

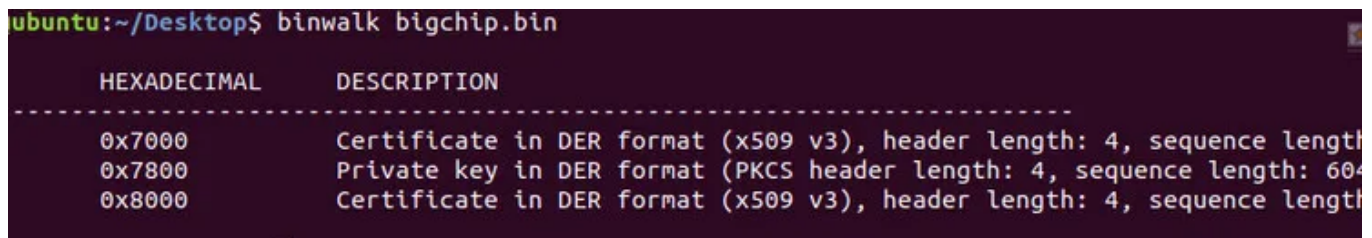
[Home](#) / [Blog](#) / Hörmann: Opening Doors for everyone...

Hörmann: Opening Doors for everyone...

28.10.2020 vulnerability

Hoermann is constantly working to optimize the quality and security of all products. The detection of potential weaknesses by SEC Consult proved helpful to further improve the entire BiSecur system.

The following article is based on research done by Tamas Jos (SEC Consult Schweiz AG).



```
ubuntu:~/Desktop$ binwalk bigchip.bin
```

HEXADECIMAL	DESCRIPTION
0x7000	Certificate in DER format (x509 v3), header length: 4, sequence length: 60
0x7800	Private key in DER format (PKCS header length: 4, sequence length: 60)
0x8000	Certificate in DER format (x509 v3), header length: 4, sequence length: 60

1. TL;DR

SEC Consult found that the **Hörmann BiSecur Gateway** product contained multiple vulnerabilities.

The most serious *remote* attack allows an attacker to **harvest credentials from users** operating the product and underlying infrastructure by exploiting a logic flaw in the custom protocol. All registered customers are affected by this vulnerability.

The most serious *local* attack allows an attacker to **hijack the session** of any user connected to the device. Its success in a matter of minutes relies on the use of a short session identifier and absence of session expiration.



Incident?

An attacker on the local network can **open any doors** using the BiSecur Gateway system. A remote attacker on the Internet can harvest and acquire all user credentials to open customer doors using

NOTE: This article contains external links to the author's personal GitHub account includes other interesting projects.

2. Introduction

A few years ago, I hired a company to install a garage door opener in a private rented garage. In the official sales catalog, they offered an additional **remote control solution** that allowed the garage door to be opened and closed via a smartphone instead of a physical key-fob. As the latter was quite an expensive option, I decided to go for the smartphone solution. The salesman explained that although he could sell me the remote opening system, the company **would not be able to install it**, due to many bad experiences in the past (let's not forget this was an official Hörmann distributor). I still decided to proceed by taking the responsibility to install it myself, and that's how the interesting story that you will discover below was born.

During the installation process, I quickly found out that the BiSecur Gateway device was not functioning properly. Every time the device was off, all user-defined settings (including WiFi profiles) were lost and reset. As a result, it would **not consistently pair with the garage door**. That's what motivated me to investigate the causes and examine what was inside the plastic casing.

3. Target Device Specification

Vendor: **Hörmann**

Product Name: **BiSecur Gateway**

Description: The BiSecur Gateway device has been identified as a Microchip microcontroller, equipped with an official Hörmann key fob, bundled with a WiFi and Ethernet interface. Thus, end-users can remotely open and close garage doors after coupling it with the motor controller. A mobile application allows users to **communicate directly with the gateway device** via a local network and/or remotely over the Internet.

Where To Buy: **Amazon**

4. Hardware

Product Picture:

CPU: **PIC32MX695F512H**



Incident?

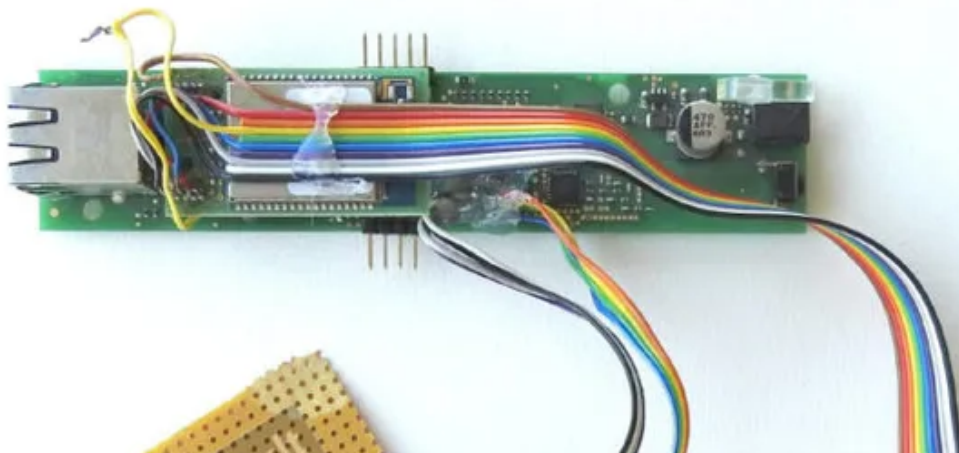
Interfaces (External): WiFi, Ethernet



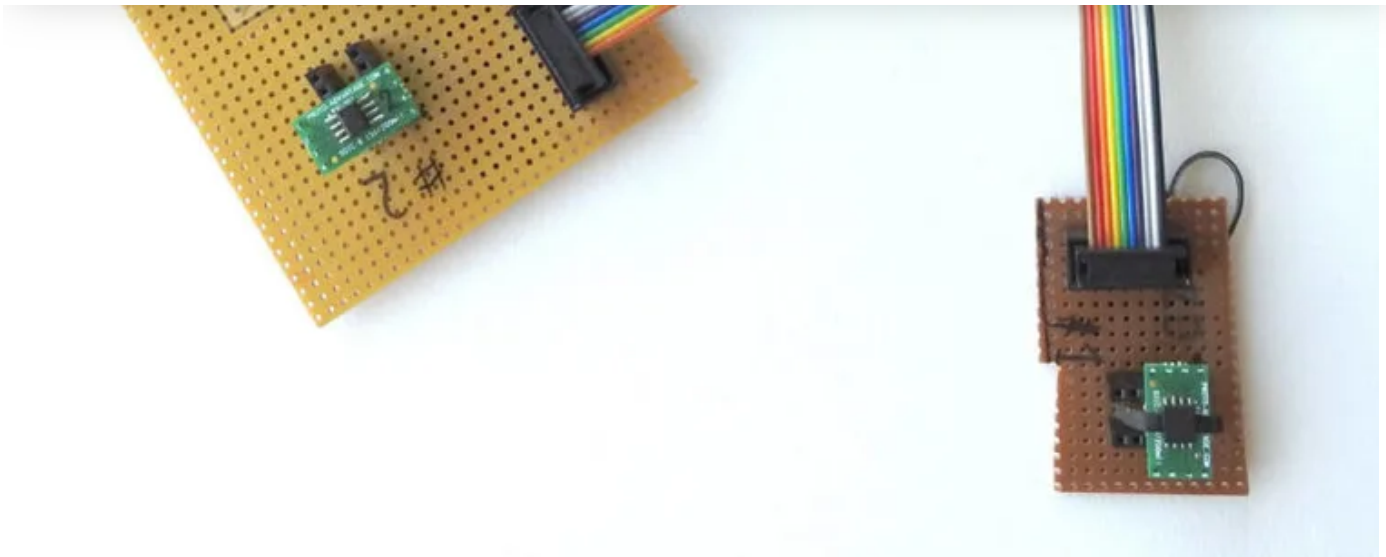
4.1. Flash Chip

The datasheet of the identified flash chip reveals the communication interface used, a Serial Peripheral Interface (SPI). This appears to be the de facto standard for such devices

4.1.1. Reading



Incident?



In an attempt to identify the necessary pins on the chip itself, I first tried directly connect a **Bus Pirate** (BP3) device to the exposed pins and interface it with the chip. This is not a standard method of reading the flash, but it works sometimes. On this occasion, this was unfortunately not the case for this particular Printed Circuit Board (**PCB**), so I had to remove the Flash chip from the PCB and place it on a breakout board.

Usually, I like to use **flashrom** for the reading process; unfortunately, in this case, the utility was of no help. For unknown reasons, the chip did not respond to any commands sent by the flashrom script. So, I had to manually configure BP3, which allowed me to successfully run the read commands. Thus, I was able to transfer the content of the to a text file in a hexadecimal format. Using a Python script, I converted it to the appropriate binary format.

4.1.2. Analysis

For a quick analysis, **binwalk**, **xxd** and **strings** were used.

When using Binwalk, the results yielded two certificates and a private key.

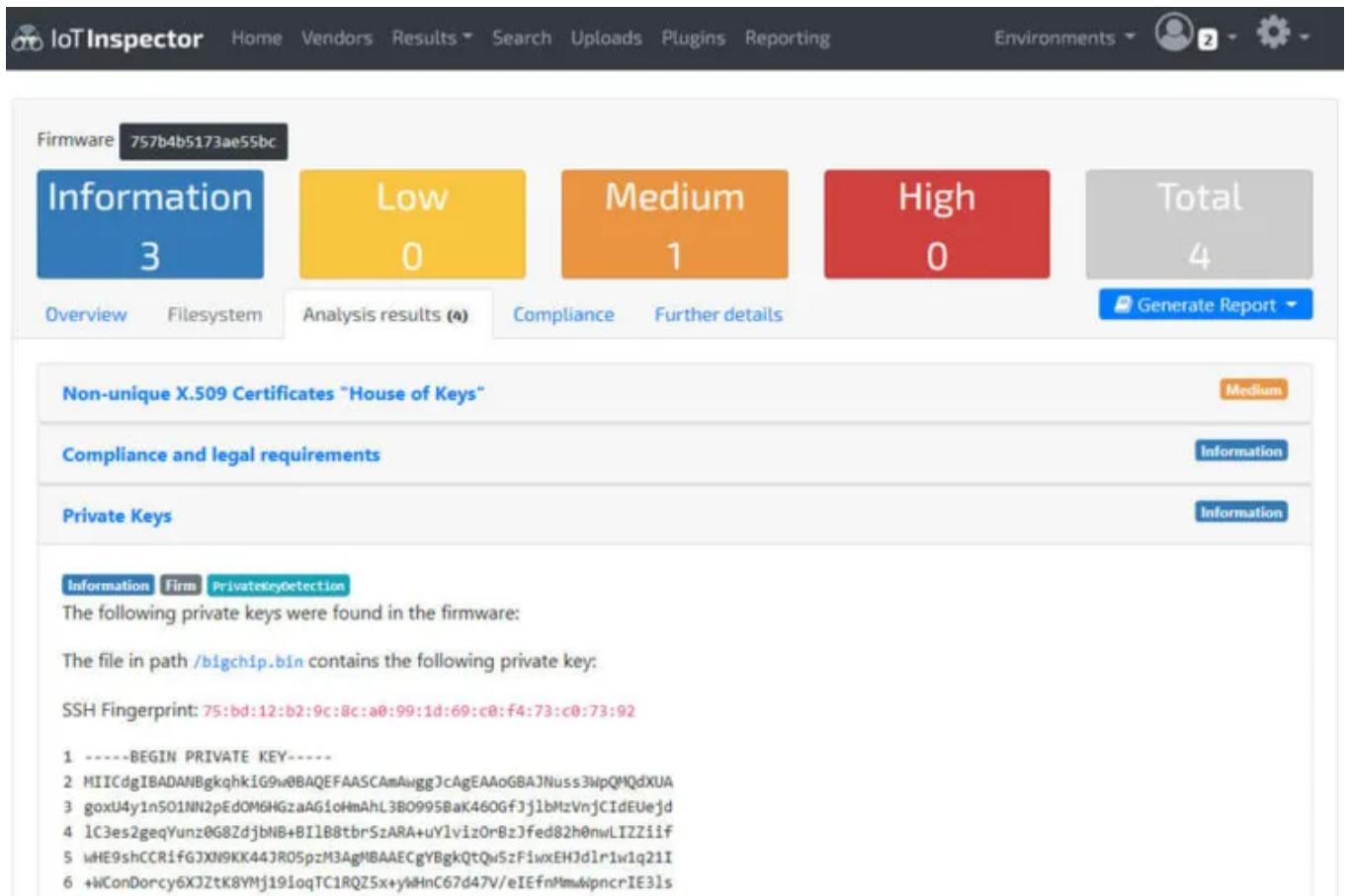
```
blisecur@ubuntu:~/Desktop$ binwalk bigchip.bin
```

DECIMAL	HEXADECIMAL	DESCRIPTION
28672	0x7000	Certificate in DER format (x509 v3), header length: 4, sequence length: 1298
30720	0x7800	Private key in DER format (PKCS header length: 4, sequence length: 1298)
32768	0x8000	Certificate in DER format (x509 v3), header length: 4, sequence length: 1298



Incident?

identify it. I chose **xxd** to investigate further.



The screenshot shows the IoT Inspector web interface. The top navigation bar includes links for Home, Vendors, Results, Search, Uploads, Plugins, Reporting, and Environments. The main content area displays the firmware ID 757b4b5173ae55bc. A summary section shows counts for Information (3), Low (0), Medium (1), High (0), and Total (4) findings. Below this, a list of findings is shown, including 'Non-unique X.509 Certificates "House of Keys"' (Medium), 'Compliance and legal requirements' (Information), and 'Private Keys' (Information). The 'Private Keys' finding is expanded, showing a list of private keys found in the firmware. The first key is highlighted, showing its SSH fingerprint and the file path /bigchip.bin.

Firmware: 757b4b5173ae55bc

Information 3 Low 0 Medium 1 High 0 Total 4

Overview Filesystem Analysis results (4) Compliance Further details Generate Report

Non-unique X.509 Certificates "House of Keys" Medium

Compliance and legal requirements Information

Private Keys Information

Information Firm PrivateKeyDetection

The following private keys were found in the firmware:

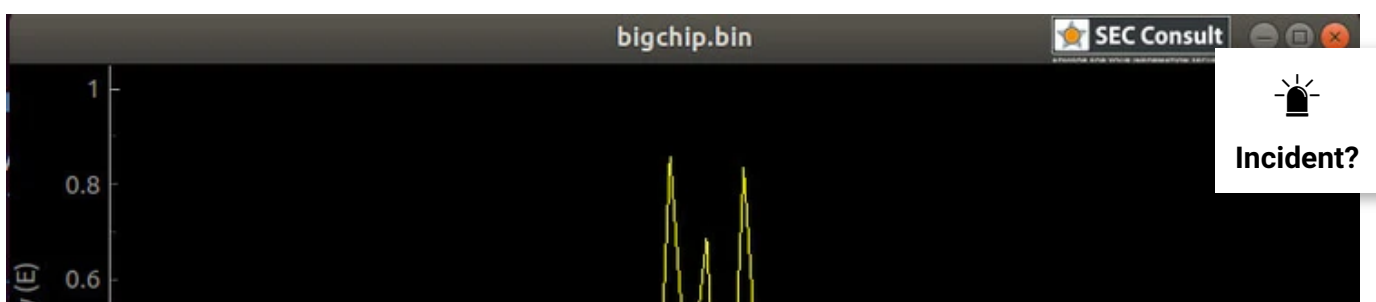
The file in path /bigchip.bin contains the following private key:

SSH Fingerprint: 75:bd:12:b2:9c:8c:a0:99:1d:69:c8:f4:73:c0:73:92

```
1 -----BEGIN PRIVATE KEY-----
2 MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAJNuss3WpQMqdxUA
3 goxtU4y1n501NN2pEdOM6HGzaAG1oHmAHl3BO995BaK46OGf3j1bMzVnjCidEuejd
4 lC3es2geqYunz0G8ZdjbnB+8I1B8tbrSzARA+uYlviz0rBz3fed82h0nwLIZZiif
5 wHE9shCCR1fGJXN9KK443RO5pzH3AgMBAAECgYBgkQtQv5zFiwxEH3d1r1w1q21I
6 +WConDorcy6XJZtK8YMj191oqTC1RQZ5x+ywHnC67d47V/eIEfnMwkipncrIE3ls
```

Additionally, entropy analysis hinted that other data was present, yet the binwalk could not fully identify it. I chose **xxd** to investigate further.

Manually checking the contents with **xxd** showed some interesting memory regions where information, later identified to be hard-coded admin credentials (username & password), was stored; in an area that looked to contain structured data. The initial hypothesis was that this location stored all user credentials in plain-text after creating new users. This hypothesis was later verified to be true.





```
bi secur@ubuntu:~/Desktop$ strings bigchip.bin
BISECUR_83362F
BISECUR_WLAN
bisecur_gw
admin
0000
BiSecur Gatew
BISECUR_83362F
BISECUR_WLAN
bisecur_gw
admin
0000
```



The strings revealed the hard-coded admin credentials as well as some additional identification data (e.g. serial number), which while valuable, was of little relevance for further analysis. The next step was to extract the certificate and private key data from the dump file, for which I wrote and used a python script, before later realizing that binwalk could have done it as well.

4.1.3. Swapping Mechanism

Since this was an initial analysis performed on a module that had been factory reset, I was concerned about what would happen if an **update process overwrote the original data**. This would potentially mean that I would have to remove the flash chip several times and read it, **risking damage** to it in the process after subjecting it to so many cycles. That's why I chose to go another route. While checking the wires between the CPU (microcontroller) and the Flash chip, I found that the same SPI bus was shared with the WiFi module as well! Good news, since the WiFi module is external, I could use its pins for my plug-in Flash chip solution.

Curious to discover this plug-in solution? It will be easier for you to understand it by looking at the image right.

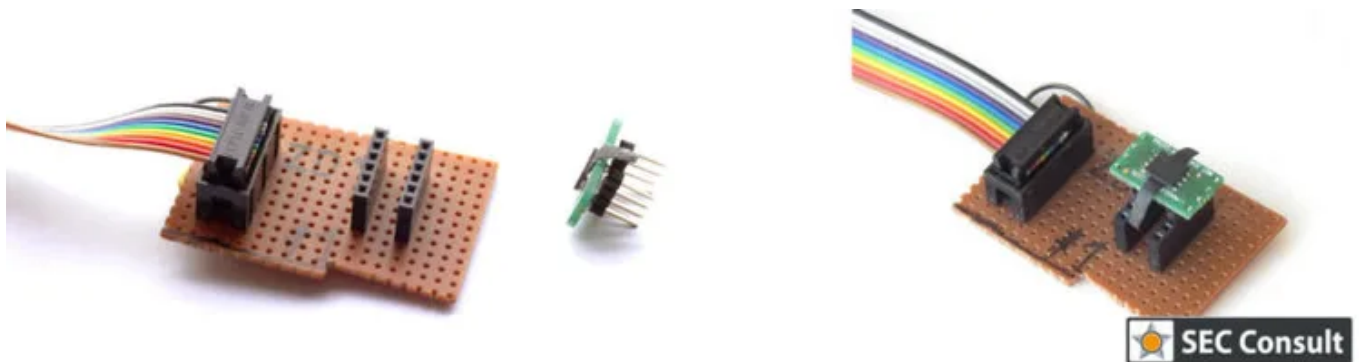


Incident?

By placing the Flash chip on a breakout board, and making a small PCB with a socket to connect to said breakout board, it was possible to create an “interchangeable” Flash chip solution. This is very convenient if you want to modify the data on the Flash, or read it multiple times because of potential

We didn't see anything of value at first glance, as per the analysis. However, my reasoning was as follows:

1. Because it's fun...
2. The presence of a private key and two certificates indicate that the SSL/TLS protocol is used for client authentication and certificate pinning. If we want to launch a **Man-in-The-Middle** (MiTM) attack, we can hope that the default SSL configuration is secure. Alternatively, we can test it and verify that the functionality and integrity of the BiSecur Gateway are protected against the replacement of certificates and user-generated private keys. This type of attack would ensure that the device **will not be able to communicate directly** with Hörmann's backend server, but only through our proxy. The proxy can then be configured to use the correct authentication hardware against the required Hörmann server, if necessary. With this configuration, we have full control of the communication channel as well as the request for outbound content to the internet.

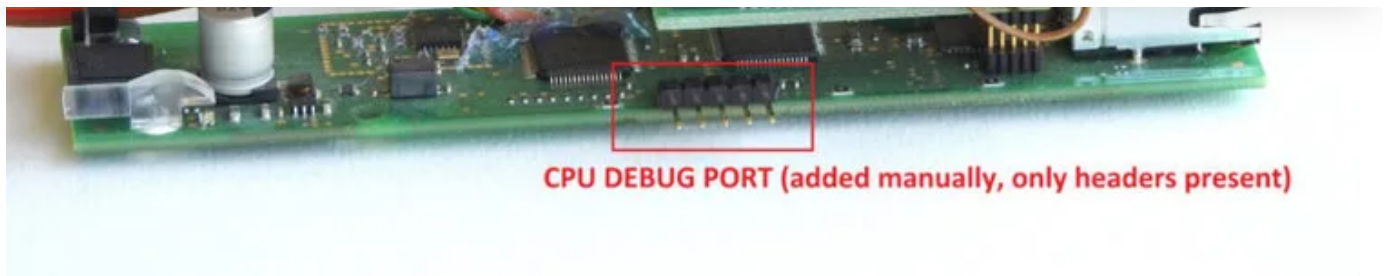


Micro-Controller

4.2.1. Reading Firmware

After analyzing the content of the Flash, we were able to confirm that the firmware resides in the microcontroller itself. In most cases, this means that it will be difficult to obtain the firmware because nowadays all microcontrollers are protected against access to non-volatile memory areas. Using this function is **activated by burning fuses** that are similar to one-time programmable bits. For information about these protection mechanisms is irrelevant in this post and, as it turned out, none of the protection functions had been activated.

**Incident?**



In its factory default state, the debug pins of the microcontroller were connected to header on the PCB, but the header had no connectors. Besides, two of the header pins were used to hold a small piece of plastic, physically holding the WiFi dongle in place. After removing the WiFi interface PCB and the placeholder pins, the header was rebuilt. Microchip sells a programmer called **MPLAB PICKIT**. This allows developers to interface with microcontrollers and **perform read/write/debug tasks**. This is a proprietary debugging interface so I recommend that you buy one if you want to tinker with Microchip's micro-controllers. After installing the necessary drivers and software, the debugger was connected to the newly created header, and... it worked. Reading the chip ID was successful, as well as reading the fuses. The fuse list showed that **no** protection was activated against firmware reading. Using the same software, we were able to dump the firmware and load it into **Ghidra**.

5. Ethernet

The ethernet port is used by the services for communication between the BiSecur Gateway device and the user' smartphone, as well as between the device and a Hörmann server on the Internet. Two different proxy configurations were therefore required to analyze the different communication channels:

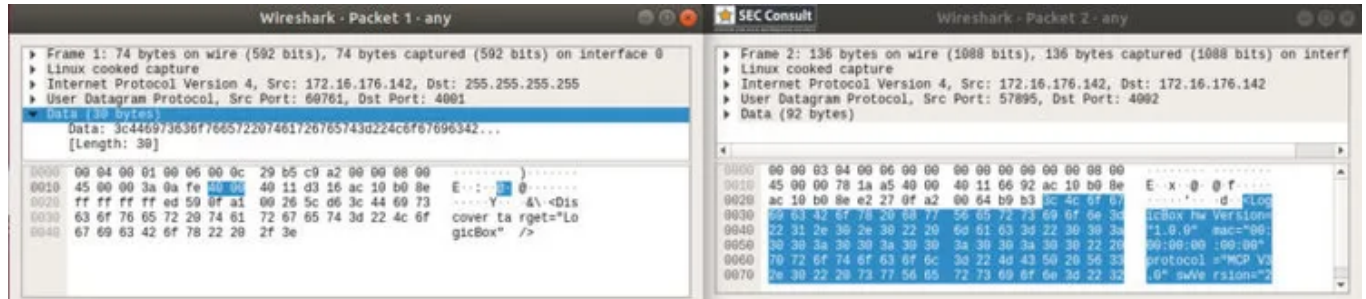
1. Passive sniffing with **Wireshark**
2. Active MiTM proxy

5.1. LAN

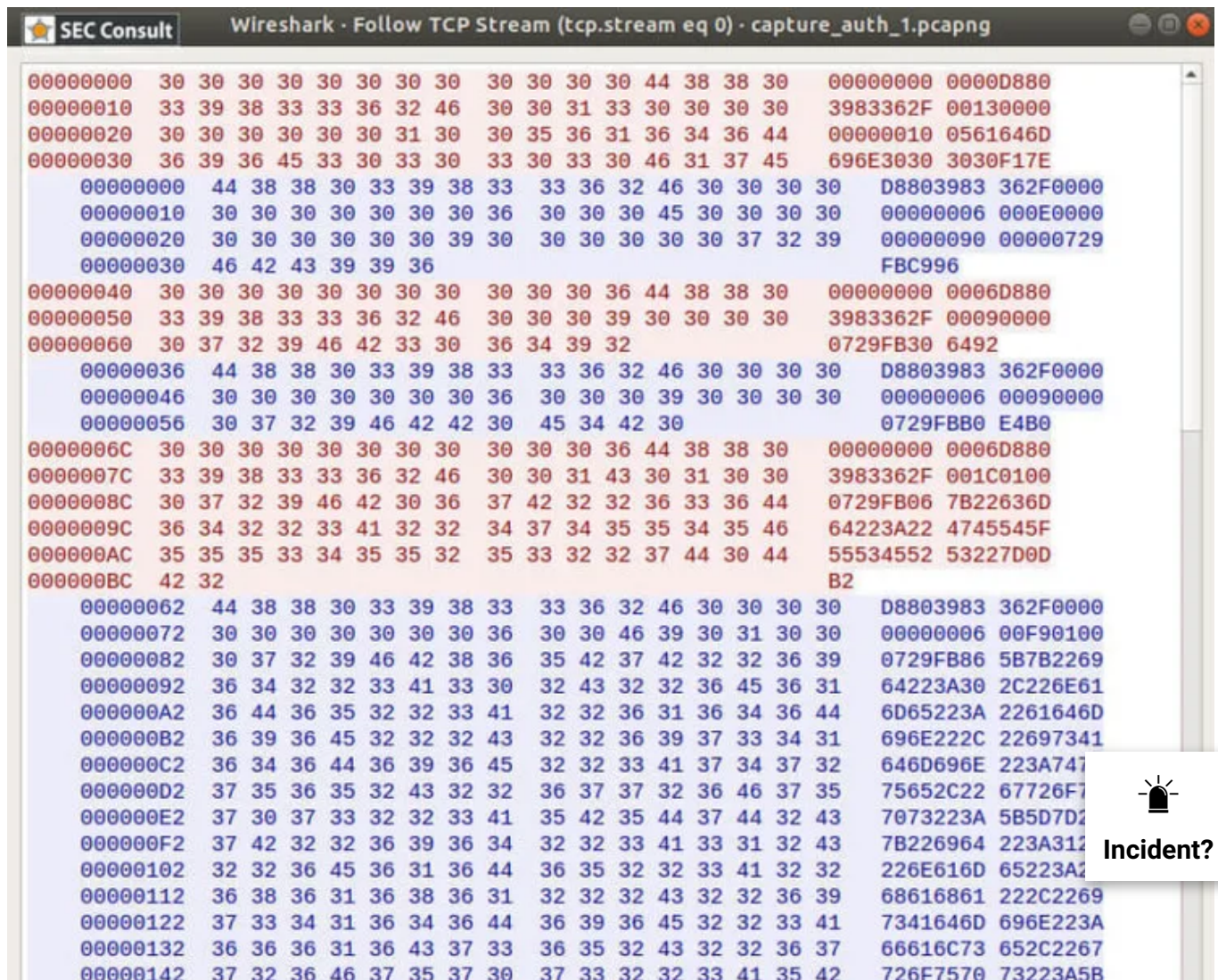
For the configuration of the LAN inspection, a Mikrotik router was set up with port mirroring copy of all packets to an analysis workstation that had Wireshark capture capabilities. The BiSecur Gateway device was then connected via Ethernet while the Android smartphone, with the **app** installed, was connected to the WIFI broadcast by the router. This allowed all traffic to be captured easily between the BiSecur Gateway device and the smartphone.

**Incident?**

that this is when the UDP traffic is transferred. By stopping the capture and checking the packets, it was found that the application was using UDP broadcast to search for devices. The broadcast traffic is an XML entity, as can be seen in the following image:



After reviewing the device search phase, I spied on the following communication. Here the configuration is the same, but this time I used the mobile application to connect to the device and perform a logon procedure.



Incident?

The stop of the capture after the end of the connection procedure allowed the analysis of the newly captured packets to continue. It turns out that the BiSecur Gateway device uses a custom text protocol, as Wireshark displayed the ASCII characters transmitted to and from the device. This was quite unusual. While searching through the data, I discovered that the strings were hexadecimal encoded bytes. So I **decoded the byte-coded traffic** and proceeded to analyze it. It was confirmed that this is a **non-standard protocol** (as could be guessed from the ASCII encoding), and this subtlety indicates that there could be **other security issues**. Just as you should not create your cryptography, you should **not unnecessarily design your custom protocol** (or at least not distribute it and use it in a consumer product). Designing a good quality protocol usually requires some effort, especially when it is subject to third-party review and testing. In 99.99% of cases, you will find that you can use an existing protocol that has been **thoroughly tested and verified**, provided with robust and implemented libraries.

Nevertheless, in an attempt to **reverse engineer** this custom protocol, most of the protocol has been deciphered by simply observing the bytes of captured traffic. Again, it is beyond the scope of this blog post to describe in detail the process of reversing a binary protocol, so I will just include how the structure was discovered:

MCP Packet (double hex-encoded)

+-----+
| SRC MAC [6 bytes] | DST MAC [6 bytes] | PAYLOAD [...] | CRC [1 byte] |
+-----+

Payload

+-----+
| LENGTH [2 bytes] | TAG [1 byte] | TOKEN [4 bytes] | COMMAND ID [1 byte] | COMMAND/RESPONSE DATA [variable] | CRC [1 byte] |
+-----+

There are some elements that we could not identify by traffic analysis alone: two-byte positions look like CRC, as the algorithm used is not easily identifiable. Moreover, I could only evaluate the connection structures at this stage. However, this information was sufficient to create a **parser**, providing a basic foundation for reading and understanding the communication, which can be transformed later into a **complete library** if necessary.

Some may wonder *how the mobile phone is going to communicate with the virtual device*. The UDP broadcast comes into play!

**Incident?**

2. The protocol is **plaintext** with no session protection or client authentication.
3. The protocol uses additional **hex** encoding for a non-apparent reason (reason why to be discovered later...).

5.1.2. Spoofing Device Discovery

As we have established, the mobile app uses UDP broadcast to discover available devices. This poses some problems:

1. UDP broadcast will only work with the same broadcast network. Alas, the device will be invisible if it's in a different broadcast range.
2. The entire discovery process is in plain text. The device response also reveals the device ID (this is the MAC address, which can also be discovered/enumerated using an ARP scan as it requires to be in the same broadcast address).
3. UDP packets can be easily spoofed.

What are the **consequences**? Using a python script, we can respond to these discovery broadcasts either as a custom device and/or as the actual device that is already on the local network.. The latter is more interesting because if we succeed, the mobile app will connect to our virtual device. And if we can implement the connection procedure, we will be able to get the connection credentials. **Remember how everything was in plain text?**

The spoofing script created and used can be found [here](#).

5.1.3. Communicating With The Device

The supported commands have been identified and can be seen to the left. All commands count as two because the same ID and format are used for the response to the specific command, but the first bit of the command identifier is set to 1 in case of response. the command structure: *"Where is the update command?"*

**Incident?**

Normally, the initial communication between the BiSecur Gateway and the smartphone application starts with the command GET_MAC followed by the command LOGIN, which includes the username

handled? Unfortunately, it doesn't seem as random as it should, usually starting with two zero bytes after successful login. Also, the protocol has no anti-brute force mechanism, unless you take into account the device's ability to self-DOS by sending packets of more than 1MBps. This means that an attacker only needs to guess 2 bytes in the best case, and only 4 bytes in the worst case. Let's not forget to mention that this TOKEN only gets invalidated in two cases:

1. The device powers completely off.
2. The LOGOUT command is sent with the token.

The LOGOUT command is never sent unless the user specifically clicks on the logout button which is "carefully hidden" on the user interface.

Now that the login sequence is noted, we can move on to the actual commands. There is not much to see, as the main function of the protocol is to open/close the doors as well as to give the user feedback on their current position. What is interesting is that the commands to open/close the doors (HM_GET_TRANSITION) require user authentication. This brings us to the next question: "*Which commands do not require authentication?*" After a fully customized implementation of the protocol, the answer came very easily. Just looping through all the commands without any arguments was enough to see that the device crashed. By rebooting the device and implementing a better iteration, with a timeout this time, it turns out that the DEBUG command causes the device to crash, and is invoked without authentication. This is a good start, what else could there be? Some interesting commands have been identified, which do not require authentication to be called successfully: DEBUG, ADD_USER, SCAN_WIFI, GET_MAC, PING, GET_USER_IDS. Let's see what we can do with them!

ADD_USER: Obviously, this allows you to create a user with a password. This wouldn't be a difficult problem to overcome without authentication, but since the device only supports a maximum of 10 user accounts, this could cause an availability issue. Furthermore, it does not appear to be in ASCII only and smartphone applications do not support any other encoding. By "not supporting", I mean if a user doesn't comply with the strict encoding rules defined in the username field, the application stops working for **ALL** users, and no one can log in anymore to operate their doors. Finally, the length of the password field in the command structure is not verified and causes a memory leak, which is not usable, however, because the credentials are written directly to the flash memory. Thus, an attacker could get a few bytes of the actual firmware using this technique, but due to the lack of protection mechanisms of the firmware initially found, there are much easier ways to read it.

**Incident?**

DEBUG: It was not possible to fully determine what this command does, other than crash the device without parameters. Since it does not require authentication, anyone on the same LAN can make the

ice can see.

```

1109 class MCPCommand(Enum) :
1110     PING = 0
1111     ERROR = 1
1112     GET_MAC = 2
1113     SET_VALUE = 3
1114     GET_VALUE = 4
1115     DEBUG = 5
1116     JMCP = 6
1117     GET_GW_VERSION = 7
1118     LOGIN = 16
1119     LOGOUT = 17
1120     GET_USER_IDS = 32
1121     GET_USER_NAME = 33
1122     ADD_USER = 34
1123     CHANGE_PASSWD = 35
1124     REMOVE_USER = 36
1125     SET_USER_RIGHTS = 37
1126     GET_NAME = 38
1127     SET_NAME = 39
1128     GET_USER_RIGHTS = 40
1129     ADD_PORT = 41
1130     ADD_GROUP = 42
1131     REMOVE_GROUP = 43
1132     SET_GROUP_NAME = 44
1133     GET_GROUP_NAME = 45
1134     SET_GROUPED_PORTS = 46
1135     GET_GROUPED_PORTS = 47
1136     GET_PORTS = 48
1137     GET_TYPE = 49
1138     GET_STATE = 50
1139     SET_STATE = 51
1140     GET_PORT_NAME = 52
1141     SET_PORT_NAME = 53
1142     SET_TYPE = 54
1143     GET_GROUP_IDS = 64
1144     INHERIT_PORT = 65
1145     REMOVE_PORT = 66
1146     SET_SSL = 80
1147     SCAN_WIFI = 81
1148     WIFI_FOUND = 82
1149     GET_WIFI_STATE = 83
1150     HM_GET_TRANSITION = 112
1151     CHANGE_USER_NAME = 67
1152     CHANGE_USER_NAME_OF_USER = 68
1153     CHANGE_PASSWORD_OF_USER = 69

```



Incident?

5.2. Internet

The capture of the communication between the BiSecur Gateway device and the Hörmann backend

initially prohibited performing a MiTM between the device and the backend server. This meant that I could only see encrypted traffic.

At this point I needed to position myself appropriately to perform a MiTM attack in the presence of SSL. Remember the Flash memory analysis? Well, I ended up generating my own **Certificate Authority** (CA), client certificate and private key, and then **inject them into the Flash chip** at the correct position in the memory.. Further traffic analysis showed that the BiSecur Gateway device was not capable to communicate with the server after this action. I then wrote a python script that acted as a generic TCP proxy, loading my malicious CA on the server side of the proxy and using the correct client certificate on the client side of the proxy. **Why not use Burp or other well-established web proxy solutions?** Actually, there have been attempts to use them, but they all failed to provide the required solution.. The reason for this is that although the device uses TCP/443 and SSL, the underlying protocol was not HTTP, but turned out to be the same custom protocol that was discovered when performing the local LAN analysis. Burp and other web proxies were therefore **not able to intercept and interpret the traffic correctly**.

After setting up the TCP proxy (which essentially transmitted traffic between the device and the server in 4096 byte increments, because at first I didn't know the protocol used), the next step was to make the BiSecur Gateway device connect to my proxy host instead of the Hörmann server. It turned out that the domain name of the server **was hard-coded in the firmware** of the device, which I did not want to change for security reasons. So, I tried to spoof the DNS responses during the initial resolution of the name. Unfortunately, it didn't work. When the device did not get an IP address within the "expected" IP range, it settled for an infinite loop of resolution of the same domain name. This is not really a common way to perform DNS resolutions, because an assumption is either a **protection mechanism or a programming error** in the firmware. After a few more hours spent solving the problem, I ended up using a USB-Ethernet dongle on my proxy server and, while it seemed unlikely, I set its IP address to the same /24 network range as the original Hörmann server. To my surprise, it worked.

At this point, I have a MiTM running between the BiSecur Gateway device and the Hörmann server; time to check the communication! As mentioned before, the device uses exactly the same protocol on the Internet as on the local network, but with an additional SSL layer. This means that if I create a working communication library for this protocol, I will be able to communicate with the remote server as a BiSecur Gateway device, provided I have the right SSL certificate and the right private key. All this is interesting, but first let's look at the **communication flow to open a door to the Internet**.

By logging traffic with the custom TCP proxy and using the partial parser written from the information collected by the LAN packet analysis, I could print the messages on the console.

**Incident?**

2. Remote Server: GET_MAC
3. Client: response GET_MAC with the client's MAC address
4. Phone login button pressed
5. Remote Server: LOGIN <unencrypted credentials>
6. Phone: successful LOGIN response indicated in the TOKEN field (changes from 0 to random integer)
7. Remote Server: GET_INFO
8. Client: GET_INFO response

Side Note: GET_MAC is used as a **keepalive**, every 30 seconds the device receives a GET_MAC message.

Nothing seems strange to you? We used a client certificate to perform strong authentication, while the client certificate itself contains the MAC address of our device in the Common Name(CN) attribute. At this point, you may wonder why the Hörmann server asks the BiSecur Gateway device to provide the MAC address again through a separate request in the established session. The first assumptions were that it is used simply to be able to perform a **keepalive** (with later reverse engineering which discovered that there was also a PING command available) or to **identify the device** based on this information? Actually, YES...

To test this initial theory, I used the (now complete) implementation of the communication library I developed to respond to the server with an incorrect MAC address when receiving a GET_MAC request. An attempt was then made to connect to the BiSecur Gateway device via a smartphone. **This did not work.** To test this hypothesis, I went to buy another BiSecur Gateway device. The reason for this decision was to test my hypothesis on privately owned devices, for obvious reasons that I would not have tested on other existing customers, nor would I have wanted to cause any damage to Hörmann's infrastructure. By installing the second device and coupling it with the smartphone application according to the normal procedure, this made it possible to **"hijack"** the secondary device via the Hörmann server. After configuring a user with a secure password on the second device of the BiSecur gateway, I started my virtual device using the SSL key of the first device to connect to the remote server, but injecting the MAC address of the second device into the GET_MAC request. After pressing the login button of the smartphone application, the virtual device **received the secret credentials** instead of the real second device. This verified that via the virtual client, I could retrieve all the credentials of all the client devices connected to the remote Hörmann server over the Internet by spoofing the MAC address after the connection.

**Incident?**

As it turned out later, the Ethernet controller chip is also purchased from Microchip and used in all

6. WIFI

The device WiFi security was not in scope for this blog post and therefore not tested.

7. SMARTPHONE MOBILE APPLICATION

The BiSecur gateway product can be used and controlled by downloading the appropriate application to your smartphone both available for **Android** and **iOS**.

7.1. Android Application

After downloading the official Android application .apk file, time was spent decompiling the dex file and checking the included binary libraries. However, it quickly turned out that this step was not needed at all, as all the libraries and classes had only one feature: to set up a **Flash** environment and execute an **SWF** file bundled in the resources folder.

However, after further decompiling the SWF file and reviewing the code, I could verify and confirm that the bundled SWF file is in fact the actual utilized application and not just another virtual environment. Also, the resources within the SWF file were found to contain some “*advanced hardcore*” protection against reverse engineering efforts. Below you can bear witness to this protection mechanism used, with the actual protection file as found, within the “assets” folder of the apk.

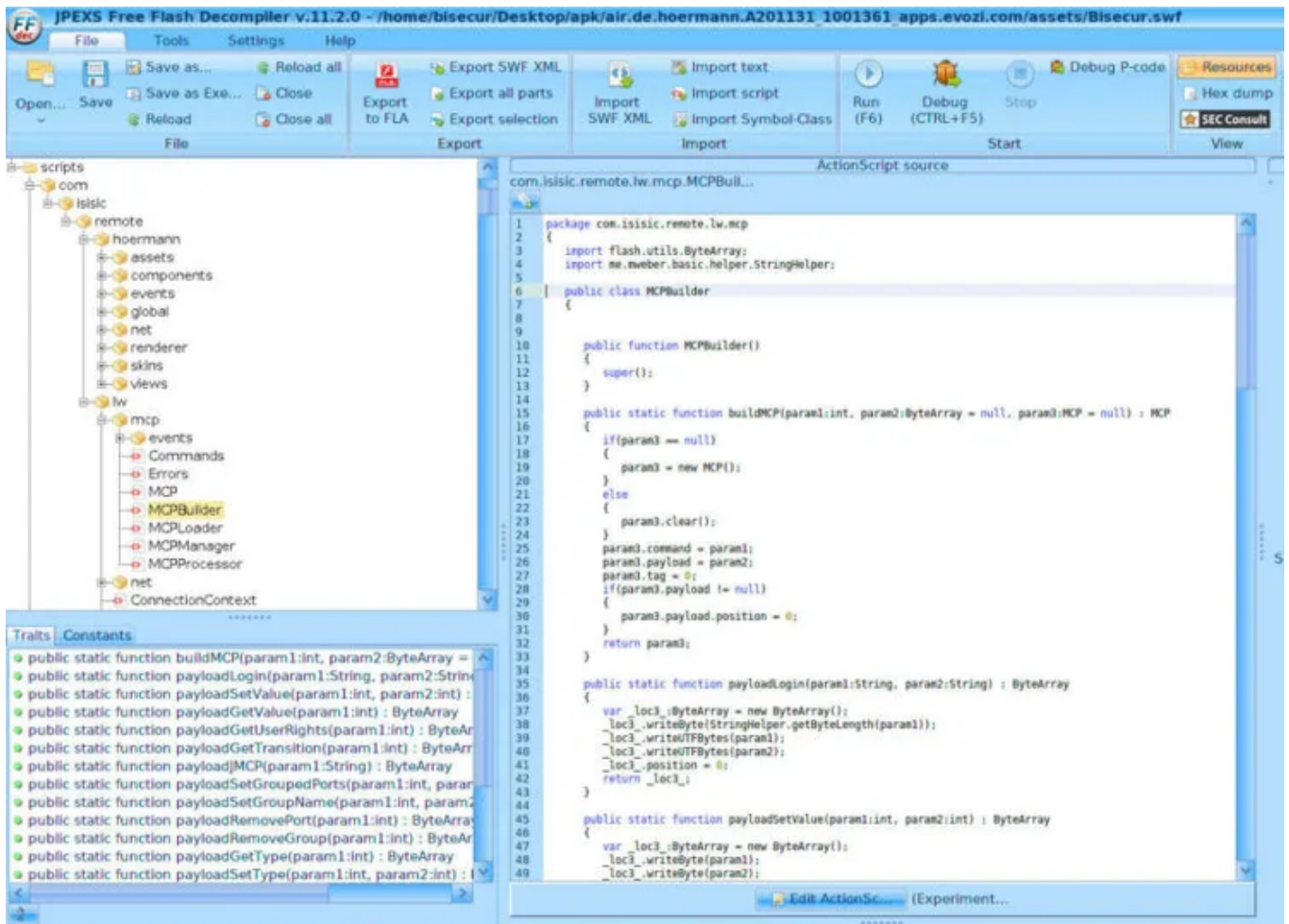


Incident?



SEC Consult

an Eviden business



It appeared to be the source code of the binary protocol used for communication, that not only contains the data structures but the utilized CRC algorithm and the list of all available commands. Now armed with this information, I was able to create a full communication library called **pysecur3**

7.2. iOS Application

The iOS app was not in scope for this blog post and therefore not tested.

8. Conclusion

The tested device (just like many other IoT devices) would require a complete redesign on a including hardware, protocol, back-end infrastructure.



Incident?

Hörmann was informed by SEC Consult about the potential security risks of the BiSecur gateway and responded promptly. Without delay, the registration option on the official BiSecur portal was switched

9. About the author

Tamas Jos is a Senior Security Consultant at SEC Consult (Schweiz) AG.

He specializes in hardware reverse engineering in particular.

10. Vendor's Statement

(Excerpt)

The algorithm of the check code for registering the BiSecur gateway has been adapted promptly after being informed by SEC Consult. The verification code is longer, more cryptic and contains random values.

All customers who registered a gateway through the portal until the security risks became known have been informed that a new verification code is required. In compliance with the necessary security requirements, a new verification code was sent to these users by post.

The password specification in the BiSecur app has been updated. After a quick update, a 10-digit password with upper- and lower-case letters, numbers and special characters is required.

In addition, the software of the BiSecur gateway has been adapted. It now has a readout protection of both controllers.

Hörmann is constantly working to optimize the quality and security of all products. The detection of potential weaknesses by SEC Consult proved helpful to further improve the entire BiSecur system.



Blogpost SEC Consult Statement Hörmann

Hörmann ist von SEC Consult über die potenziellen Sicherheitsrisiken des BiSecur Gateways informiert worden und hat daraufhin sofort reagiert. Unverzüglich wurde die Registrierungsmöglichkeit am offiziellen BiSecur Portal abgeschaltet und die Fertigung der BiSecur Gateways vorläufig eingestellt. Die Produktentwickler haben mit höchster Priorität die genannten Schwachstellen geprüft und in kurzer Zeit alle relevanten Sicherheitsrisiken behoben.

Zum einen ist aus dem Algorithmus des BiSecur-Portals...

Hörmann KG Verkaufsgesellschaft
Tore · Türen · Zargen · Antriebe

Telefon: [REDACTED]

Telefon: [REDACTED]

Telefon: [REDACTED]

E-Mail: [REDACTED]

Download Texte und Bilder:
www.hoermann.de/presse



Incident?

kanntwerden der Sicherheitsrisiken angemeldet haben, wurden darüber informiert, dass ein neuer Prüfcode erforderlich ist. Unter Beachtung der erforderlichen Sicherheitsanforderungen wurde diesen Nutzern ein neuer Prüfcode postalisch zugestellt.

Zum anderen ist die Passwortvorgabe in der BiSecur App aktualisiert. Nach einem kurzfristigen Update ist ein 10-stelliges Passwort mit Groß- und Kleinbuchstaben, Zahlen sowie Sonderzeichen erforderlich.

Außerdem ist die Software des BiSecur Gateways angepasst. Sie verfügt nun über einen Ausleseschutz der beiden Controller.

Hörmann arbeitet stetig daran, die Qualität und die Sicherheit aller Produkte zu optimieren. Die Aufdeckung potenzieller Schwachstellen durch SEC Consult erwies sich als hilfreich, um das gesamte BiSecur System weiter zu verbessern.

[← Back](#)

[Legal Notice](#) [Privacy Statement](#) [Jobs](#)

SEC Consult is one of the leading consultancies in the field of cyber and application security. The company specializes in information security management, NIS security audits, penetration testing, ISO 27001 certification support, Cyber Defence and secure software certification. SEC Consult is part of Eviden, an Atos business.

**Incident?**