SSRF vulnerabilities caused by SNI proxy misconfigurations

SSRF vulnerabilities caused by SNI proxy misconfigurations - Author not stated, invicti.com.

- Published: date not stated
- Original: <https://www.invicti.com/blog/web-security/ssrf-vulnerabilities-caused-by-sni-proxy-misconfigurations/>
- Current location: <https://www.invicti.com/blog/web-security/ssrf-vulnerabilities-caused-by-sni-proxy-misconfigurations>
- Preserved from: https://www.invicti.com/blog/web-security/ssrf-vulnerabilities-caused-by-sni-proxy-misconfigurations (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

SSRF vulnerabilities caused by SNI proxy misconfigurations

Table of Contents

A typical task in complex web applications is routing requests to different backend servers to perform load balancing. Most often, a reverse proxy is used for this. Such reverse proxies work at the application level (over HTTP), and requests are routed based on the value of the `Host` header (`:authority` for HTTP/2) or parts of the path.

One typical misconfiguration is when the reverse proxy directly uses this information as the backend address. This can lead to server-side request forgery (SSRF) vulnerabilities that allow attackers to access servers behind the reverse proxy and, for example, steal information from AWS metadata. I decided to investigate similar attacks on proxy setups operating at other levels/protocols – in particular, SNI proxies.

What is TLS SNI?

Server Name Indication (SNI) is an extension of the TLS protocol that provides the foundation of HTTPS. When a browser wants to establish a secure connection to a server, it initiates a TLS handshake by sending a `ClientHello` message. This message may contain an SNI extension field that includes the server domain name. In its `ServerHello` message, the server can then return a

certificate appropriate for the specified server name. The typical use case for this is when there are multiple virtual hosts behind one IP address.

What is an SNI proxy?

When a reverse proxy (more correctly, a load balancer) uses a value from the SNI field to select a specific backend server, we have an SNI proxy. With the widespread use of TLS and HTTPS in particular, this approach is becoming more popular. (Note that another meaning of SNI proxy refers to the use of such proxies to bypass censorship in some countries.)

There are two main options for running an SNI proxy: with or without SSL termination. In both cases, the SNI proxy uses the SNI field value to select an appropriate backend. When running with SSL termination, the TLS connection is established with the SNI proxy, and then the proxy forwards the decrypted traffic to the backend. In the second case, the SNI proxy forwards the entire data stream, really working more like a TCP proxy.

A typical SNI proxy configuration

Many reverse proxies/load balancers support SNI proxy configurations, including Nginx, Haproxy, Envoy, ATS, and others. It seems you can even use an SNI proxy in Kubernetes.

To give an example for Nginx, the simplest configuration would look as follows (note that this requires the Nginx modules `ngx_stream_core_module` and `ngx_stream_ssl_preread_module` to work):

```
stream {
    map $ssl_preread_server_name $targetBackend {
        test1.example.com backend1:443;
        test2.example.com backend2:9999;
    }
    server {
        listen 443;
        resolver 127.0.0.11;
        proxy_pass $targetBackend:443;
        ssl_preread on;
    }
}
```

Here, we configure a server (TCP proxy) called `stream` and enable SNI access using `ssl_preread on`. Depending on the SNI field value (in `$ssl_preread_server_name`), Nginx will route the whole TLS connection either to `backend1` or `backend2`.

SNI proxy misconfigurations leading to SSRF

The simplest misconfiguration that would allow you to connect to an arbitrary backend would look something like this:

```
stream {
    server {
        listen 443;
        resolver 127.0.0.11;
        proxy_pass $ssl_preread_server_name:443;
        ssl_preread on;
    }
}
```

Here, the SNI field value is used directly as the address of the backend.

With this insecure configuration, we can exploit the SSRF vulnerability simply by specifying the desired IP or domain name in the SNI field. For example, the following command would force Nginx to connect to *internal.host.com*:

```
openssl s_client -connect target.com:443 -servername "internal.host.com" -crlf
```

In general, according to RFC 6066, IP addresses should *not* be used in SNI values, but in practice, we can still use them. What's more, we can even send arbitrary symbols in this field, including null bytes, which can be useful for exploitation. As you can see below, the server name can be changed to an arbitrary string. Though for this specific Nginx configuration, unfortunately, I did not find a way to change the backend port:

Another class of vulnerable configurations is similar to typical HTTP reverse proxy misconfigurations and involves mistakes in the regular expression (regex). In this example, traffic is forwarded to the backend if the name provided via SNI matches the regex:

```
stream {
    map $ssl_preread_server_name $targetBackend {
        ~^www.example\.com $ssl_preread_server_name;
    }
    server {
        listen 443;
        resolver 127.0.0.11;
        proxy_pass $targetBackend:443;
        ssl_preread on;
    }
}
```

This regex is incorrect because the first period character in `www.example.com` is not escaped, and the expression is missing the `$` terminator at the end. The resulting regex matches not only

www.example.com but also URLs like *www.example.com.attacker.com *or *wwwAexample.com*. As a result, we can perform SSRF and connect to an arbitrary backend. While we can't use the IP address directly here, we can bypass this restriction simply by telling our DNS server that *www.example.com.attacker.com* should resolve to 127.0.0.1.

Potential directions for SNI proxy research and abuse

In a 2016 article about scanning IPv4 for open SNI proxies, researchers managed to find about 2500 servers with a fairly basic testing approach. While this number may seem low, SNI proxy configurations have become more popular since 2016 and are widely supported, as evidenced even by a quick search of GitHub.

As a direction for further research, I can suggest a couple of things to think about for configurations without TLS termination. An SNI proxy checks only the first `ClientHello` message and then proxies all the subsequent traffic, even if it's not correct TLS messages. Also, while the RFC specifies that you can only have one SNI field, in practice, we can send multiple different names (TLS-Attacker is a handy tool here). Because Nginx only checks the first value, there could (theoretically) be an avenue to gain some additional access if a backend accepts such a `ClientHello` message but then uses the second SNI value.

Avoiding SNI proxy vulnerabilities

Whenever you configure a reverse proxy, you should be aware that any misconfigurations may potentially lead to SSRF vulnerabilities that expose backend systems to attack. The same applies to SNI proxies, especially as they are gaining popularity in large-scale production systems. In general, to avoid vulnerabilities when configuring a reverse proxy, you should understand what data could be controlled by an attacker and avoid using it directly in an insecure way.