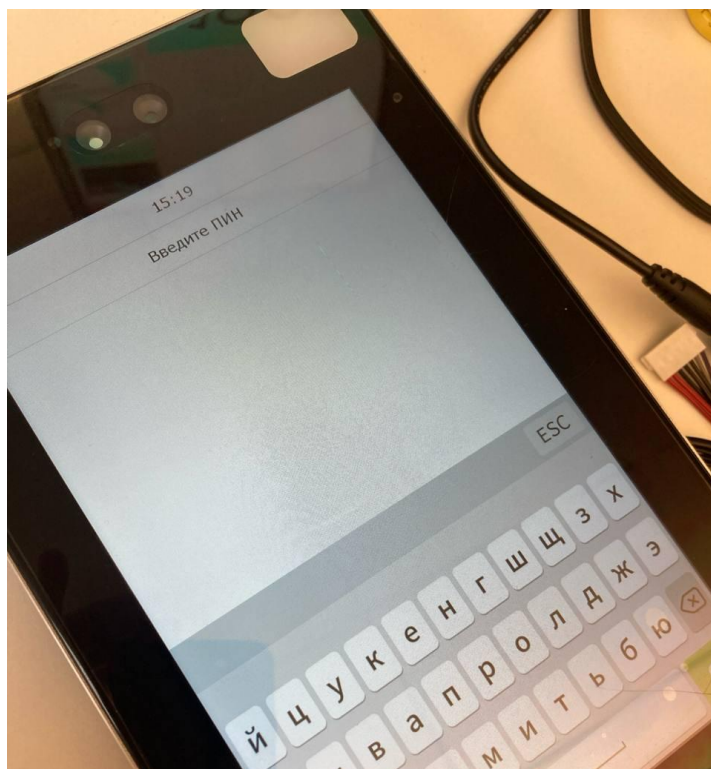
- RS232
- RS485 (unused)
- Wiegand In/Out

A regular (non-privileged) user has few options in terms of interacting with the device: they can only tap one of the two on-screen buttons that you can see in the photograph below.



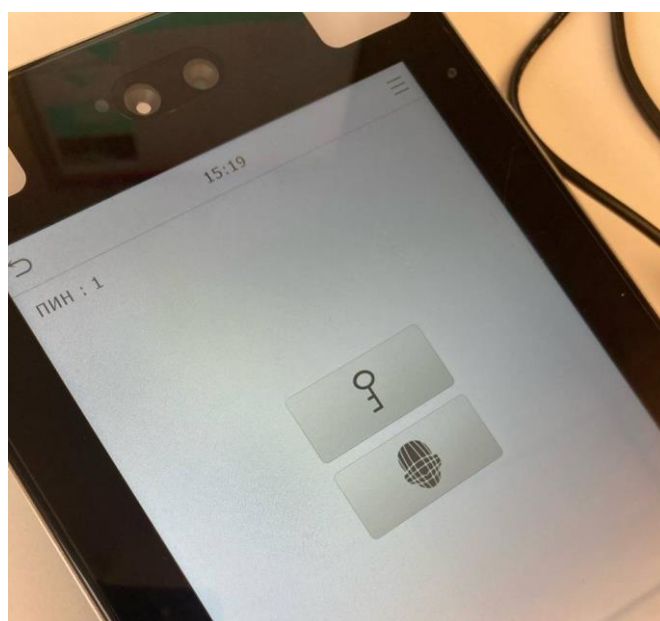
Available touchscreen buttons

Tapping a button brings up a prompt for PIN, which is the user's unique ID in our case.



User ID input interface

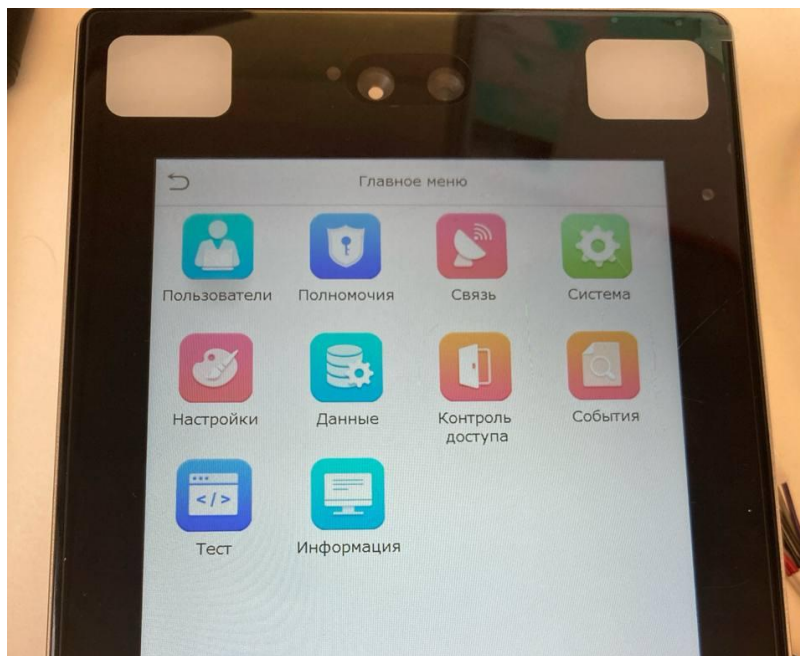
If a valid (existing) ID is entered, the screen displays available user-specific authentication options. The example shows a user with the ID 1 and two authentication methods: biometrics and password.



Authentication methods available to the user with the ID 1

That is the extent of what a non-administrator or unauthenticated user can do with the terminal.

The options available to an administrator are more interesting. With administrator privileges, we can control nearly all of the device settings. The image below shows the maximum-access menu.



Administrator's device setup menu

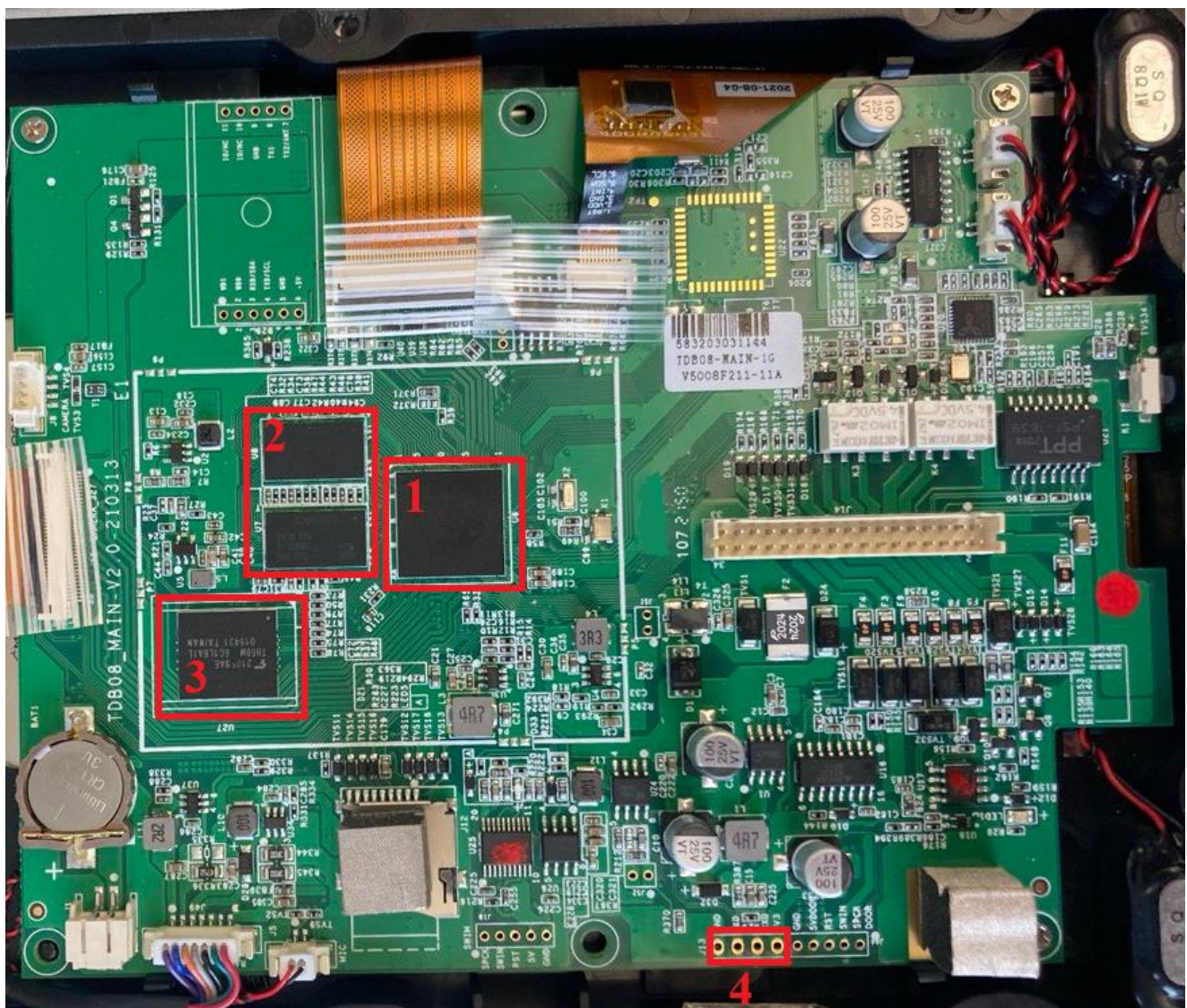
The administrator menu can be used to add new users, manage their levels of access, and change the network and facial scanner settings. As you will see below, administrator access allows for achieving all of the security analysis objectives listed in the previous section. Getting that level of access requires passing authentication as an administrator.

Black box analysis

Circuit analysis

Our engineering analysis will begin with black box analysis, and namely, circuit analysis. The photograph below shows the circuit board with the following components that we are interested in.

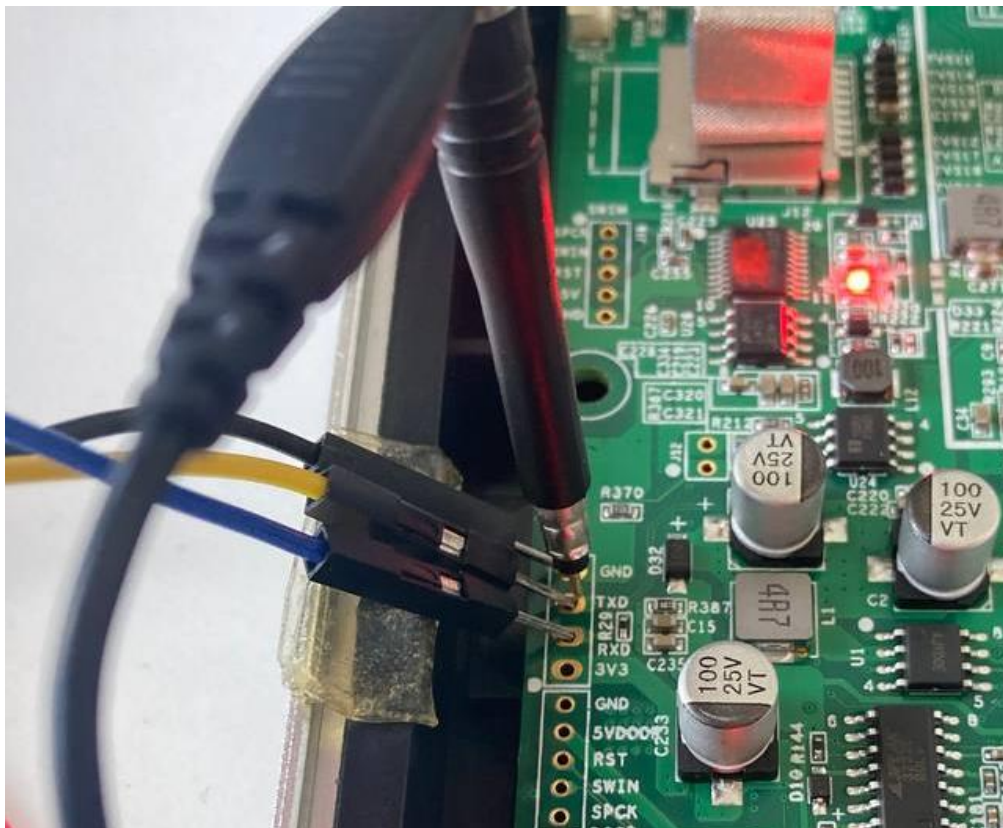
- 1. SOC (HI 3516 DV300)
- 2. RAM (K4B4G16E-BCMA, 4Gb)
- 3. Flash memory (THGBMJG6C1LBAI, 8Gb, BGA-153)
- 4. UART



Circuit board

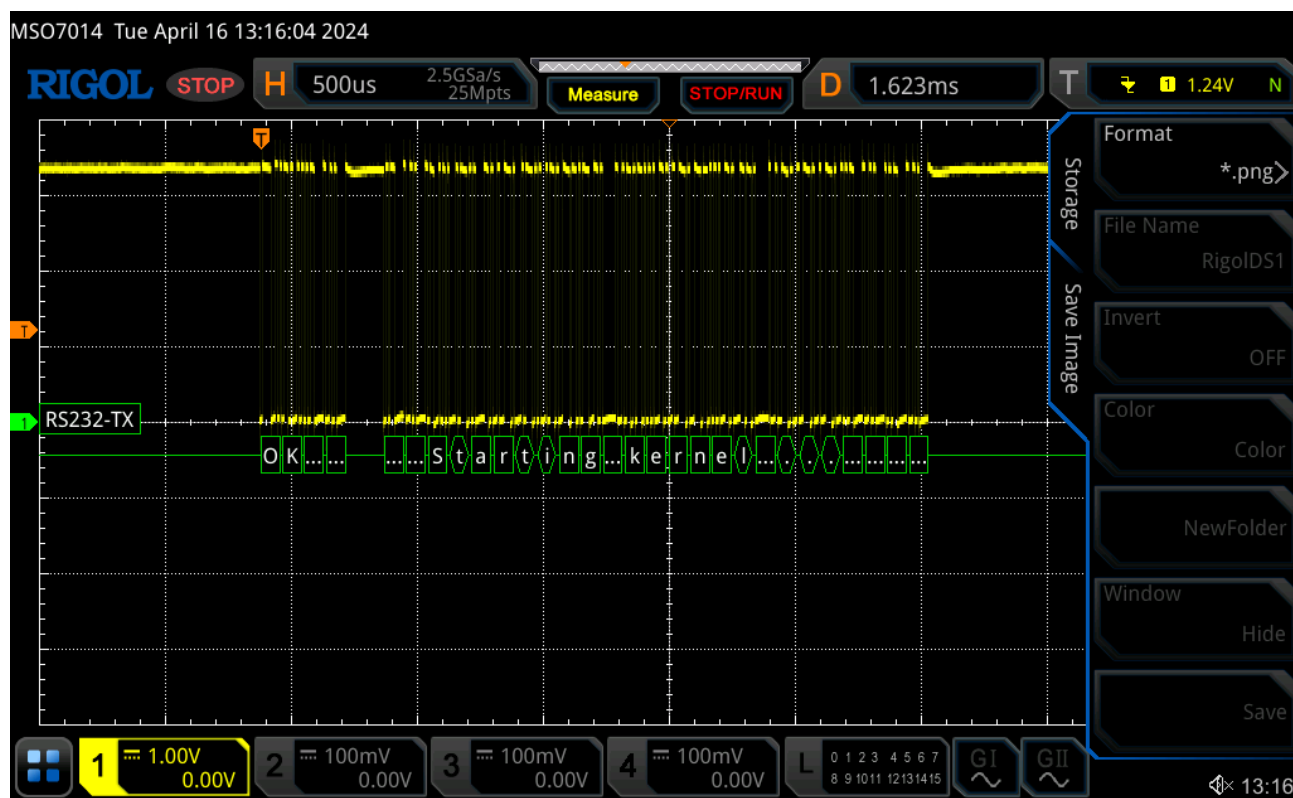
You may notice that the circuit board has many [test points](#). That said, we are only interested in the ones marked with the number 4, as those are the location of a [universal asynchronous receiver-transmitter \(UART\)](#) that we can use to communicate with the device. The flash memory, marked with the number 3, is of interest as well, as it holds the entire firmware in unencrypted form.

To check that we had recognized the UART correctly, we used an oscilloscope to connect to what we had identified as the TX port through which the device sends data externally.



Oscilloscope connection to UART

After calculating the UART data rate and setting the oscilloscope to that value, we saw that this was indeed a UART, and the device was sending a boot log through it.



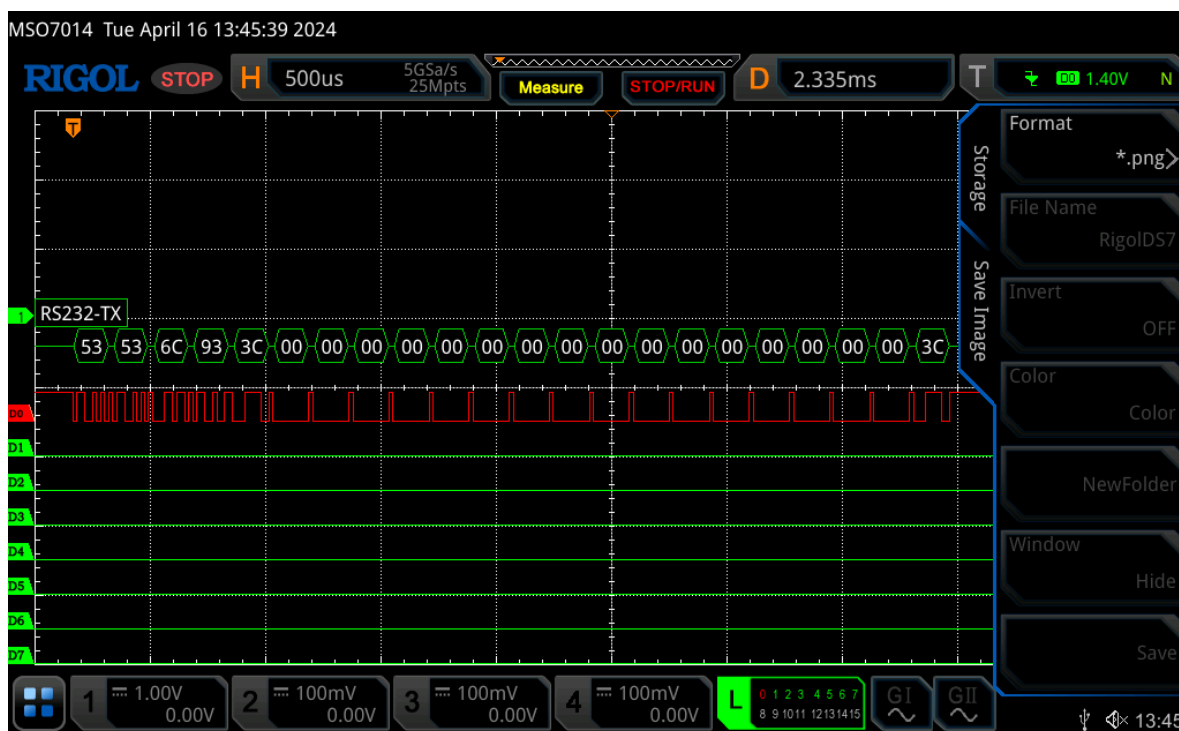
Boot log

Next, we connected to the UART using a PC, which helped us to view the full boot log and identify the bootloader as U-Boot.

```
System startup
Uncompress Ok!
U-Boot 2016.11-svn1034 (Aug 10 2020 - 14:57:32 +0800)
Relocation Offset is: 0f6c3000
Relocating to 8fec3000, new gd at 8fe22ef0, sp at 8fe2
```

UART connection from a PC

The bootloader configuration prevents any attempts at interrupting startup (bootdelay = -2) or interacting with it in any other way. However, having waited some time after the device booted up, we found that the UART switched to a different baud (bits per second) rate of 115,200 from 57,600 as the device began to send uniform packets, which suggested the use of an unknown protocol.



The unknown protocol as used by the UART

Every packet began with a 0x53 0x53 byte, and the fifth byte was always identical to the final one. An online search for these two brought up nothing. Sending similarly formatted packets to the device yielded nothing, either.

Network analysis

Another type of black box analysis is scanning network ports. We can use Nmap, a publicly available network scanner utility, to see which ports are open, and try to identify the services running on these and their versions. The screenshot below shows the TCP ports open on the biometric terminal.

```

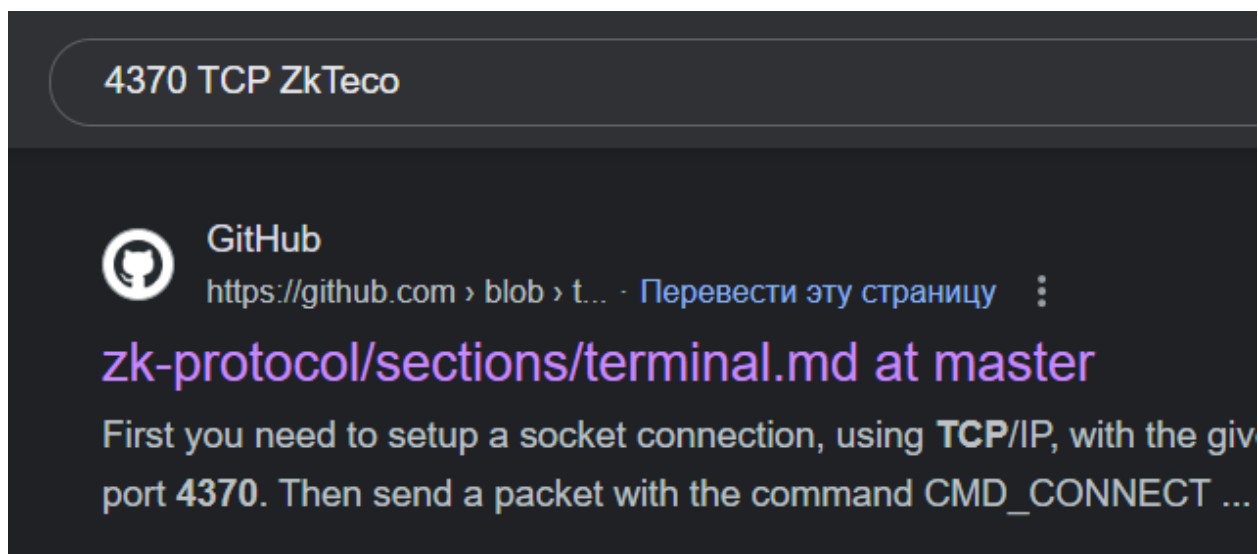
→ ~ sudo nmap -p0-65535 -sV -sS 192.168.1.201
Starting Nmap 7.80 ( https://nmap.org ) at 2024-04-
Nmap scan report for 192.168.1.201
Host is up (0.0012s latency).
Not shown: 65533 closed ports
PORT      STATE SERVICE      VERSION
3718/tcp  open  ssh          Dropbear sshd 2018.76
4370/tcp  open  elpro_tunnel?
6668/tcp  open  irc?
MAC Address: 00:17:61:12:2B:06 (Private)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

```

Open ports

You may notice that the device supports SSH on a non-standard port. In theory, we could connect to that if we get hold of the right credentials. We could potentially extract those from the firmware by using a dictionary attack or brute-forcing the password hash.

Besides, there were two services that could not be identified automatically. The service running on port 6668/TCP was Tuya Server, but we could not find out its purpose. The service running on port 4370/TCP was more interesting as it used the vendor's proprietary protocol supported by many of its devices. After searching the web for the protocol, we found that there was [documentation available](#), making our analysis much easier.



Searching for the protocol on port 4370/TCP

Camera and QR code scanner analysis

Our overview of the device mentions that it supports QR code authentication. We decided to see what happened if a code we presented to the device contained invalid data that could disrupt the processing logic. We were able to achieve a result by making the device scan a QR code that contained malicious SQL code.

A basic SQL injection resulted in the device recognizing us as a valid user.

We further noticed that making the device scan a QR code containing 1 KB of data or more caused it to go into an emergency reboot, which suggested that some of its components had experienced overflow. More on this in the reverse engineering and firmware analysis section.

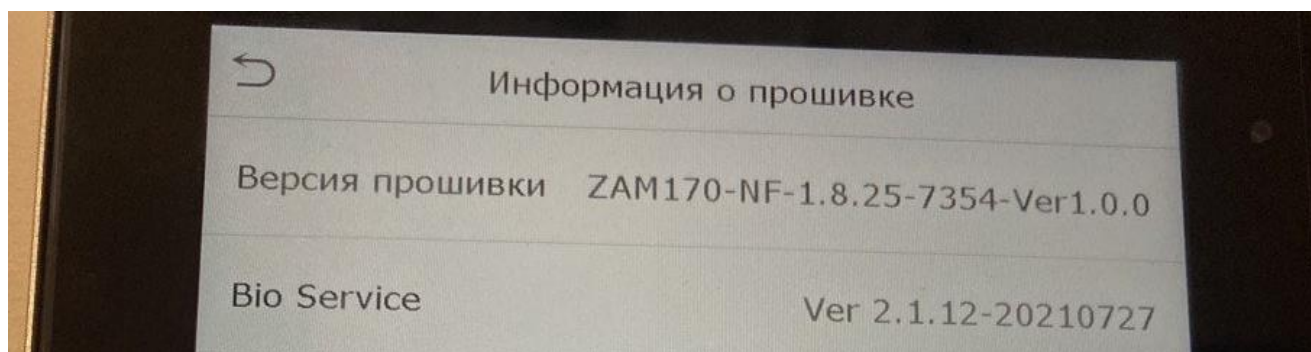
Getting and unpacking the firmware

The vendor's website will not let just anyone download the latest version of the firmware. You can download a PDF file containing the update algorithm, but it is protected with a password that we could not find on any public websites.

Therefore, we had two options for obtaining the firmware: removing the flash memory and dumping it with a programmer, or trying to find a copy on the web.

Searching the web for the firmware

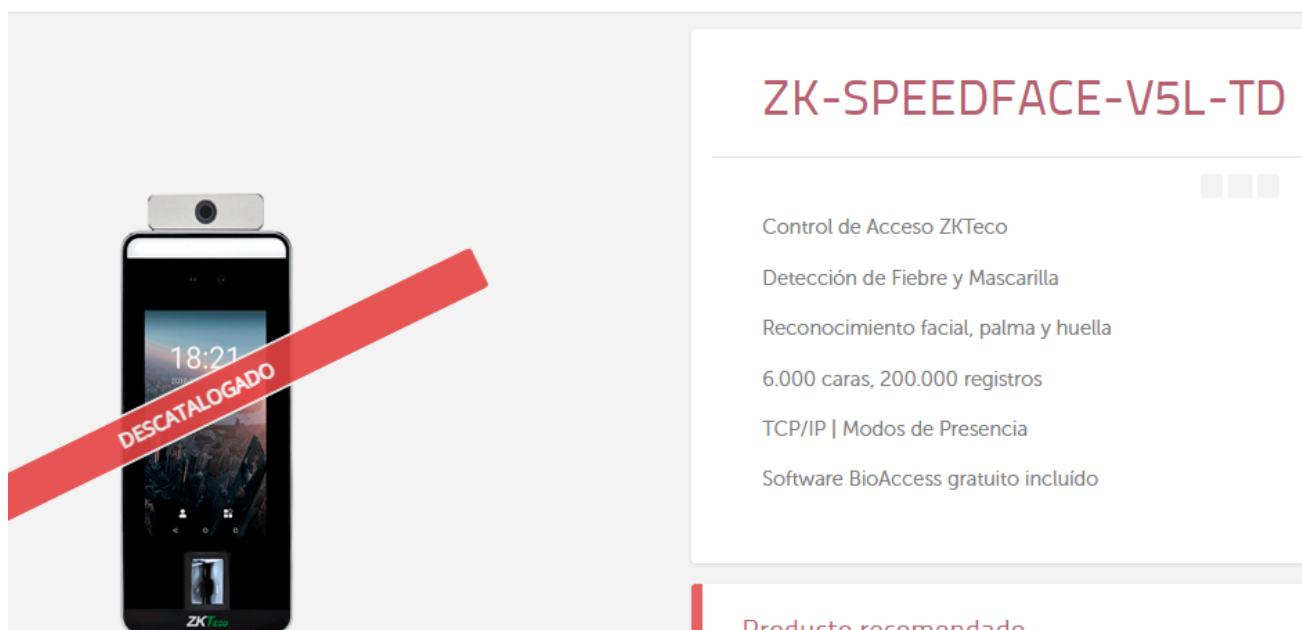
To start searching for the firmware, we needed to find out its name and rough version. We were analyzing an unused device fresh out of the box, so we had administrator access to it. Therefore, we could view the device details and find the current firmware version.



Firmware details as seen in the setup menu

The version we had was ZAM170-NF-1.8.25-7354-Ver1.0.0. We used that string and parts of it for our web search.

After running some sophisticated Google search queries, we found a few devices on international distributors' websites that looked a lot like our terminal.



A similar device on an international distributor's website

We also found the firmware, albeit it was an earlier version.



+ FW-zkteco ZAM170 Devices UUpdate 1.5.8

Same-series firmware

The firmware was just enough for us to figure out how the update worked. Having downloaded and analyzed the firmware, we found that the update itself was part of a text file to be transformed by specialized software.

```

1  [Upgrade]
2  FWVersion=ZAM170-NF-Ver1.5.7-6928-02
3  [Options]
4  ZAM170_TFTFirmwareVersion=Ver 8.00 Oct 19 2020
5  FirmwareLength=72600039
6  FirmwareProtype=push
7  FirmwareChecksum=2867
8  [FirmwareData]
9  Data0=1B55E57FBEEF4D5F7D23F6A85780490A13E78226A2BF503819F1E
10 Data1=B44912818D988A451FC68963E9E9600202D668A3557300CBC4FF3
11 Data2=5D5AF59006B1E90F4A0EBACC45739CE45BA06AF9C17C1A602CCAF
12 Data3=48EEE9513DC9543C304C36E1DFFD61593FE5C4387EC099807B60A
13 Data4=DEE4DB39C8C1796F48E665064A06BFC1E37B17CE9A65E10DD5991
14 Data5=F601CB55C1D55BEBDDDD1AC73049F7F51DAEB7837D67B8C99C2F71

```

[Update text file](#)

The transformation process was not too sophisticated, with the hexadecimal text records contained in the "DataX" variables converted to the byte format to produce firmware.

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	1B	55	E5	7F	BE	EF	4D	5F	7D	23	F6	A8	57	80	49	0A	[Ue.snM_)#цEWBI.
00000010	13	E7	82	26	A2	BF	50	38	19	F1	BB	27	35	F0	FB	97	.з,&ŷiP8.с»'5рм-
00000020	22	0A	BD	B9	90	9F	DE	1E	DD	E5	FD	12	FC	32	ED	7A	".SNђuЮ.Зез.ь2нз
00000030	A9	F9	50	4A	D5	8B	1E	B0	82	78	B9	66	8F	76	16	E4	©мPJX<.°,хNfЦv.д
00000040	B0	0F	37	E5	43	02	D1	2E	FF	EE	9B	C8	7E	BA	B5	C8	°.7eC.C.яо>И~срИ
00000050	64	96	32	86	37	17	9F	5B	16	8B	25	C5	17	F1	0A	83	d-2†7.ц[.<%E.с.ђ
00000060	87	55	AC	43	E8	B9	FC	69	9F	32	D6	C6	6C	22	F5	26	†U~CиNђu2ЦX1"x&
00000070	3F	12	ED	AC	63	51	FB	8C	48	37	C5	F1	81	B8	54	B8	?..н~сQыEH7EcђeTё
00000080	E5	44	B2	56	A4	18	81	7B	12	6B	AC	21	89	77	68	0E	eDIVx.ђ{.k~!%wh.
00000090	F7	1F	00	CF	BA	2B	99	E9	3B	8B	DB	8D	01	B2	68	1F	ч...Пе+™ŷ;<HĲ.Ih.
000000A0	77	50	96	E2	87	A7	7B	1D	B3	23	16	6E	7D	CD	D9	57	wP~в+\$(.i#.n)HIIIW
000000B0	E6	80	E5	C0	39	B1	0A	6E	0E	9E	9F	58	64	55	51	35	жTeA9†.n.ђuXdUQ5
000000C0	5A	21	CD	AC	BE	1A	D9	48	2F	FC	1B	EA	75	84	FC	04	Z!H~s.IIIH/ь.ku,,ь.
000000D0	DE	8F	A5	AE	64	57	60	2A	CD	18	B2	15	6B	B5	26	26	KUIt@dW`*H.I.ku&&

[Update binary](#)

A quick analysis of the file found that it was encrypted. This led us to examine other files in the archive.

A closer inspection revealed that the device supported partial firmware updates that affected only certain libraries and executables. We found a smaller update package like that inside a directory shipped with the firmware archive that we had downloaded from the distributor website.

```
→ ico file mtblock.tgz
mtblock.tgz: gzip compressed data,
→ ico
```

Partial update archive

```
→ unpack tar xvf mtblock.tgz
data/standalonetabledesc.xml
lib/libzkdbpushconvert.so
lib/libcjson.1.0.so
service/standalonecomm
→ unpack
```

Partial update files

Through a quick analysis of the "standalonecomm" executable, we found that the file handled requests received on port 4370/TCP. The executable also had firmware update functionality. The handler invoked a "zkfp_ExtractPackage" file extractor function that was external to the executable.

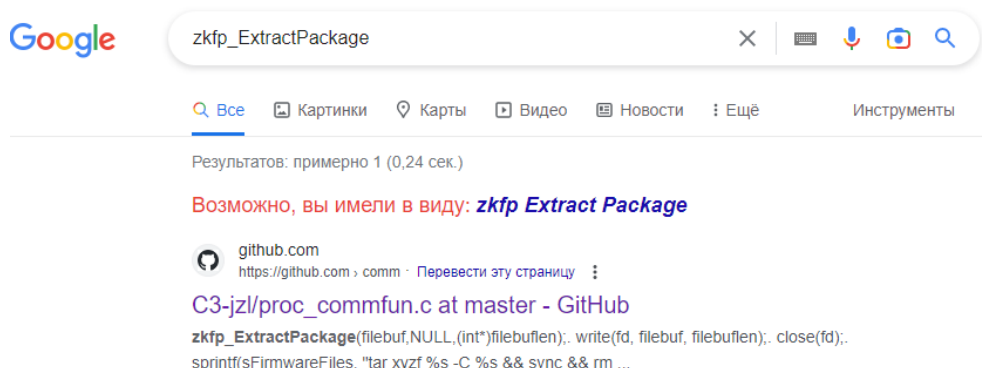
```
PrintLog(4, "commu.c", "ProcessUpdateFirmwareCmd", 0x1289, "
zkfp_ExtractPackage((int)UpdateFileBuffer, 0, UDiskLength);
PrintLog(4, "commu.c", "ProcessUpdateFirmwareCmd", 0x128C, "
write(FirmwareUpdateFd, UpdateFileBuffer, UDiskLength);
close(FirmwareUpdateFd);
sprintf(UnpackCmd, "tar xvzf %s -C %s && sync && rm %s -rf",
if ( systemEx(UnpackCmd) )
```

Update file extract code

```
extern:0007E768 ; int __fastcall zkfp_ExtractPackage(_DWORD, _DWO
extern:0007E768          IMPORT __imp_zkfp_ExtractPackage |
```

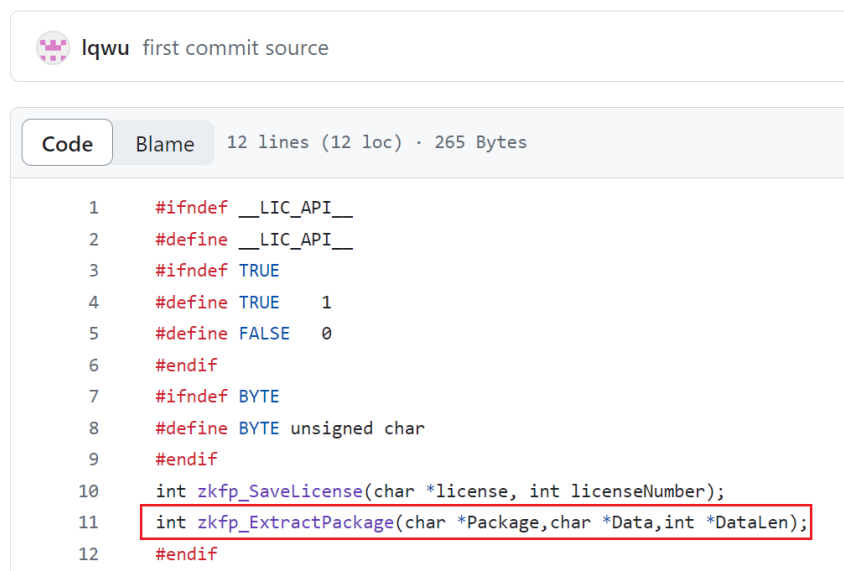
External update image extract function

We failed to find the function in any of the other update files, so we resorted to searching the web. This took us to a repository that had the function in its header file.



Searching for the extract function

[C3-jzl](#) / [lib](#) / [libdlcl.h](#)

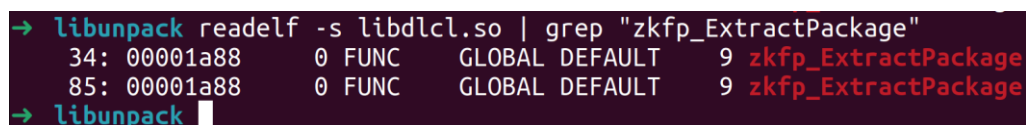


The extract function inside the header file

We found a library with the function implemented inside the same repository.

[C3-jzl](#) / [platform](#) / [zem500](#) / [libdlcl.so](#)

The library with the extract function inside the repository



Searching for the extract function inside the library

After analyzing the extract function, we found that it was also used for decrypting the firmware. The screenshot below shows the decrypt code.

```

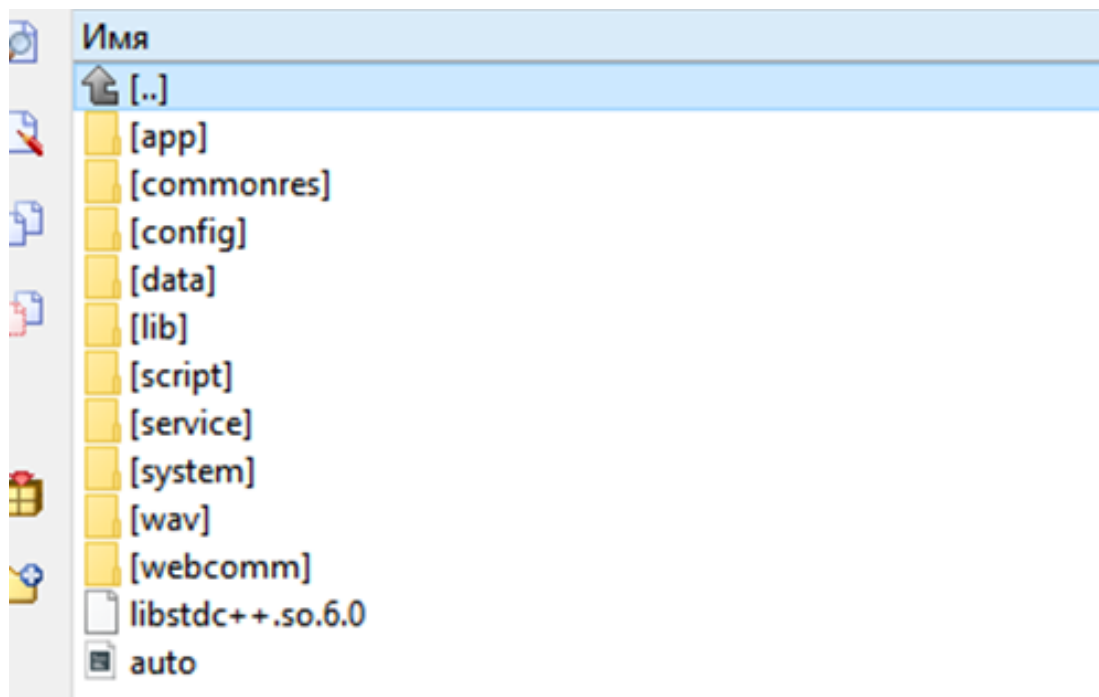
35  *(_DWORD *)&key[8] = v9;
36  *(_DWORD *)&key[12] = v10;
37  *(_DWORD *)&key[16] = PackageSize;
38  v11 = &key[19];
39  do
40  {
41      pKey = &key[idx2];
42      tmp1 = *v11 ^ idx2;
43      tmp2 = key[idx2++];
44      --v11;
45      *pKey = (tmp1 + tmp1 % 0xFu) ^ tmp2;
46  }
47  while ( idx2 < 10 );
48  for ( i = 2; i < (int)PackageSize; *v16 = v17 ^ v18 )
49  {
50      v16 = &Package[i];
51      v17 = Package[i];
52      v18 = key[i % 20];
53      ++i;
54  }
55  v19 = (char *)PackageSize + (_DWORD)Package;
56  *Package = 0x1F;
57  Package[1] = 0x8B;
58  v20 = *(_DWORD *)&key[4];
59  v21 = *(_DWORD *)&key[8];
60  v22 = *(_DWORD *)&key[12];
61  *((_DWORD *)v19 - 4) = *(_DWORD *)key;
62  *((_DWORD *)v19 - 3) = v20;
63  *((_DWORD *)v19 - 2) = v21;
64  *((_DWORD *)v19 - 1) = v22;
65  return 1;
66 }

```

Update file decrypt code

The encryption used XOR with a key consisting of the last 16 bytes of the update file and the file size. It appeared that now we had all the data we needed to generate a key and decrypt the firmware.

Once decrypted, the file turned out to contain an update only for some of the executables, libraries and configuration files.



Decrypted update archive

This was not too much of an issue, as the executable that handled incoming data on port 4370/TCP – the one we were looking for – was among the contents of the downloaded archive. We still wanted the full firmware, so we tried the other option: reading the flash memory.

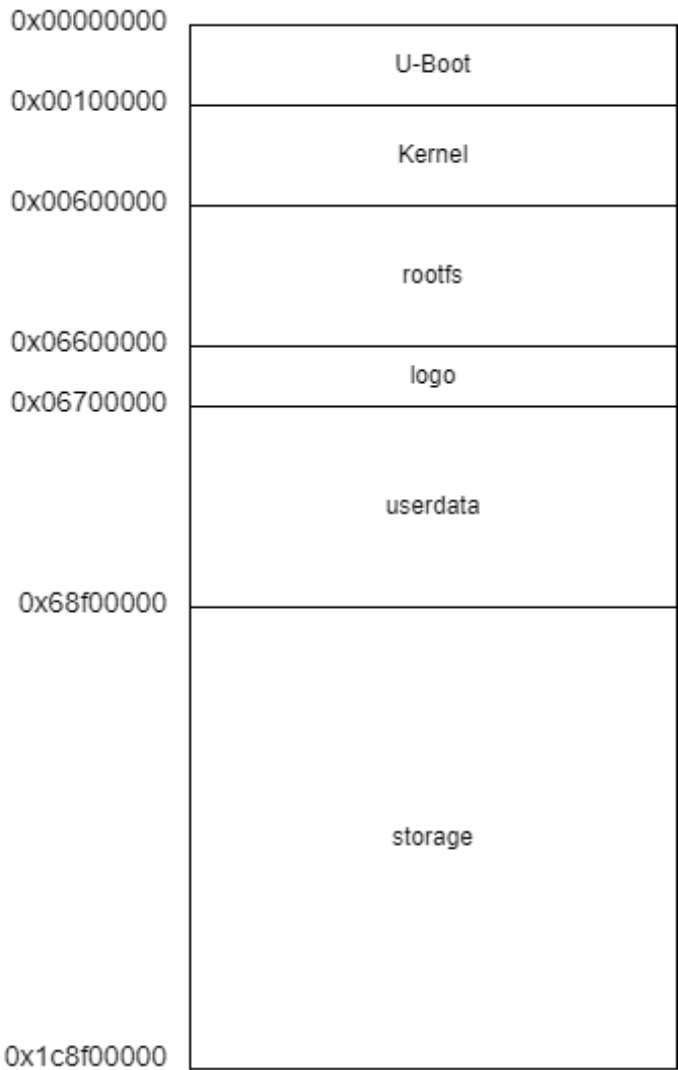
Getting the firmware from the flash memory

As mentioned at the beginning of this section, one could pull a copy of the firmware from the flash memory located on the circuit board.



The flash memory on the circuit board

The memory was an eMMC inside a BGA-153 package that was easy to find a programmer clip for, online. Reading the flash memory gave us a file that contained various sections as shown below.



Flash memory structure

The section names were generally self-explanatory, but we still ran binwalk, a publicly available utility for data container analysis, to make sure they were correct. The binwalk output is shown below.

```
→ biometric_term binwalk dump_flash.bin
```

DECIMAL	HEXADECIMAL	DESCRIPTION
20784	0x5130	gzip compressed data, has original file name: "u-boot.bin",
1048576	0x100000	uImage header, header size: 64 bytes, header CRC: 0xDF885750
008000, Entry Point: 0x80008000, data CRC: 0xA7B58E81, OS: Linux, CPU: ARM, image type: OS		
1048640	0x100040	Linux kernel ARM boot executable zImage (little-endian)
1075780	0x106A44	gzip compressed data, maximum compression, from Unix, last m
4913224	0x4AF848	Flattened device tree, size: 18024 bytes, version: 17
6291456	0x600000	Linux EXT filesystem, blocks count: 24576, image size: 25165
31730686	0x1E42BFE	Unix path: /var/run/dbus/system_bus_socket

The binwalk output for the flash memory dump

Besides all the executables and a Linux kernel, the flash memory contained the credentials of the system's only two users.

```
→ ext-root cat etc/shadow
root:$6$aTemDmvBzN6H/o3G$ZVSWuL
zkteco:$6$aTemDmvBzN6H/o3G$ZVSW
→ ext-root
```

The contents of /etc/shadow

Assuming the users accessed the device via SSH, we tried brute-forcing the hashes to get their passwords. We successfully obtained the password for the user "zkteco" who indeed had SSH access to the terminal.

```
→ ~ ssh -p3718 zkteco@192.168.1.201
zkteco@192.168.1.201's password:
It seems that your account's password is never changed.
Please change it as soon as possible.
Hello, zkteco.
[zkteco@zam170]~$ id
uid=1000(zkteco) gid=1000(zkteco) groups=1000(zkteco)
[zkteco@zam170]~$
```

Logging in with credentials via SSH

Unfortunately, this user did not have the highest privileges, but we still got access to a number of sensitive system files and a list of running services.

```
347 root      0:02 /mnt/mtdblock/service/licdm
448 root      0:05 /mnt/mtdblock/service/hub
461 root      2:41 /mnt/mtdblock/service/devs
466 root      0:08 /mnt/mtdblock/app/mginit/mginit
528 root      1:32 /mnt/mtdblock/app/main/main
569 root      0:09 /mnt/mtdblock/service/biometric
686 root      0:03 /mnt/mtdblock/service/pushcomm
692 root      0:02 /mnt/mtdblock/service/standalonecomm
695 root      0:12 /mnt/mtdblock/service/tuyacomm
696 root      0:00 /mnt/mtdblock/service/spytuya
```

Executables running on the device

The main service is named "main". It controls everything that is displayed on the screen and talks to other necessary services through a service named "hub". The latter is a message broker of sorts that provides a convenient interface for services to communicate. A further service of interest is "pushcomm": an HTTP client that sends requests to a server specified in the device configuration. In other words, the client can be used to attack the device if the attacker can make the device talk to a web server that they control. Read on to find out about attacks that can be implemented by using this method. Also, note that all the services are running with the highest privileges, which makes hijacking the device much easier as any vulnerability that allows code or command execution gives the attacker the highest privileges.

Analyzing the protocol on port 4370/TCP

We chose the standalonecomm service as the main object for our analysis as it implements the vendor's proprietary protocol on port 4370/TCP and contains commands of interest to an attacker that may be implemented improperly.

As mentioned at the beginning of this article, protocol documentation is available from a GitHub repository, which significantly simplifies analysis as one can apply the knowledge to disassembled code to find the handler of the command one is interested in.

The protocol structure is fairly simple and typical. A packet consists of a header and a payload. The payload is also divided into a header and data, with the latter largely determined by the command. In some cases, it is a four-byte number, and in others, a string or dataset. A detailed description of the protocol design can be found in the [publicly available document repository](#).

Protocol authentication and its issues

The protocol's interesting features include user authentication, which requires knowing the password set on the device. On our device, the password is called "COMKey" and set by the administrator. The password is set to 0 by default, that is, there is no password, and all requests can be run without any authentication.

Besides, COMKey can be an integer from 0 to 999999, so there is a limited number of possible passwords that can be brute-forced over the network. We came across the restriction while analyzing the code that sets the password.

```
15 CreateInputWnd(a1, StrByID, 0, v10, 16, 6, 0, 999999, 2, 0);
16 if ( !strcmp((const char *)v11, (const char *)v10) )
17     return 1;
18 SetSubItemTextByItem(a2, a3, 1, "*****");
19 v9 = strtol((const char *)v10, 0, 0xA);
20 SetOptionIntValue((int)"COMKey", v9);
21 if ( !GetOptionIntValue("IsSupportZKEcoProFunOn", 0) )
22     return 1;
23 AddOpLog(80, 0, 0, v11, v10);
24 return 1;
25 }
```

COMKey set code

The method used for generating a so-called "MAC" (Message Authentication Code) for protocol authentication is not secure enough either. The generation process relies on reversible operations, so if

we can monitor traffic on the network, we can recover the password once the client is authenticated successfully. The generation code is shown in the screenshot below.

```

1 int __fastcall sub_1F520(int COMKey, int SessionId)
2 {
3     int v2; // r2
4     int i; // r3
5     int v4; // r12
6     int v5; // r2
7     int v7; // [sp+4h] [bp-4h]
8
9     v2 = 0;
10    for ( i = 0; i != 32; ++i )
11    {
12        v4 = COMKey >> i;
13        if ( (v4 & 1) != 0 )
14            v2 = (2 * v2) | 1;
15        else
16            v2 *= 2;
17    }
18    v5 = SessionId + v2;
19    BYTE2(v7) = BYTE2(v5) ^ 0x53;
20    HIBYTE(v7) = HIBYTE(v5) ^ 0x4F;
21    LOWORD(v7) = v5 ^ 0x4B5A;
22    return __ROR4__(v7, 16);

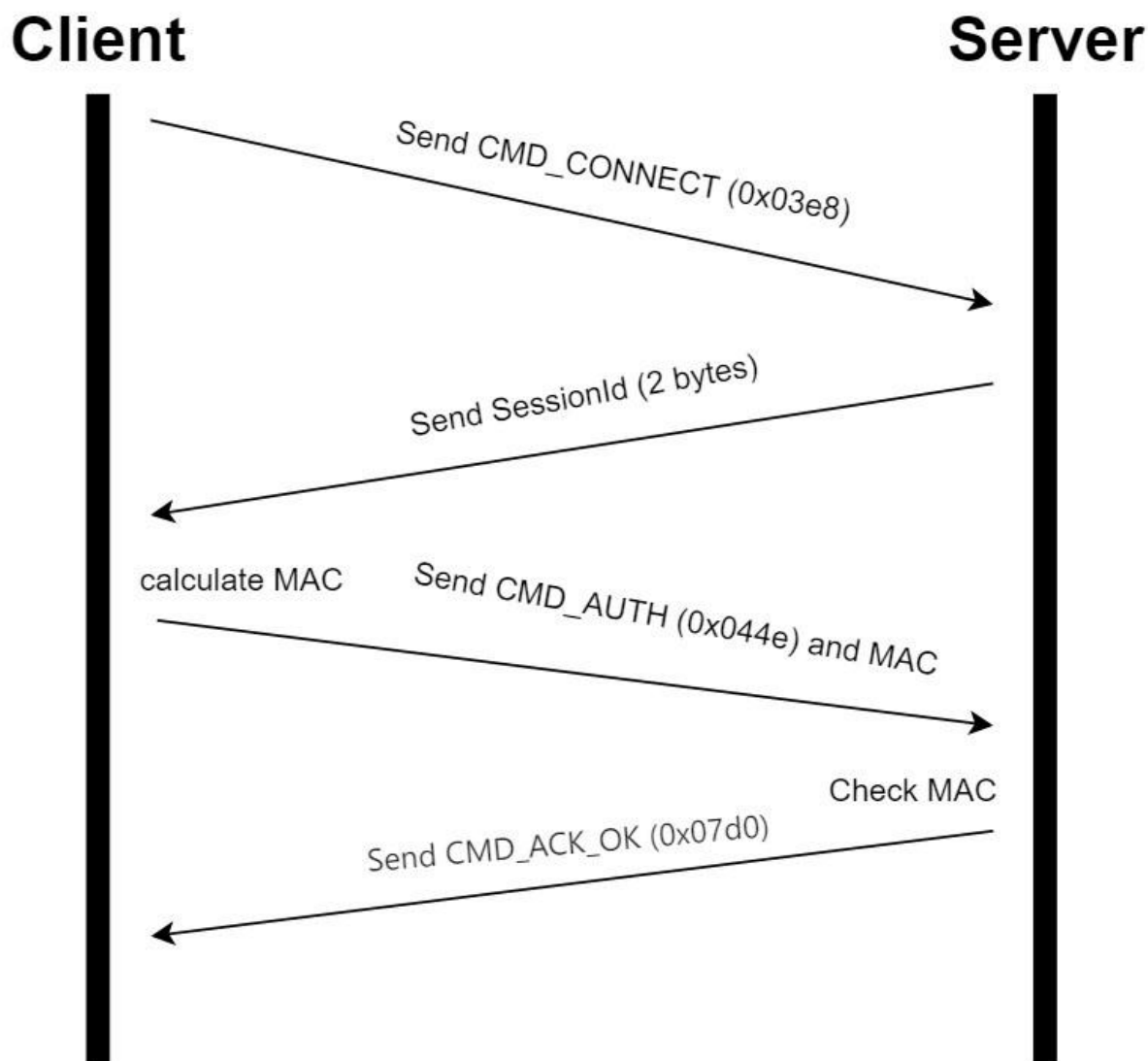
```

MAC generation code

The SessionId variable is a two-byte value generated by the server and sent to the client, so it can calculate a MAC from the COMKey and return the resulting value to the server.

Another password-related security risk is that the COMKey is stored unencrypted in the device database, so an arbitrary file read vulnerability would let us find it out and authenticate over the protocol. Another possible scenario is logging in via SSH and reading the database to obtain the protocol password without a network brute-force attack.

The diagram below illustrates the protocol authentication mechanism.

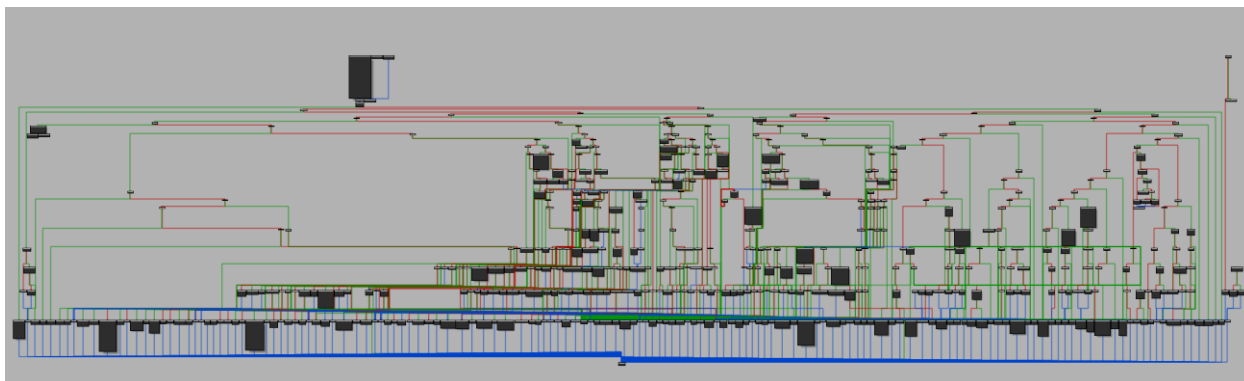


Protocol authentication mechanism

The client sends a connect command (`CMD_CONNECT`), and the server returns two bytes that represent a `SessionId` and are combined with the `COMKey` to generate a MAC. The client sends the MAC with a `CMD_AUTH` command, and the server validates that. If the MAC is found to be valid, the server responds with `CMD_ACK_OK`, and the client is now free to use all available server commands within the current TCP session.

Vulnerability analysis of command handlers

All commands that become available as a result of successful authentication are handled by one large function with a command ID switcher inside. Below is what its graphic representation looks like.



A graphic representation of the command handler

Analyzing the function does not involve any great complexity: this is only a matter of time and attention.

We immediately singled out commands whose names contained the words "DOWNLOAD", "UPLOAD", "DELETE" or "UPDATE" as relevant analysis objects.

For example, CMD_DOWNLOAD_PICTURE downloads a user image. It accepts a file name as an argument, which it does not validate in any way before inserting in the file open function. This allows passing, say, directory traversal characters as a file name to fetch an arbitrary system file. The handler code is shown in the screenshot below.

```
if ( Cmd == CMD_DOWNLOAD_PICTURE )
{
    *UserPacketPayload = -47;
    UserPacketPayload[1] = 7;
    n[0] = 0x100000;
    memset(s, 0, 0x80u);
    GetAdPicturePath(s, 128, UserPacketPayload + 8);
    PrintLog(8, (int)"commu.c", (int)&unk_5BEA8, 6774, "file name= %s");
    v62 = malloc(n[0] + 1);
    v46 = v62;
    if ( !v62 )
    {
        *UserPacketPayload = 118;
        v12 = 8;
        UserPacketPayload[1] = 19;
        return v12;
    }
    memset(v62, 0, n[0] + 1);
    if ( !ReadPhotoBuf((const char *)s, (int)v46, (int *)n) )
```

Image download handler

The command can be used to obtain /etc/shadow, as standalonecomm is running with the highest privileges.

We detected several file read vulnerabilities after finding further commands that passed file names without any filtering. We also found a function that allowed uploading files to arbitrary paths. Given the privileges granted to the service, the function can be leveraged to gain unlimited access to the device.

An analysis of CMD_DELETE_PICTURE revealed the possibility of embedding shell commands due to the name of the image to be deleted being inserted directly into the command, which was then passed to the "system" function.

```
memset(s, 0, 0x100u);
*UserPacketPayload = -47;
UserPacketPayload[1] = 7;
memset(n, 0, 0x96u);
if ( UserPacketPayload[8] == 'A'
    && UserPacketPayload[9] == 'L'
    && UserPacketPayload[10] == 'L'
    && !UserPacketPayload[11] )
{
    GetAdPicturePath(n, 150, "");
}
else
{
    GetAdPicturePath(n, 150, UserPacketPayload + 8);
}
EL_369:
v12 = 8;
snprintf((char *)s, 0x100u, "rm %s", (const char *)n);
system((const char *)s);
sync();
*UserPacketPayload = -48;
UserPacketPayload[1] = 7;
return v12;
```

Image delete handler

We wrote PoC scripts to confirm that the vulnerability can be exploited. See below for script output.

```
→ scripts python3 cmd_inject.py 192.168.1.201 "id > /tmp/1"
→ scripts python3 read_file.py 192.168.1.201 /tmp/1
uid=0(root) gid=0(root)
```

PoC script output

We also found several buffer overflow vulnerabilities associated with the use of insecure strcpy/sprintf functions and a lack of copied buffer size validation in the "memcpy" function. We will use the example of the CMD_CHECKUDISKUPDATEPAGE handler to examine the issue.

```

23
24 *payload = -47;
25 payload[1] = 7;
26 memset(v22, 0, sizeof(v22));
27 v19 = 0LL;
28 v20 = 0;
29 v21 = 0LL;
30 memset(v26, 0, 0x400u);
31 if ( GetOptionStrValue("~Platform", "ZMM100", &v19, 20) )
32     sprintf(v22, "%s%s", "ZEM200", "_FirmwareVersion=");
33 else
34     sprintf(v22, "%s%s", (const char *)&v19, "_FirmwareVersion=");
35 v6 = v13;
36 memcpy(v26, payload_data, payload_size);
37 v7 = v13;
38 PrintLog(4, (int)"commu.c", (int)&aProcesscheckud, 5951, "%s\n", v26);

```

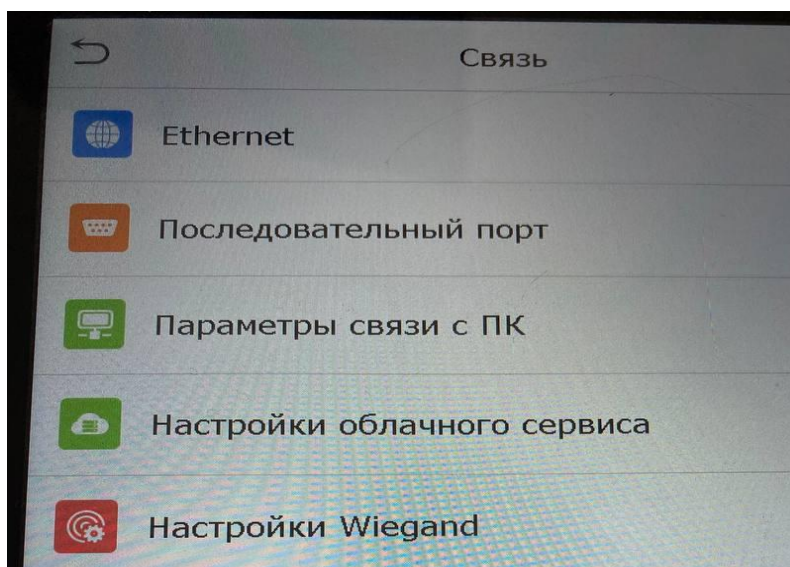
CMD_CHECKUDISKUPDATEPACKPAGE handler

The vulnerability stems from the fact that when copying data from a user network packet, the handler uses the packet size specified by the user. The destination buffer is located in the stack and has a size of 1028 bytes. The user specifying a greater data size in the packet results in a buffer overrun. The executable has no stack overflow protection. Malicious actors can exploit the vulnerability to invoke a [ROP chain](#) and execute arbitrary code that opens remote access to the device.

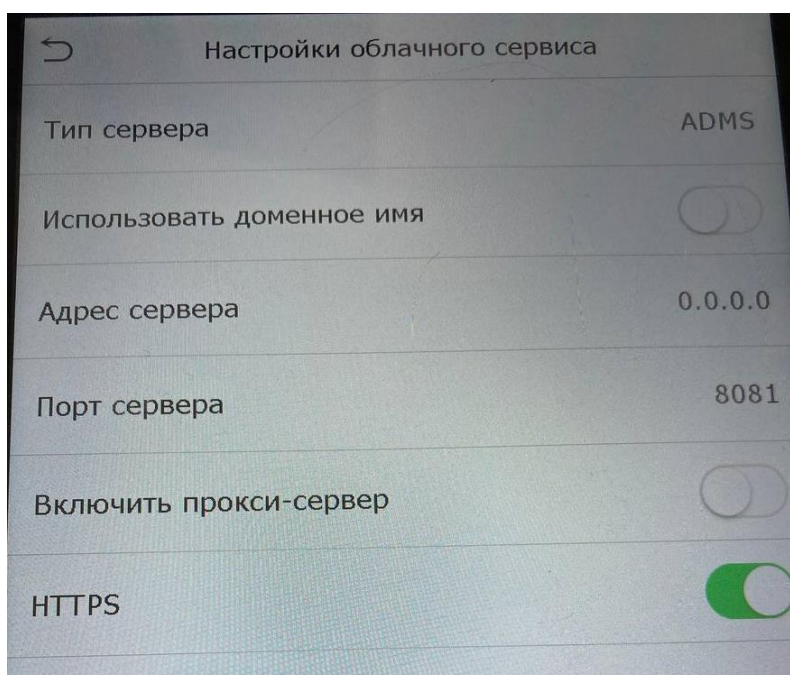
Finally, we discovered SQL injection vulnerabilities virtually everywhere a string value passed by the user inside a network packet was directly inserted into a database query.

pushcomm analysis

As mentioned above, the pushcomm service sends requests to a server specified in the device configuration. To set up the server address, the administrator goes to the "COMM" menu and opens "Cloud Server Setting". The administrator defines an IP address to connect to and a port, also enabling other options as required. The screenshots below show the configuration menu.



COMM menu



Cloud Server Setting menu

An analysis of the executable showed that it was prone to the same issues as standalonecomm. However, exploiting the flaws requires spinning up a web server and making the device talk to it. There is more than one way to do this: by changing settings in the database or the admin menu, or via ARP spoofing.

Note that one of the pushcomm commands is named "SHELL", and it runs any commands on the device.

```

331 if ( !strncasecmp(CmdData, "SHELL", 5u) )
332 {
333     strncpy(v146, CmdData, 5u);
334     if ( strncasecmp(CmdData + 6, "reboot", 6u) )
335     {
336         memset(OS_cmd, 0, 0x64u);
337         memset(s, 0, 0x400u);
338         memset(OutFilePath, 0, 0x400u);
339         memset(SN, 0, 0x40u);
340         GetZKRamdiskPath(OutFilePath, 1024, "shell.out");
341         snprintf(OS_cmd, 0x64u, "%s >%s 2>&1", CmdData + 6,
342         ret = system(OS_cmd);

```

SHELL handler

All it takes to execute the command is spinning up a web server and implementing the following handler.

```

35 @app.route('/iclock/getrequest', methods=['GET'])
36 def getrequest():
37     global cmd
38     args = request.args
39
40     if cmd == 0:
41         cmd = 1
42         return "C:118:SHELL:ls -la > /tmp/123;"
43     return 'OK'

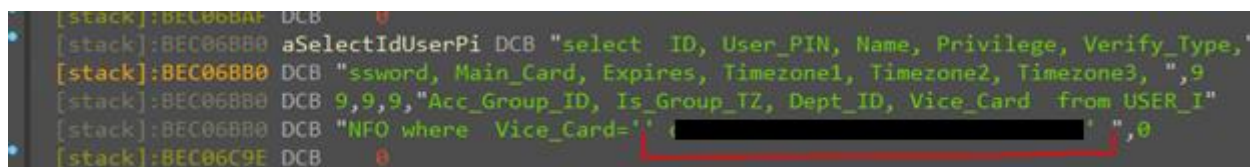
```

Example of a handler to invoke SHELL

Overall, there is considerable overlap between pushcomm and standalonecomm code, especially in terms of database queries.

QR code handler analysis

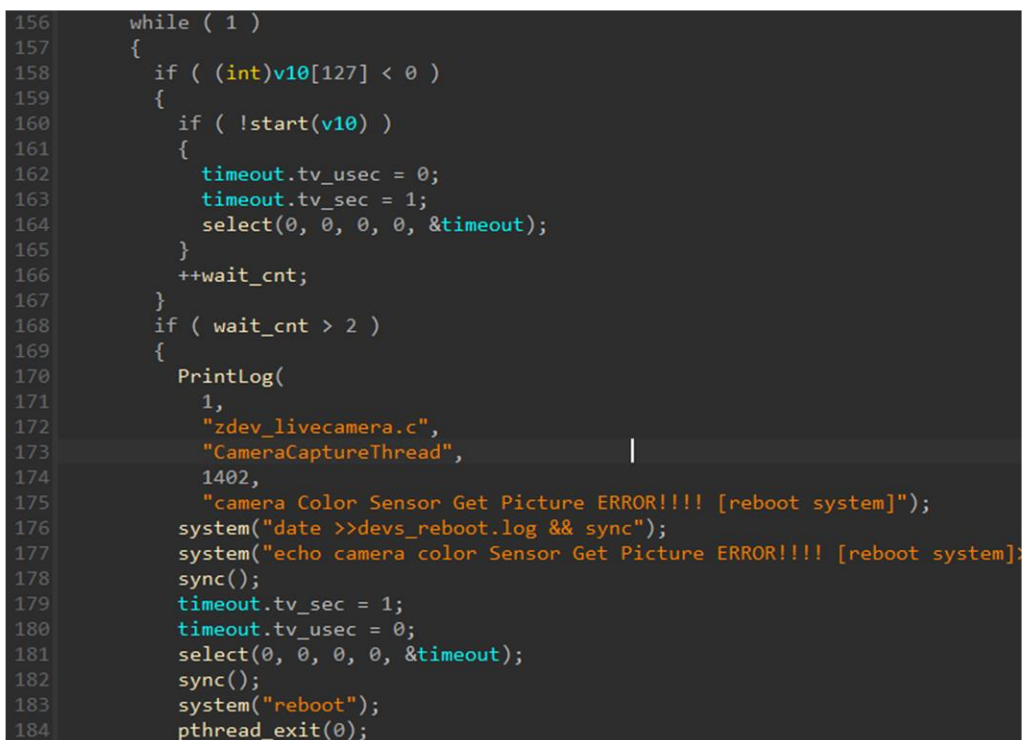
At the beginning of the article, we mentioned that the device authenticated us as a different user when we made it scan a QR code with SQL injection. However, as we analyzed the code, we found that the size of data that a QR code could contain was limited to 20 bytes. This prevents [complex UNION and SELECT injections](#) that can be used to obtain arbitrary data from various fields in the database. The database query that was generated when the device scanned our malicious QR code (code with SQL injection in our case) is shown in the screenshot below.



```
[stack]:BEC06B8F DCB 0
[stack]:BEC06B80 aSelectIdUserPi DCB "select ID, User_PIN, Name, Privilege, Verify_Type, "
[stack]:BEC06B80 DCB "ssword, Main_Card, Expires, Timezone1, Timezone2, Timezone3, ",9
[stack]:BEC06B80 DCB 9,9,9,"Acc_Group_ID, Is_Group_TZ, Dept_ID, Vice_Card from USER_I"
[stack]:BEC06B80 DCB "NFO where Vice_Card=' ' ",0
[stack]:BEC06C9E DCB 0
```

Database query when using the QR code

We also found that we could cause the device reboot by making it scan a QR code that contained a lot of data. Looking at the code, we saw this was due to a piece of code that was waiting on camera data being unable to receive it within a predefined period of two seconds and sending a "reboot" command in response to what it perceived as a malfunction.



```
156 while ( 1 )
157 {
158     if ( (int)v10[127] < 0 )
159     {
160         if ( !start(v10) )
161         {
162             timeout.tv_usec = 0;
163             timeout.tv_sec = 1;
164             select(0, 0, 0, 0, &timeout);
165         }
166         ++wait_cnt;
167     }
168     if ( wait_cnt > 2 )
169     {
170         PrintLog(
171             1,
172             "zdev_livecamera.c",
173             "CameraCaptureThread",
174             1402,
175             "camera Color Sensor Get Picture ERROR!!!! [reboot system]");
176         system("date >>devs_reboot.log && sync");
177         system("echo camera color Sensor Get Picture ERROR!!!! [reboot system]");
178         sync();
179         timeout.tv_sec = 1;
180         timeout.tv_usec = 0;
181         select(0, 0, 0, 0, &timeout);
182         sync();
183         system("reboot");
184         pthread_exit(0);
185     }
186 }
```

Camera data wait code

Conclusion

Biometric devices designed to improve physical security can both offer convenient, useful features and introduce new risks for your IT system. When advanced technology like biometrics is enclosed in a poorly secured device, this all but cancels out the benefits of biometric authentication. Thus, an insufficiently configured terminal becomes vulnerable to simple attacks, making it easy for an intruder to violate the physical security of the organization's critical areas.

Our analysis of the ZkTeco biometric terminal yielded a total of 24 vulnerabilities. Many of those were similar, stemming from an error in the database wrapper library. We generalized these as "multiple vulnerabilities" and stated the type and cause, arriving at a smaller number of CVEs.

In terms of the cold statistics, the results are as follows:

- 6 SQL injection vulnerabilities
- 7 buffer stack overflow vulnerabilities
- 5 command injection vulnerabilities
- 4 arbitrary file write vulnerabilities
- 2 arbitrary file read vulnerabilities

The descriptions of the vulnerabilities we detected are available in the Kaspersky research team's [GitHub repository](#).