Vacuums in the Cloud: Analyzing Security in a Hardened IoT Ecosystem

Fabian Ullrich, Jiska Classen, Johannes Eger, Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt
{fullrich, jclassen, jeger, mhollick}@seemoo.de

Abstract

With the advent of robot vacuum cleaners, mobile sensing platforms entered millions of homes. These gadgets not only put "eyes and ears" into formerly private spaces, but also communicate gathered information into the cloud. Furthermore, they reside inside the customer's local network. Hence, they are a prime target for attacks and if compromised become a privacy and security nightmare. Vendors are aware of robots being a target of interest; they employ various security mechanisms against tampering with devices and recorded data in the cloud.

In this paper, the *Neato BotVac Connected* and *Vorwerk Kobold VR300* ecosystems are analyzed and the robot firmware is reverse engineered. To achieve the latter, a technique to bypass the devices' secure boot process is presented revealing the firmware, which is then dissected to evaluate device-specific secret key generation and to trace vulnerabilities. We present flaws in the secret key generation and provide insight on the occurrence and exploitation of a buffer overflow, which give an attacker complete control not only in the local network but also via the robots' cloud interface. Eventually, multiple attacks based on the findings are described and security implications are discussed. We shared our findings with the vendors, who further increased their otherwise commendable security mechanisms, and hope more vendors can take away valuable lessons from this highly complex Internet of Things (IoT) ecosystem.

1 Introduction

Smart vacuum cleaning robots contain many interesting sensors, such as laser scanners, ultrasonic distance sensors, cameras, and bump detectors. Combined with cleaning schedules and result maps, this exposes a substantial amount of private information about their owner—accordingly, iRobot's announcement to sell Roomba data for marketing purposes has been discussed controversially [2]. With the customer's shipping address, WiFi configuration, or simply position data collected by the robot or the app, apartment size and furniture can be associated with locations. All market leading vacuum cleaning robots are embedded into complex IoT ecosystems. Xiaomi robots were subject of significant security research including an open source replacement for their proprietary cloud components [7]. However, Xiaomi uses a different infrastructure, hardware architecture, and operating system—meaning that while Neato and Xiaomi are functionally similar, the security measures have to be different.

With a market share of 62.7 million € in 2017, Neato is one of the bigger players on the vacuum robot market [20]. Vorwerk acquired Neato in 2017 and sells a robot with almost identical firmware, which is part of the vacuum cleaning product line *Kobold* listed with 800 million € revenue in total.

Even though vacuum cleaning robots are a target of interest, not much security research has been published on Neato, except for an exploit chain for attackers in the same local network [10, 11]. For Vorwerk robots, which are re-branded Neato robots with minor hardware and settings upgrades, not a single Common Vulnerabilities and Exposures (CVE) number is known. Despite robot vacuums not being broadly considered to be security critical, Neato put plenty of advanced security techniques in place, which are interesting to analyze and learn from for all kinds of IoT devices. We show that the combination of multiple vulnerabilities in different components leads to a powerful attack compromising the entire ecosystem. Our main contributions are:

- Secure boot bypass and extraction of Random Access Memory (RAM) with previous contents,
- execution of commands on robots over the cloud with superuser privileges without authorization,
- massive entropy reduction in robot secret key generation,
- decryption of RSA private keys on robots and enabling arbitrary command and control channels over their cloud,
- information leakage of end-user smartphone's public Internet Protocol (IP) addresses.

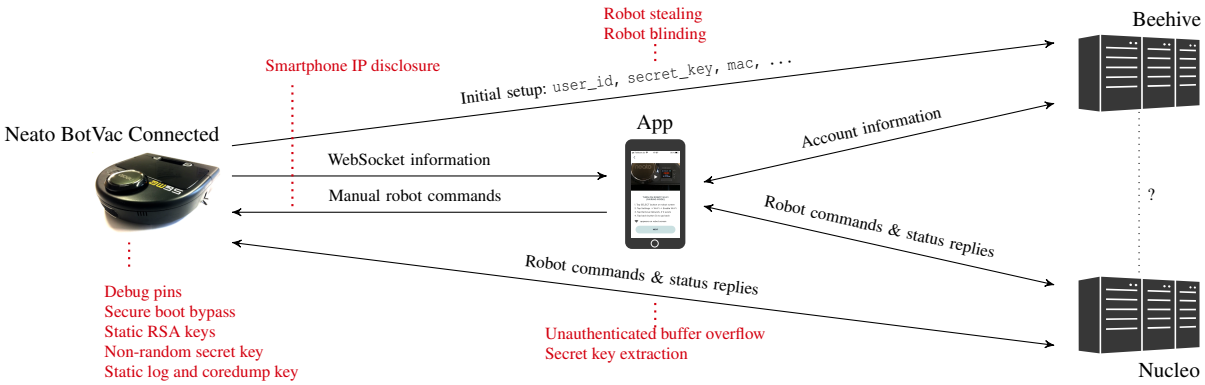

Figure 1: The Neato ecosystem. During normal usage, the smartphone application only communicates through a cloud infrastructure with the robot. Vulnerabilities found in certain components or communication paths are marked red.

All security issues were communicated to the vendor. Remote exploitation over the cloud was filtered immediately, and firmware updates fixing bugs closer to hardware were rolled out until June 2019. Adequate patches on some encryption flaws required changes in the device manufacturing process.

Our research highlights that obscurity and the usage of strong encryption methods without proper knowledge of the underlying schemes does not provide sufficient security. Our analysis and recommendations are practice-oriented and of interest to other vendors and security analysts. This is especially relevant as some technologies utilized by Neato are also present in the automotive and other interconnected sectors. We encourage vendors to work on open solutions for security and establish better standards. In the long run this will solve security issues similar to the ones found in our analysis.

This paper is structured as follows: Section 2 gives a brief overview on the robot ecosystem components and the messages exchanged with special emphasis on security. While Section 3 focuses on exploitation with local hardware or network access, Section 4 shows cloud exploitation techniques. Security implications of our findings are discussed in Section 5. Section 6 concludes our work.

2 The Neato and Vorwerk Ecosystems

Advanced smart robot vacuum cleaners are able to clean the house almost autonomously. The only parts requiring interaction are emptying the dirt bin and starting the robot. Neato advertises the connected series with the ability to control the robot and get notifications via smartphone as well as obtaining a cleaning summary [16]. To achieve this, the ecosystem features multiple components displayed in Figure 1. The smartphone app communicates with two different application programming interfaces (APIs), namely *Beehive* and *Nucleo*. *Beehive* (`beehive.neatocloud.com`) manages all account-related tasks, such as account information and linked robots. The robot itself communicates with the *Beehive* cloud during initial pairing to exchange authentication information. In contrast, *Nucleo* (`nucleo.neatocloud.com`) provides robot functionality. It receives commands from the smartphone app and forwards them to the robot. In reverse direction, *Nucleo* provides the app with robot status messages. Moreover, a subset of commands can be sent to the robot directly without using the cloud in manual mode. The robot platform itself is based on a QNX system [3].

Product Name	Firmware Version	Build Date
Neato BotVac Connected	2.2.0_296_	Dec 12 2016
Neato BotVac Connected D7	4.2.0-102	Jul 12 2018
	4.4.0-72	Dec 22 2018
Vorwerk Kobold VR300	4.2.2-137	Aug 30 2018
	4.2.5-166	Jan 11 2019

Table 1: Firmware version comparison. *Neato* is used to test new features, *Vorwerk* is getting stable releases only.

Neato, like most vendors, does not provide public details about the inner life of their ecosystem. All information in this section was gathered either by sniffing network traffic or by reverse engineering the *Android* smartphone app. Additionally, Neato provides a developer API documentation [17].

We dissected multiple products, which we compare in Section 2.1. Section 2.2 explains further details on the *Beehive* cloud while Section 2.3 focuses on the *Nucleo* cloud.

2.1 Product Comparison

To confirm and compare our findings, we tested three vacuum cleaning robot models. Vorwerk robots are very similar to Neato; in fact they use the same firmware just with different settings applied. Firmware versions are listed in Table 1. The Vorwerk *Beehive* and *Nucelo* cloud are hosted on `beehive.ksecosys.com` and `nucleo.ksecosys.com`. Vorwerk hardware has some upgrades compared to Neato hardware, such as more ultrasonic sensors, a larger battery, different power and speed settings for normal and eco vacuum cleaning mode, a green brush, and rainbow colored LEDs.

2.2 Beehive Cloud

Beehive is responsible for all account-related information. It holds the `secret_key`s of all robots that are connected to a user account. The `secret_key` is required for communicating with the *Nucleo* cloud and to send manual commands to the robot. *Beehive* is hosted at *Amazon* [1] via *Heroku* [18].

2.3 Nucleo Cloud

The *Nucleo* API represents the interface for smartphone↔robot communication. It provides robot status information and forwards commands to the *Neato BotVac Connected*. The *Nucleo* infrastructure is hosted with *Amazon* via *Heroku* as well. All messages to the API are payload-based with the constant path `/vendors/neato/robots/[robot_serial]/messages`.

Authentication In the *Nucleo* API, `Authorization` and `Date` header fields are used for authentication. The authorization field is generated by concatenating the message body to the date and afterwards concatenating the result to the lower case `robot_serial`. Then the SHA-256 Keyed-Hash Message Authentication Code (HMAC) [6, 9] of the resulting string is computed, using the robot `secret_key`. The computation instructions in pseudo-code are given in Listing 1.

On sending a message to a distinct robot, the server checks for `Date` header validity. If a date is provided and lies within a 20 min time window of the current time, the server checks for presence of the `Authorization` header. When providing the "NEATOAPP" string in the `Authorization` header but not giving a HMAC or an invalid HMAC, the server's response is `{"Could not find robot_serial for specified vendor_name"}`. This behavior also applies when trying to access random `robot_serial`s, which means it is not possible to confirm the existence of `robot_serial`s in the Nucleo API. Note that the server does not check the `Authorization` header validity itself, instead the request is forwarded to the robot which then validates the HMAC. To summarize, anyone who is in possession of a `robot_serial` and `secret_key` is able to send commands to the corresponding robot.

```
authorization = "NEATOAPP " + signature
signature = HMAC_SHA256(robot_secret_key,
    UTF8_encoded(string_to_sign))
string_to_sign = lower(robot_serial) + "\n" +
    date_header + "\n"  + body
```

Listing 1: Neato *Nucleo* authentication between smartphone and backend [17].

3 Local Vulnerabilities

We use a new secure boot bypass technique in Section 3.1 to extract system image and RAM contents. Static system image analysis in Section 3.2 is the base for findings across the remaining part of the paper, which includes extensive `secret_key` generation entropy reduction in Section 3.3.

3.1 Bypassing Secure Boot

Access to the firmware running on the robot is required for more advanced security research. This includes but is not limited to analysis of mutually authenticated Transport Layer Security (TLS) connections and debugging of running applications. Neato is using a custom *AM335x* chip with secure boot enabled [14]. The flash chip only contains an encrypted and signed system image; a physical attacker cannot simply unsolder the chip to obtain it. Firmware updates use the same encrypted and signed format. Changing and debugging firmware on a robot therefore requires bypassing secure boot.

We start with a hardware analysis in Section 3.1.1, which leads to a hidden boot menu in Section 3.1.2. We exploit this to extract Neato's system image (see Section 3.1.3). The described security issue is filed as CVE-2018-20785. It has been fixed in the *Neato BotVac Connected D7* firmware version 4.4.0-72, all other versions listed in Table 1 are vulnerable. As of June 2019, also the *Vorwerk* firmware has been fixed.

3.1.1 Hardware Analysis

After teardown of the *Neato BotVac Connected*, further analysis of all components is required. The motherboard has many testing points and undocumented chips, with an important part of it depicted in Figure 2. We connected all testing points to a logic analyzer, and rebooted the robot multiple times. Most of the testing points do not show any interesting output, some might be for sensors and other features not relevant for system access. In addition to test points, the right front of the motherboard contains three extra large unlabeled pins, which are a serial interface with clock, read, and write. The ground needs to be taken from another source, such as the internal Universal Serial Bus (USB) port. It prints the following text during startup and remains silent afterward:

```
BotVac Connected "<Fast Memory>"(84038)
```

The *Neato BotVac Connected D7* features a similar serial port, it is located on the motherboard's left and already has four connection pins. Access to the *Vorwerk Kobold VR300* serial port does not require disassembling it, it is located next to the USB port.

The main chip on the *Neato Botvac Connected* is the *SNI20952608ZCE* in the center of Figure 2. Since it is a custom chip there is no information or documentation available. The form factor *962B* leads to the *AM335x* chip family manufactured by Texas Instruments.

3.1.2 Hidden Boot Menu

During startup, QNX first loads an Initial Program Load (IPL), which then loads an Image File System (IFS) containing the operating system. Instead of a QNX IPL, Linux typically uses the more common setup of MLO and U-Boot on *AM335x*. Either way, these setups enable to boot an operating system. Secure boot uses encryption keys configured on the *AM335x* decrypt and verify both, IPL and IFS, prior to execution.

Neato's standard setup is to simply boot the IFS containing the firmware from the external flash chip without showing any options. While testing the partially documented Neato USB command line [15], we found that `TestMode On` followed by `SetSystemMode PowerCycle` enables a hidden IPL boot menu on the serial interface:

```
BotVac Connected "<Fast Memory>"(84038)
Press enter twice within the next 2 seconds for
    boot menu
**Commands:
Press 'M' to load IFS from main image flash
Press 'X' for serial download, using XModem-1k
Press 'S' to scan existing memory without download
```

Note that `XModem-1k download` makes the robot download a system image from the serial port and boot it. This does not enable system image extraction. On uploading a valid QNX IFS no secure boot checks are performed, representing a bypass. Loading a *BeagleBone* QNX IFS from *Foundry27* for *AM335x* works in principle, it prints some information on the serial interface but crashes when initializing processors.

The original *AM335x Foundry27* board support package includes a very similar IPL procedure in `src/hardware/ipl/boards/am335x/main.c`. It enables sending an IFS over the QNX serial protocol `sendnto`. Neato's implementation represents a variation and more *AM335x*-based products might be affected by the secure boot bypass. The more common setup of MLO and U-Boot also allows to load arbitrary images by default. In general, vendors relying on secure boot should carefully check for such contradicting setups.

3.1.3 Memory Extraction

We exploit the secure boot bypass to launch a cold boot attack, which is possible as neither the *AM335x* chip nor the IPL reset memory contents during startup. When recompiling and booting the *Foundry27* image, the main issue is that hardware

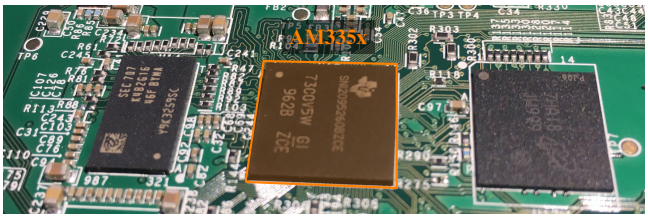

Figure 2: Part of the *Neato BotVac Connected* motherboard.

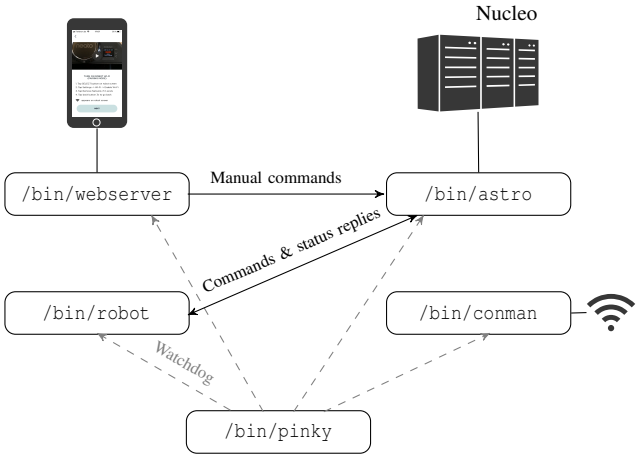

Figure 3: Core tasks running on *Neato BotVac Connected* and their most important actions.

cannot be initialized. This is since memory is mapped to different addresses on the Neato chip variant and any invalid memory access freezes the chip.

At this stage an attacker is able to print memory contents until an invalid memory access is made. In our experiments, increasing the serial interface speed up to 1 Mbit/s worked. Despite slow speed and partially unknown memory layout, this poses a huge attack surface for cold boot attacks.

By default, QNX is located at address `0x81000000`. First memory dumps show that despite booting our own QNX IFS this memory region still contains some of the original Neato IFS contents. We relocate our custom QNX IFS to a free memory region and perform a full memory dump of the upper 2 GB of the address space. This area contains the same 256 MB, which are mapped 8 times to the memory region. Each mapping contains the full Neato IFS and any RAM contents present before the reboot, including WiFi credentials.

3.2 Static Firmware Analysis

Neato's IFS is a rich source for information and enables static firmware analysis. It contains all important binaries and shell scripts coordinating them. For example, `/bin/robot` is called by a script `/etc/brain.sh` and the watchdog `/bin/pinky` is started via `/etc/pinky.sh`. An overview of the main tasks is shown in Figure 3. `robot` is the largest binary and responsible for the main features, starting at the interactive command line and ending at all cleaning logic. It receives *Nucleo* cloud commands from `astro` and generates replies. The manual drive mode which is issued locally via the smartphone app is first redirected from `webserver` to `astro` and then processed similarly. `conman` manages WiFi connections. Messages are exchanged via proprietary `libNeatoIPC` calls as well as standard QNX message queues. In the following we focus on those components relevant for further findings.

Testing the robot's wireless interfaces produced coredumps that are separate files in the so-called "Blackbox Logs", which can be copied to a USB stick connected to the Neato USB interface. Such a coredump is the entry point for our analysis in Section 4.2. For security reasons, these coredumps are encrypted. A key question for further research is how this encryption works. Shell scripts in the Neato IFS copy and rotate these logs, encryption is performed by `/bin/rc4_crypt` called without arguments. Reverse engineering `rc4_crypt` reveals the hard-coded password `*^JEd4W!I`. This password is valid for each of the three tested robots. We assume the same password is used throughout all firmware versions. We published this vulnerability as CVE-2018-17177.

The robot features a number of encrypted private RSA keys which are located in `/var/keys/`. One of the keys is named `vendorPrivateKeyProduction` and is used for authenticating the robot against the cloud components. By examining the key decryption procedure in the robot binaries, we were able to decrypt the keys and launched several tests as described in Section 4.1. Due to the high impact of these keys becoming public, we agreed to not disclose details on the decryption process.

3.3 Secret Key Generation

First hints for the `secret_key` generation are found in the `astro` binary since it features a `GenerateSecretKey` function. This function calls multiple other functions to finally call `NeatoCrypto_GenerateRobotPassword`, which is imported from `NeatoCrypto`. After renaming and adjusting type declarations, we were able to reconstruct the function as shown in Listing 2. The first important observation is the usage of the OpenSSL library for computing the final `secret_key`, which appears to be a SHA-1 [5] hash. Type declarations were derived from the OpenSSL documentation. Contents passed to SHA-1 are the following blocks of bytes:

`t_shift[0-3]` The first block is set to the current Unix time in seconds.

`t_shift[4-6]` is set to constant 0.

`t_shift[7]` is set to constant 16.

`t_shift[8]` is set to a random value with some mathematical operation.

`t_shift[9]` is set to a random value with multiple mathematical operations. Since one of them is an `AND` with the value `0x3F`, the first two bit of this byte are set to constant 0.

`t_shift[10-15]` is the robot's Media Access Control (MAC) address, which is a 6 B hex-string.

In the next step, a hash for the 16 B `t_shift` variable is computed. If computing the digest was successful, the hash is then stored at the address contained in

```
ciphers = OpenSSL_add_all_ciphers(v8)
OPENSSL_add_all_algorithms_noconf(ciphers)
EVP_MD_CTX_init(&v18)
rnd = rand()
t_shift[0:3] = time_now
t_shift[4:6] = 0
t_shift[7] = 16
t_shift[8] = rnd + rnd / 0xFFFF
t_shift[9] = ~(~((((unsigned int)(unsigned short)
  (rnd + rnd / 0xFFFF) >> 8) & 0x3F) << 25) >> 25)
t_shift[10:15] = robot_MAC
v_sha1 = EVP_sha1()
if ( !EVP_DigestInit(&v18, v_sha1)
  || !EVP_DigestUpdate(&v18, t_shift, 16)
  || !EVP_DigestFinal(&v18, robot_key, 20))
    return 7000
filehandle = fopen(*emmc_keys_robotkey, "w")
fwrite(robot_key, 1, 16, filehandle)
```

Listing 2: Secret Key Generation.

the pointer `emmc_keys_robotkey` which leads to the file `/emmc/keys/RobotKey`. The first 16 B of the `robot_key` variable are then written to that file.

In Section 5.1.3 we show that the `robot_serial` (and thereby the `robot_MAC`) has to be considered as known, which eradicates the entropy of `t_shift[10:15]`. The same is true for the constant values, eliminating the entropy for `t_shift[4:7]`.

The bytes `t_shift[8:9]` are random, but one of them has two bits constantly set to 0. Given the random number generator (RNG) was properly seeded, this would add 14 bit of entropy to the `secret_key`. Since the RNG is not seeded, these two bytes do not add entropy in case the attacker knows the state of the RNG. Our experiments show that the RNG state is always the same after rebooting or resetting a robot. We were able to confirm, that newly unboxed robots have the same RNG state as well. Note that the RNG state is the same amongst all robots and not limited to single devices.

The first 4 B of the `t_shift` structure are the Unix time in seconds, which is close to the time where the robot connects to the *Beehive* cloud for the first time. Since all other values for the `secret_key` generation are known, the entropy consists of timestamp entropy plus 14 bit from the random values in case the RNG state is not known. If the RNG state at the time of `secret_key` computation is known, the entropy exclusively relies on the timestamp. Further security implications and attacks are discussed in Section 5.1.3.

4 Remote Vulnerabilities

In Section 4.1 several attacks on the robot interface of the Neato and Vorwerk cloud are presented. Section 4.2 analyzes a buffer overflow vulnerability in the robot, which is triggered by sending forged requests to the cloud.

4.1 Robot Backend Vulnerabilities

Obtaining the private RSA keys for the Neato and the Vorwerk cloud (compare Section 3.2) offers a new attack vector against the ecosystems. We give an overview of the message types that are exchanged between robot and cloud and discuss resulting security problems.

4.1.1 Authentication

Robots authenticate against the backend by using RSA signatures in the `Authorization` header, as detailed in Listing 3. The header consists of two parts, separated by the `:` character. The first part is the `robot_serial` and the second part the base64 encoded RSA signature. The signature signs similar content as the `Authorization` header presented in Section 2.3. It signs the `robot_serial`, the Hypertext Transfer Protocol (HTTP) request method, the request Uniform Resource Identifier (URI), and the request body. Since some of the backend messages exchange the `secret_key` discussed in Section 3.3, backend requests do not utilize it and instead entirely rely on the RSA private key.

Robot Impersonation Attack The RSA private key is identical for all robots and the robots do not submit any other authentication factor. This enables an attacker to send messages to the backend on behalf of any robot with a known `robot_serial`. Problems caused by this issue are explained in the following sections.

4.1.2 Beehive

The *Beehive* backend is exclusively used for linking a robot to a new account and exchanging the new secret key. Under normal circumstances a robot only has to send a request to the backend once in a lifetime. A linking request contains several information about the robot, among others the robot's `secret_key` and the `user_id` of the account which the robot will be linked to. A linking request as sent by a robot is displayed in Listing 4.

Robot Stealing Attack The Robot Impersonation Attack enables an attacker to send such linking requests on behalf of arbitrary robots. This allows to change the user account to which a robot is linked to. With the means presented in Section 3.3 it is possible to compute the current valid `secret_key` and fully take over control of a robot.

Changing Keys Possession of the private RSA key also allows for changing the `secret_key` stored in the cloud for a given `robot_serial`, which results in the Robot Blinding Attack described in Section 4.1.3.

```
string_to_sign = utf8(robot_serial + "\n" +
 http_req_method + "\n" + req_URI + "\n" +
 date_time "\n" + req_body)
utf8 = convert_to_utf8(string_to_sign)
signature = sign_rsa_sha256(utf8, private_key)
enc_signature = base64encode(signature)
```

Listing 3: Robot signature, used between robot and backend.

```
POST /links HTTP/1.1
Host: beehive.neatocloud.com
Authorization: NEATOBOT OPS2[red]-508cb1[red]:
    jRB31k+D0qu7TkgwDkPyezmhn8XrEzoKoVXJq1W[short]
User-Agent: Neato-Robot

{"user_id":"95ac1f189b434[red]",
"mac_address":"508cb1[red]",
"model":"BotVacConnected",
"secret_key":"295DC2BA26A221060[red]",
"firmware":"2.2.0","name":"Test"}
```

Listing 4: Neato *Beehive* robot link (shortened).

Information Leakage With the means to impersonate arbitrary robots, we were also able to test the server's error handling in regard to information leakage. When providing the backend with a random `user_id` an `{"user_id":"invalid"}` error is returned. This allows to verify the existence of `user_id`s in the cloud. Since `user_id`s are randomly generated and are 32 B long, enumeration should not be feasible.

The same cloud behavior is achieved by sending random `robot_serial`s to the server. Since the MAC part of the serial is easy to obtain, only the first part resembling the production date remains unknown. The production date has little entropy and therefore can be bruteforced in an online attack against the backend. When a valid MAC with a wrong production date is sent, the server answers with an `{"mac_address":"taken"}`.

Fake Identities There is no verification if vacuum cleaners were actually produced, it is possible to add arbitrary robots with a forged `robot_serial` via the *Beehive* `/links` path.

4.1.3 Nucleo

The *Nucleo* cloud backend forwards app commands to the robots and receives their status updates. A robot↔*Nucleo* connection is a bidirectional stream, one connection is used for multiple messages. This portion of *Nucleo* is not HTTP conform. When a robot logs into the cloud with a request to `/vendors/neato/login` (compare Listing 5), the cloud answers with `transfer-encoding: chunked` and does not give a `Content-Length`. It also sends a `Connection: close` header, which normally ends the connection. If the connection is not closed from client side, the server starts to send

heartbeat messages and control commands. The connection is used as a socket from this point on.

The data format of one message in this socket connection is a length field followed by the `Authorization` and `Date` header data explained in Section 2.3. Afterwards the payload is sent. An example exchange is displayed in Listing 6.

Input Validation The backend is not filtering the payload sent by the app or robot, since the payloads of frontend messages are identical. It is even possible to change data types in the JavaScript Object Notation (JSON), which leads to abnormal behavior in the app. This might represent an attack vector against the app.

Smartphone IP Disclosure The most interesting functionality provided by the app is manual control, where a user directly controls a robot via websocket. To start manual mode, the smartphone sends a message to the *Nucleo* cloud, requesting the IP and port where it should connect to. The robot then sends a response with this information and additionally communicates which Service Set Identifier (SSID) it is connected to. The manual driving mode should only work in the same WiFi and local network, as stated by the manufacturer. By impersonating a robot and sending a forged external IP and port, we were able to make the victim's smartphone connect to our given IP address. Neither the IP range nor the SSID are verified to be local before connecting to the given IP. This poses a severe threat since the victims public IP address is leaked in this attack. With this information, an attacker is able to derive geolocation information or target the victim's smartphone directly.

```
POST /vendors/neato/login HTTP/1.1
Host: nucleo.neatocloud.com
Date: Wed, 06 Mar 2019 16:21:28 GMT
Authorization: NEATOBOT OPS2[red]-508cb1[red]:WqJ7
    +FTo1D19jTlV9bD2gbfjCmS93rBKE[short]
User-Agent: Neato-Robot
```

Listing 5: Neato *Nucleo* robot login (shortened).

```
87
Wed, 06 Mar 2019 16:21:27 GMT|de6a3344c1c[short]
{"reqId": "117","cmd": "getRobotState"}

246
{"version":1,"reqId":"183","result":"ok",
 "error":"ui_alert_linkedapp","state":1,
 [shortened]
 {"modelName":"BotVacConnected",
 "firmware":"2.2.0"}}
```

Listing 6: Neato *Nucleo* command.

```
POST /vendors/neato/robots/OPS234[short]/messages
    HTTP/1.1
Accept: application/vnd.neato.nucleo.v1
Content-type: application/json
Authorization: NEATOAPP aabbccddeeff[short]
Date: Tue, 06 Nov 2018 13:37:00 GMT
Host: nucleo.neatocloud.com:4443

AAAAAAAAAAAAAAAAAAAAAAAA....
```

Listing 7: Shortened request crashing `astro`.

Forwarding the unchecked `Authorization` headers and payloads to the robot has several side-effects:

Robot Blinding Attack First, changing the `secret_key` of a robot as described in Section 4.1.2 results in a Denial of Service (DoS) attack. The robot will still receive heartbeat packets and is convinced it is still connected, but it is not able to validate any packets with payload. Therefore the robot cannot be controlled via the cloud anymore.

Secret Key Extraction Second, in Section 5.1.3 an offline attack against a robot's `secret_key` with manual driving packets is explained. Since the messages sent from *Nucleo* backend to the robots are not additionally encrypted and can be leaked by impersonating the robot, this enables an offline attack as well. The severity of this attack is further increased due to eliminating the constraint of having access to the victim's local network.

Buffer Overflow Third, the buffer overflow explained in 4.2 could as well be prevented by not forwarding the header information to the robots unchecked.

4.2 Buffer Overflow

We discovered a buffer overflow vulnerability in the *Neato BotVac Connected* firmware version 2.2.0 and developed a proof of concept (PoC) exploit, which enables arbitrary unauthenticated remote code execution over the *Nucleo* cloud with root permissions.

4.2.1 Vulnerability

Commands can be sent to robots over the *Nucleo* cloud as described in Section 2.3. If the forwarded request body contents are too long, `astro` crashes locally on the robot and the QNX utility `dumper` automatically creates a core dump. Listing 7 shows a request that triggers the buffer overflow. It is triggered for arbitrary HMACs, meaning it does not require valid authentication. The only required information is the robot's serial number.

We analyze the binary files `webserver`, `astro`, and the core dump of the latter to locate the root cause. The Link Register (LR) is used to save the return address if a function is called in an ARM program [12]. The return address points to the next instruction to execute after the function call returns.

The LR of the `astro` core dump belongs to the function `HMAC_CTX_cleanup`. It is part of the library `OpenSSL`, but the problem occurs earlier in program execution. We examine the call graph of `HMAC_CTX_cleanup` shown in Figure 4 to locate the issue, which indicates that `astro` verifies the HMAC. Note that the function name `VerifySignature` is misleading here; examination of the binary file reveals that this function calculates a HMAC. Since the crash occurs during HMAC verification, the exploit can be triggered unauthenticated.

Listing 8 shows the part of `VerifySignature` that contains the buffer overflow vulnerability. The `NeatoCrypto_UTF8_Encode` function encodes an input string in UTF-8 and stores it in the buffer `utf8_str`. This is done without verification if the input string fits into the target buffer. The input string `str` can contain 7265 B, while the target buffer `utf8_str` is only 2048 B, thus causing the overflow.

The input string is generated by concatenating the robot's serial number, the date header, and the JSON-formatted command to be executed. By sending long and invalid command strings, attackers can overwrite values on the stack. The Program Counter (PC) register, which holds the currently executed instruction on ARM [13], retrieves its content from the stack in function epilogues. The corresponding stack value is overwritten by the payload at the position 2032. Moreover, a malicious payload controls registers `r4–r11`.

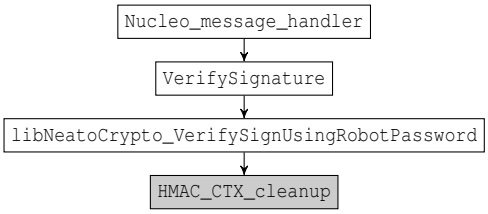

Figure 4: Simplified call graph to the function `HMAC_CTX_cleanup`.

```
VerifySignature(*this, const char *cmd, const char
     *date, const char *robot_serial, const char *
    signature) {
  char *str; //str_to_sign
  char *utf8_str;
  //[...]
  memset(&utf8_str, v11, 0x800);
  memset(&str, 0, 0x1C61u);
  robot_serial_lower = str_lowercase(v9);
  snprintf(&str, 0x1C61u, "%.*s\n%.*s\n%.*s", 32,
    robot_serial_lower, 64, cmd_date, 7168,
    cmd_str);
  len = strlen(&str);
  NeatoCrypto_UTF8_Encode(&str, len, &utf8_str);
  //[...]
}
```

Listing 8: Extract of the source code of the reverse engineered function `VerifySignature`.

4.2.2 Proof of Concept Development

The QNX 6.5 operating system running on *Neato BotVac Connected* robots supports various exploit mitigation techniques. However, neither Address Space Layout Randomization (ASLR) nor Data Execution Prevention (DEP) are enabled by default [21]. Thus, the buffer overflow exploit development is simplified. Nonetheless, request body payloads must satisfy two conditions. First, a string containing null bytes (`0x00`) is rejected by the *Nucleo* cloud. Second, only American Standard Code for Information Interchange (ASCII) characters are accepted (byte values `0x00–0x7f`).

To execute arbitrary commands, our proof of concept aims at executing `system` [4]. Since the `astro` binary runs as root, commands are executed with the same privileges. `astro` is using the shared library `libc.so.3`, which provides the `system` function at address `0x0101b0a8`. However, this address contains non-ASCII bytes. In addition, `system` requires the command to be passed in `r0`, which is not directly controlled by the buffer overflow. Therefore, we construct a Return Oriented Programming (ROP) chain containing five gadgets, which are located at valid addresses. An `ADD` gadget solves the non-ASCII issue by adding up ASCII values to non-ASCII ones, which ultimately leads to root Remote Code Execution (RCE).

5 Discussion

In this section the most important findings and their security implications and impact are discussed. We continue using the same structure as in the actual findings and first discuss local vulnerabilities in Section 5.1, remote vulnerabilities in Section 5.2, and then discuss remaining issues in Section 5.3.

5.1 Local Vulnerabilities

In this section security implications related to findings in Section 3 are discussed.

5.1.1 Secure Boot Bypass

Fixing the bootloader has a high potential in bricking robots. If anything fails during the bootloader update, there is probably no recovery for customers. However, a hard-coded password and a memory reset would be sufficient to stop an attack at this stage in the first place. Neato's solution to this issue rolled out in version 4.4.0-72. It fixes this issue by detecting if the robot is in factory mode and skips the boot menu if not.

5.1.2 Static Firmware Analysis

On evaluating the binaries we found that they are not obfuscated at all—they even contain symbols which made analysis fast and efficient. The efforts required to fully understand and exploit an ecosystem could be significantly increased by obfuscation, while the costs for doing so are fairly low.

5.1.3 Secret Key Generation

Secret keys are computed based on robot MAC addresses, a timestamp, and an unseeded random element. In this section we discuss the security of serial numbers and the resulting implications on the secret key.

Serial Number Security A lot of the security and authentication depends on a robot's serial number—which turns out to be fairly easy to find out. First, serial numbers are printed on the robot's packaging and the robot itself. The former enables attackers to go into shops and note their numbers or checking someone's trash. The latter is often photographed when selling used robots. Second, serial numbers can be enumerated as shown in Section 4.1.2. The only information they contain is the MAC address and manufacturing date, with the former being observable in—for instance—the network and the latter being within a limited time range. Taking this into consideration, a robot's serial number is not a secret and security solely relies on `secret_key` security.

Secret Key Security Considering the findings in Section 3.3, where we presented the `NeatoCrypto_GenerateRobotPassword`, several derivations are made. If an attacker knows in which year a robot was first connected to the cloud, the timestamp contributes $log_2(365 \times 24 \times 60 \times 60) = 25$ bit of entropy. In contrast, if he is able to tell the timestamp with an accuracy of $\pm 1/2$ h, the number of unknown bits is $log_2(60 \times 60) = 12$. Table 2 gives an overview of different timespans and the resulting entropy and average number of tries needed for bruteforcing a `secret_key`, depending on a known respectively unknown RNG state.

As visible in Table 2, an attacker is able to test all possible `secret_keys` for a given `robot_serial` against the cloud, if the RNG state is known and the time of first connection can be determined with a 30 min accuracy. If the time is not known to a sufficient accuracy, it is still possible to execute an online attack distributed over a longer period of time.

For an unknown RNG state, testing against the Nucleo cloud would not work, since the server will start to blacklist all IP addresses generating massive amounts of faulty authenticated messages. If the attacker is able to record a manual driving session on the local network of the victim, an offline attack on the unencrypted authentication header is possible [19]. The offline attack has to break the hash of the authentication header, which contains the `secret_key`, `robot_serial`, `date`, and message body (compare Section 2.3). Since all information except for the `secret_key` is known, this represents a valid attack. With the `secret_key` extraction described in Section 4.1.3 an offline attack is possible on arbitrary robots with known `robot_serial`.

RNG state	Timespan	Entropy	Avg. # tries	Bruteforce
known	±30 min	12 bit	2048	0.000,15 s
	±1/2 year	25 bit	16,777,216	1.1 s
unknown	±30 min	26 bit	33,554,432	2.2 s
	±1/2 year	39 bit	2.75×10^{11}	5 h

Table 2: Complexity computations for `secret_key`.

Gosney has shown that a setup with 8x Nvidia GTX 1080 Ti achieves 15,187.1 kHashes/s for PBKDF2-HMAC-SHA256 [8], which is used for the hash of the authentication header, hence average attack duration for given timestamp accuracies can be computed and are presented in Table 2. The time calculations show that the `secret_key` of a robot can be easily brute-forced as soon as an `Authorization` header with all related data for the offline attack is given.

This new attack allows to remotely control a robot via the *Nucleo* API, since knowledge of the `secret_key` and the `robot_serial` is sufficient to forge a valid authentication header. This is an upgrade to an attack presented in [19], since for the new attack the robot is not paired with a new account. Instead the old account connection stays intact and the attack is thereby very hard to detect.

Since this is found in the `NeatoCrypto` library, this attack is valid for the most recent Neato and Vorwerk vacuum robots as well. With the `secret_key` being the root of trust in their cloud environment, this has further security implications.

5.2 Remote Vulnerabilities

In this section the security implications of the findings presented in Section 4 are discussed.

5.2.1 Robot Backend Vulnerabilities

We revealed several problems in the backend and the common root of problem is the usage of an identical RSA private key on all devices. Having different devices using the same private key defeats the purpose of public key cryptography—a symmetric cryptography scheme would be sufficient. Since RSA consumes a lot of computation time, this also allows to send lots of random `Authorization` headers to the backend, which results in an amplified Denial of Service (DoS) attack.

With an attacker being able to send and receive arbitrary data from both the app and the robot cloud interface, the server could be exploited as a botnet command and control server. An attacker could register multiple fake robot vacuums with arbitrary `robot_serial`s and use them as an interfacing point for control data. Since knowledge of a `robot_serial` is enough to communicate with a fake robot, the app interface would be used by the botnet's bots in this scenario. An attacker could also register high amounts of fake robots sending regular traffic to execute a stealthy Distributed Denial of Service (DDoS) attack. This would cause a high message load on the backend that is almost impossible to be filtered.

5.2.2 Remote Command Execution

An attacker could use the command execution on the robot to extract sensitive information from the robot like maps and other sensor readings. This violates the privacy of the victim. An attacker gains access to the victim's local network via the vulnerability. The access can be used to attack further devices inside the local network which would be protected by the firewall of the network otherwise. The attacker can also use the exploit to execute a DoS attack against the robot if he executes a command which bricks the robot. Last but not least the exploit could be used to build a botnet consisting of Neato robots, which can be utilized for DDoS attacks.

5.3 General

We finalize the discussion with general security-related observations.

Update Process The update process for the older connected series is not user-friendly. Updating the *Neato BotVac Connected* requires users to download firmware to a correctly formatted USB stick which has to be then plugged in with a special cable. Finally, the update process has to be invoked manually. We assume most customers will not update their robots, unless they are provided with a over-the-air (OTA) update functionality, as it is featured by the most recent models. Customers could be forced to do updates by locking their robots, however this might cause angry customer feedback.

Responsible Disclosure Establishing contact with Neato turned out to be more of an obstacle than expected. After first contacting them about our findings on March 12, 2018, customer support repeatedly claimed to have forwarded our messages to development, which was later disputed. Even sending a PoC Python script for the cloud-based unauthenticated command execution was ignored. In fact, it was not until November 27, 2018 when we were finally approached by Neato core development. Afterwards, the issues were fixed in a timely fashion and Neato even provided us with the latest *Neato BotVac Connected D7* for further testing.

6 Conclusion

In this paper we presented multiple attacks on the Neato and Vorwerk ecosystems, which ultimately lead to various possible security breaches. Amongst them are those to start and pause the robot, extract a map of the victim's apartment, read arbitrary sensor data, leak customer public IP addresses, and even gain access to the local network of a customer.

On a first glance, Neato did a very good job in securing their connected vacuum cleaners: secure boot and mutual authentication between robots and the cloud exceeds what is done for most IoT products. Their robots are in the upper price range and the customer receives premium security features in return. Dissecting their robots was not only challenging, but also now provides interesting insights in building secure IoT applications and other large-scale connected scenarios.

However, Neato followed a hard shell, soft interior approach. Once an attacker is inside the system with access to all binaries, many security best practices were missing: no privilege separation, no operating system security features, no binary obfuscation, and hand-written cryptography. Especially the cryptography part looks well-done from the outside and represents solid security without knowledge of the exact parameters used in the computation. Their implementation illustrates how scientifically proven strong theoretical security primitives still fail in practice. The mistakes were present over multiple years without being noticed by the developers, thus indicating a need for better books and standards to be guided.

Our analysis shows that cryptography on current devices is flawed and does not provide sufficient security. Neato will need to strictly filter invalid requests in the cloud to prevent attacks, which is not possible for targeted attacks on individual robots. To maintain security in a targeted attack scenario, users need to protect their robot's `serial_id` by any means: they should buy robots exclusively from the vendor, not throw away the original packaging (or shred it), use a separate WiFi, and prevent physical access to the robot by any third parties.

From a customer perspective, more research on which data is shared depending on the vacuum cleaner model would be interesting. Collected and accessible data contains a lot of information and meta data about the customer's everyday life. This includes the location of residence, apartment size and furniture, number of inhabitants including pets, unencrypted traffic exchange on the robot's WiFi, daily scheduling, and more. This information is sensible enough to be worth all the protection mechanisms discussed in this paper.

Finally, we want to encourage the use of the knowledge gathered in this paper by the existing Neato and Vorwerk communities to build smartphone apps and gateways with additional features and capabilities.

Acknowledgments

We thank Neato for taking our responsible disclosure serious, patching their systems as fast as possible, and providing us with more testing hardware after the first report. Moreover, we thank Max Maass for co-supervising Fabian's Master thesis.

This work has been supported by the DFG as part of project C.1 within the RTG 2050 "Privacy and Trust for Mobile Users" as well as the German Federal Ministry of Education and Research (BMBF) and the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) under the joint grant for the National Research Center for Applied Cybersecurity.

References

[1] Amazon Web Services, Inc. Amazon Web Services. https://aws.amazon.com/. Accessed 2019-02-25.

[2] Maggie Astor. Your Roomba May Be Mapping Your Home, Collecting Data That Could Be Shared. *NY Times*, July 2017.

[3] Blackberry QNX. QNX Operating Systems. https://blackberry.qnx.com/en. Accessed 2019-03-26.

[4] Blackberry QNX. system() - Execute a system command. http://www.qnx.com/developers/docs/6.5.0SP1.update/com.qnx.doc.neutrino_lib_ref/s/system.html.

[5] P. Jones D. Eastlake. US Secure Hash Algorithm 1 (SHA1). https://tools.ietf.org/html/rfc3174. Accessed 2019-03-04.

[6] T. Hansen D. Eastlake. US Secure Hash Algorithms (SHA and HMAC-SHA). https://tools.ietf.org/html/rfc4634. Accessed 2018-05-20.

[7] Dennis Giese and Daniel Wegemer. Unleash your Smart-home Devices: Vacuum Cleaning Robot Hacking. https://media.ccc.de/v/34c3-9147-unleash_your_smart-home_devices_vacuum_cleaning_robot_hacking, December 2017.

[8] Jeremi M Gosney. 8x Nvidia GTX 1080 Ti Hashcat Benchmarks. https://gist.github.com/epixoip/ace60d09981be09544fdd35005051505. Accessed 2018-05-20.

[9] R. Canetti H. Krawczyk, M. Bellare. HMAC: Keyed-Hashing for Message Authentication. https://www.ietf.org/rfc/rfc2104.txt, 1997.

[10] Jason Kielpinski. Security in a Vacuum: Hacking the Neato Botvac Connected, Part One. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2018/march/security-in-a-vacuum-hacking-the-neato-botvac-connected-part-1/, March 2018.

[11] Jason Kielpinski. Security in a Vacuum: Hacking the Neato Botvac Connected, Part Two. https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2018/april/security-in-a-vacuum-hacking-the-neato-botvac-connected-part-2/, April 2018.

[12] ARM Ltd and ARM Germany GmbH. Assembler User Guide: General-purpose registers. http://www.keil.com/support/man/docs/armasm/armasm_dom1359731135351.htm.

[13] ARM Ltd and ARM Germany GmbH. Assembler User Guide: Program Counter. http://www.keil.com/support/man/docs/armasm/armasm_dom1359731138020.htm.

[14] Amrit Mundra and Hong Guan. Secure Boot on Embedded Sitara Processors. http://www.ti.com/lit/wp/spry305a/spry305a.pdf, September 2018.

[15] Neato. Programmer's Manual v3.1. https://github.com/jeroenterheerdt/neato-serial/blob/master/XV-ProgrammersManual-3_1.pdf, 2015.

[16] Neato Robotics. Botvac Connected. https://www.neatorobotics.com/robot-vacuum/botvac-connected-series/botvac-connected/. Accessed 2018-05-20.

[17] Neato Robotics. Neato Developer Network. https://developers.neatorobotics.com/. Accessed 2018-05-20.

[18] Salesforce.com. Heroku Cloud Application Platform. https://www.heroku.com/. Accessed 2019-02-25.

[19] Fabian Ullrich. Analysing and Evaluating Interface, Communication, and Web Security in Productive IoT Ecosystems. Master thesis, TU Darmstadt, May 2018. Supervised by Jiska Classen and Max Maass.

[20] Vorwerk. Annual Report 2017. https://annual-reports.vorwerk.com/fileadmin/pdf/Vorwerk-Annual-Report-2017.pdf, 2018.

[21] J. Wetzels and A. Abbasi. Dissecting QNX. https://recon.cx/2018/brussels/resources/slides/RECON-BRX-2018-Dissecting-QNX.pdf, 2018.