

CS 253: Web Security

DNS Rebinding

Admin

- Assignment 4 is due Friday, December 3 at 11:59pm
- Sample final exams are on the course website
- Leave course feedback

Business ▶ **The Channel**

Demo shows how web attack threatens fabric of the universe

All hail the power of DNS rebinding

By [Dan Goodin](#) 9 Apr 2008 at 03:16

16 

SHARE ▼



RSA Showing how the web's underpinnings can be abused to attack assets presumed to be secure, a researcher unveiled a website that can log into a home router and change key settings, such as administrator passwords and servers used to access trusted web destinations.

Rather than creating a trojan or other piece of specialized malware to access servers or other devices behind a firewall, researcher Dan Kaminsky, a director of penetration testing firm IOActive, showed how a



Get Unlimited Wired Access

[SUBSCRIBE](#)

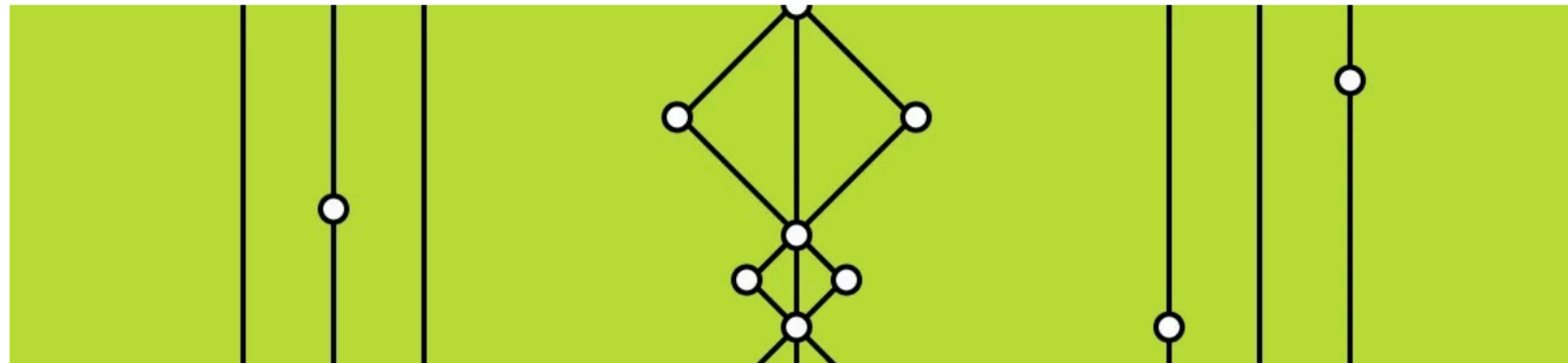
LILY HAY NEWMAN

SECURITY

06.19.2018 04:51 PM

Millions of Streaming Devices Are Vulnerable to a Retro Web Attack

Using a technique called DNS rebinding, one amateur hacker found vulnerabilities in devices from Google, Roku, Sonos, and more.



DNS rebinding attacks

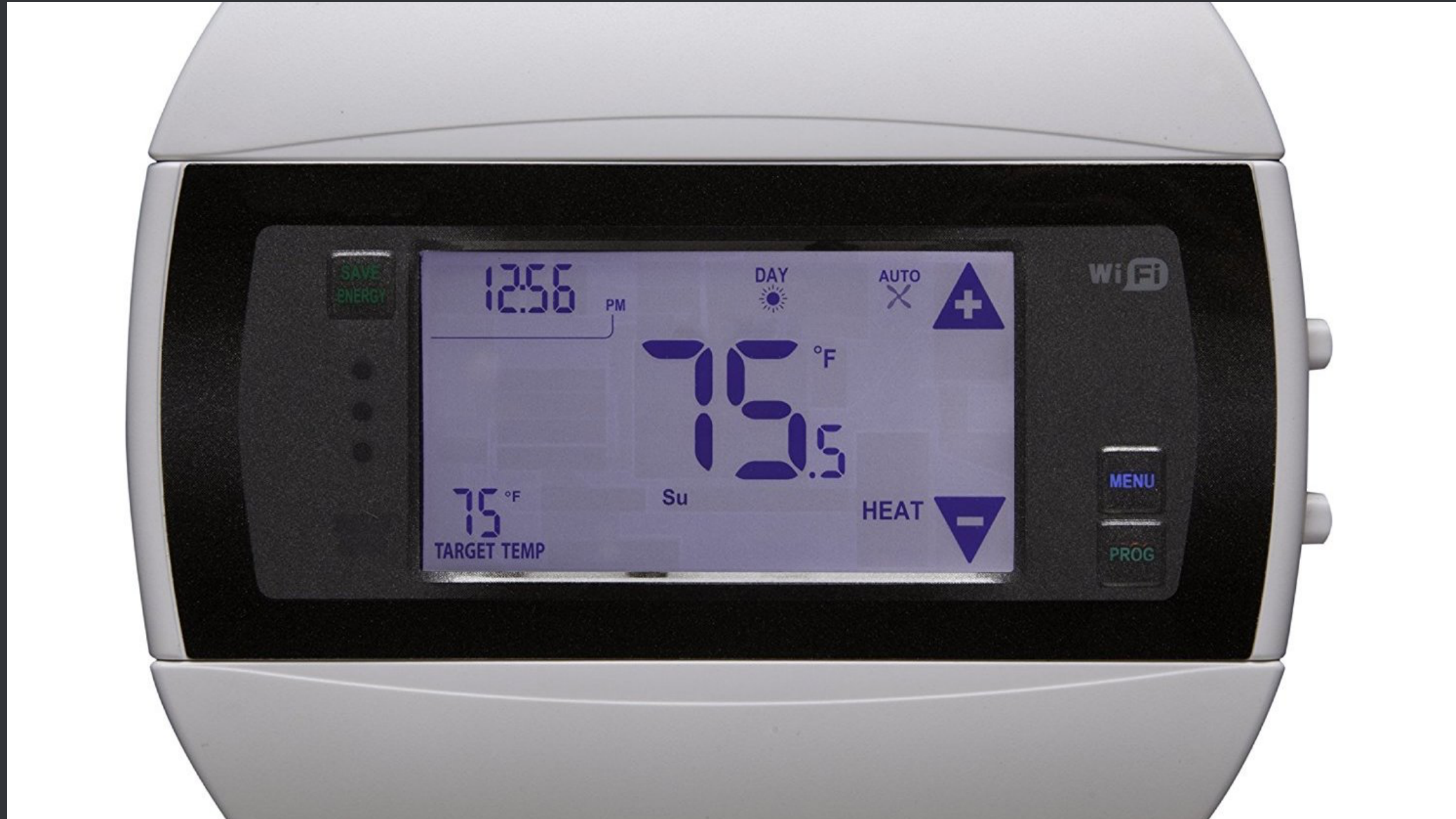
- Pretty much every IoT device is/was vulnerable
- Pretty much every local server is/was vulnerable
- For a long time, folks did not take DNS rebinding attacks seriously
- Very effective attack, and yet very little awareness about it in industry

DNS rebinding attacks

- 165 million printers, or 66 percent, are vulnerable to DNS rebinding attacks. The company named Hewlett Packard, Epson, Konica, Lexmark, and Xerox as examples of representative manufacturers shipping vulnerable printers.
- 160 million, or 75 percent, of IP cameras by manufacturers such as Axis Communications, GoPro, Sony, and Vivotek are vulnerable.
- 124 million, or 77 percent, of IP phones are vulnerable; manufacturers include Avaya, Cisco, Dell, NEC, and Polycom.
- 28 million, or 57 percent, of smart TVs – Roku-integrated, Samsung, and Vizio – are vulnerable.
- 14 million, or 87 percent, of switches, routers and access points are vulnerable; manufacturers include Cisco, Netgear, Extreme, Aruba, and Avaya.
- 5.1 million, or 78 percent, of streaming media players and smart speakers by Apple, Google, Roku, and Sonos are vulnerable.²

²<https://www.armis.com/resources/iot-security-blog/dns-rebinding-exposes-half-a-billion-iot-devices-in-the-enterprise/>

Radio Thermostat CT50



DNS rebinding attacks

The Radio Thermostat CT50 & CT80 devices have by far the most consequential IoT device vulnerabilities I've found so far. These devices are some of the cheapest "smart" thermostats available on the market today. I purchased one to play with after being tipped off to their lack of security by CVE-2013-4860, which reported that the device had no form of authentication and could be controlled by anyone on the network. [...]

That assumption turned out to be correct and the thermostat's control API left the door wide open for DNS rebinding shenanigans. It's probably pretty obvious the kind of damage that can be done if your building's thermostat can be controlled by remote attackers. The PoC at <http://rebind.network> exfiltrates some basic information from the thermostat before setting the target temperature to 95° F. That temperature can be dangerous, or even deadly in the summer months to an elderly or disabled occupant. Not to mention that if your device is targeted while you're on vacation you could return home to a whopper of a utility bill.¹

¹ <https://medium.com/@brannondorsey/attacking-private-networks-from-the-internet-with-dns-rebinding-ea7098a2d325>

Recall: User joins a zoom call (vulnerable)

Local Server
localhost:19421

Client

Server
zoom.us

Local Server
localhost:19421

Client

Server
zoom.us

Local Server
localhost:19421

Client

Server
zoom.us

GET /j/123 HTTP/1.1



Local Server
localhost:19421

Client

Server
zoom.us

GET /j/123 HTTP/1.1

HTTP/1.1 200 OK
<!doctype html>

Local Server
localhost:19421

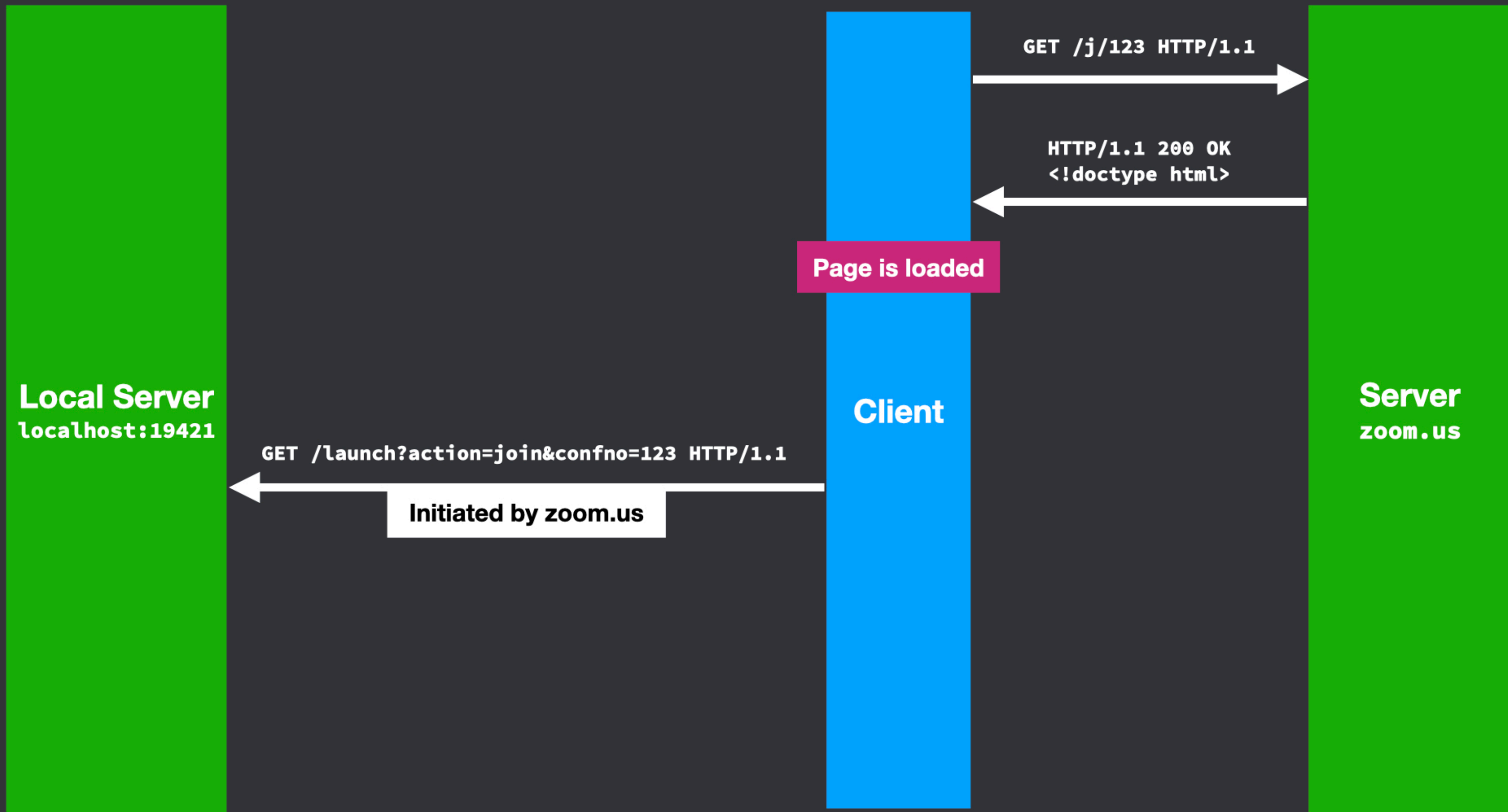
Client

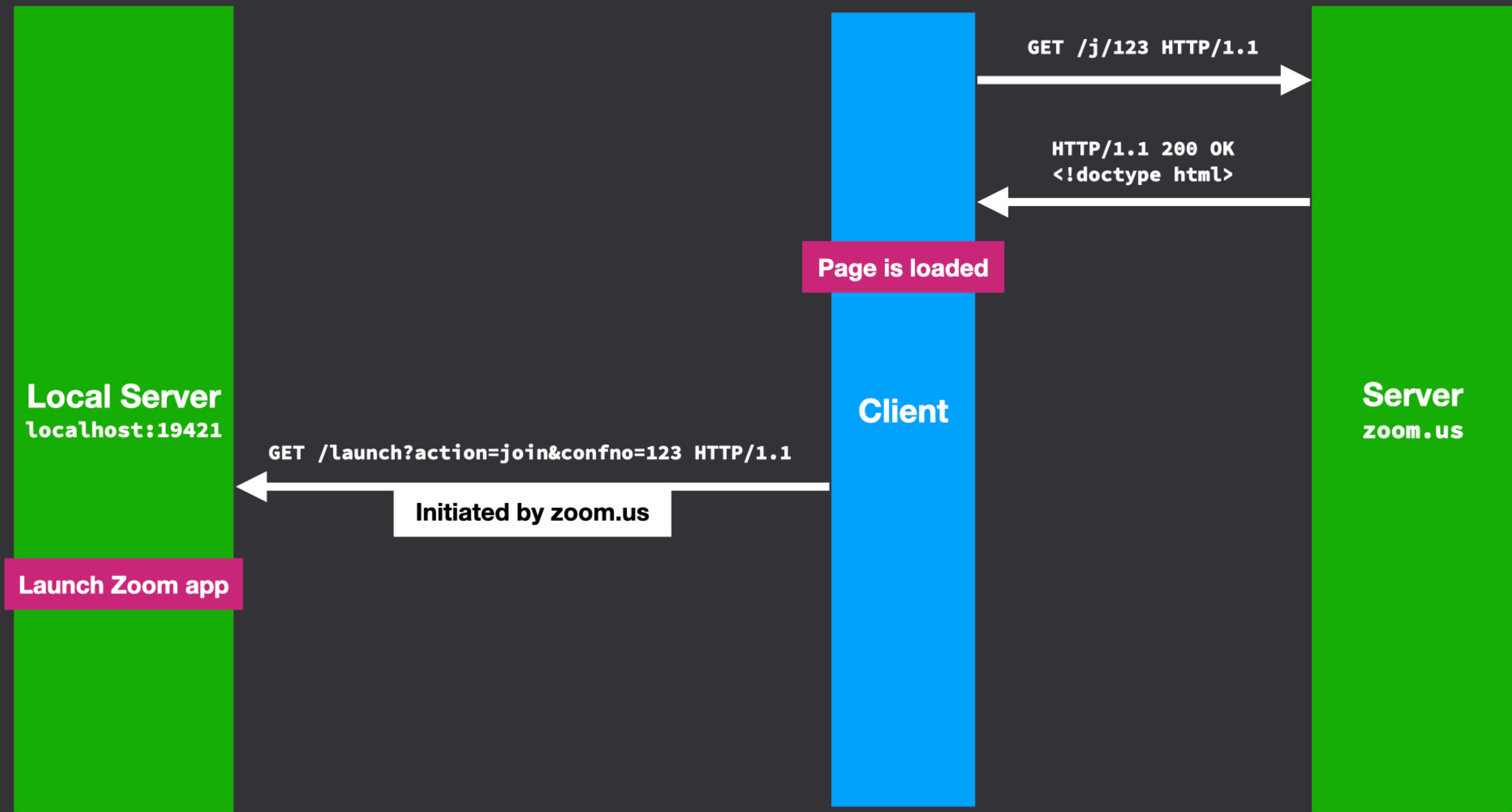
Server
zoom.us

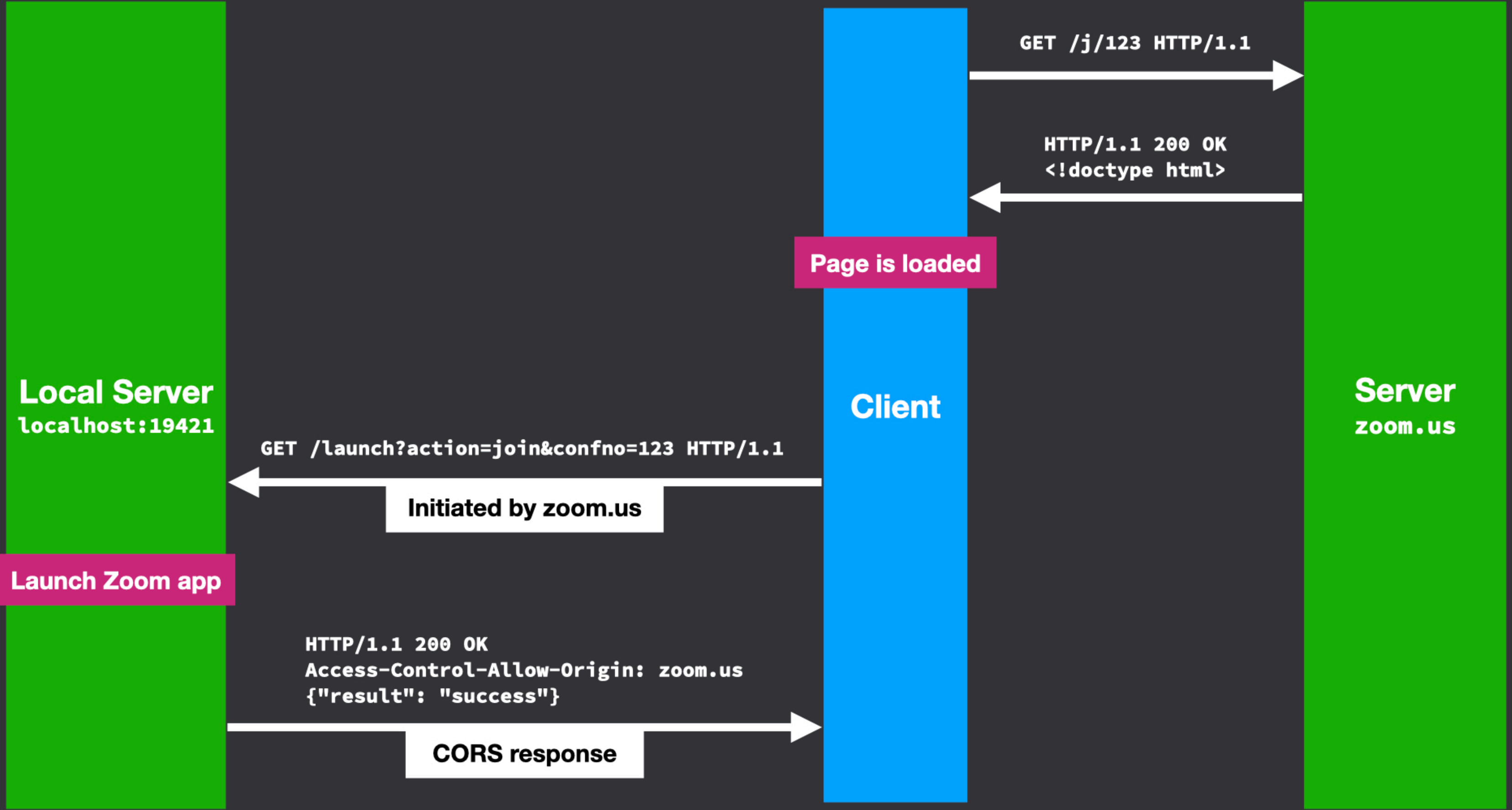
GET /j/123 HTTP/1.1

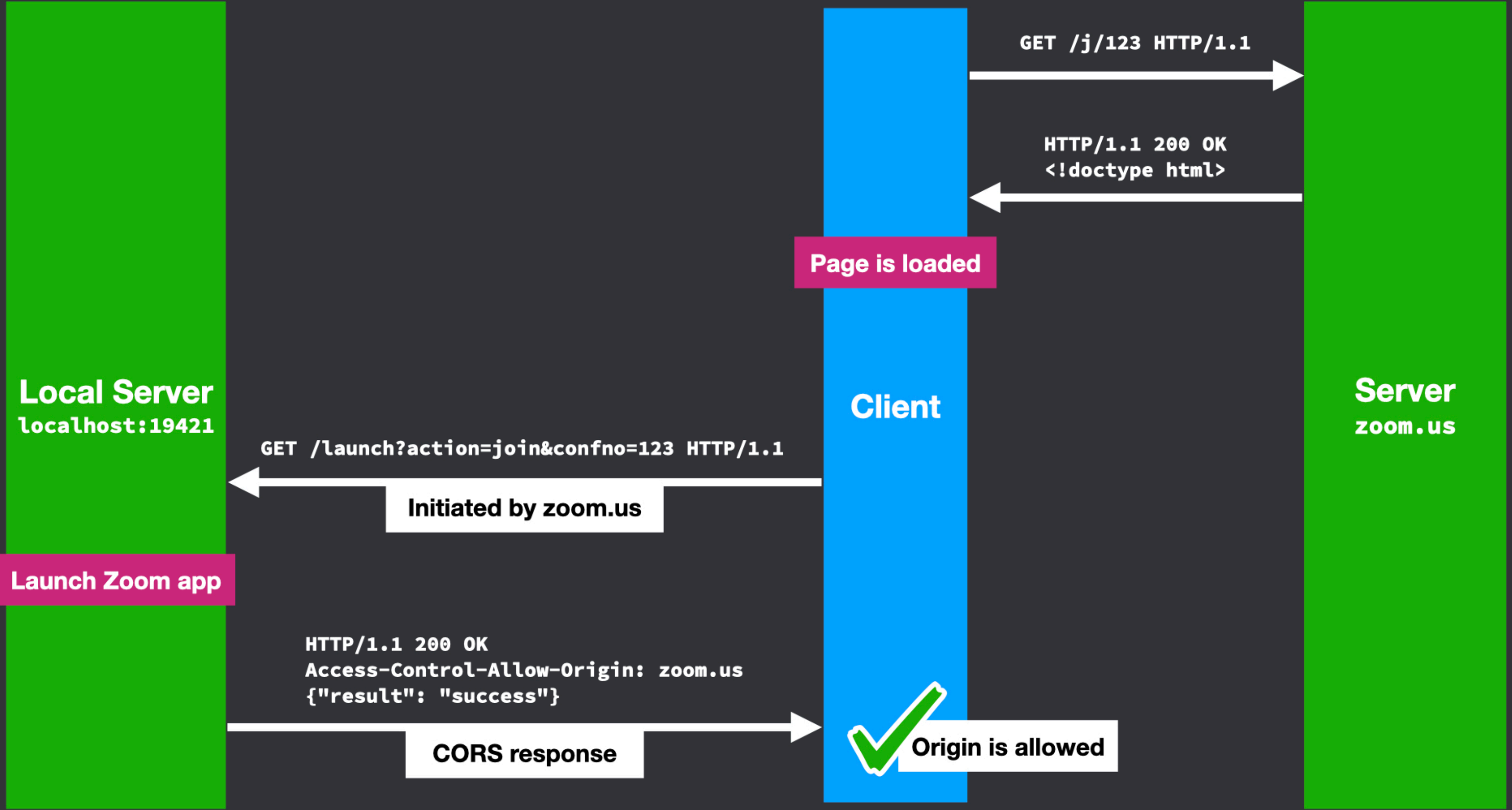
HTTP/1.1 200 OK
<!doctype html>

Page is loaded









Attacker joins user into a zoom call

Local Server
localhost:19421

Client

Server
attacker.com

Local Server
localhost:19421

Client

Server
attacker.com

Local Server
localhost:19421

Client

GET / HTTP/1.1

Server
attacker.com



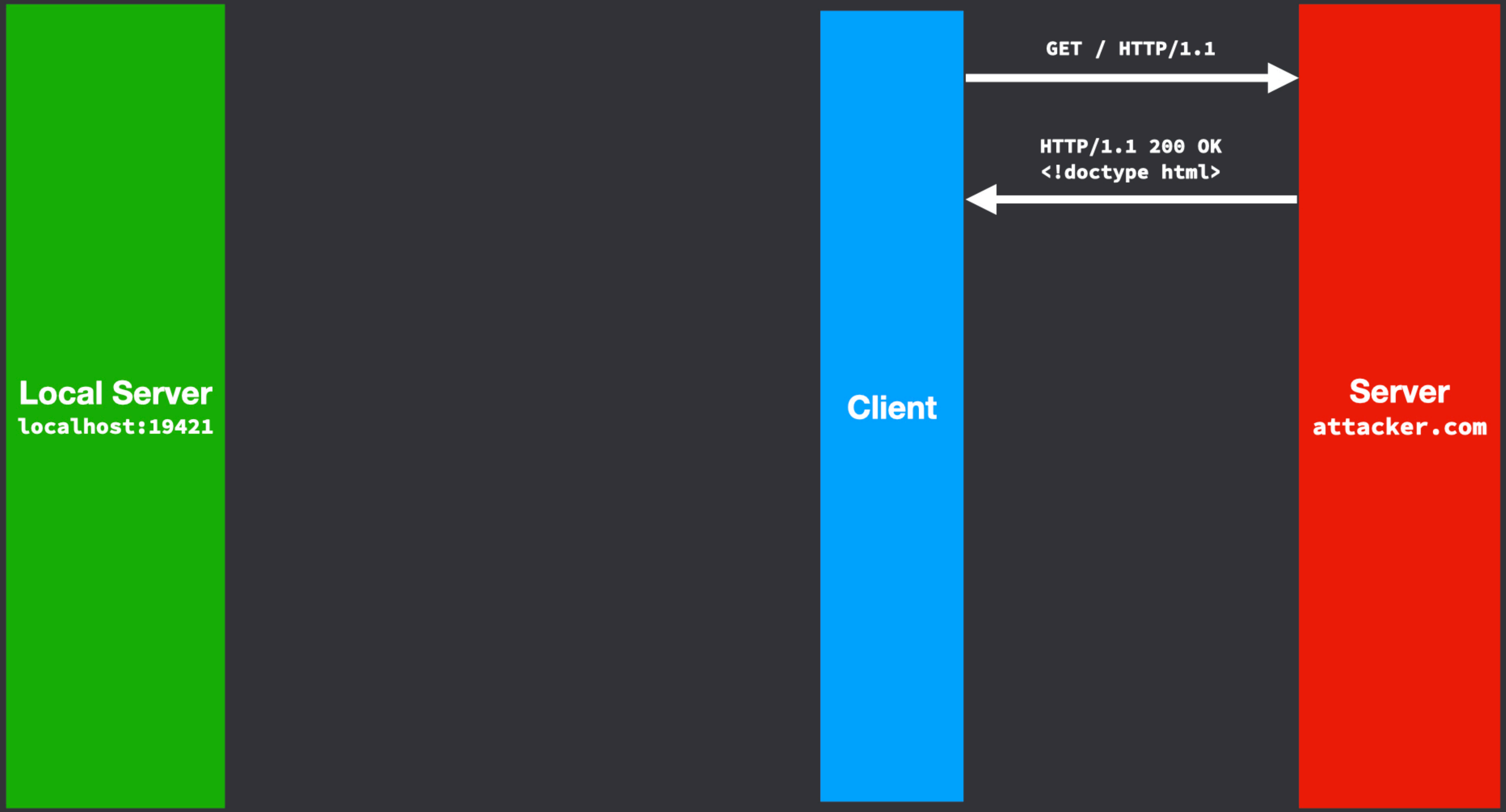
Local Server
localhost:19421

Client

Server
attacker.com

GET / HTTP/1.1

HTTP/1.1 200 OK
<!doctype html>



Local Server
localhost:19421

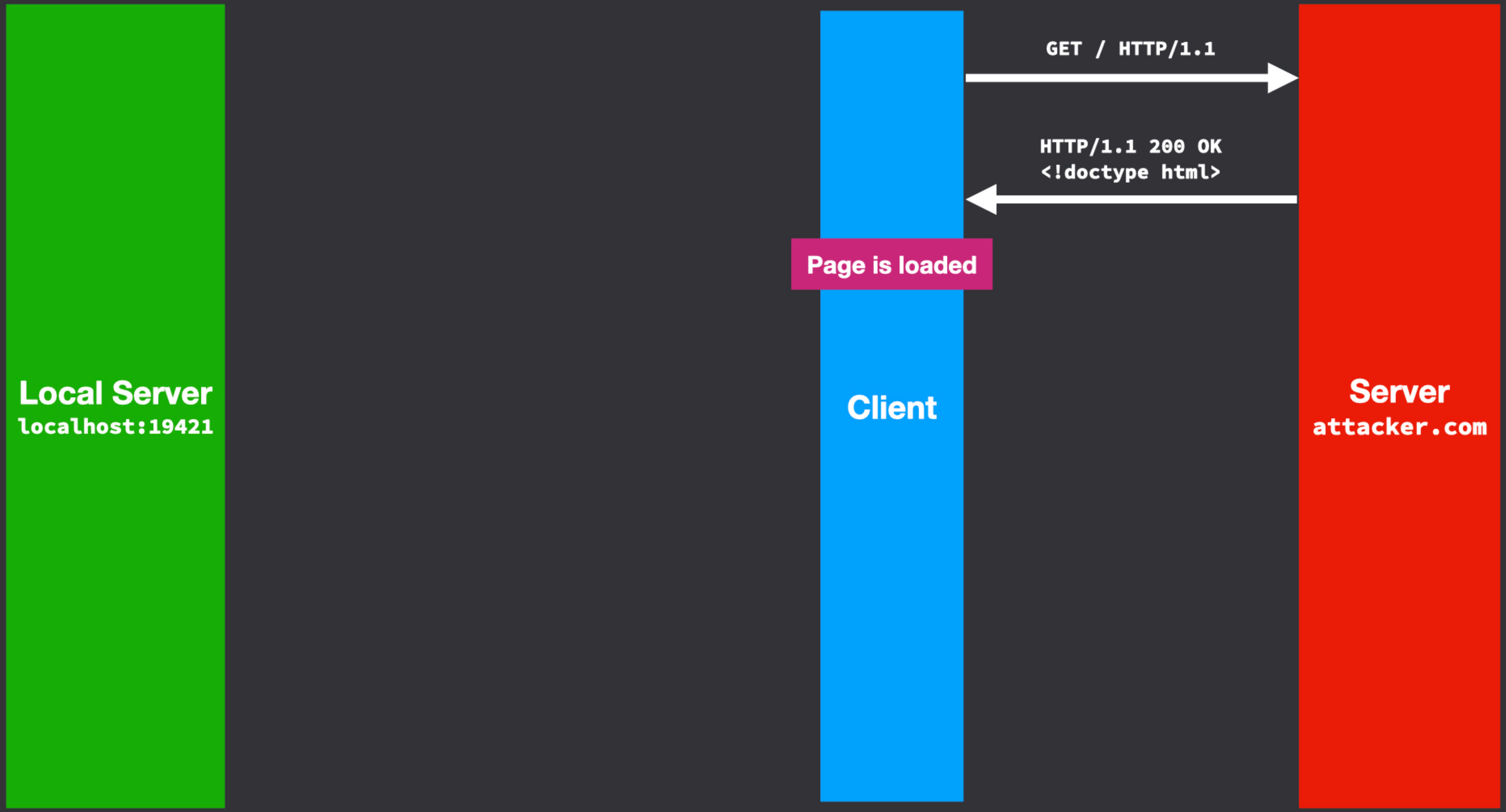
Client

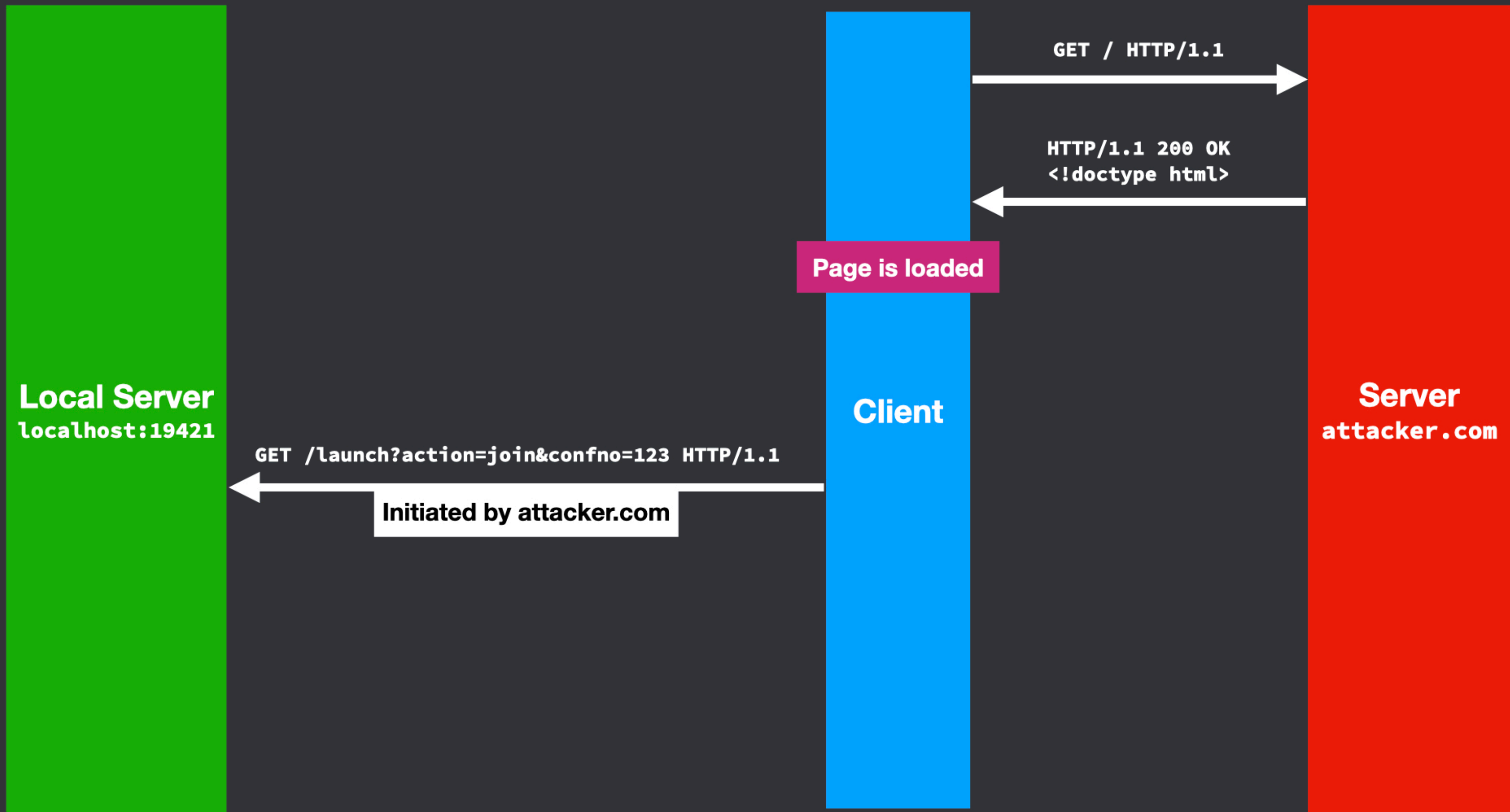
Server
attacker.com

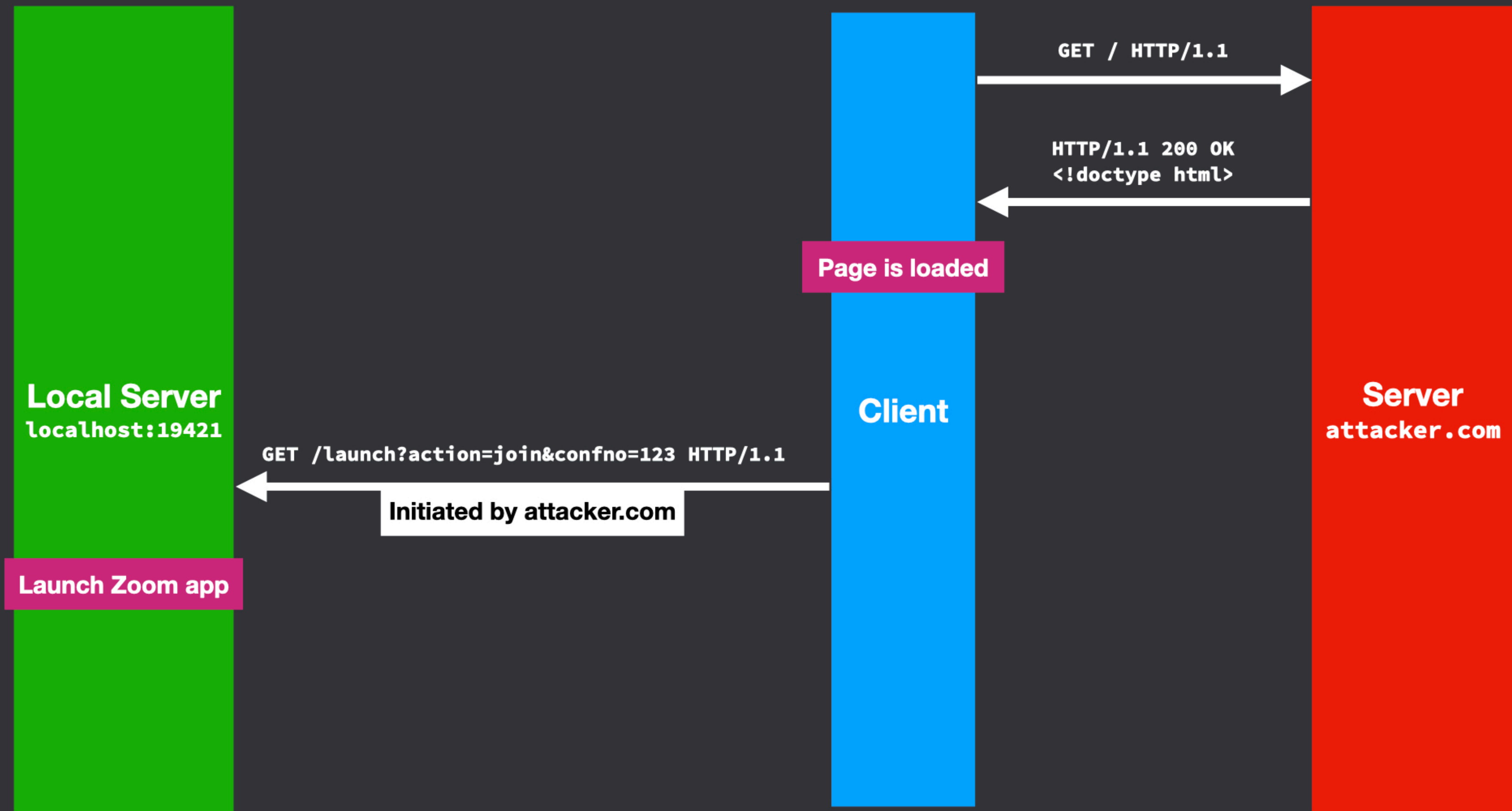
GET / HTTP/1.1

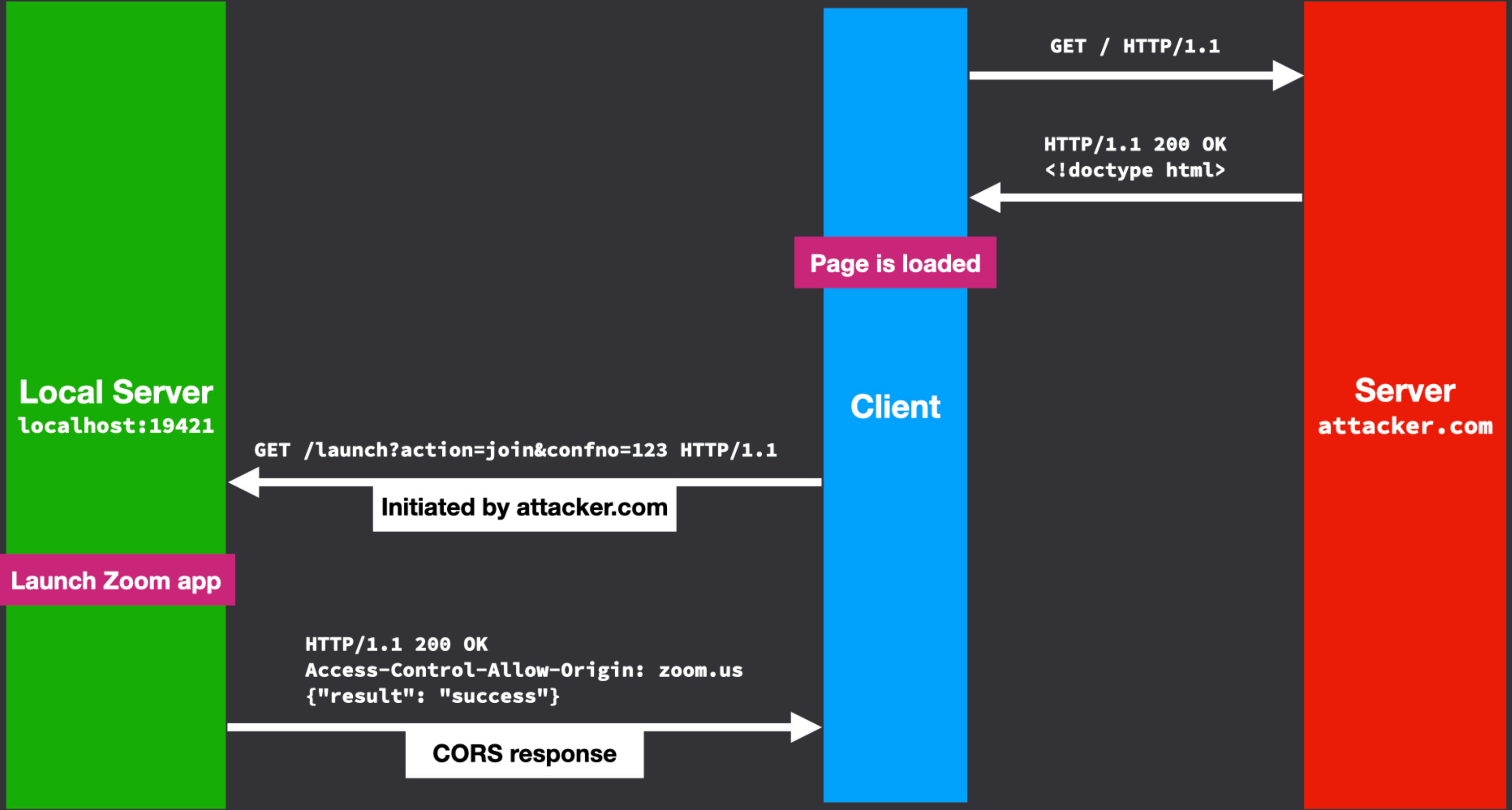
HTTP/1.1 200 OK
<!doctype html>

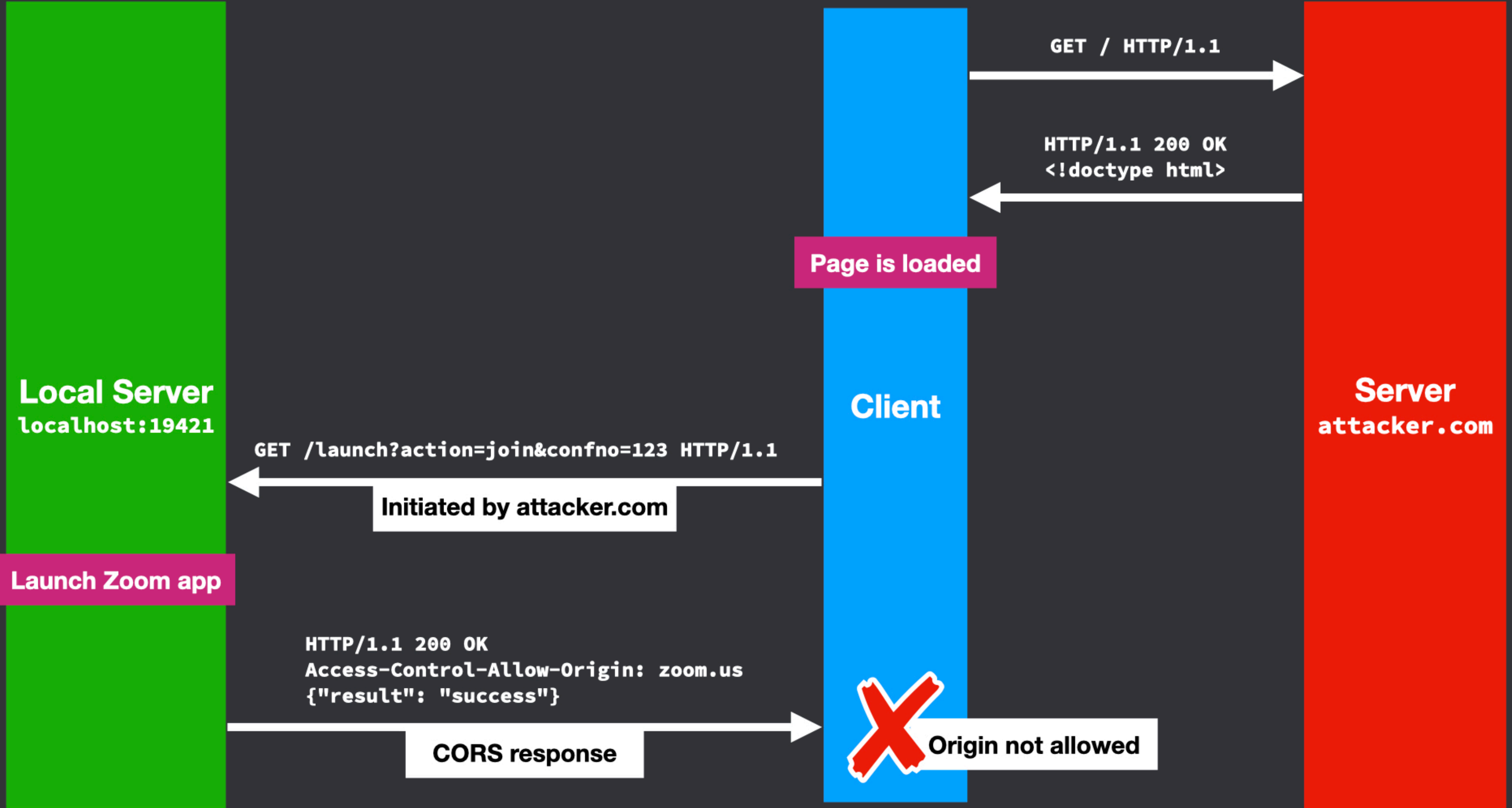
Page is loaded











Let's fix the issue

- **Best solution:** Remove the local HTTP server. Register a **zoom://** protocol handler
- However, let's assume we need to keep the local HTTP server (probably a bad idea). How can we secure it?
 - **Idea:** Require user interaction before joining, don't allow host to automatically enable video
 - **Idea:** Only allow **zoom.us** to communicate with the local server

User joins a zoom call (local server inspects Origin header)

Local Server
localhost:19421

Page is loaded

Client

Server
zoom.us

GET /i/123 HTTP/1.1

Host: zoom.us

HTTP/1.1 200 OK
<!doctype html>

Local Server
localhost:19421

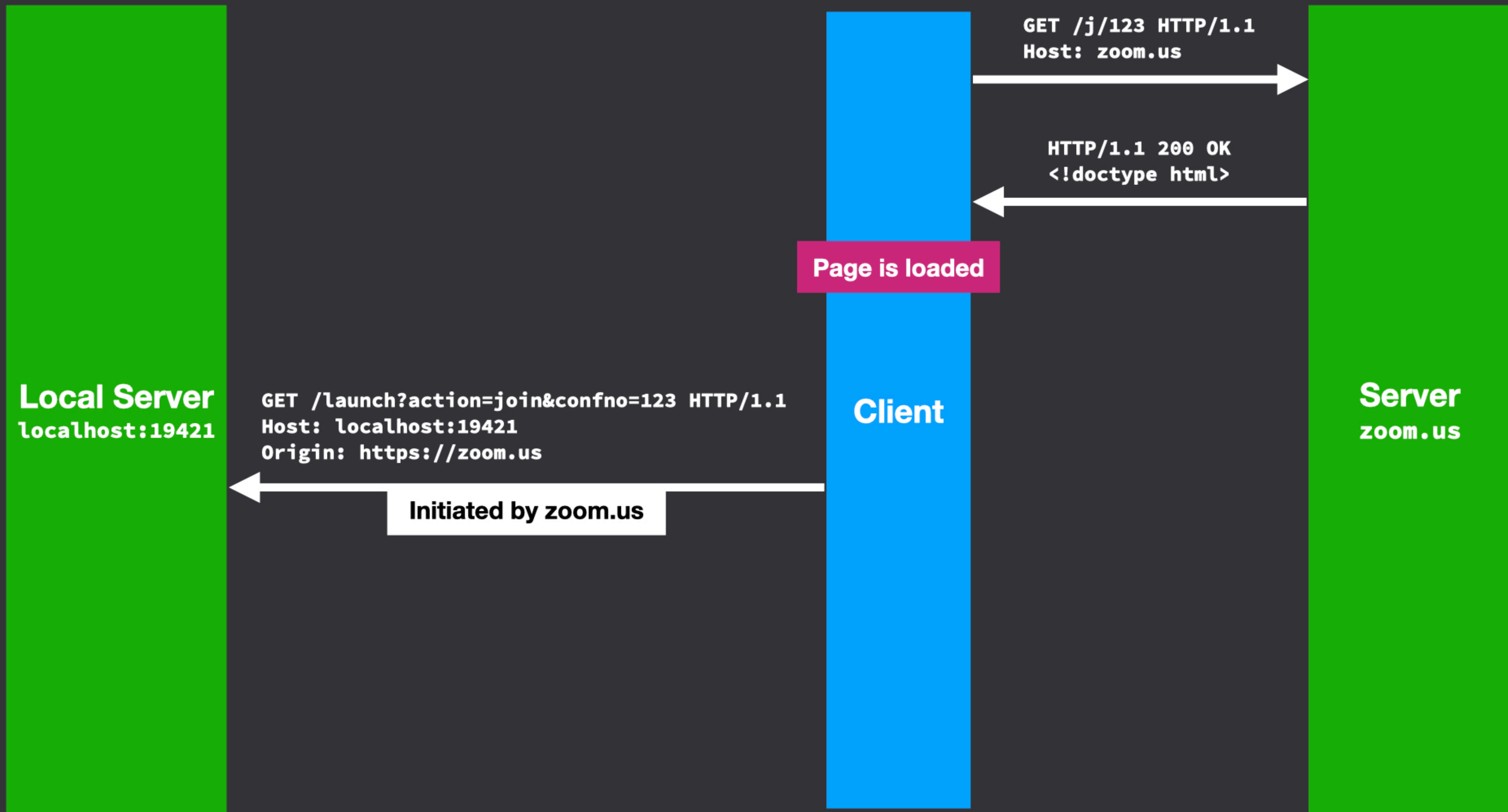
Client

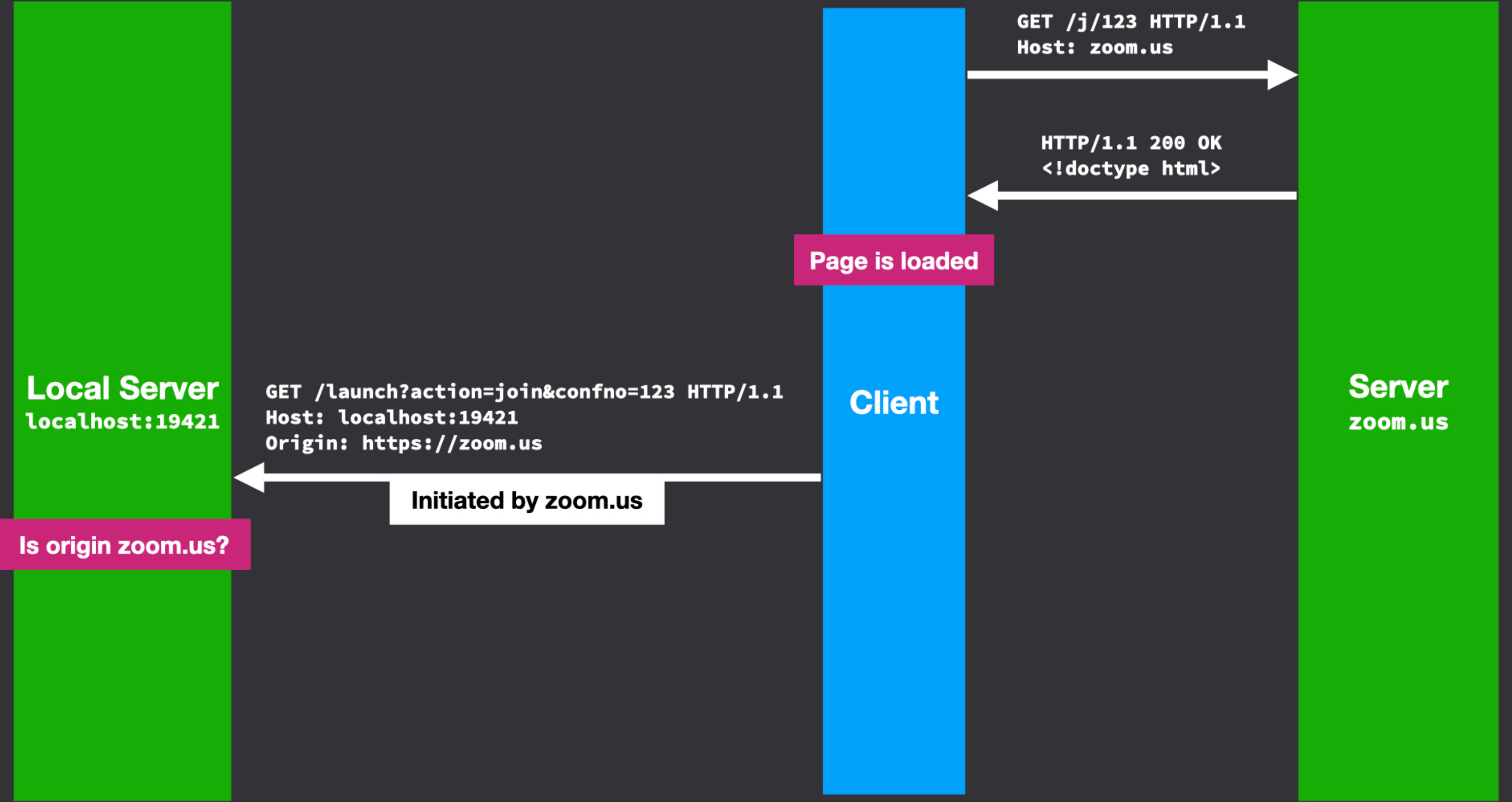
Server
zoom.us

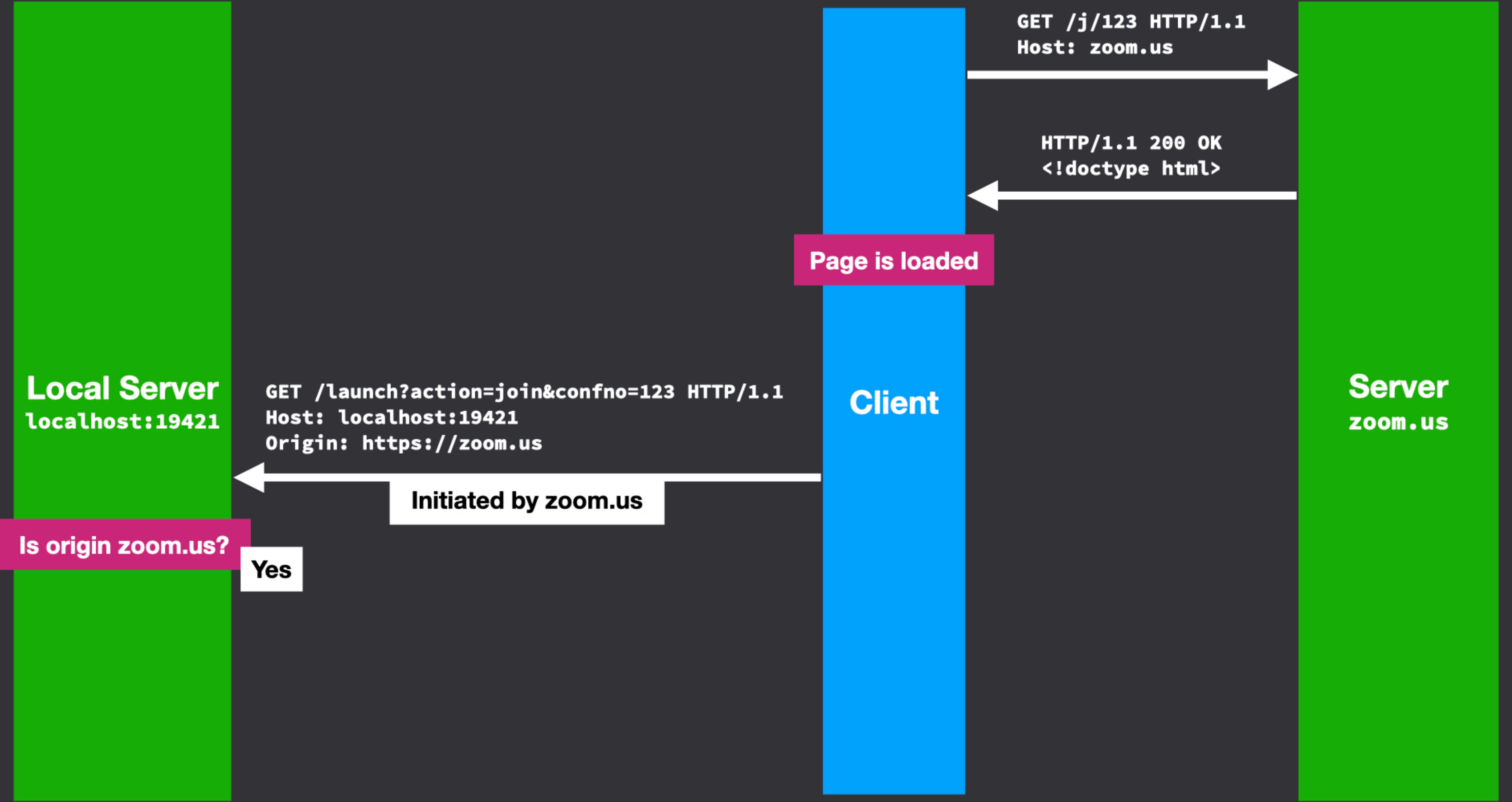
GET /j/123 HTTP/1.1
Host: zoom.us

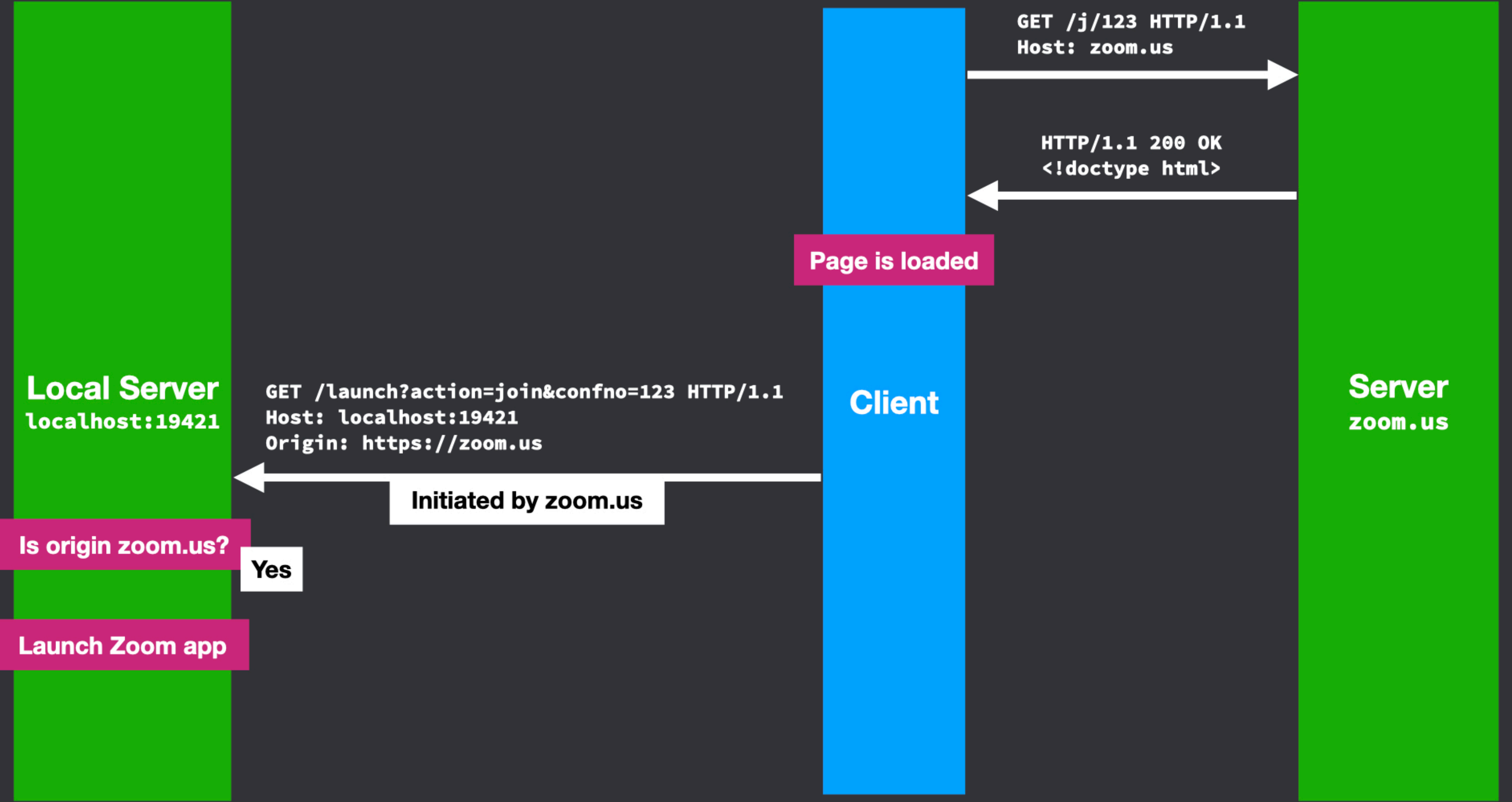
HTTP/1.1 200 OK
<!doctype html>

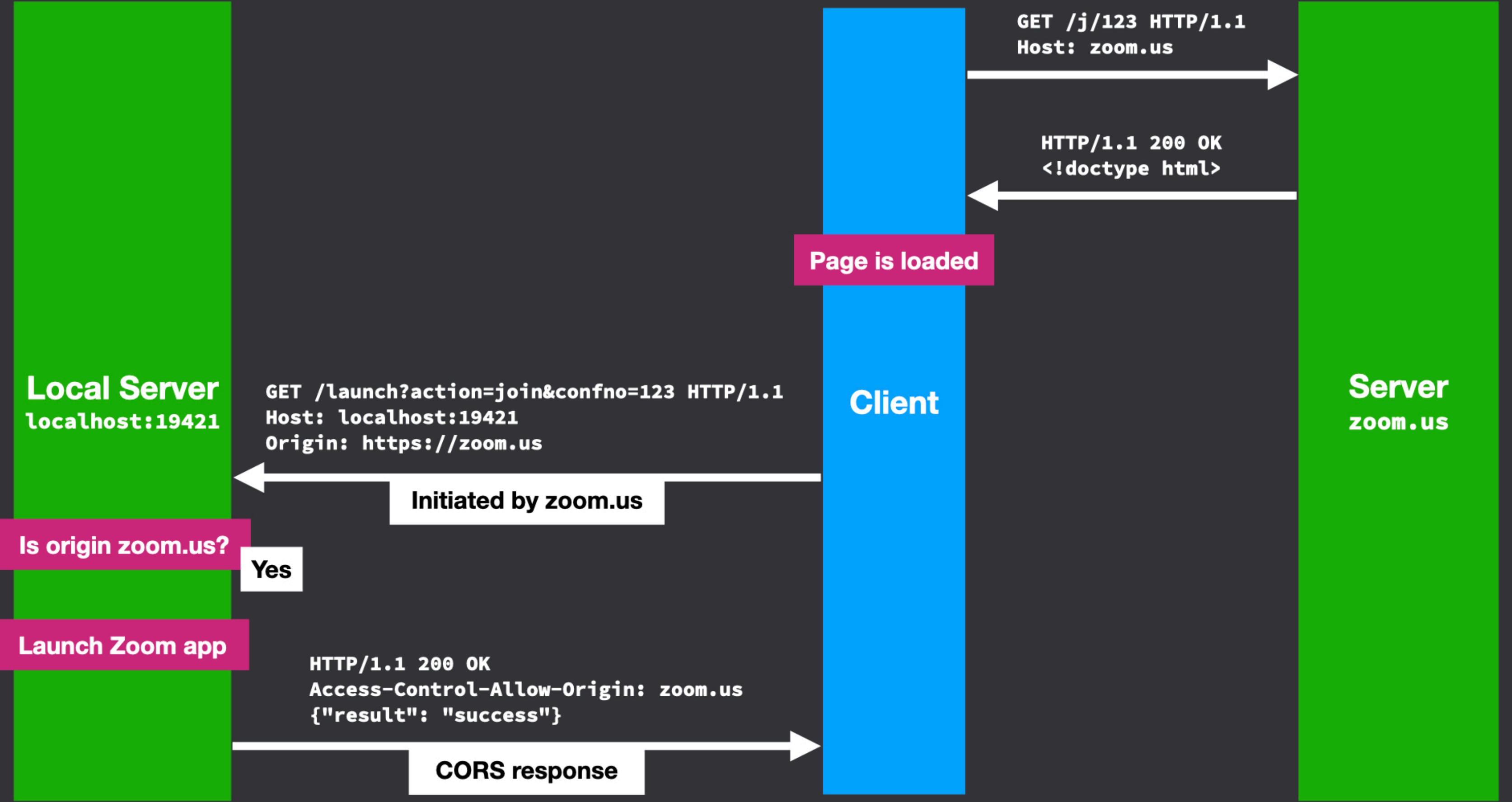
Page is loaded

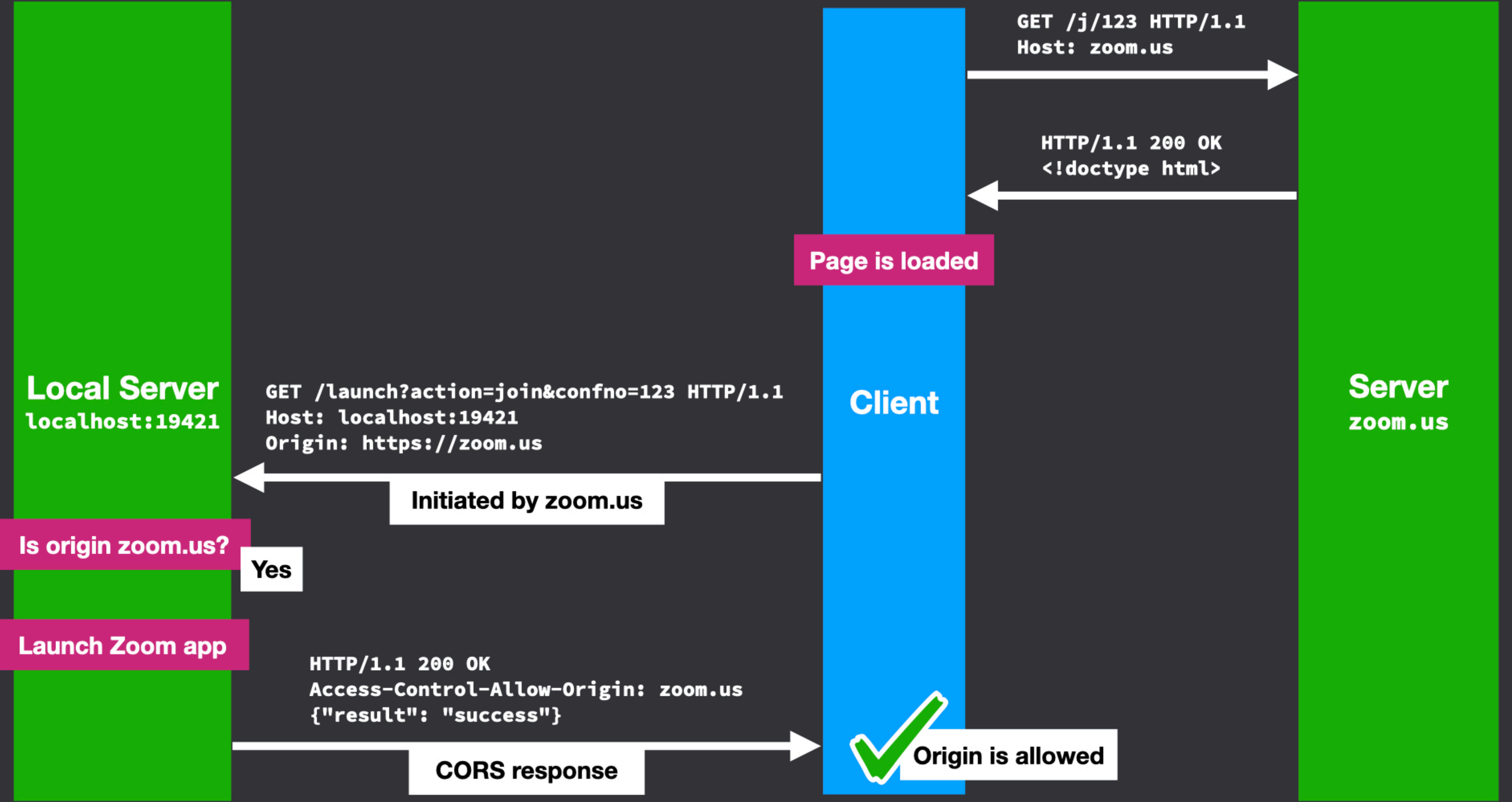












**Attacker joins user into a zoom call
(local server inspects Origin header)**

Local Server
localhost:19421

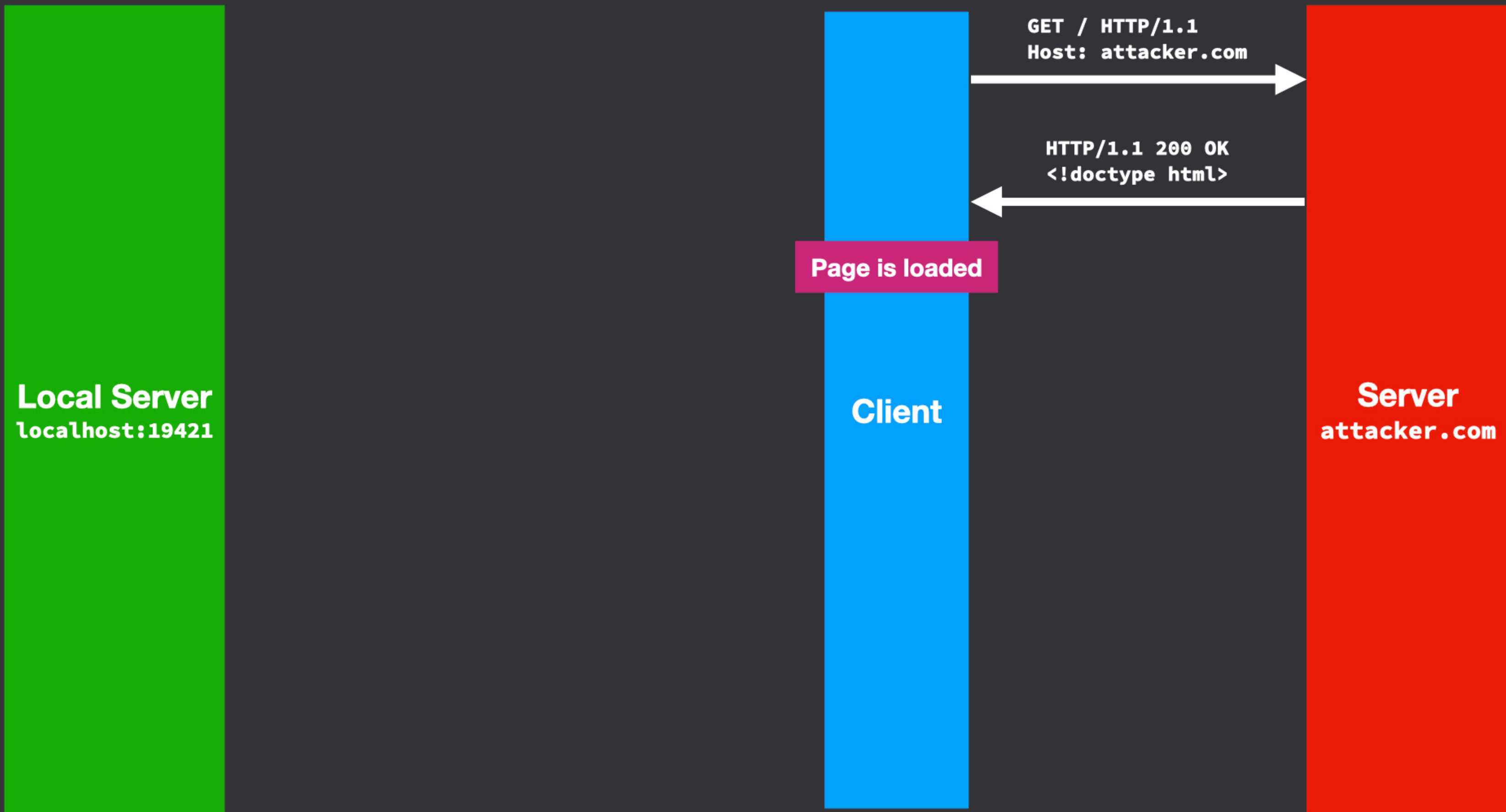
Client

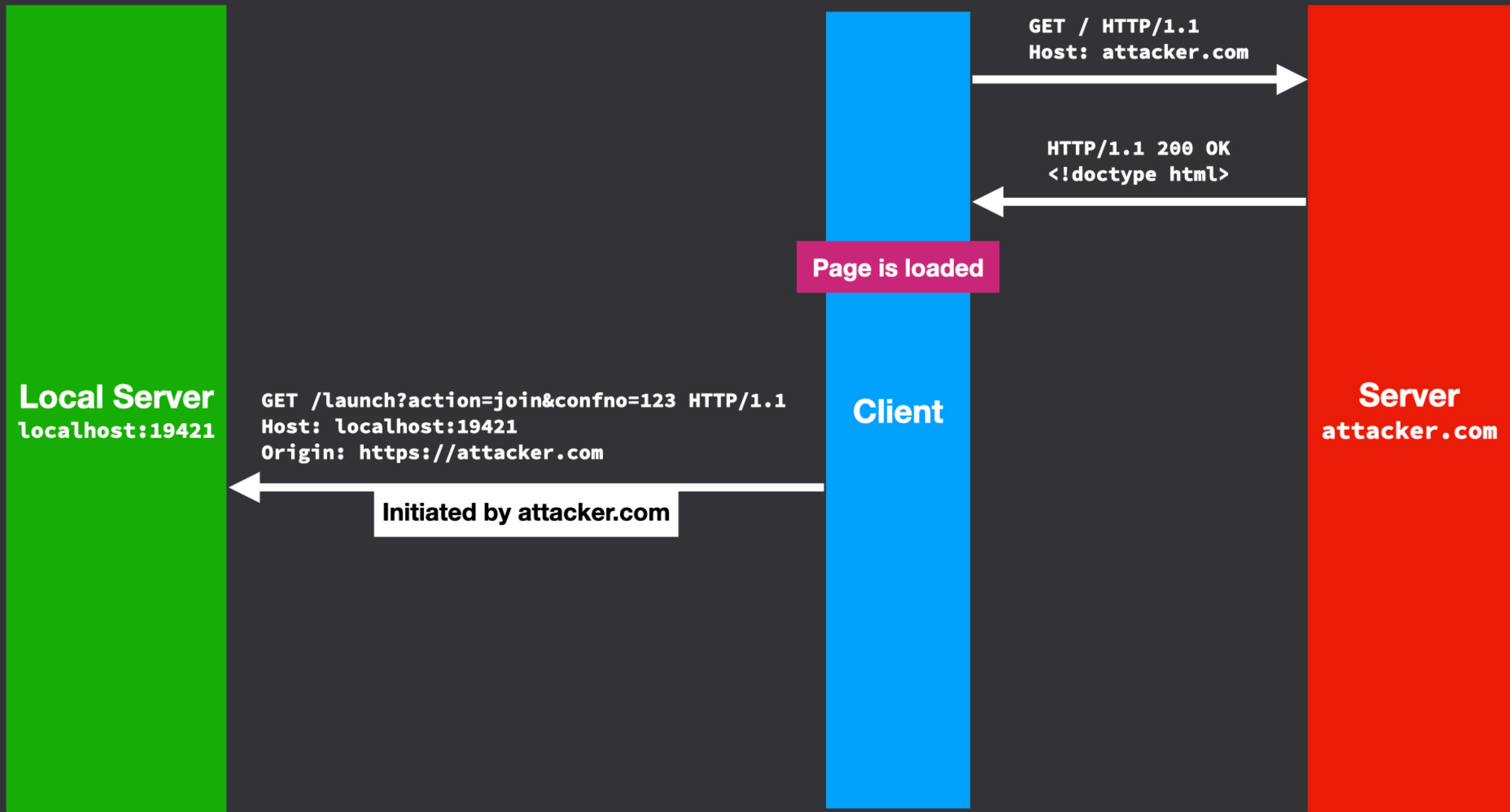
Server
attacker.com

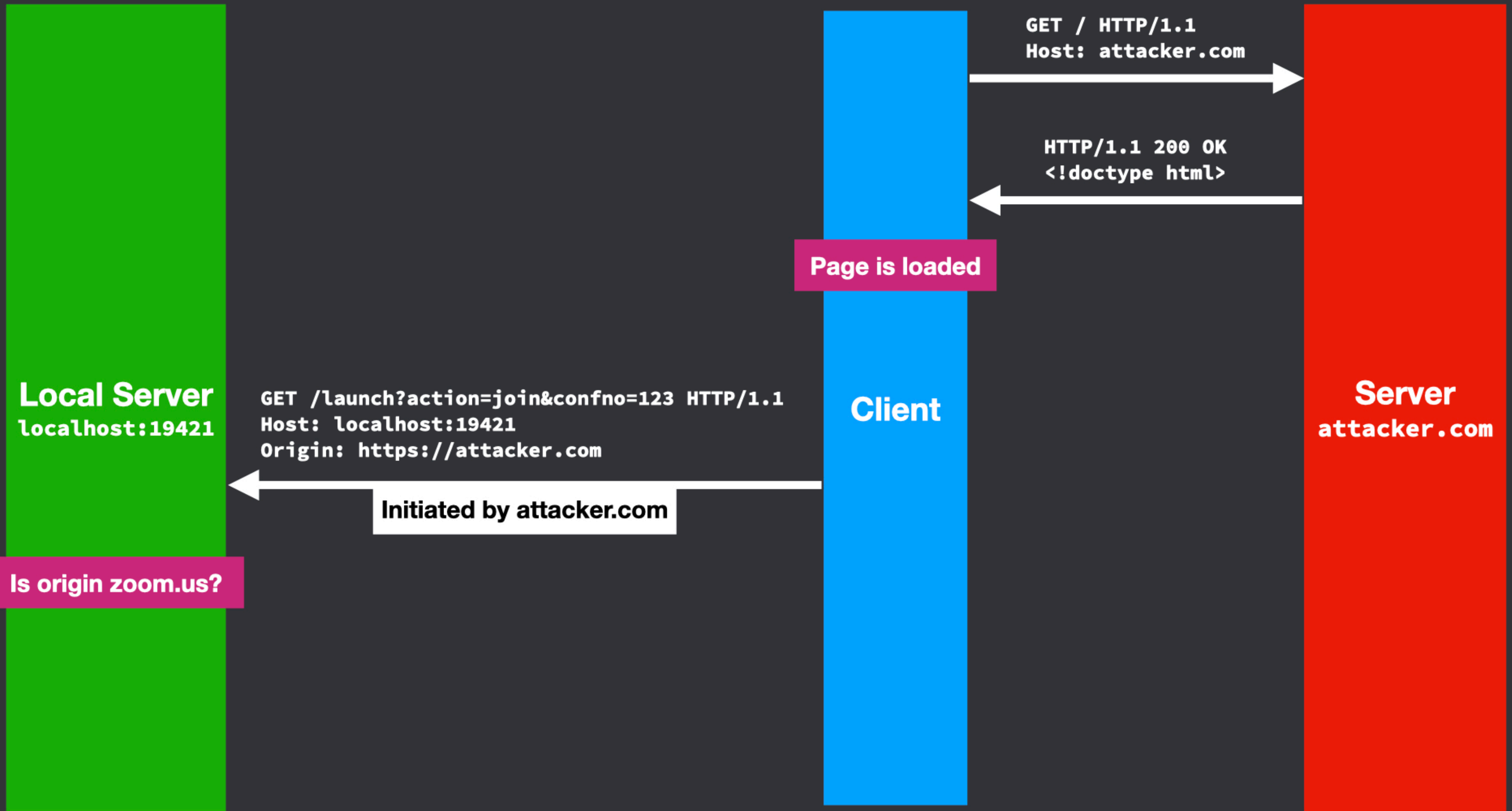
GET / HTTP/1.1
Host: attacker.com

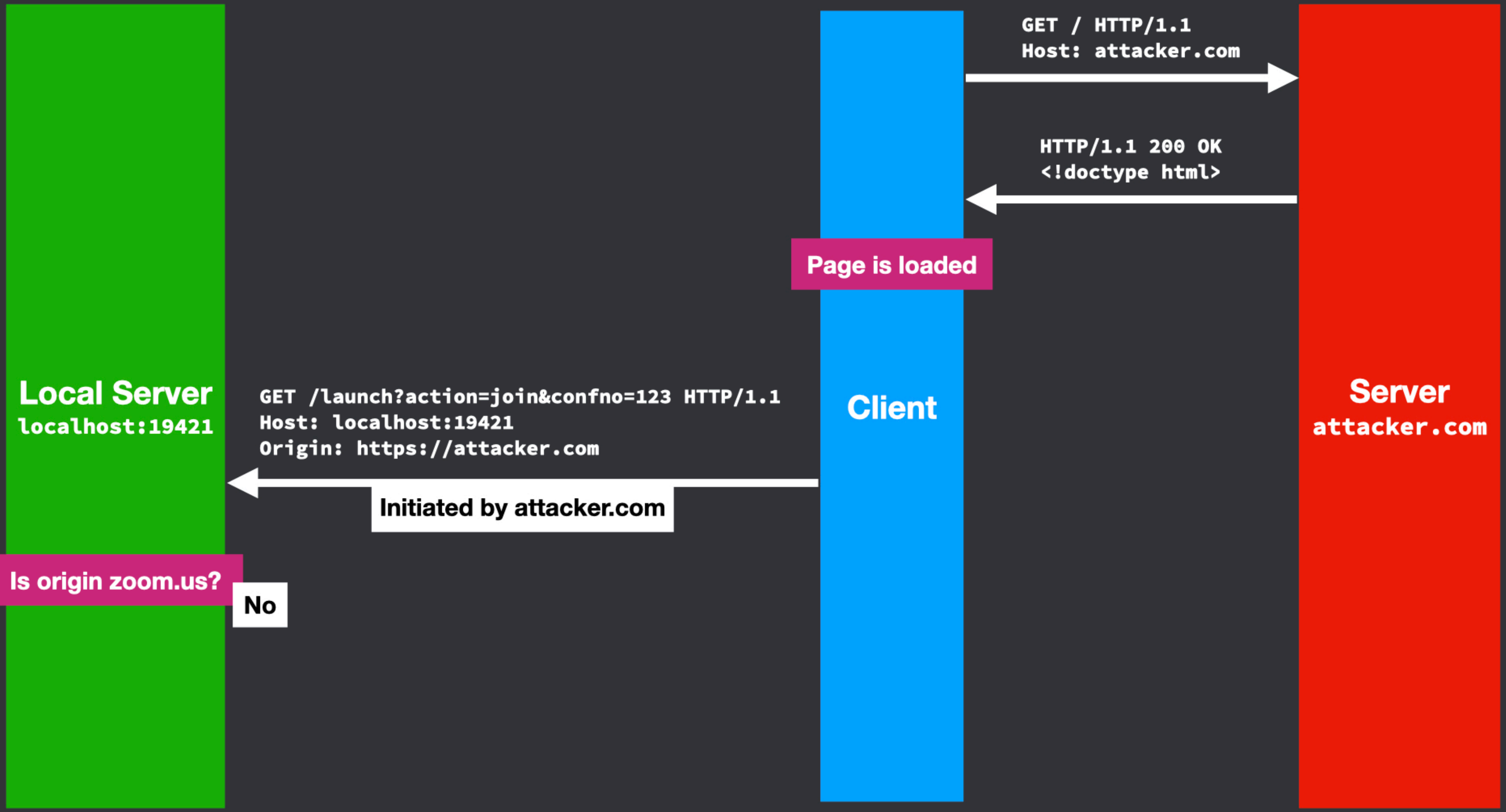
HTTP/1.1 200 OK
<!doctype html>

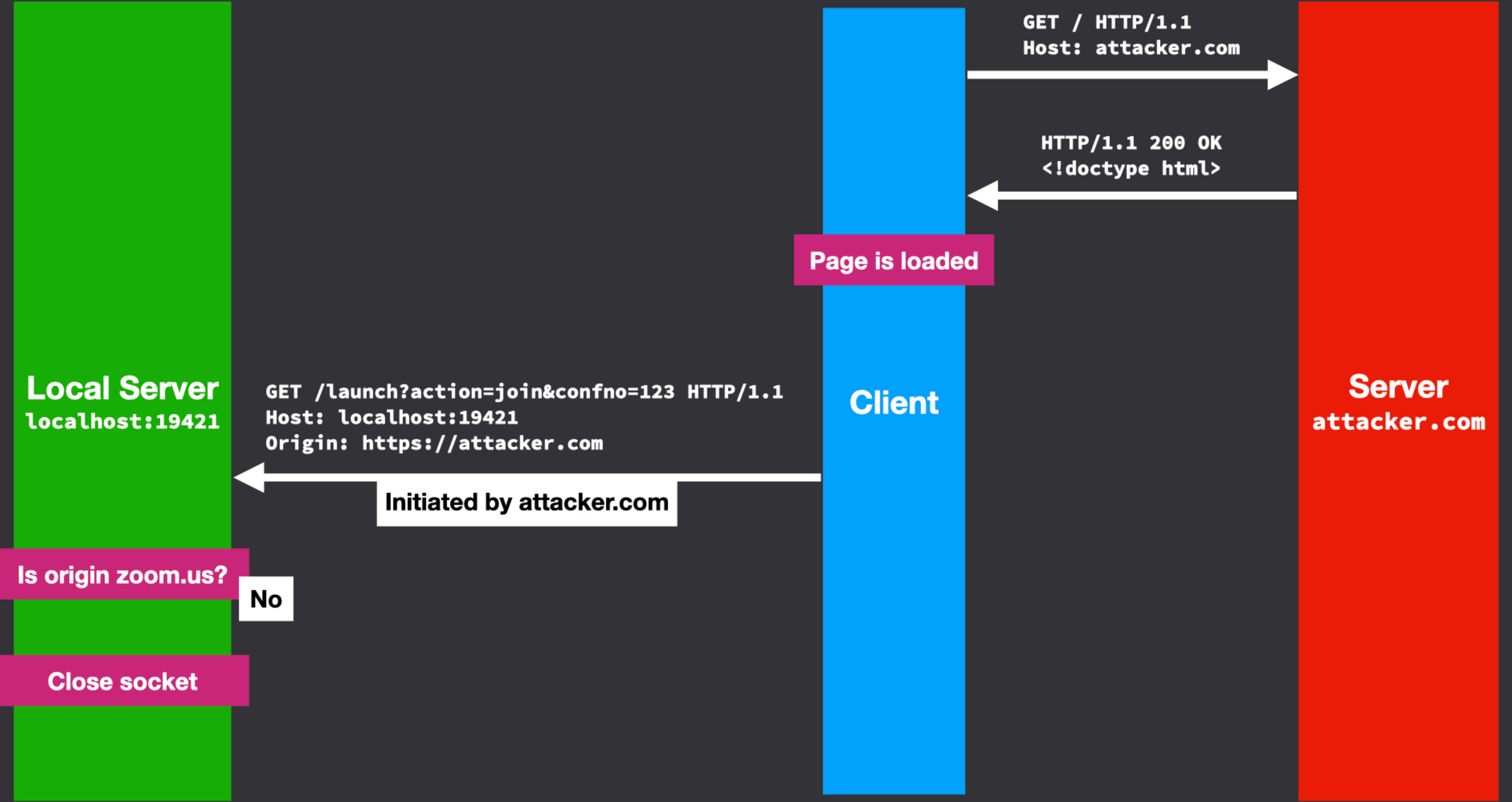
Page is loaded











Problems with inspecting **Origin** header

- **Problem:** Browser doesn't always add **Origin** header
 - for "simple" requests (e.g. `` or `<iframe>` tags)
 - for same origin requests (e.g. `fetch()` to same origin)
 - Very old browsers
- **Solution:** block requests where **Origin** header is omitted
- **Solution:** change the endpoint to require a "preflighted" request so that **Origin** header is always sent (e.g. change the HTTP method to PUT)

"Simple" HTTP requests

- An HTTP/1.1 **GET**, **HEAD** or a **POST** is the request method
- In the case of a **POST**, the **Content-Type** of the request body is one of **application/x-www-form-urlencoded**, **multipart/form-data**, or **text/plain**
- No custom HTTP headers are set (or, only CORS-safelisted headers are set)

"Preflighted" HTTP requests

- Before a "preflighted" requests can be sent to the target server, the browser must check that it is safe to send
- Browser sends **OPTIONS** request first to ask the server if the request we want to send is okay
- If server doesn't support **OPTIONS** (either because it is old or because it doesn't want to support preflighted requests) then, preflighted requests are **denied**

What happens if the browser does not preflight "non-simple" requests

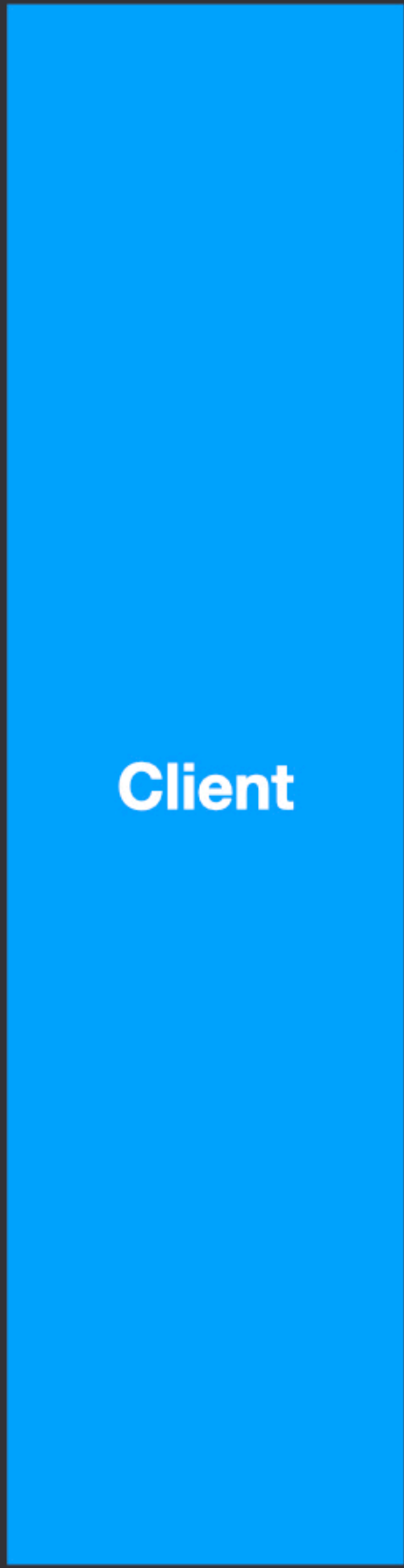
Client

Server
victim.com

Client

Server
attacker.com

Server
victim.com



Client

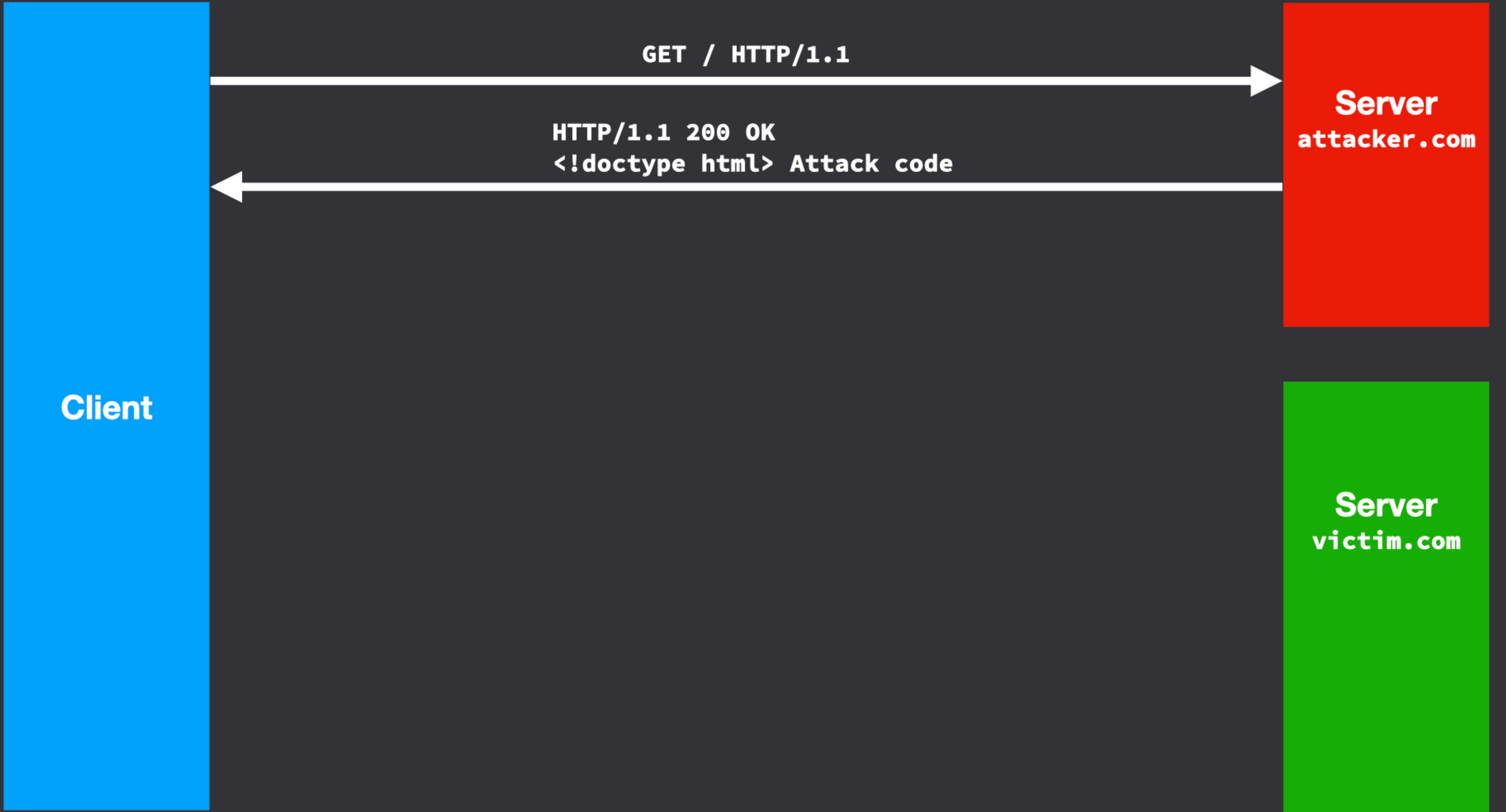
GET / HTTP/1.1



Server
attacker.com



Server
victim.com



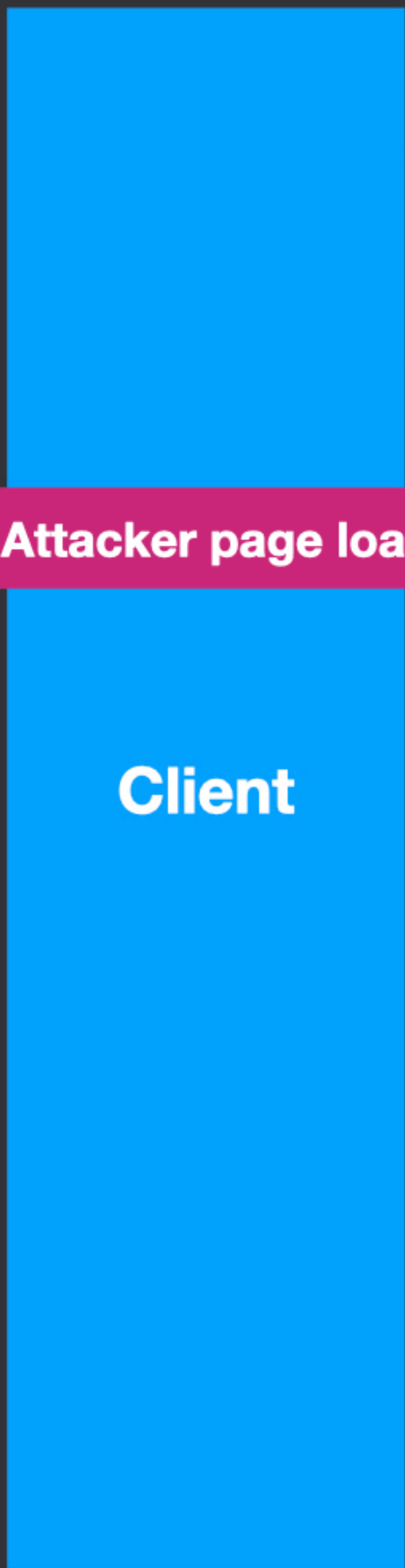
Client

Server
attacker.com

Server
victim.com

GET / HTTP/1.1

HTTP/1.1 200 OK
<!doctype html> Attack code



GET / HTTP/1.1

HTTP/1.1 200 OK
<!doctype html> Attack code

Attacker page loads

Client

Server
victim.com

Server
attacker.com



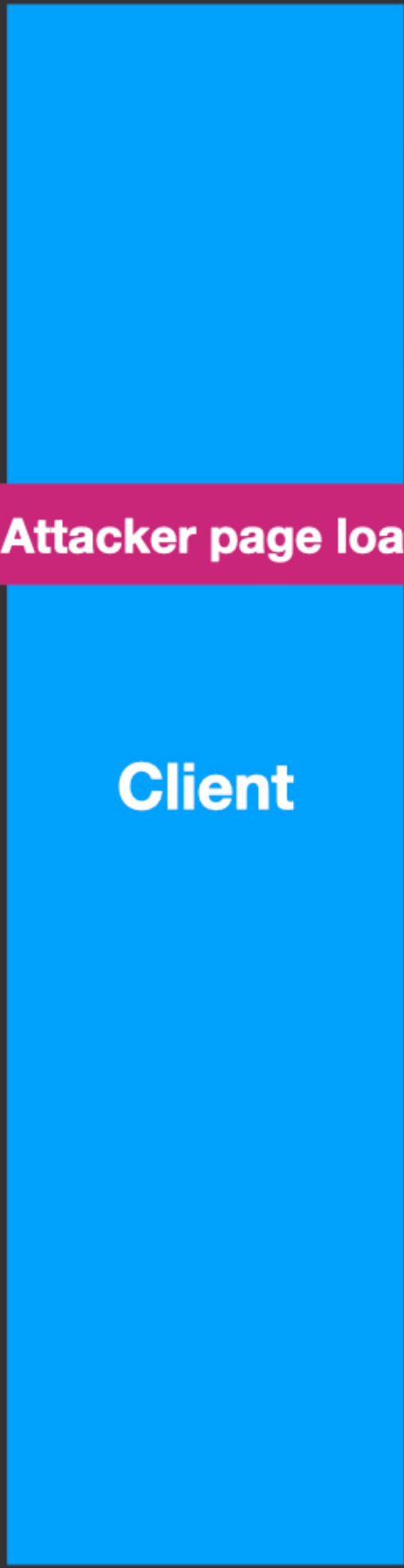
Attacker page loads

Client

Server
attacker.com

DELETE /item/42 HTTP/1.1
Cookie: sessionId=123

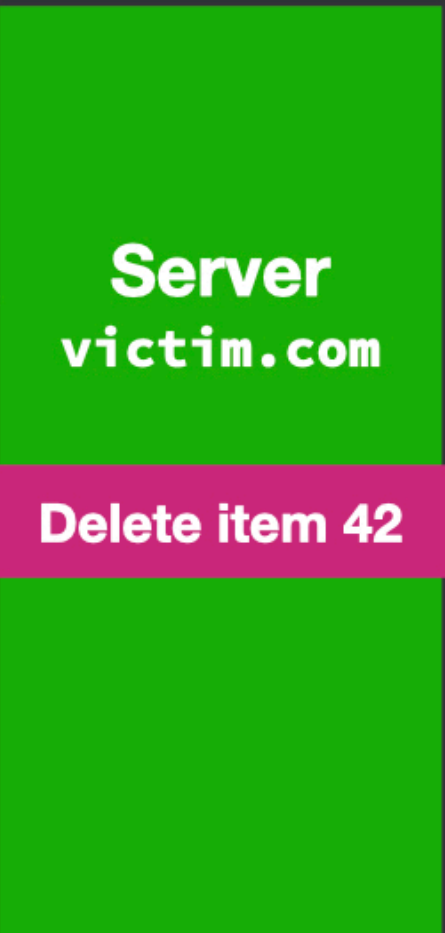
Server
victim.com



Client



Server
attacker.com



Server
victim.com

Attacker page loads

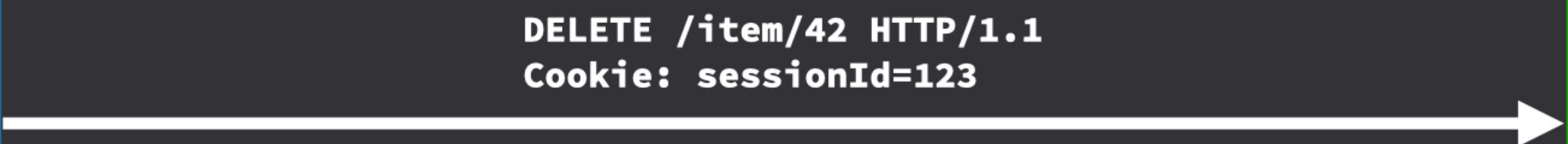
Delete item 42



GET / HTTP/1.1



HTTP/1.1 200 OK
<!doctype html> Attack code



DELETE /item/42 HTTP/1.1
Cookie: sessionId=123





Introducing the OPTIONS request

- Browser sends **OPTIONS** request first to ask the server if the request we want to send is okay
- If server doesn't support **OPTIONS** (either because it is old or because it doesn't want to support preflighted requests) then, preflighted requests are **denied**
- Let's see how it can protect our local HTTP server

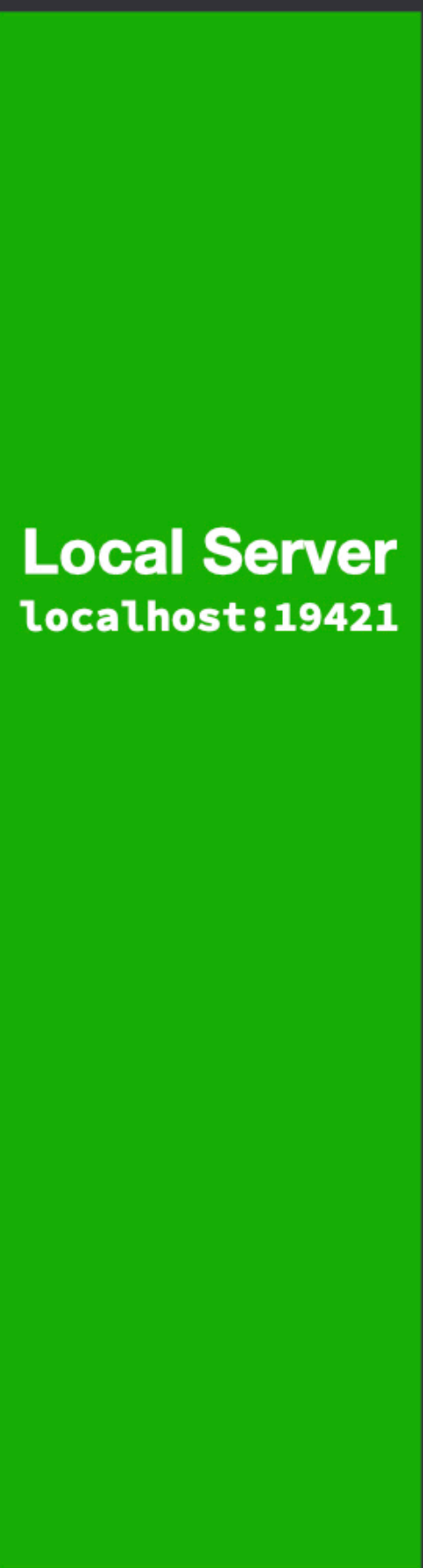
User joins a zoom call (local server requires "preflighted" request)

Local Server
localhost:19421

Client

Server
zoom.us



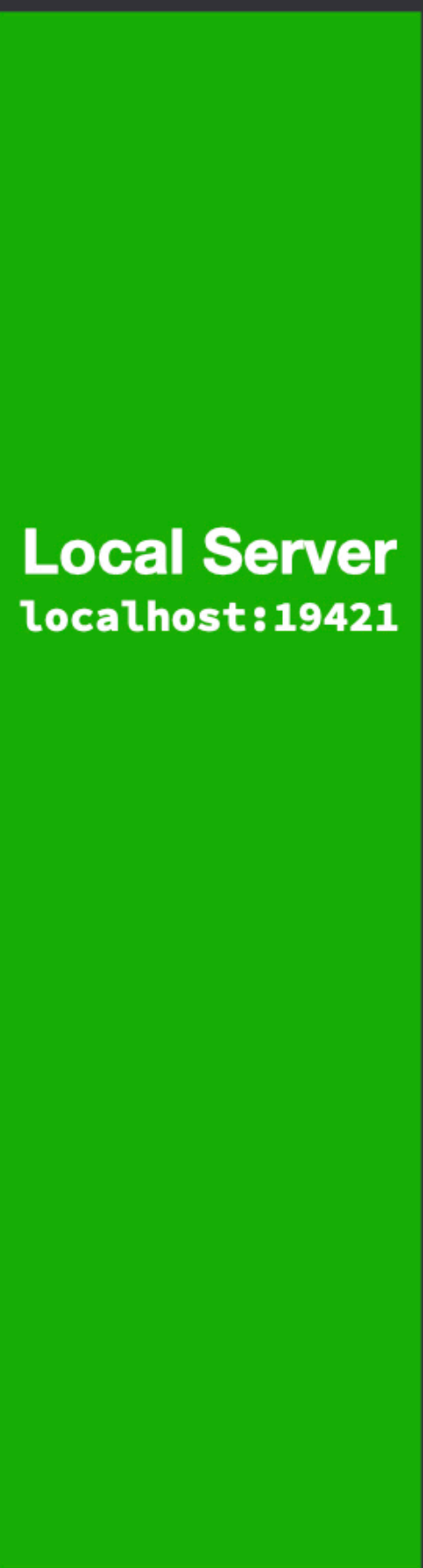


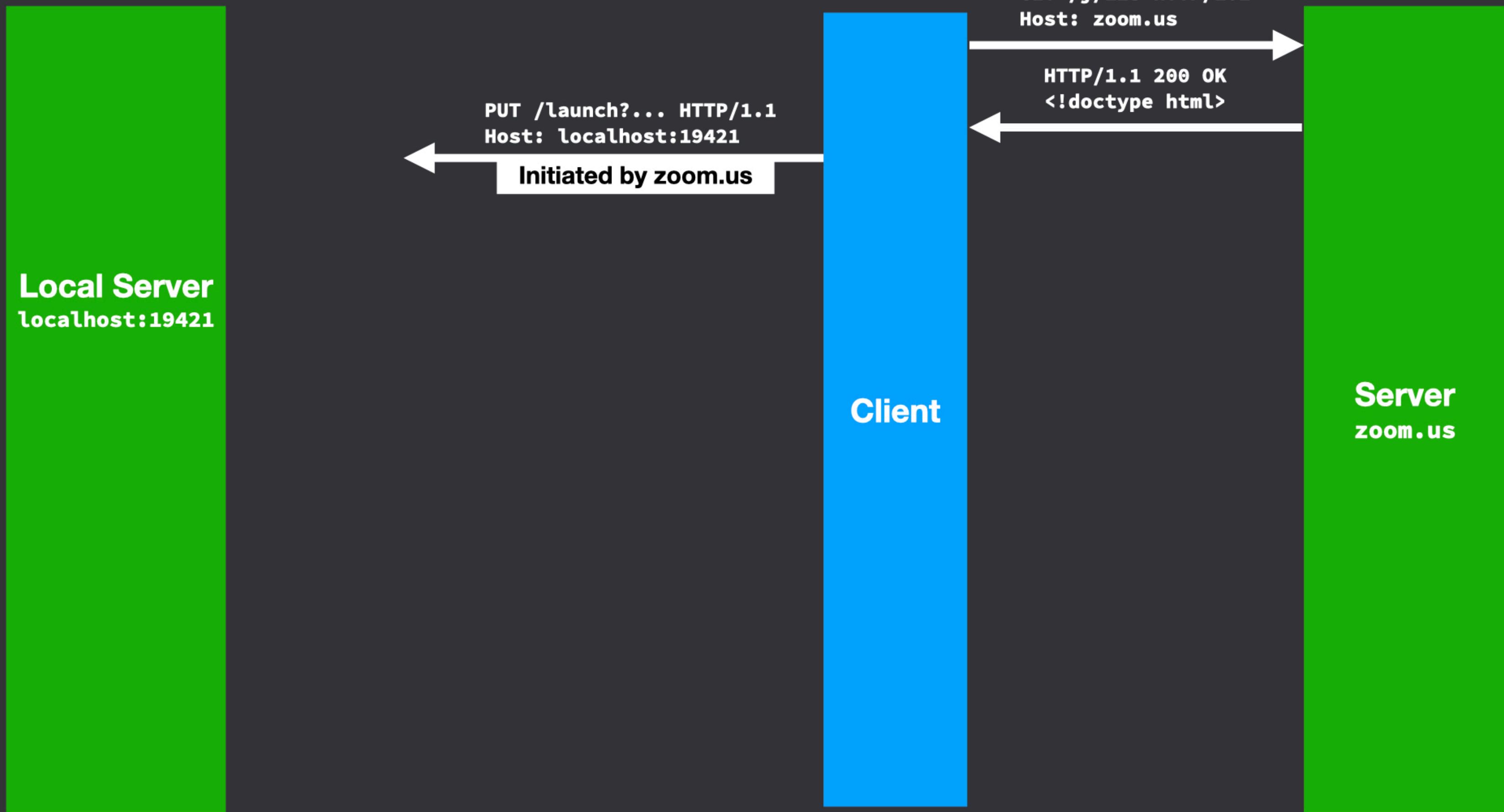
GET /j/123 HTTP/1.1
Host: zoom.us

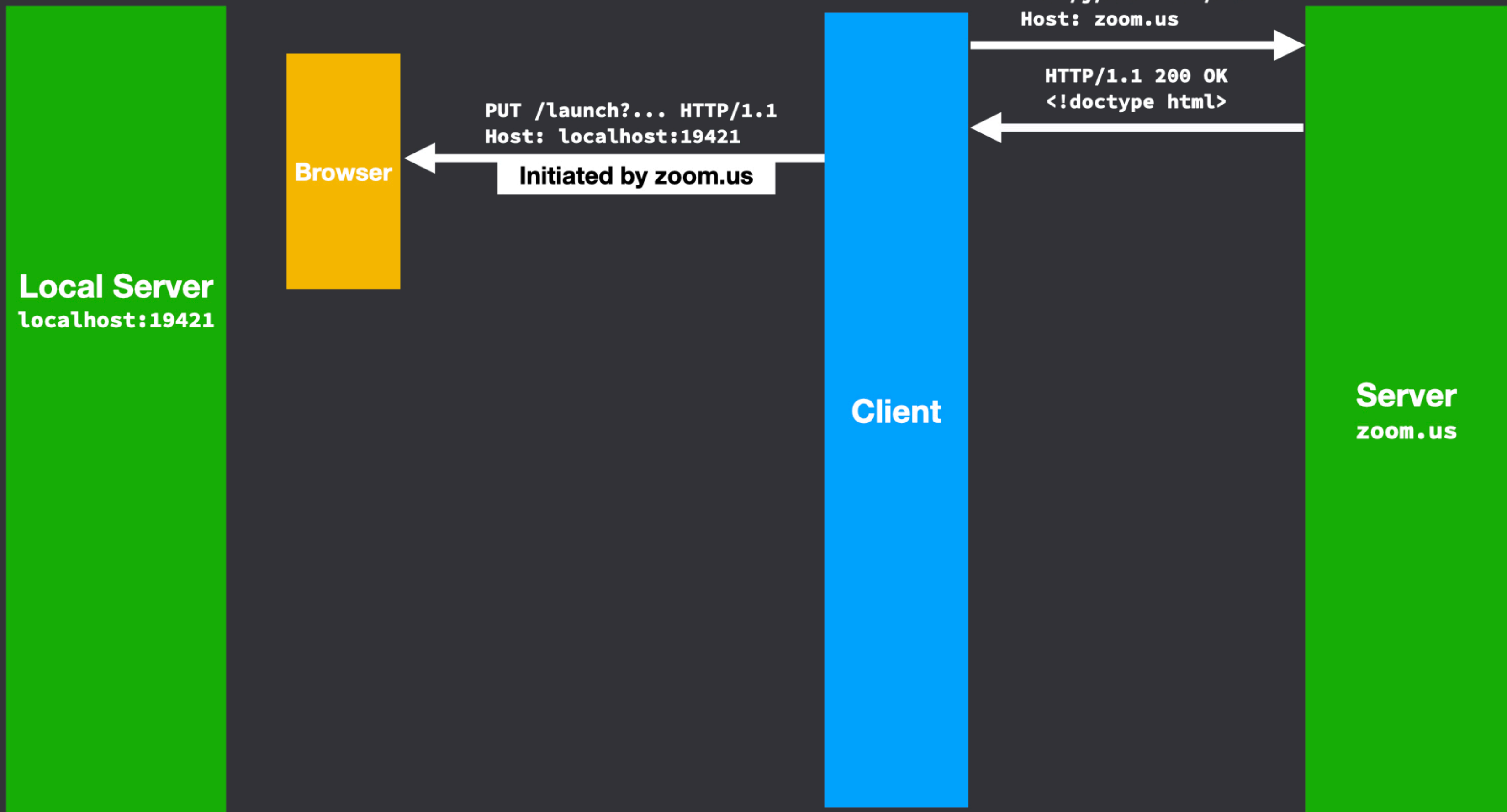


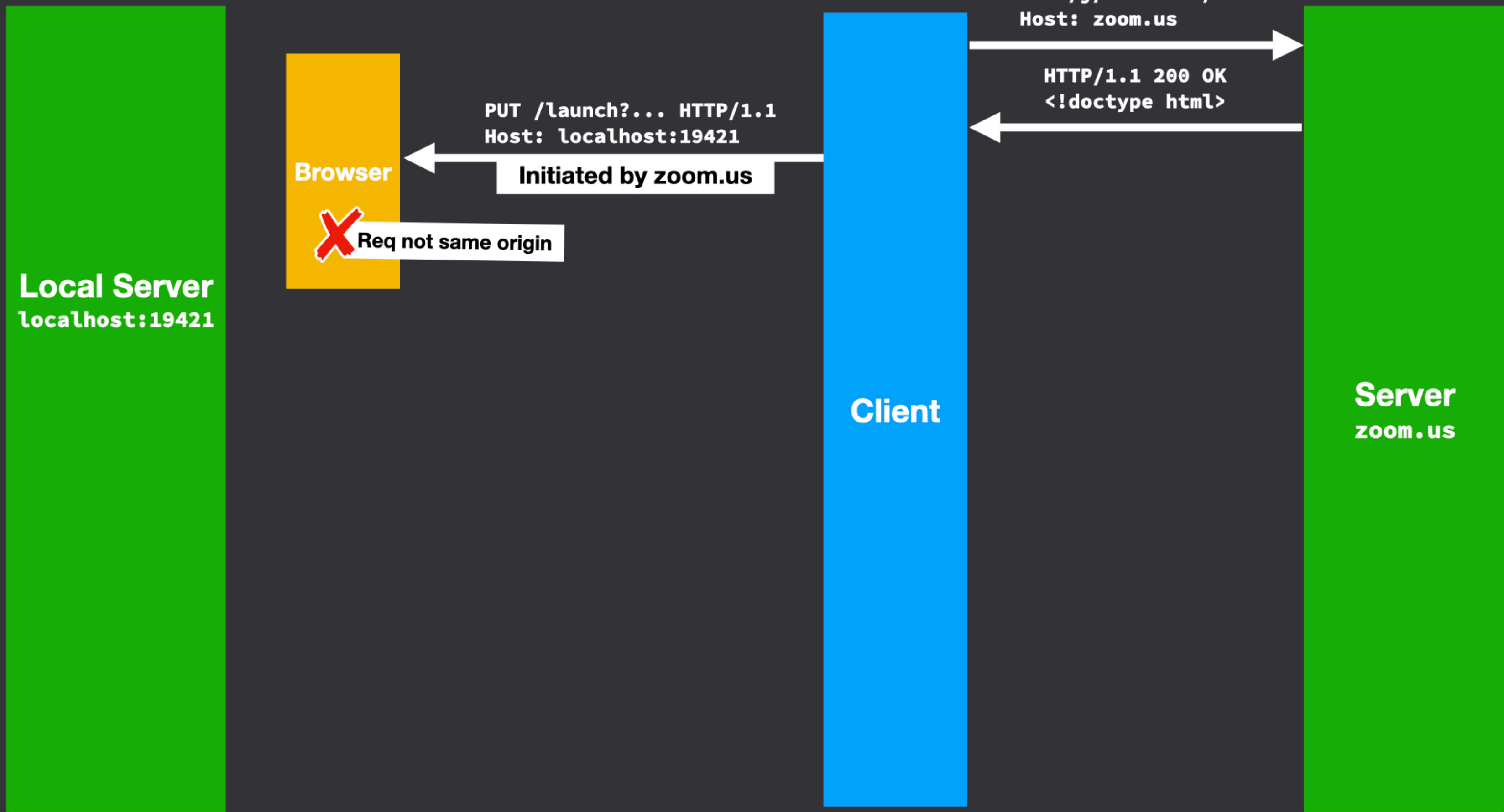
A white arrow pointing from the Client box to the Server box.

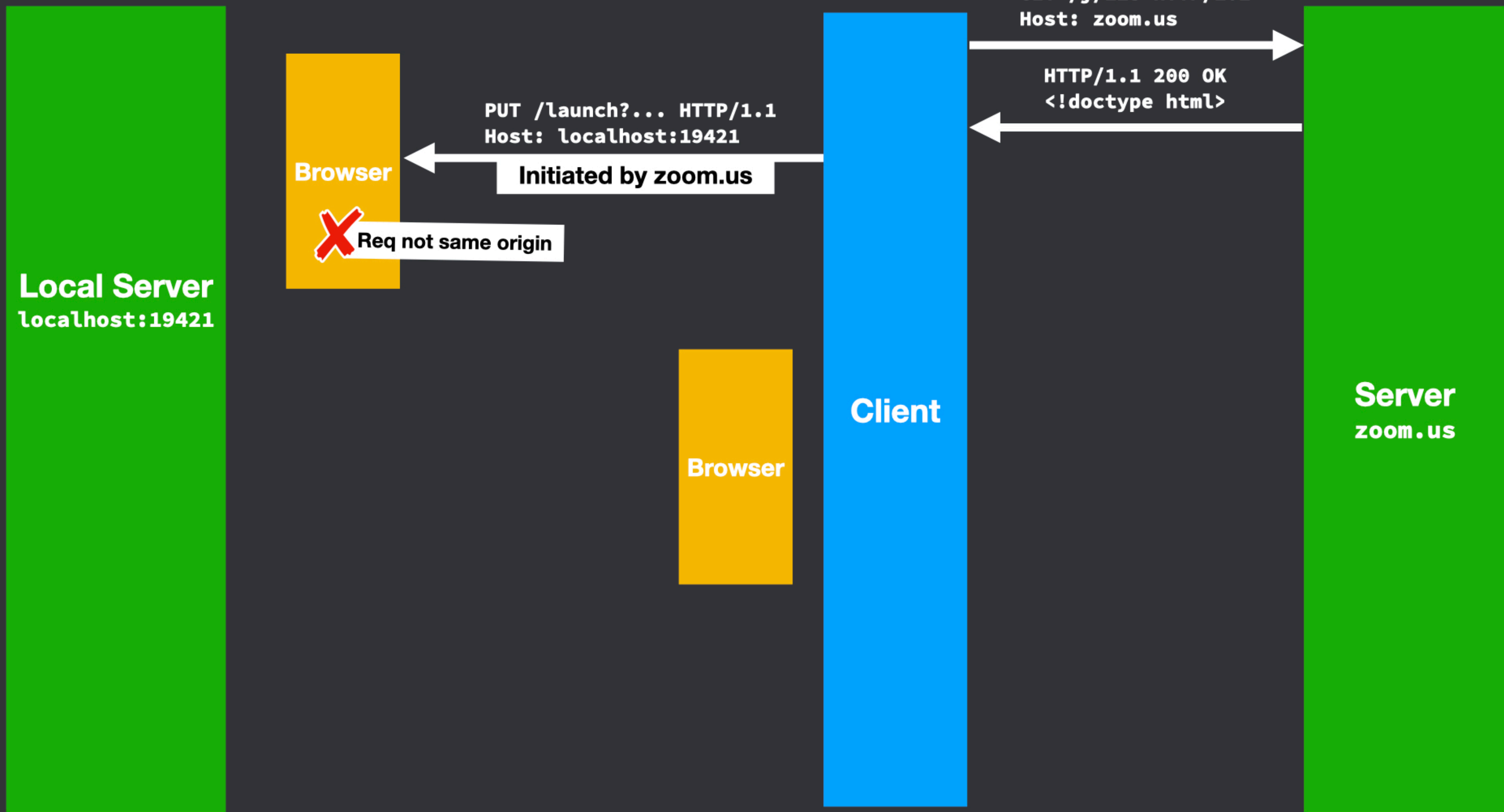


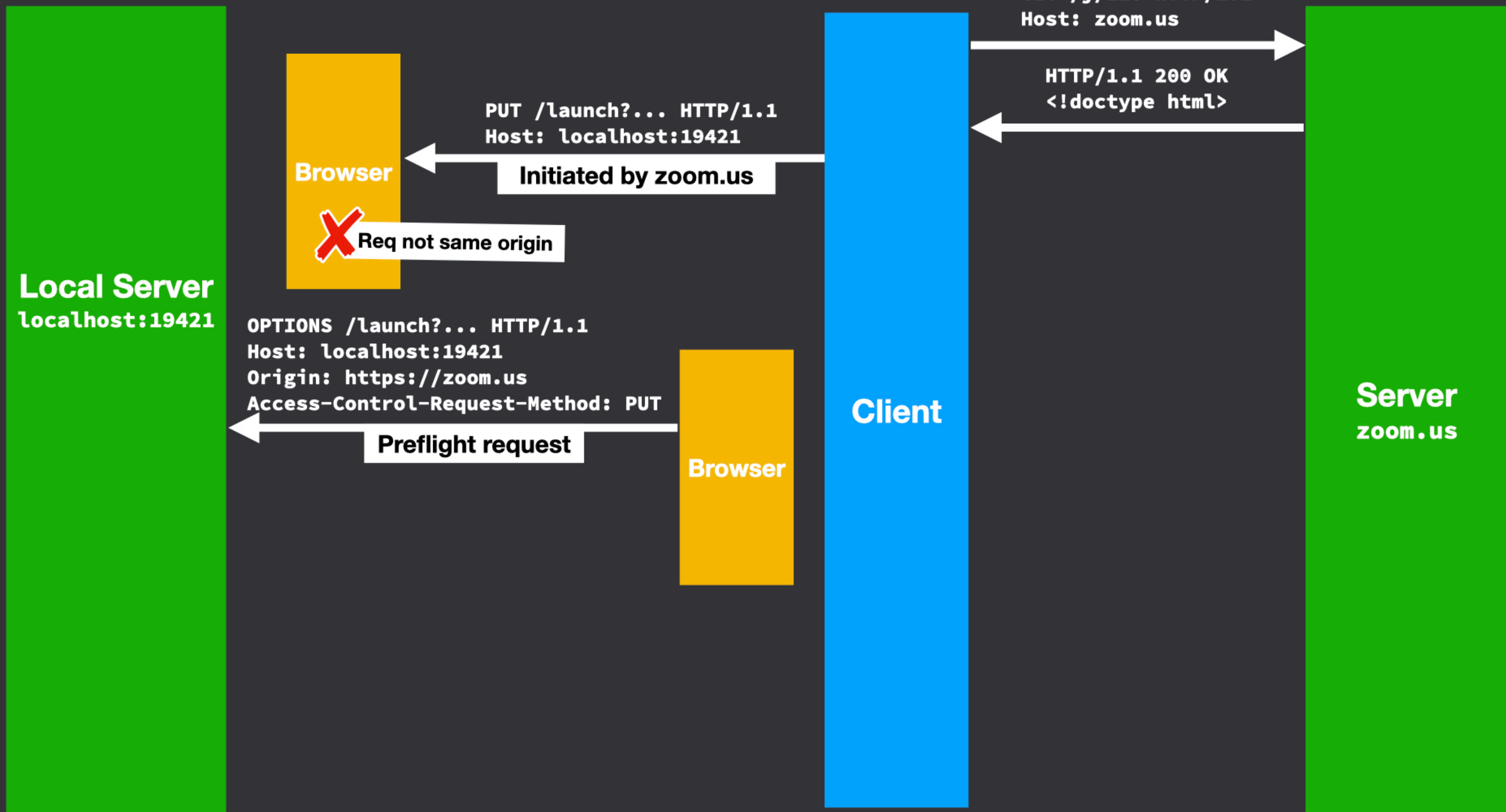


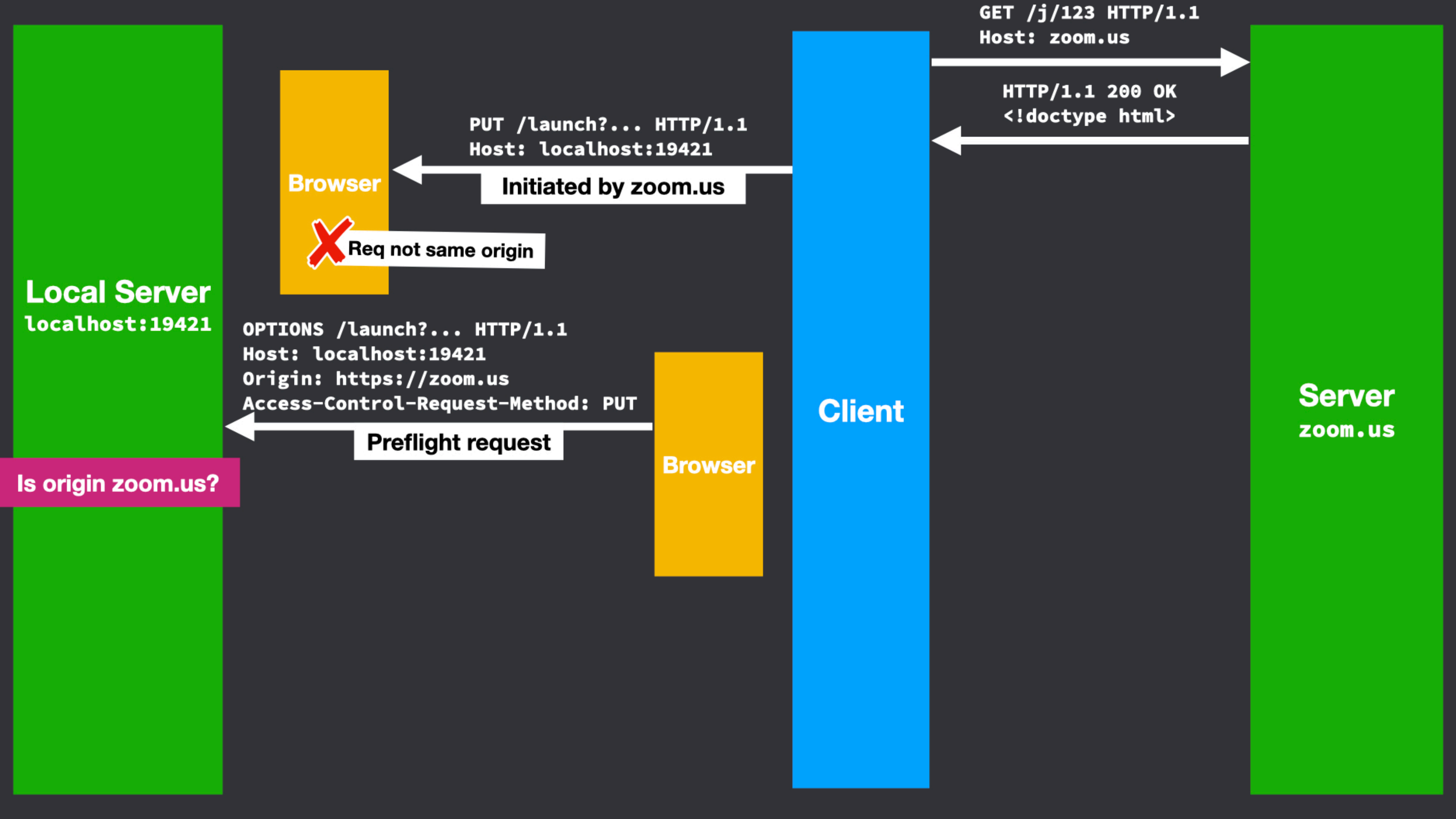


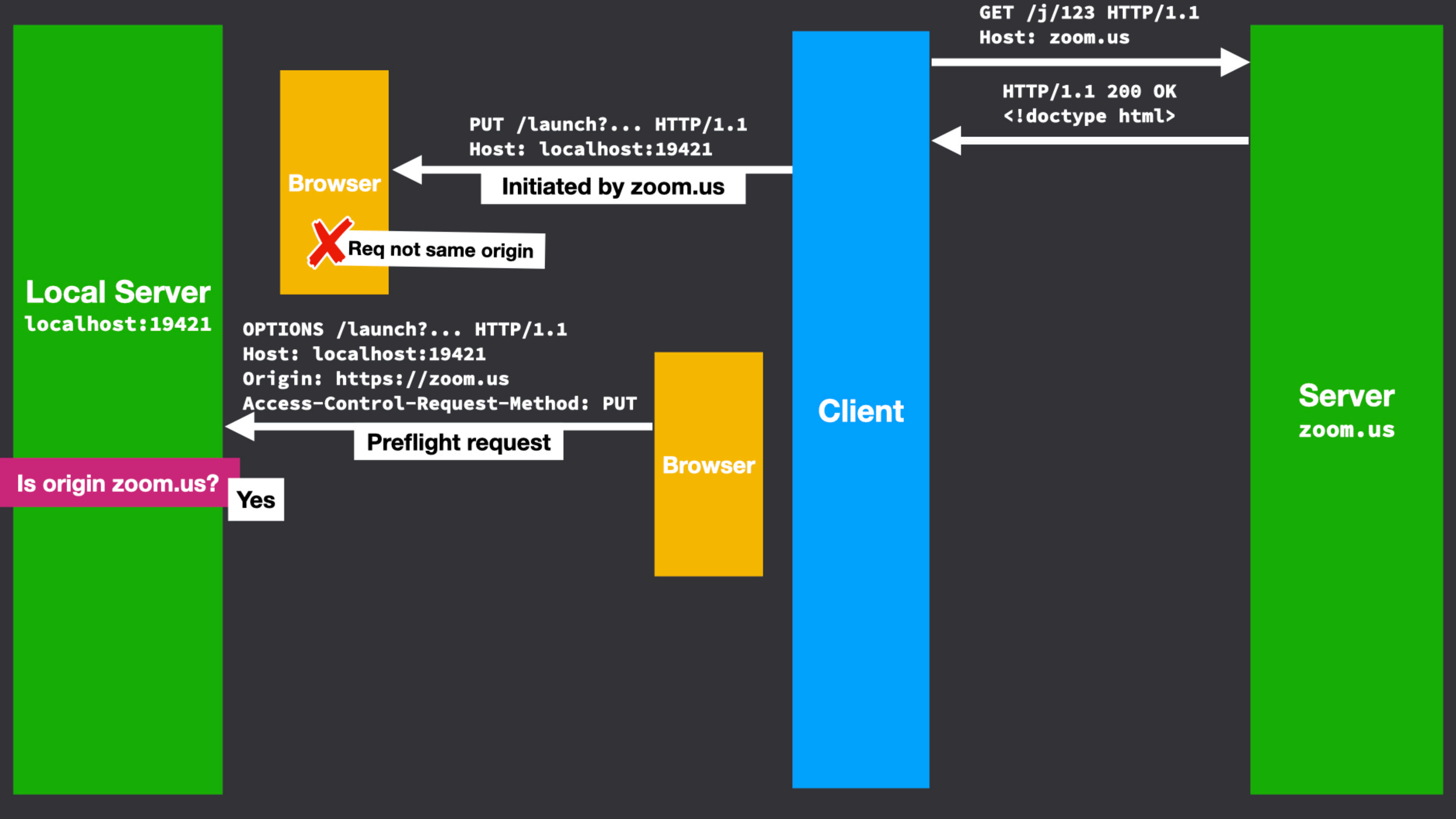


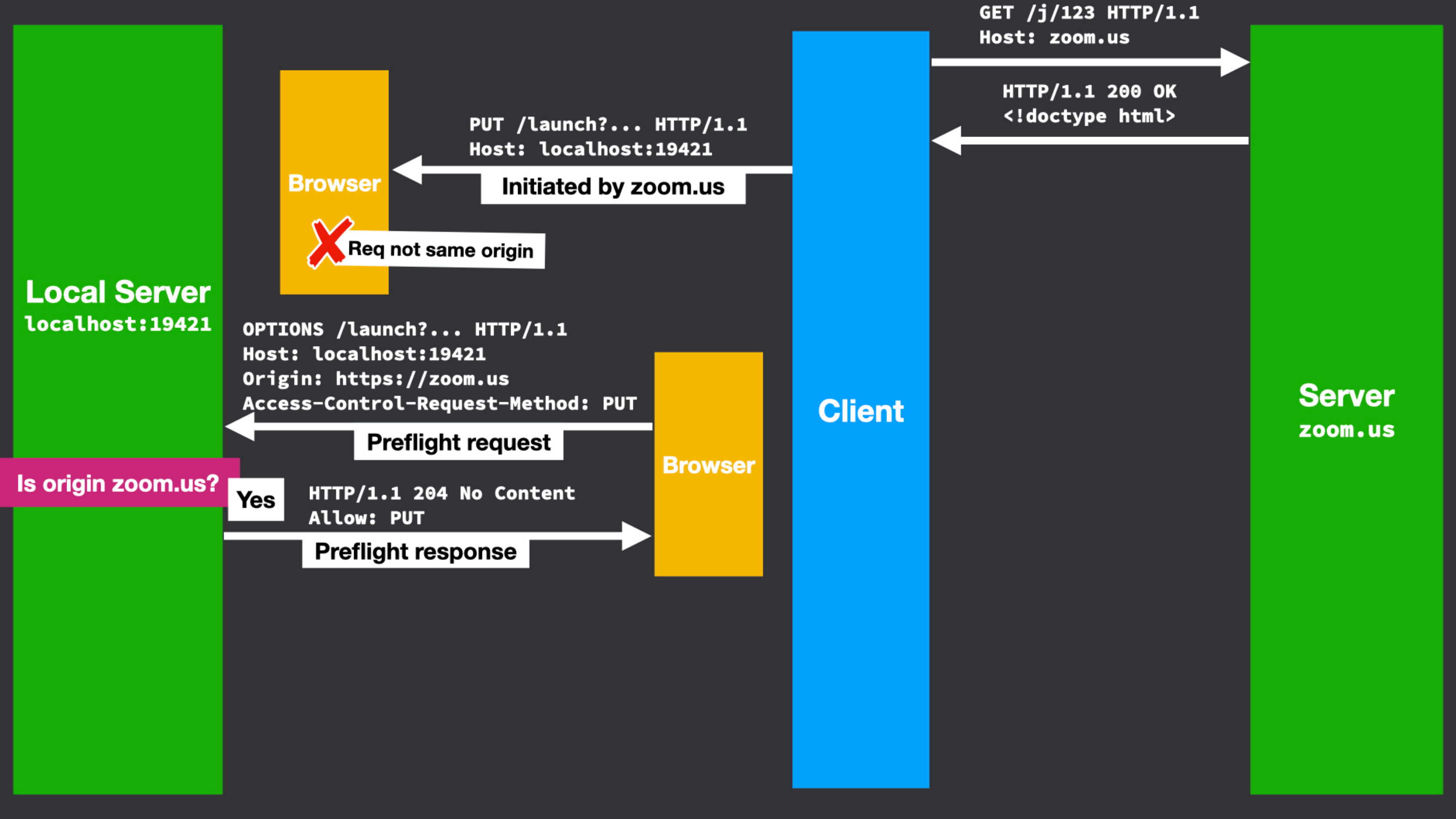


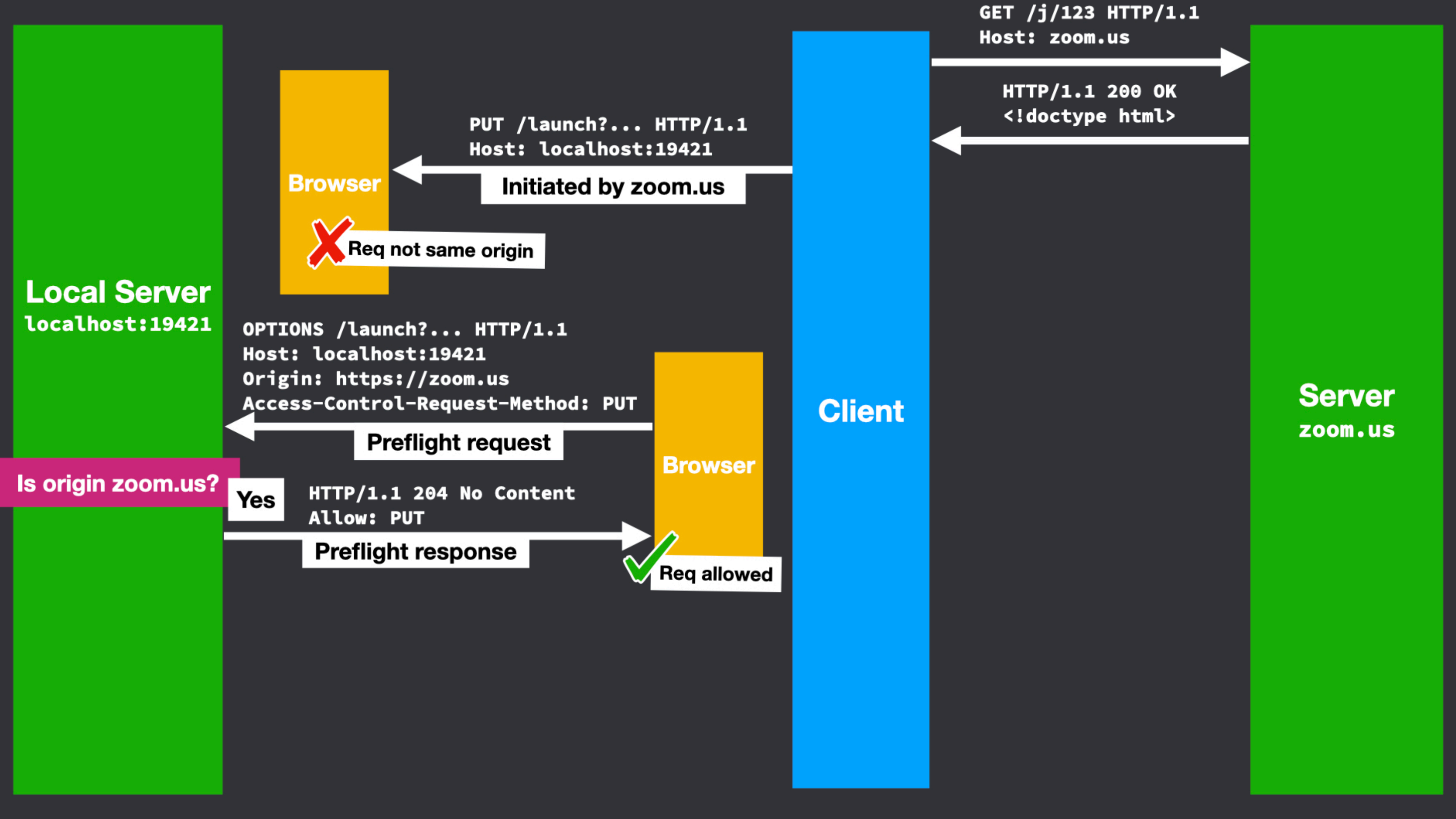


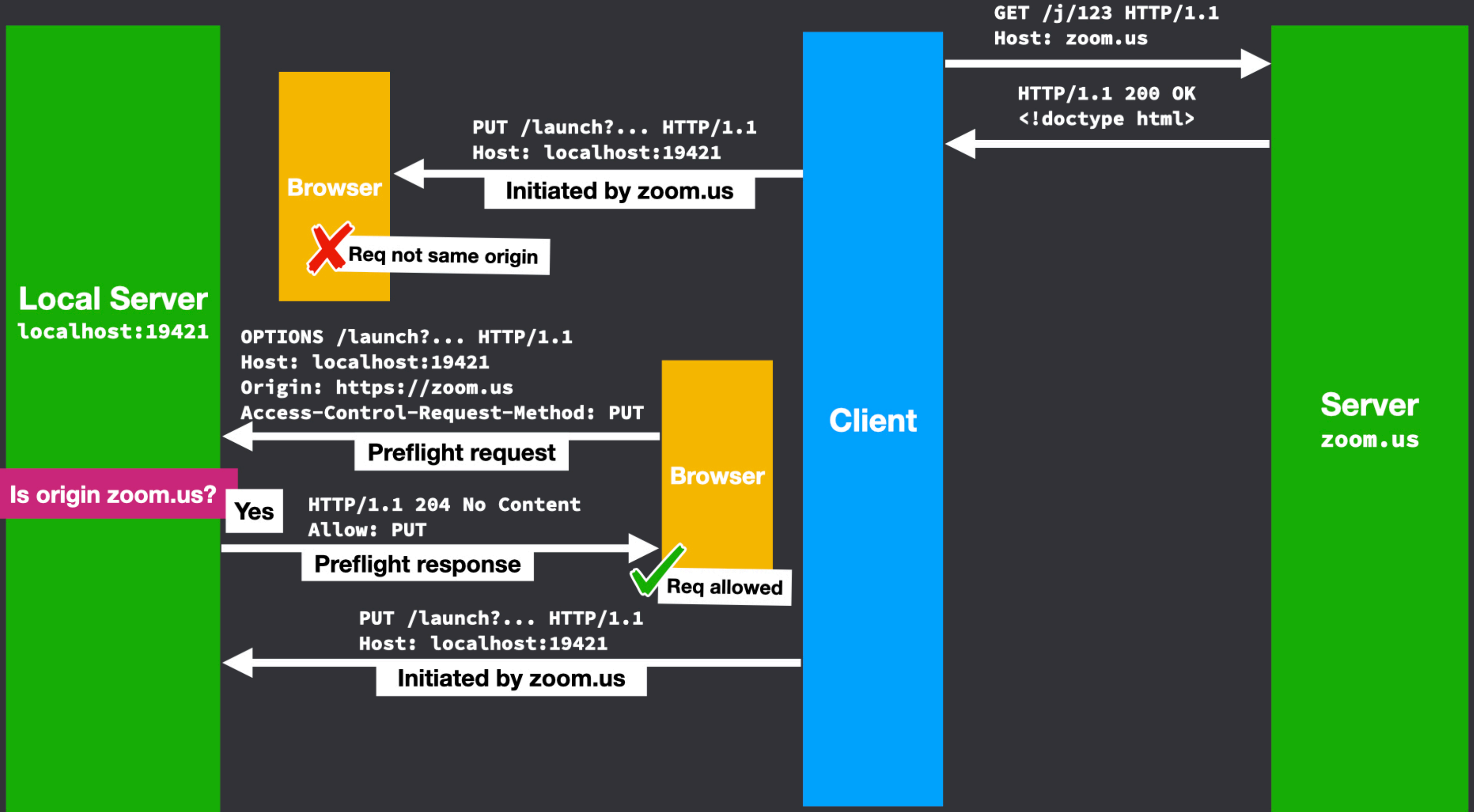


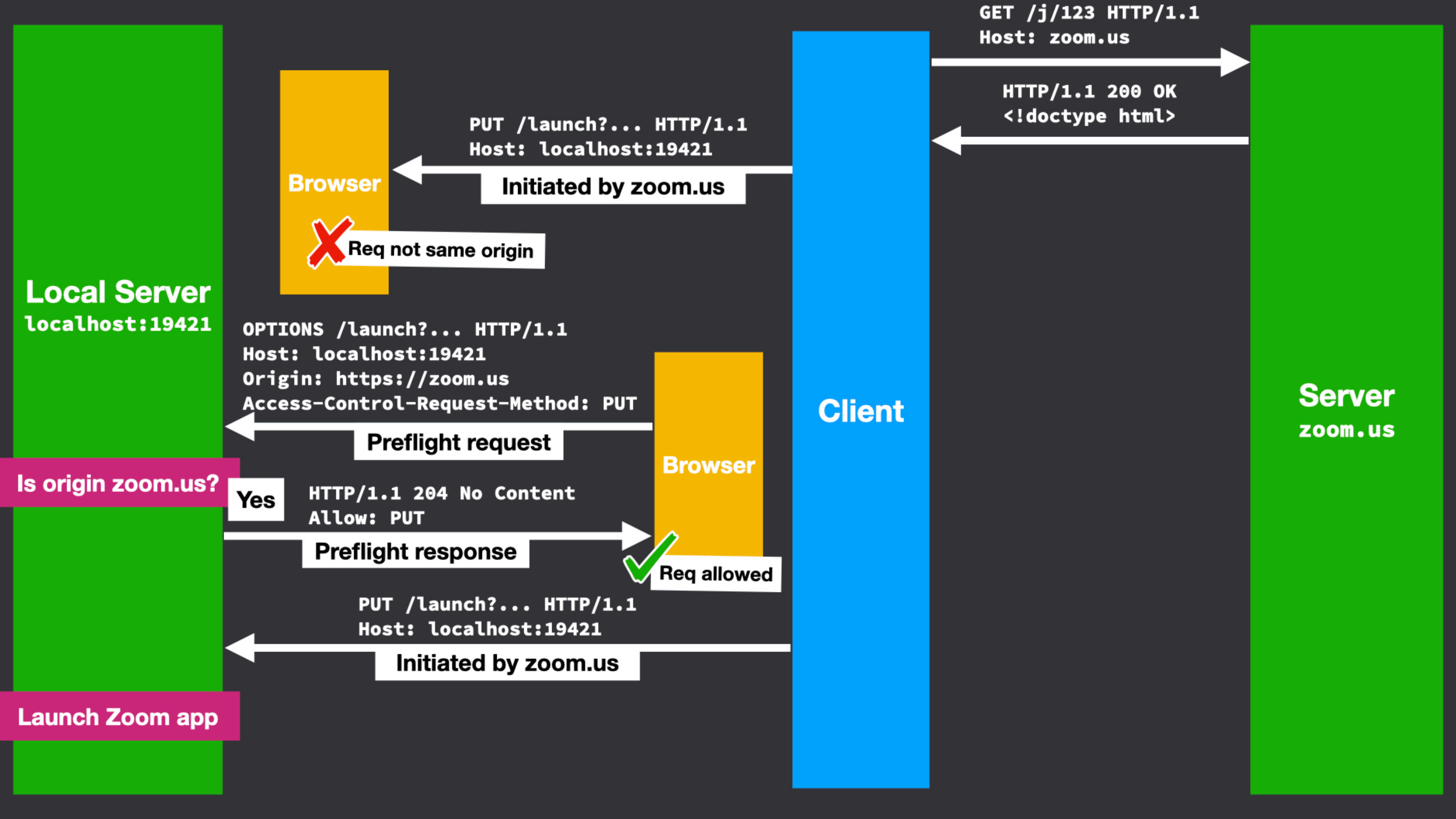


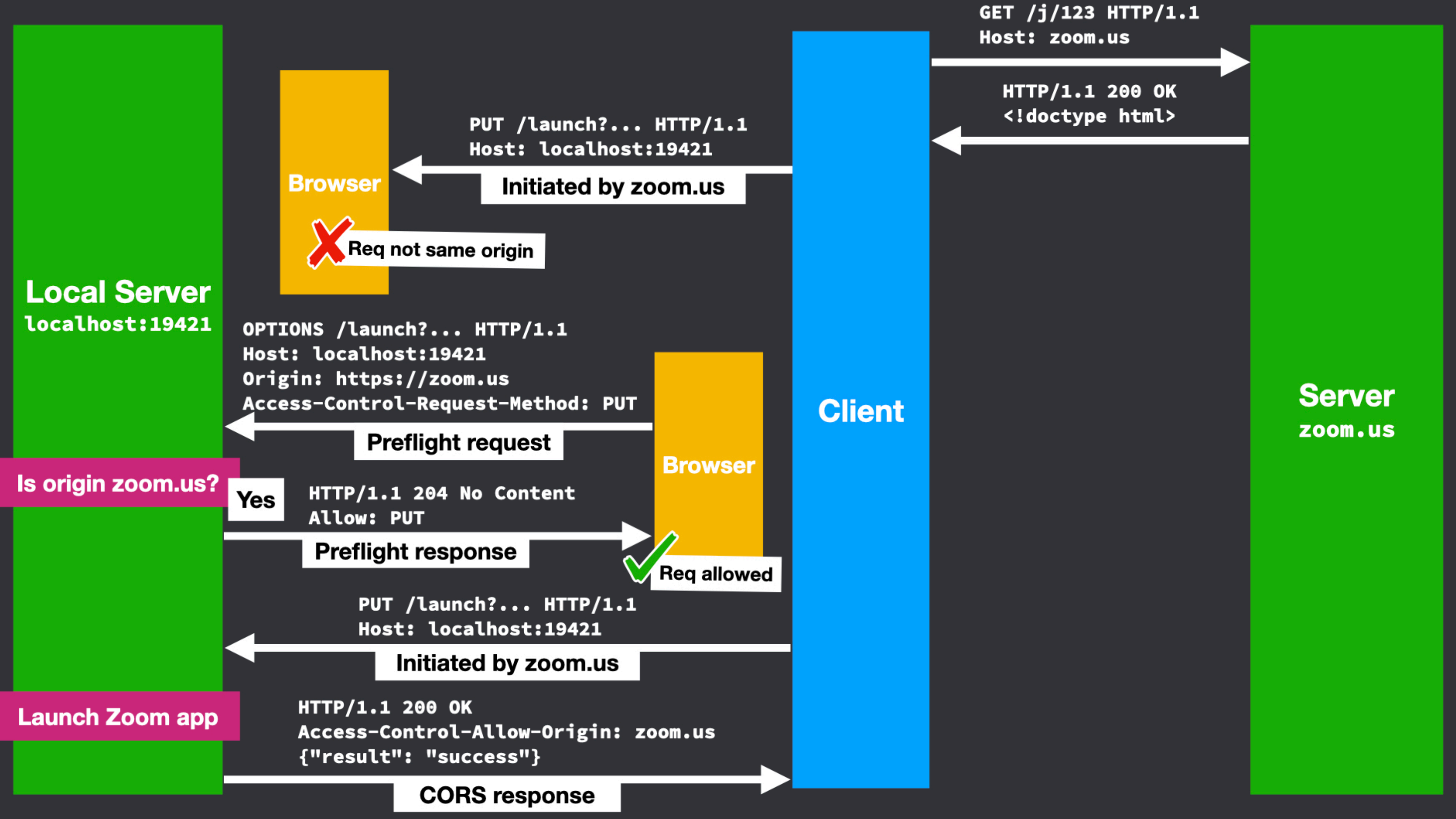


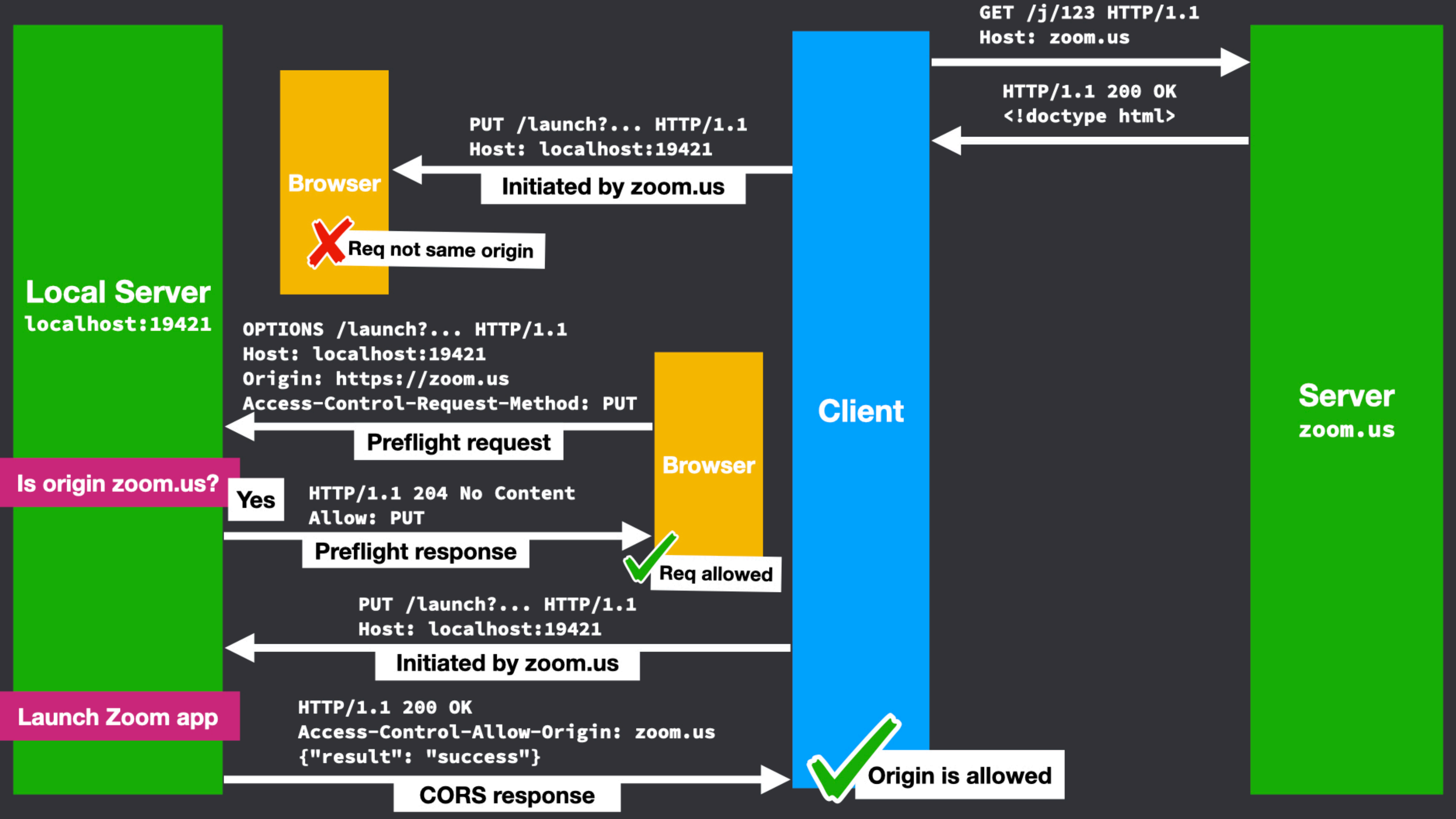










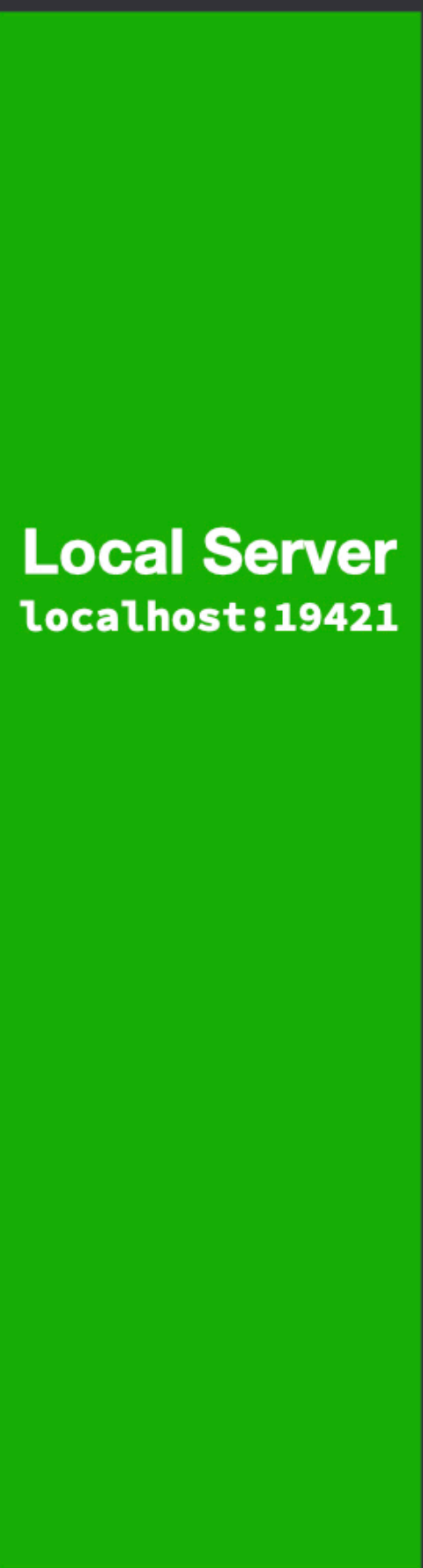


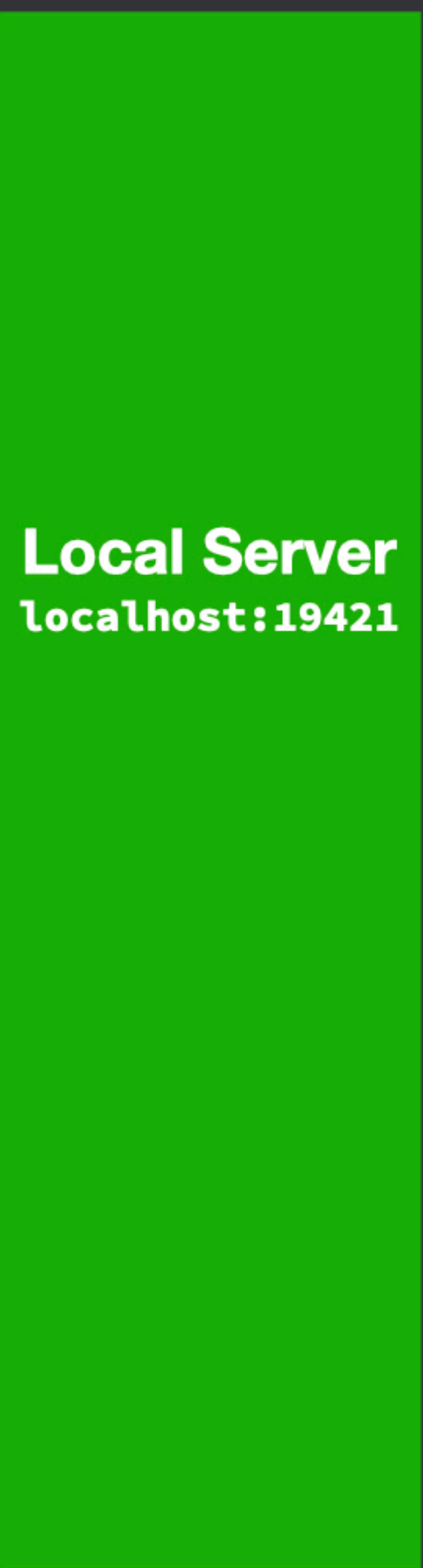
Attacker joins user into a zoom call (local server requires "preflighted" request)

Local Server
localhost:19421

Client

Server
attacker.com





GET / HTTP/1.1
Host: attacker.com



A white arrow pointing from the Client box to the Server box.



Local Server
localhost:19421

Client

Server
attacker.com

GET / HTTP/1.1
Host: attacker.com

HTTP/1.1 200 OK
<!doctype html>

Local Server
localhost:19421

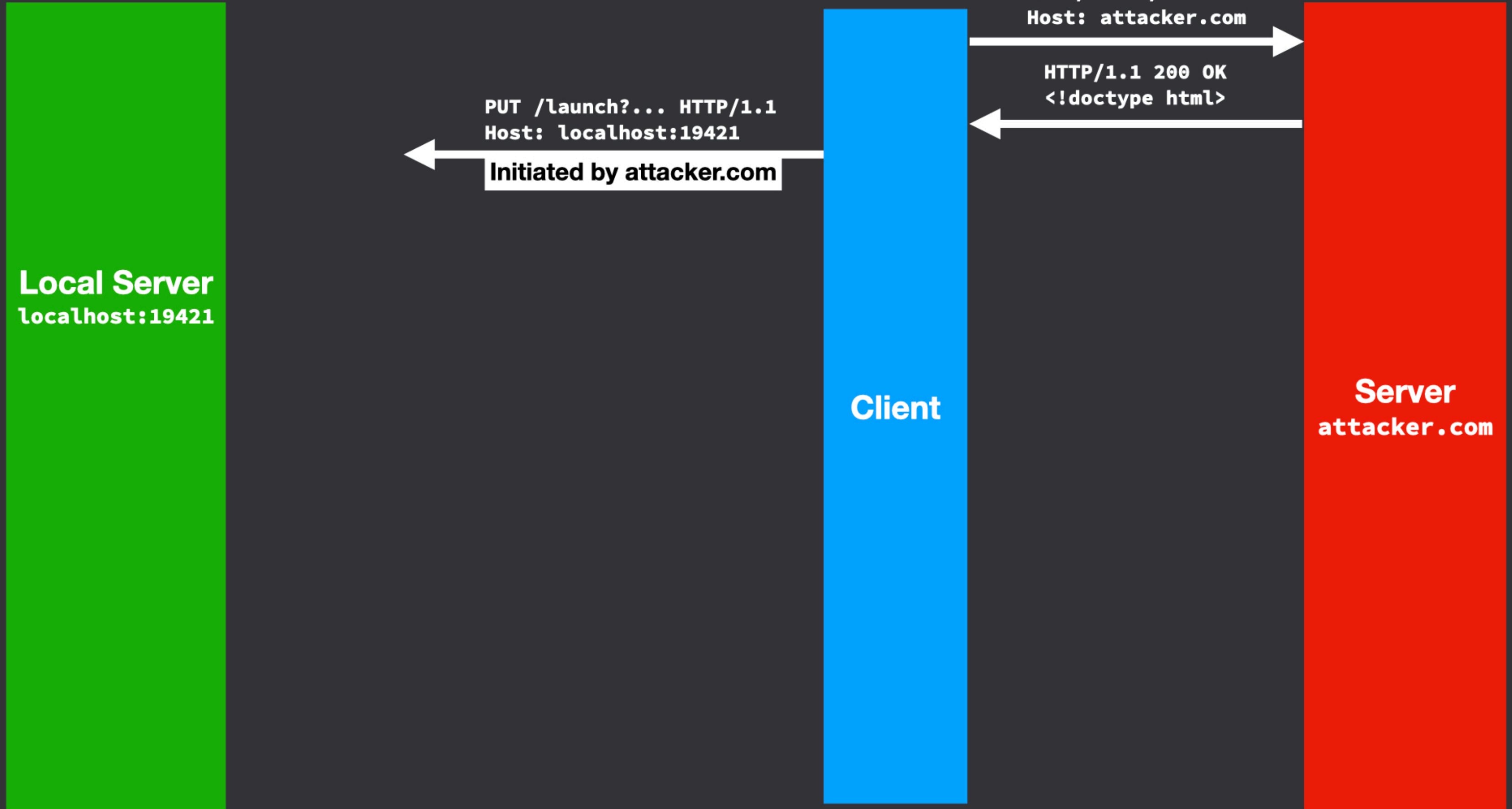
Client

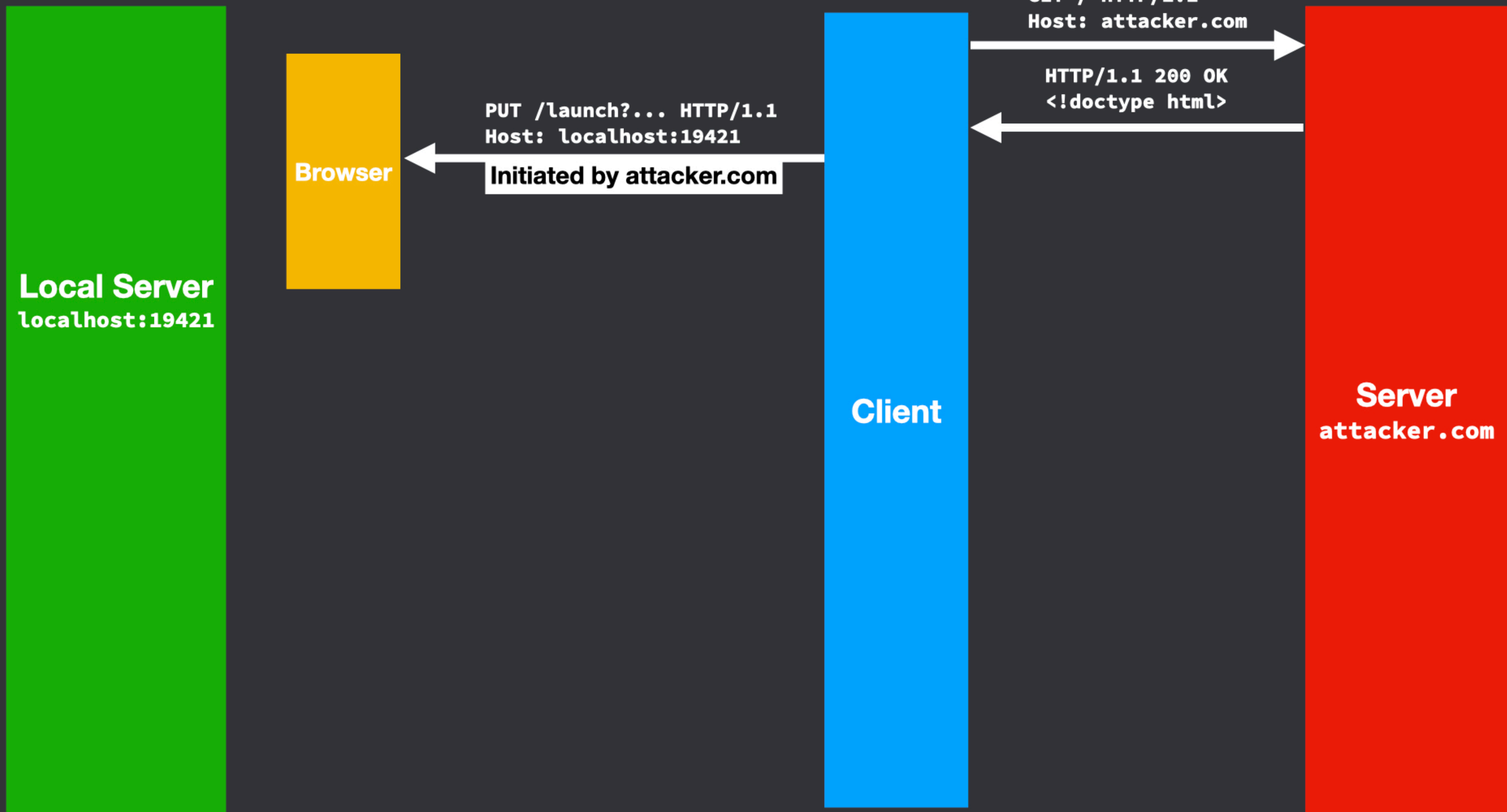
Server
attacker.com

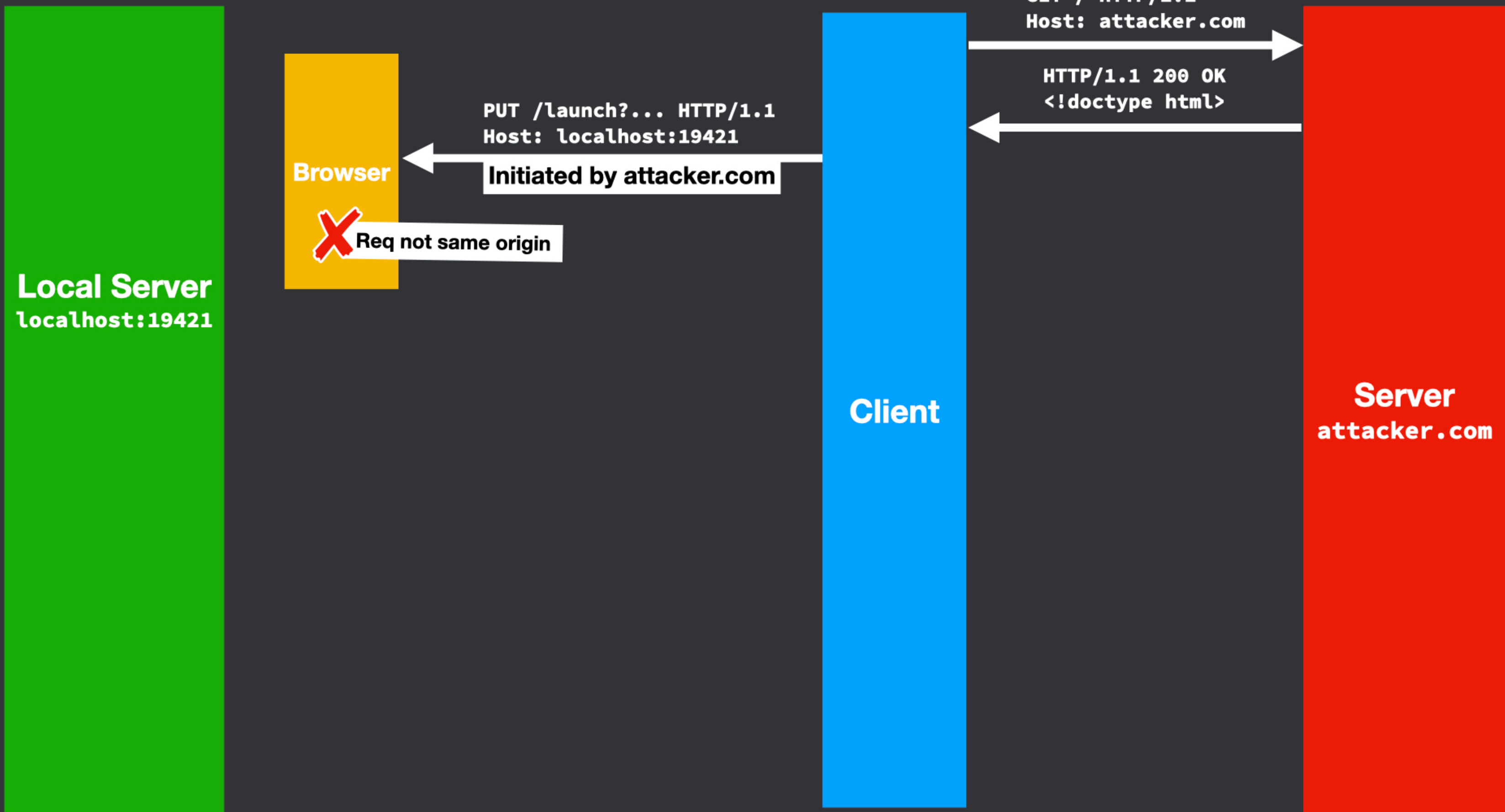
PUT /launch?... HTTP/1.1
Host: localhost:19421
Initiated by attacker.com

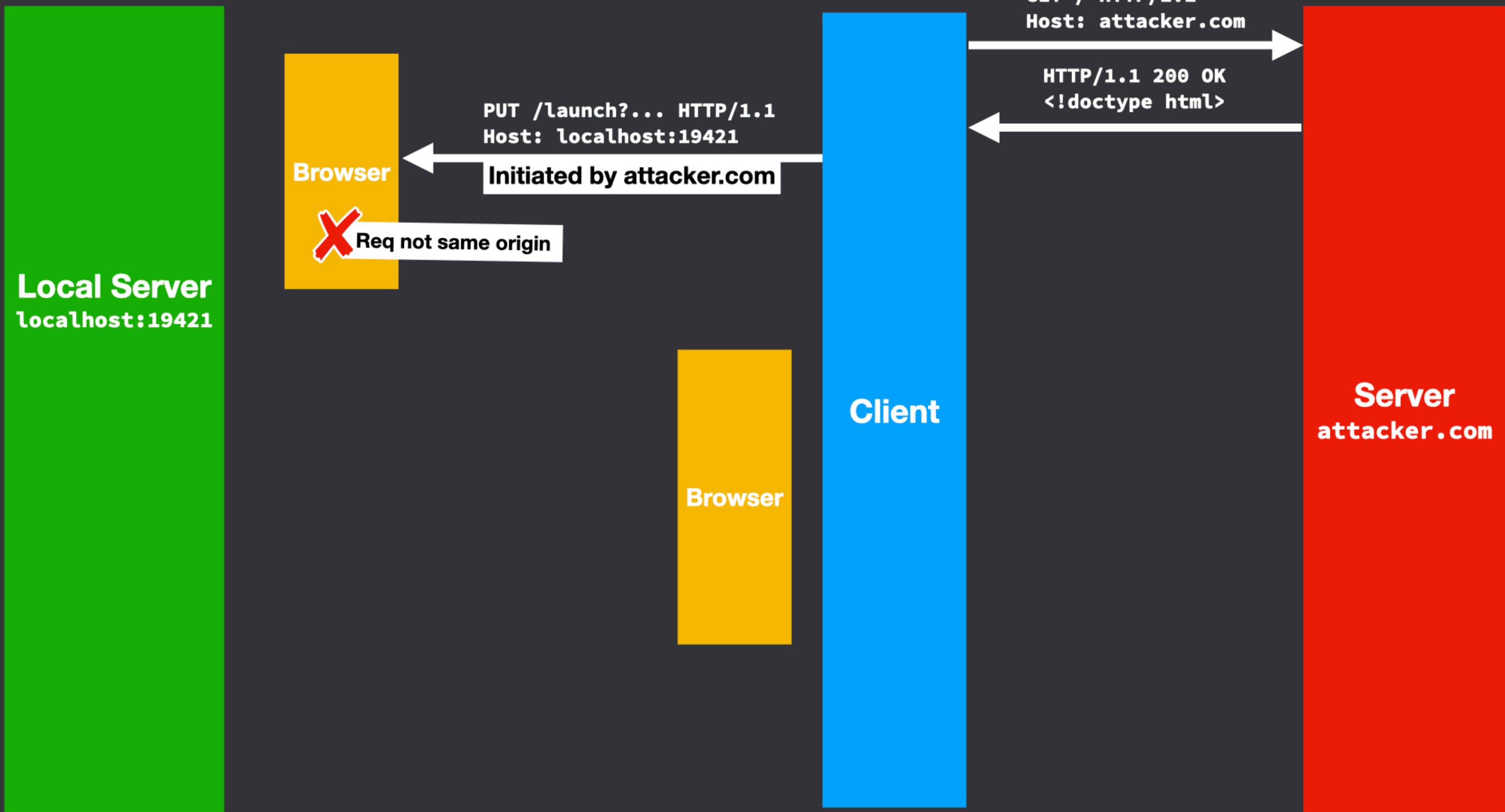
GET / HTTP/1.1
Host: attacker.com

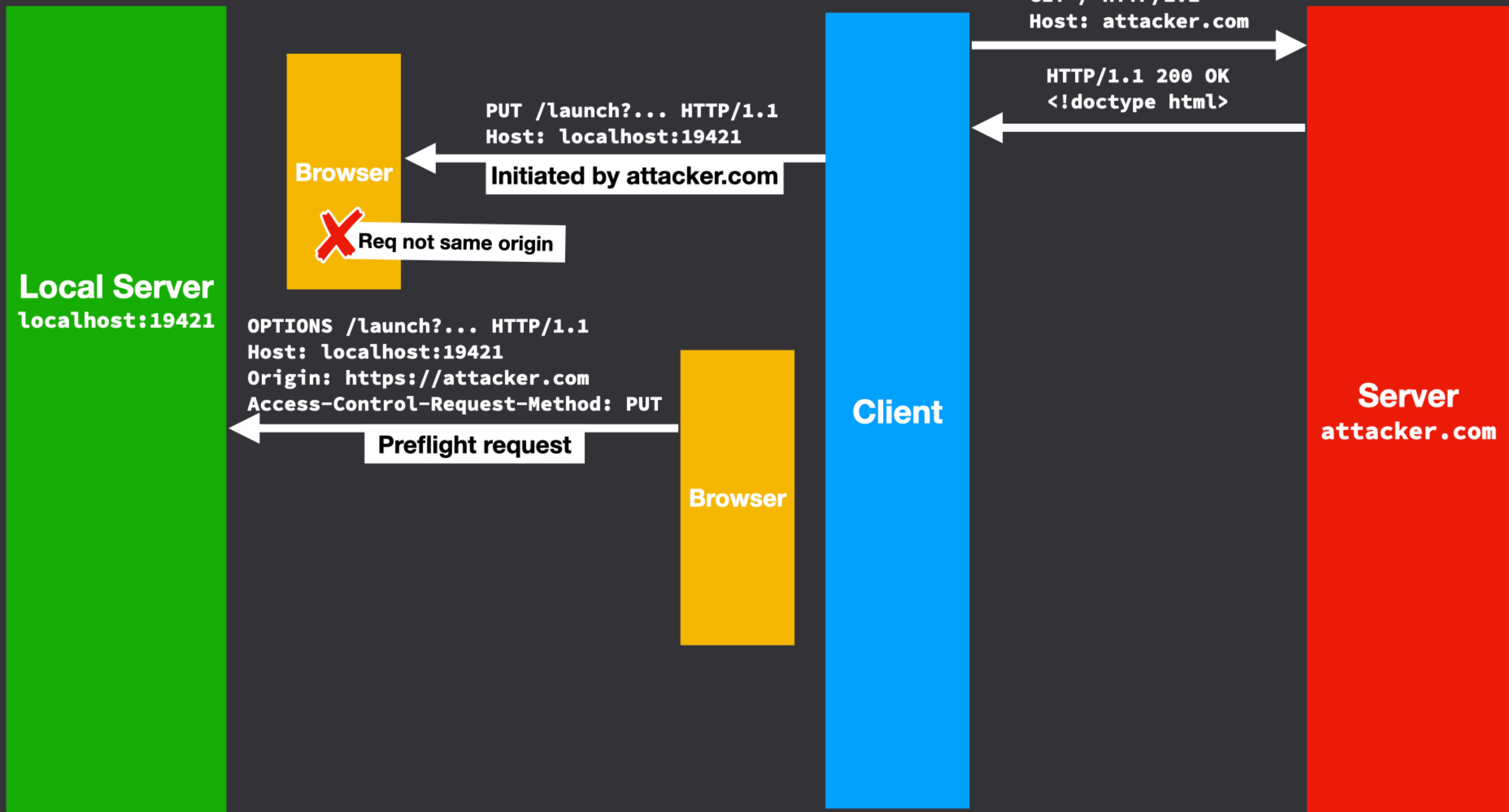
HTTP/1.1 200 OK
<!doctype html>

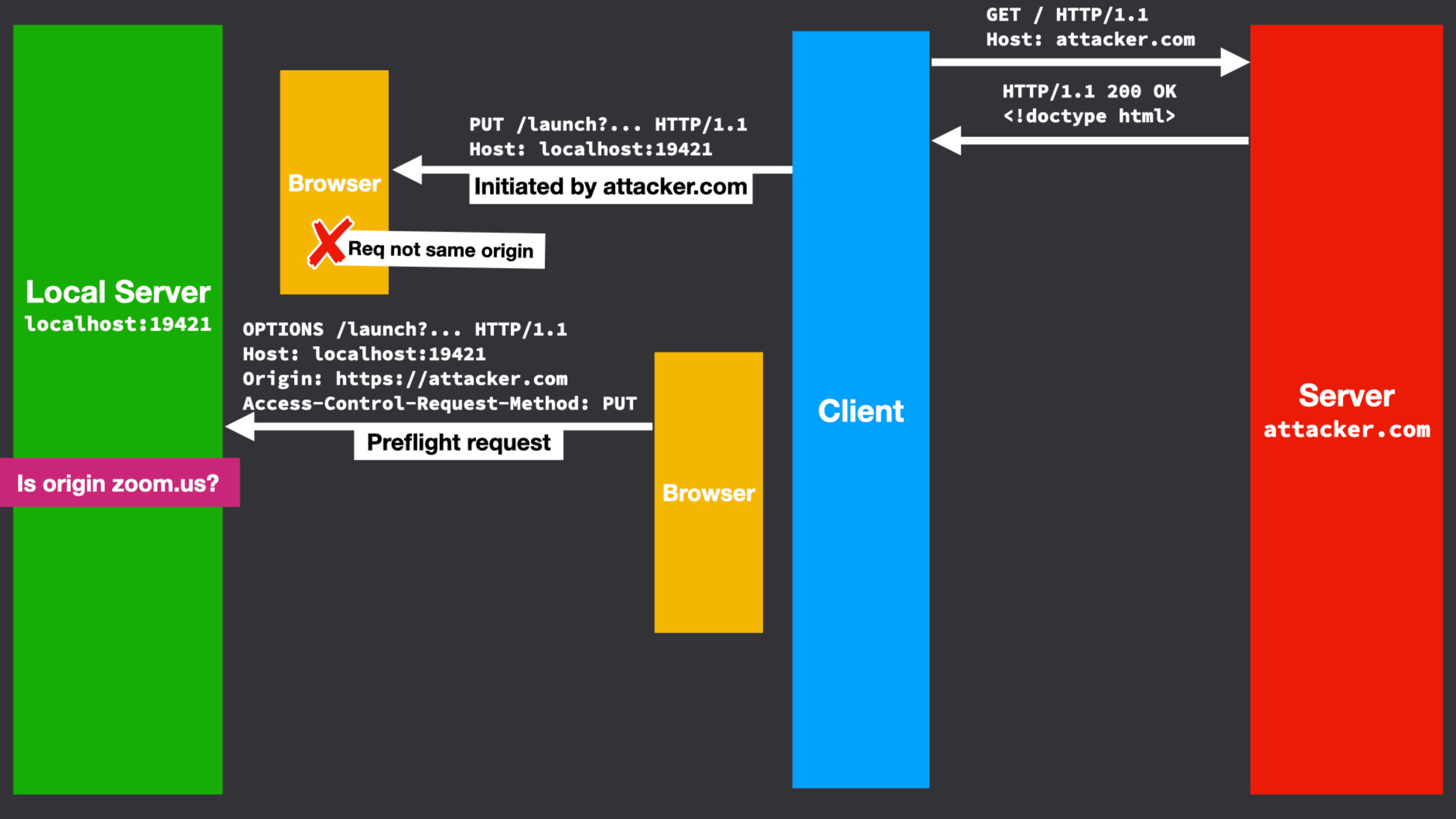


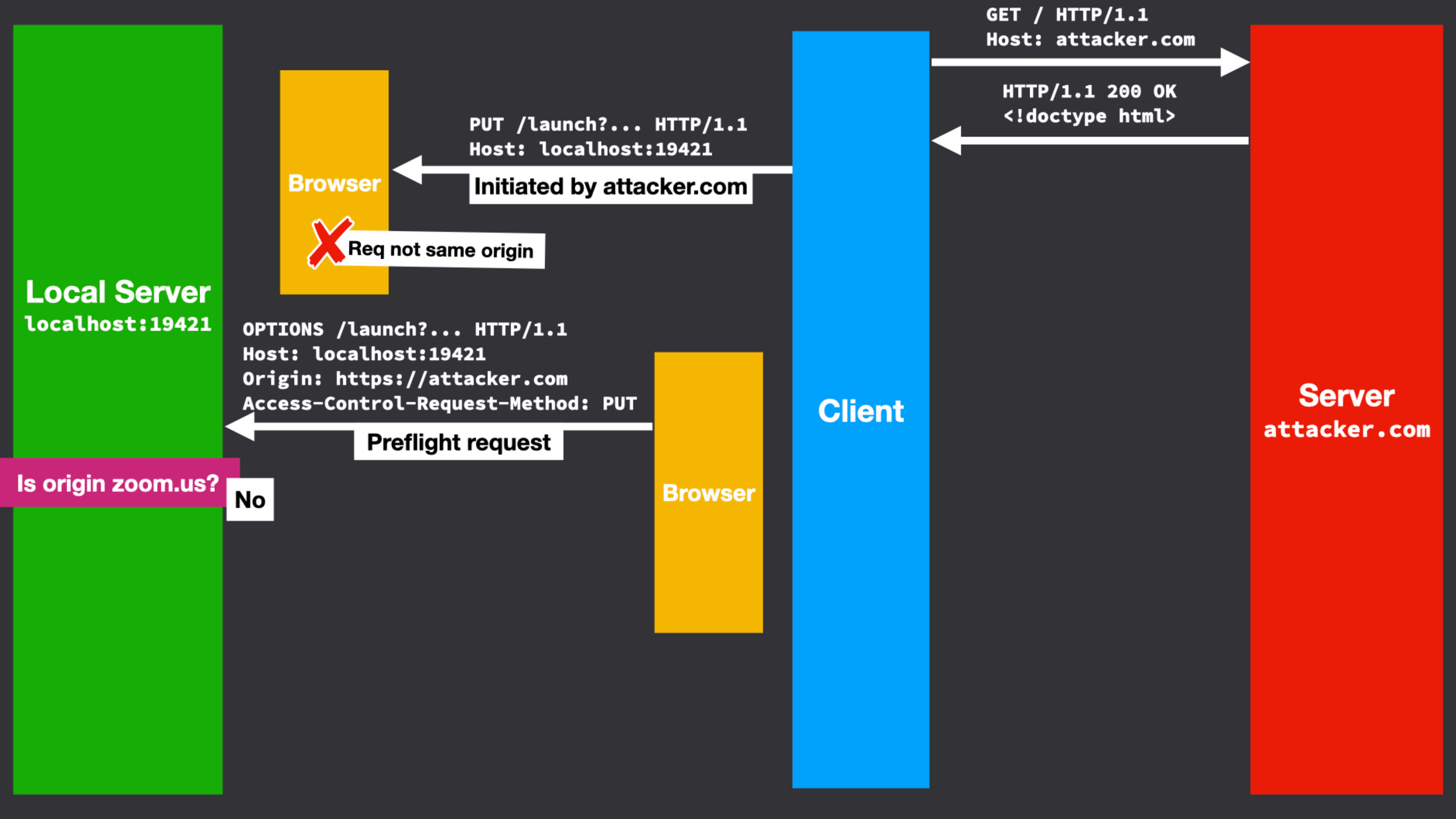


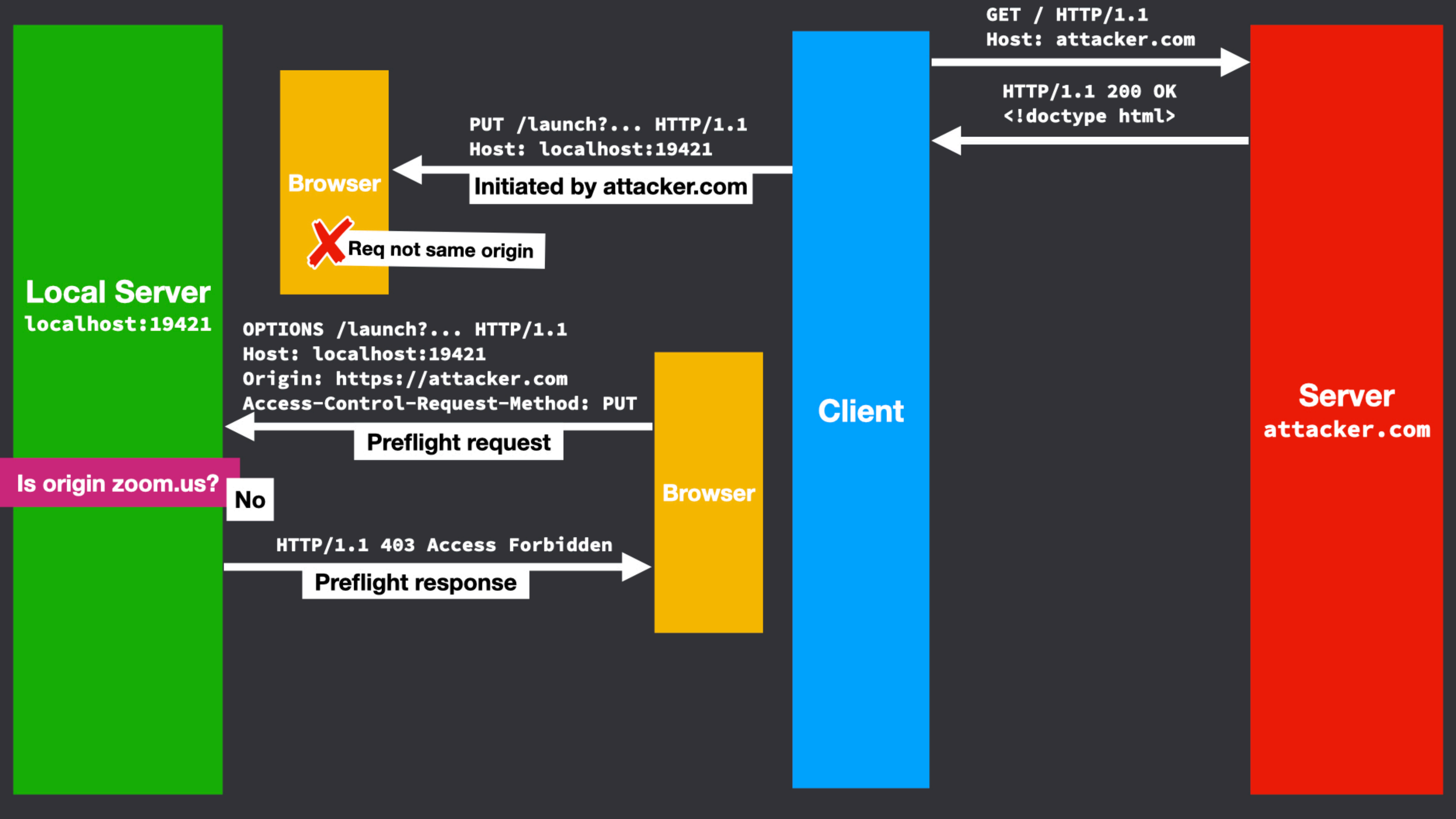


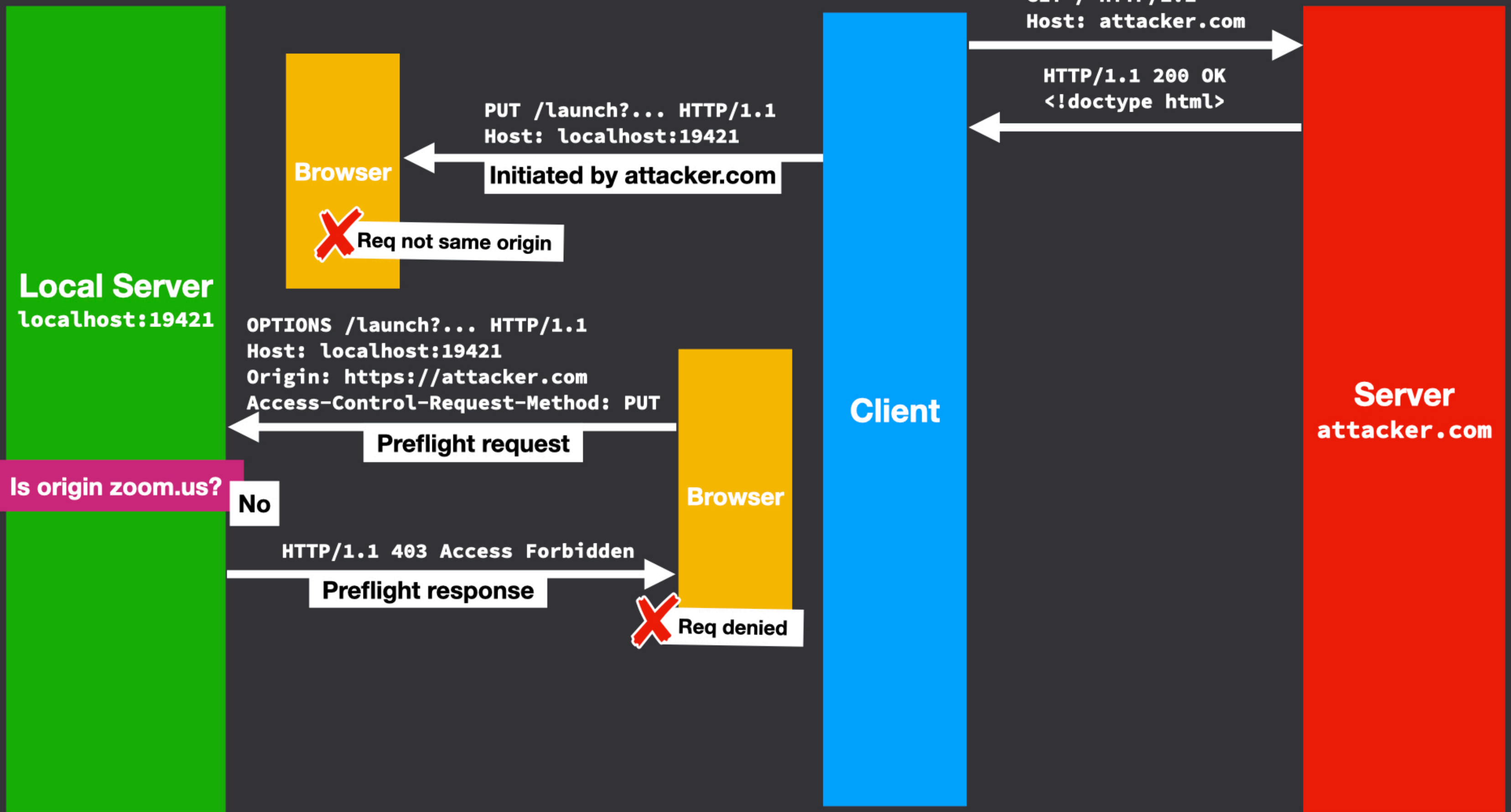












Who can still launch the app from the local server?

- Preflight requests allow the local server to prevent **PUT** requests from any origin except **zoom.us**
- But native apps running on the same device don't have to follow the browser's rules (i.e. send a preflight **OPTIONS** request, etc.)
 - The browser enforces that sites can't tamper with the **Origin** header, but a native app (e.g. a Node.js or Python script) can make a request and set the **Origin** header to **https://zoom.us**

Demo: Using the browser as a proxy to make requests to local servers

Demo: Using the browser as a proxy to make requests to local servers

```
victimApp.get('/', (req, res) => {  
  res.set('Access-Control-Allow-Origin', '*')  
  exec(COMMAND, err => {  
    if (err) res.status(500).send(err)  
    else res.status(200).send('Success')  
  })  
})
```

```
attackerApp.get('/', (req, res) => {  
  res.send(`  
    <!doctype html>  
    <html>  
      <body>  
        <h3>Welcome to attacker.com!</h3>  
        <img src='http://localhost:8080' />  
      </body>  
    </html>  
  `)  
})
```

```
victimApp.get('/', (req, res) => {
  res.set('Access-Control-Allow-Origin', '*')
  exec(COMMAND, err => {
    if (err) res.status(500).send(err)
    else res.status(200).send('Success')
  })
})

attackerApp.get('/', (req, res) => {
  res.send(`
    <!doctype html>
    <html>
      <body>
        <h3>Welcome to attacker.com!</h3>
        <script>
          fetch('http://localhost:8080')
            .then(res => res.text())
            .then(text => document.body.innerHTML += '<br />' + text)
            .catch(err => document.body.innerHTML += '<br />' + err)
        </script>
      </body>
    </html>
  `)
})
```

```
victimApp.get('/', (req, res) => {
  exec(COMMAND, err => {
    if (err) res.status(500).send(err)
    else res.status(200).send('Success')
  })
})

attackerApp.get('/', (req, res) => {
  res.send(`
    <!doctype html>
    <html>
      <body>
        <h3>Welcome to attacker.com!</h3>
        <script>
          fetch('http://localhost:8080') // RESPONSE IS OPAQUE
            .then(res => res.text())
            .then(text => document.body.innerHTML += '<br />' + text)
            .catch(err => document.body.innerHTML += '<br />' + err)
        </script>
      </body>
    </html>
  `)
})
```

```

victimApp.put('/', (req, res) => {
  exec(COMMAND, err => {
    if (err) res.status(500).send(err)
    else res.status(200).send('Success')
  })
})

attackerApp.get('/', (req, res) => {
  res.send(`
    <!doctype html>
    <html>
      <body>
        <h3>Welcome to attacker.com!</h3>
        <script>
          fetch('http://localhost:8080', { method: 'PUT' }) // WILL FAIL
            .then(res => res.text())
            .then(text => document.body.innerHTML += '<br />' + text)
            .catch(err => document.body.innerHTML += '<br />' + err)
        </script>
      </body>
    </html>
  `)
})

```


Enterprise firewalls



Internet

The diagram consists of a dark gray background. In the upper right, there is a light gray cloud shape containing the word 'Internet'. In the lower right, there is a light gray square shape containing a browser icon and the word 'Browser'. The browser icon is a simple rectangle with a smaller rectangle inside, representing the address bar and content area, with three small circles in the top right corner representing window controls.

Browser





Internet



Browser













Unsolicited incoming connection





Unsolicited incoming connection







Browser



Internet

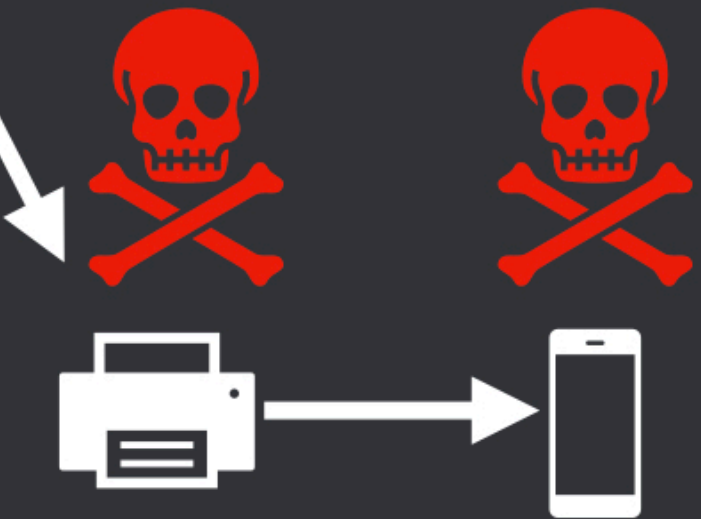






Internet

Browser

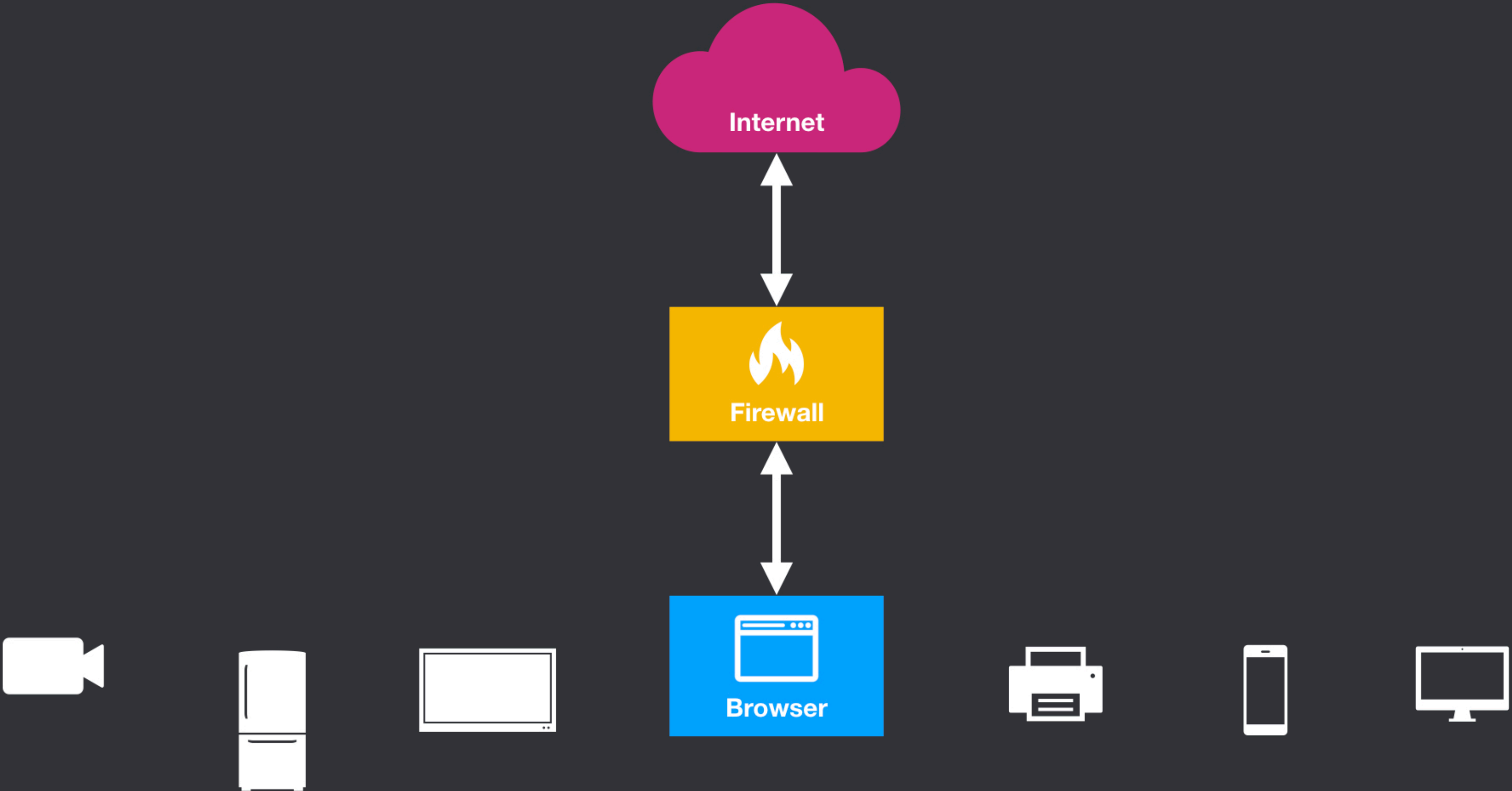


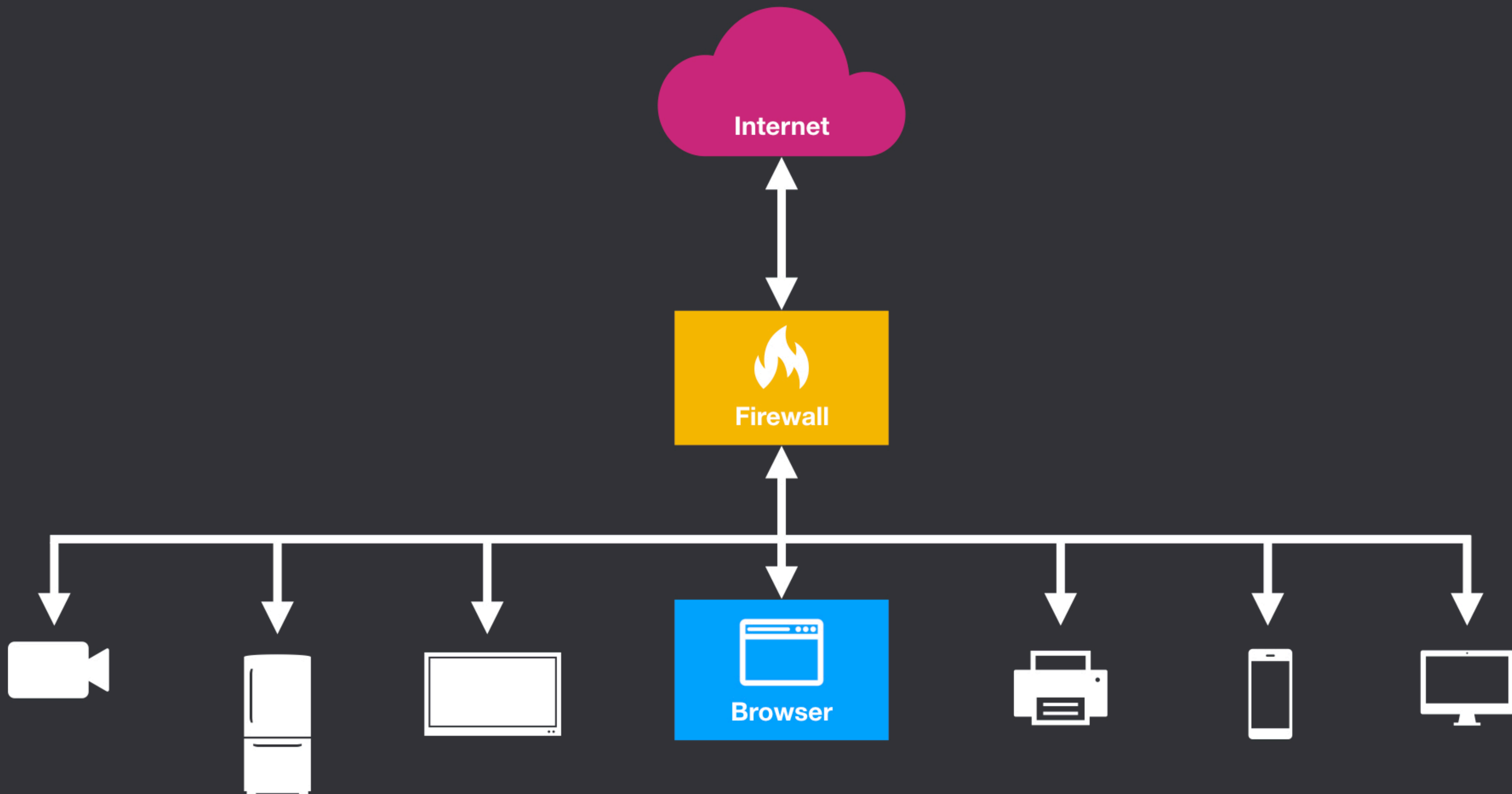


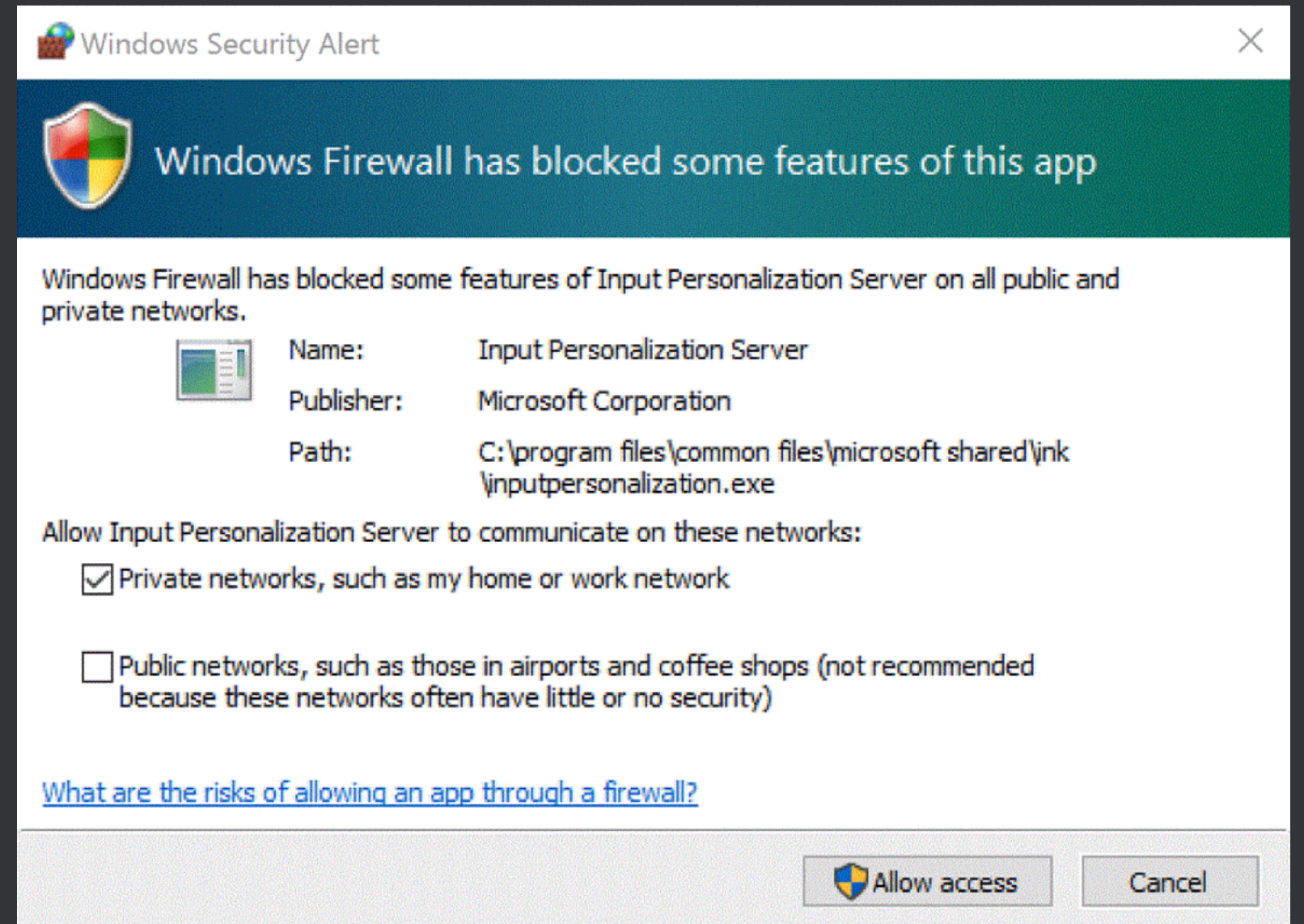
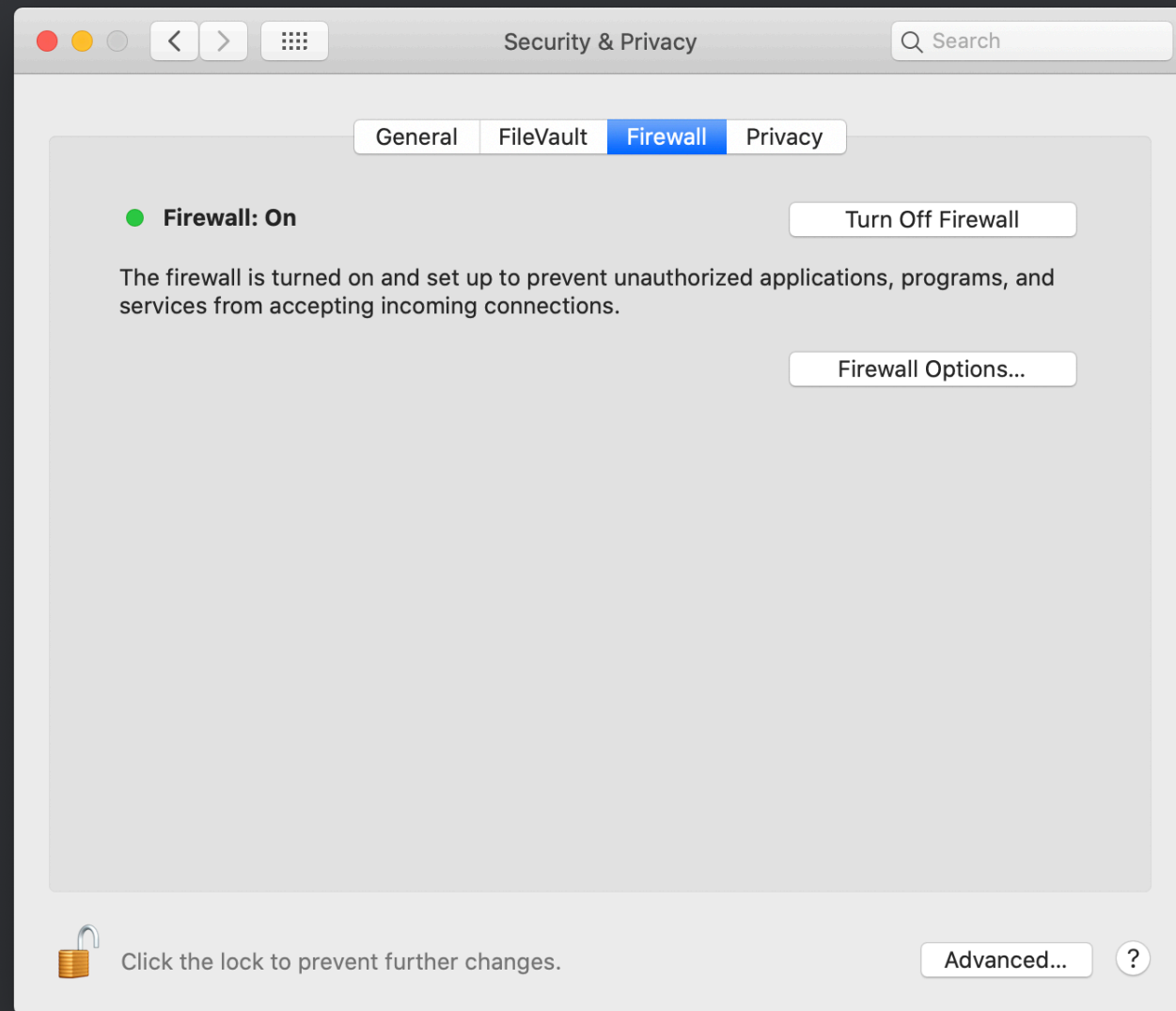




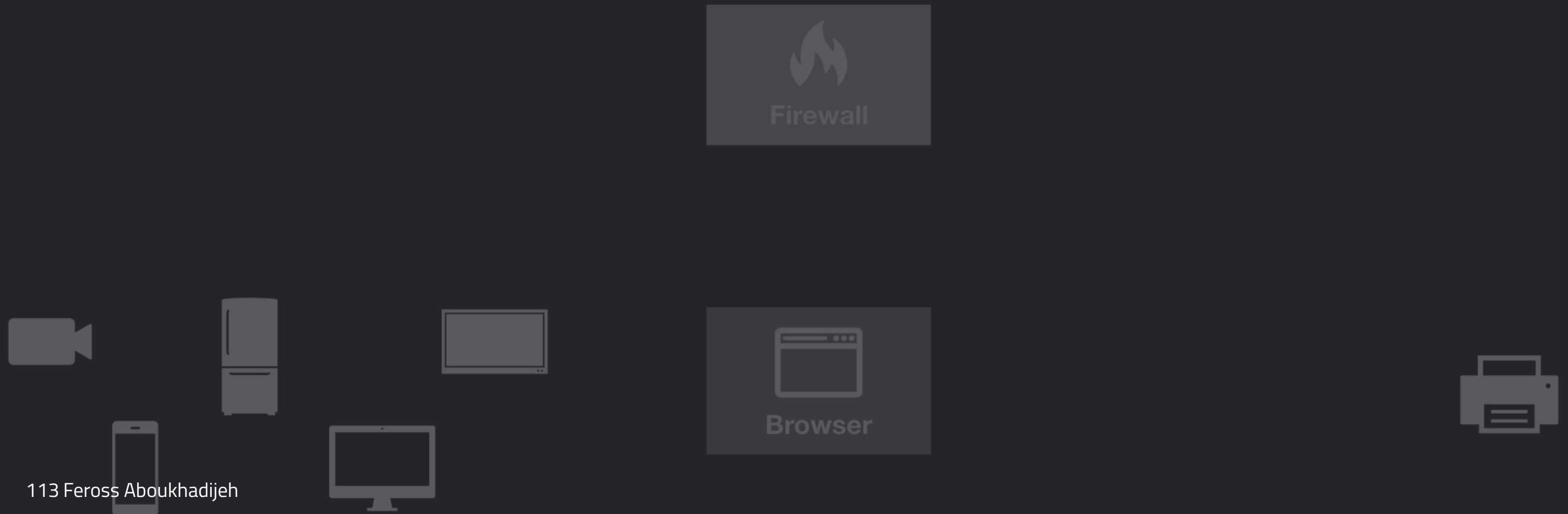






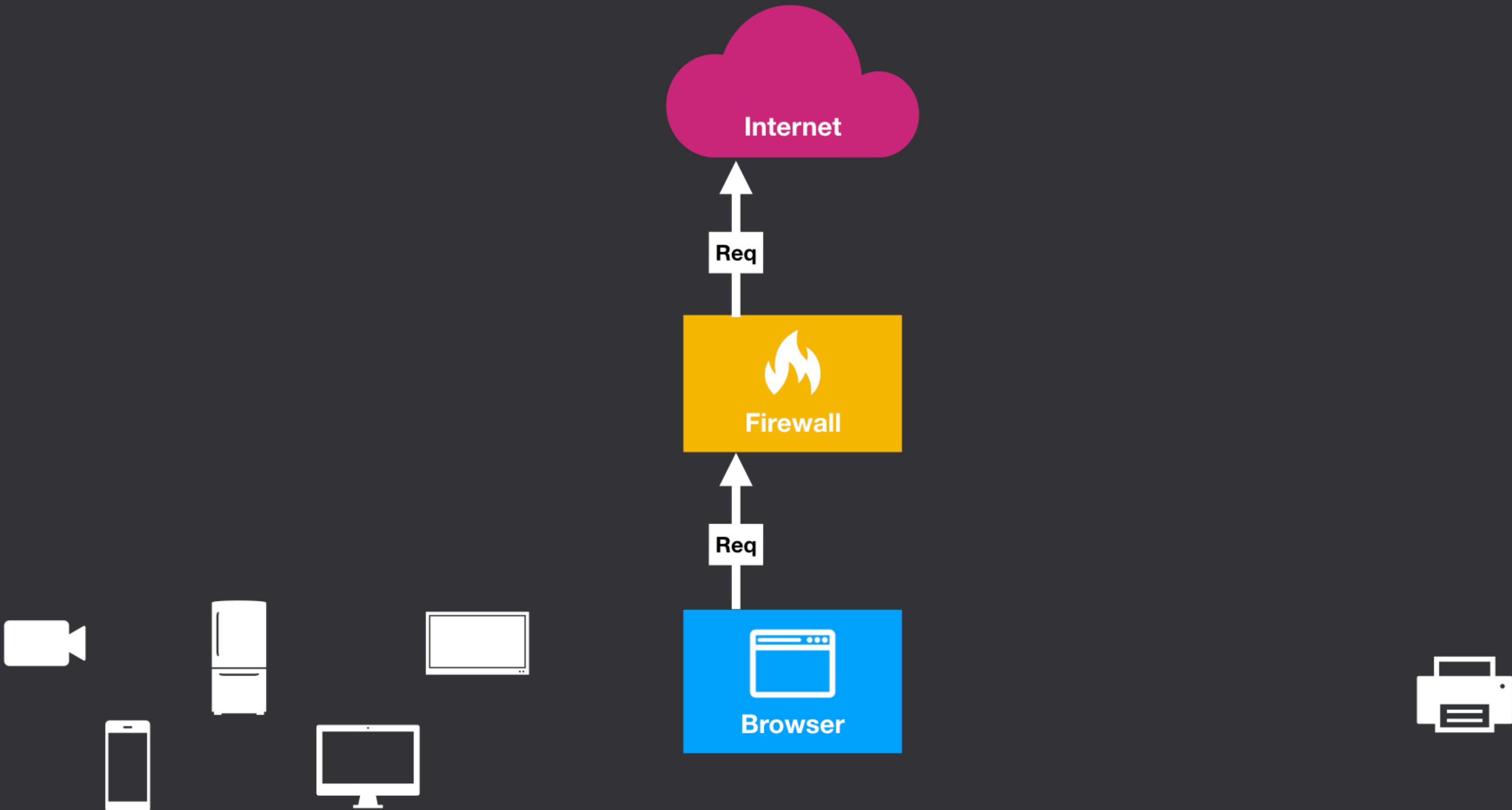


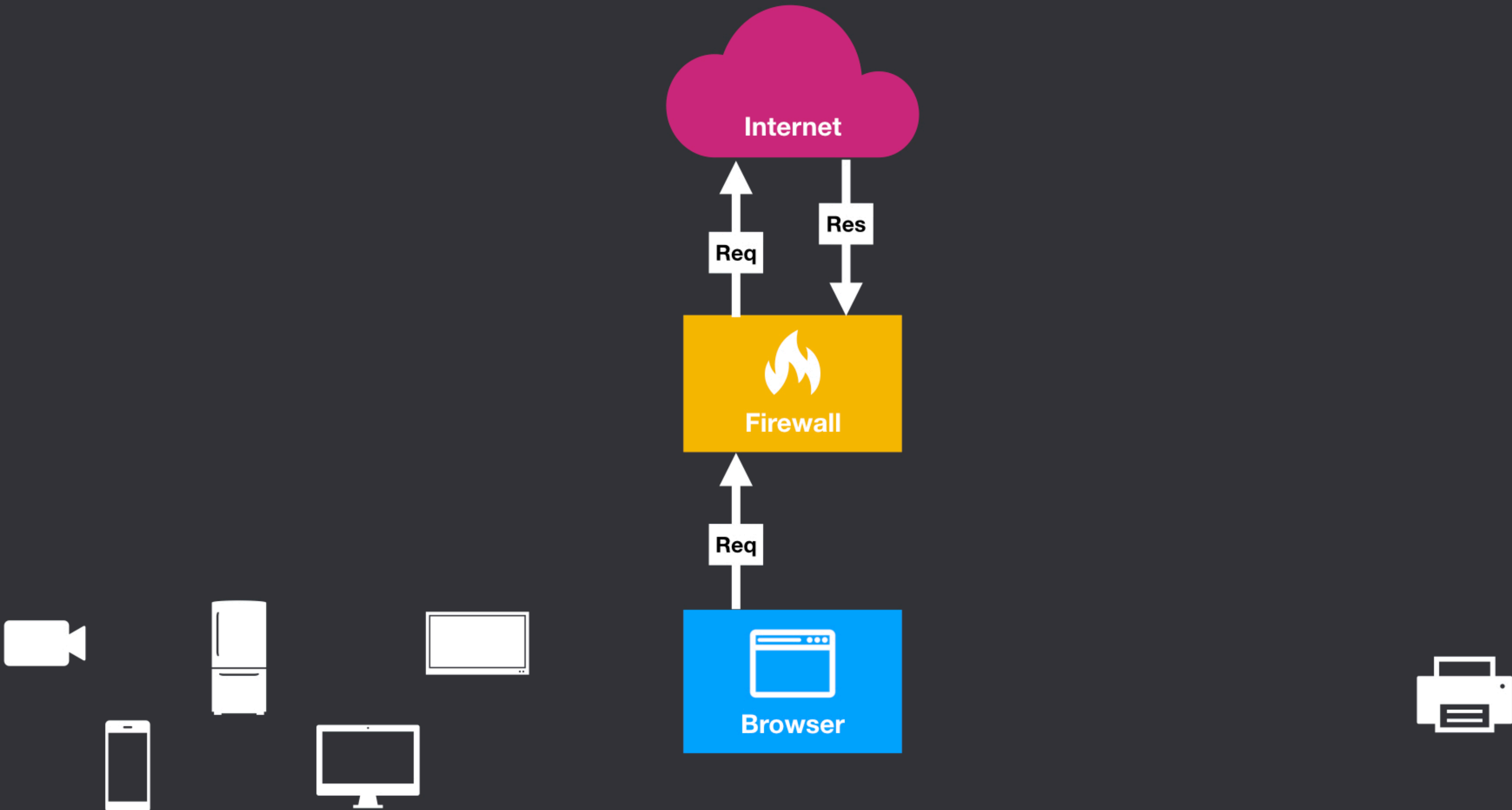
Using the browser as a proxy to make requests to the local network

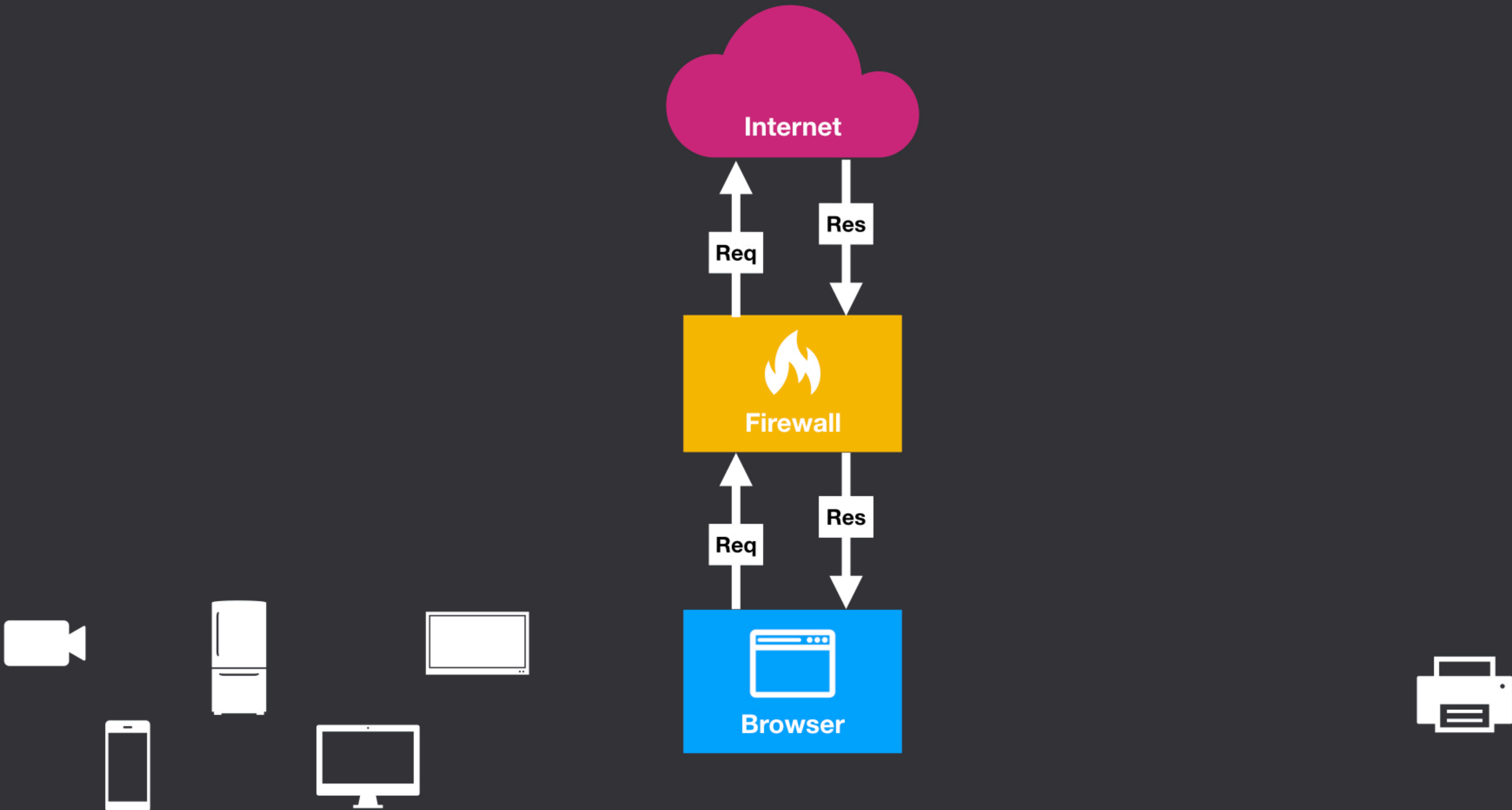


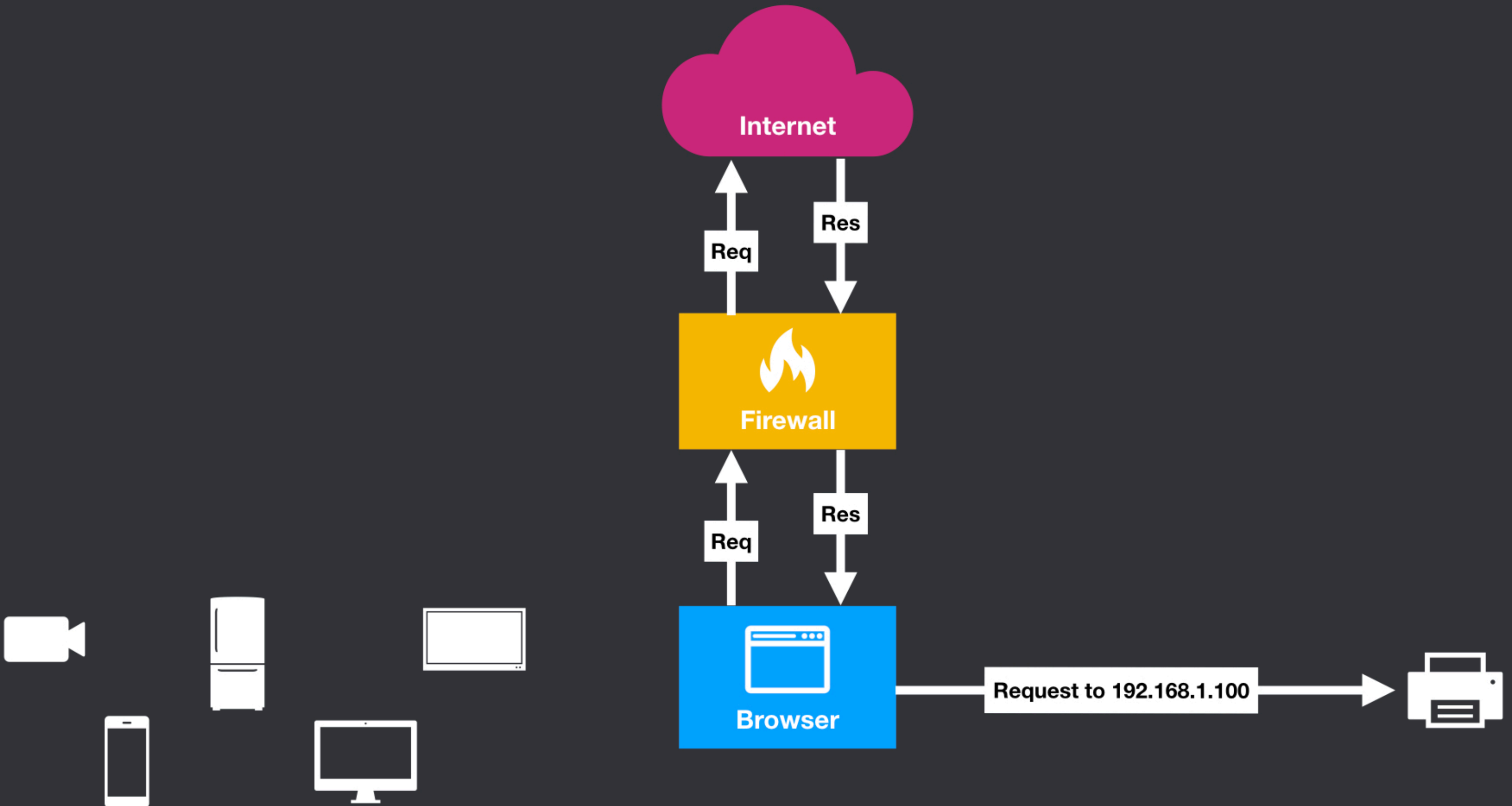


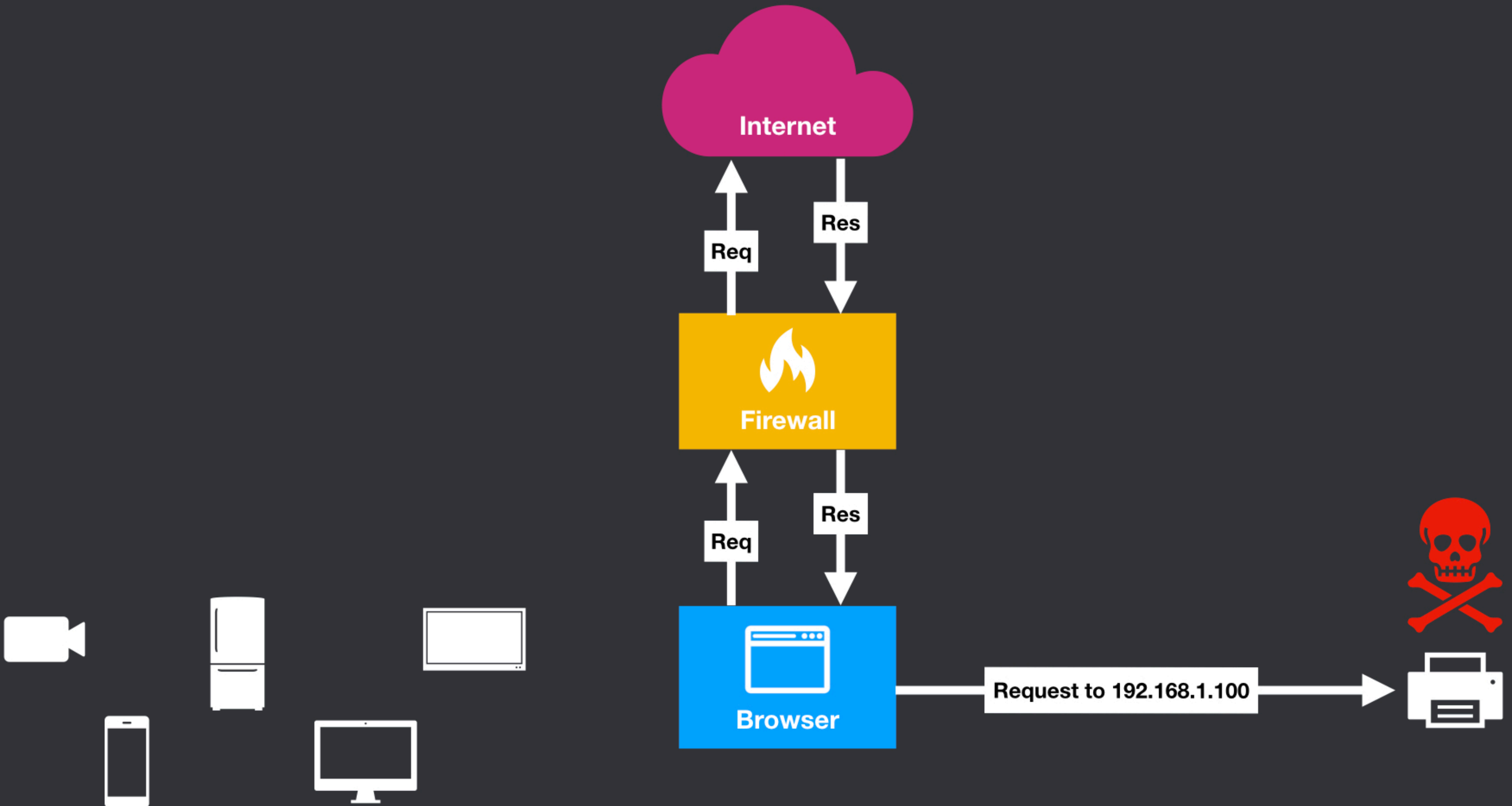




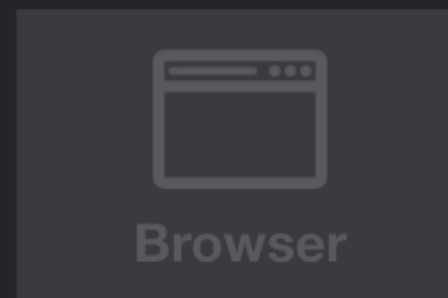
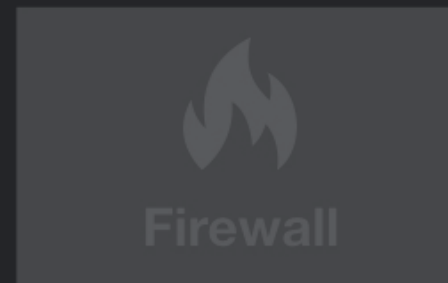






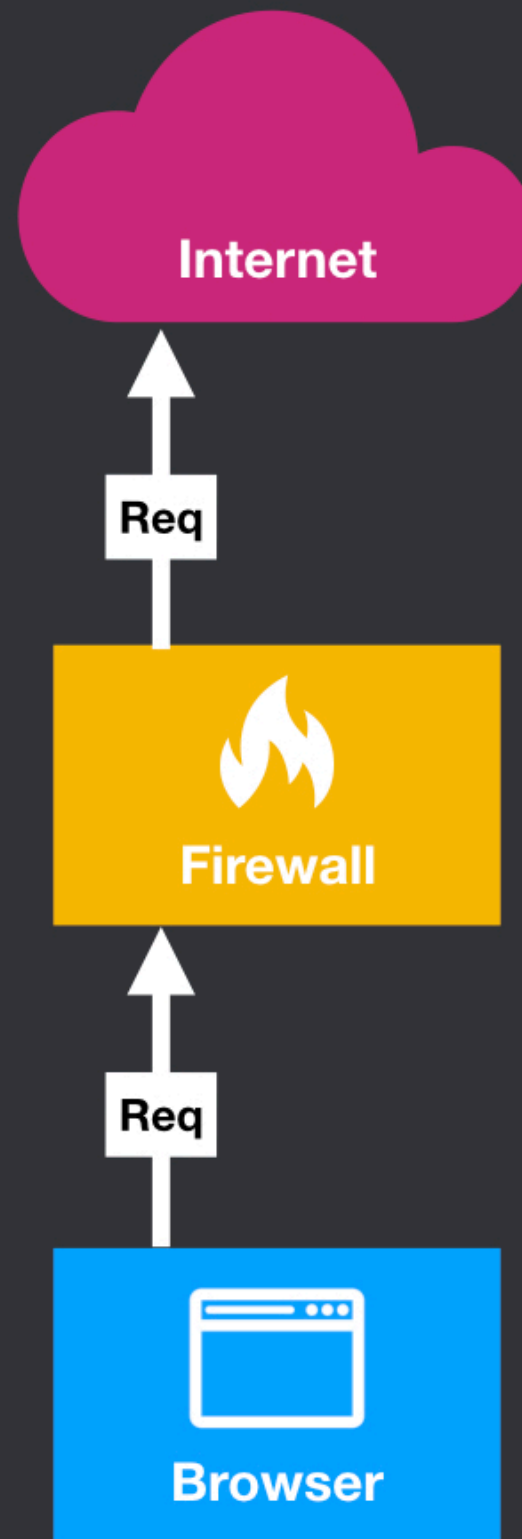


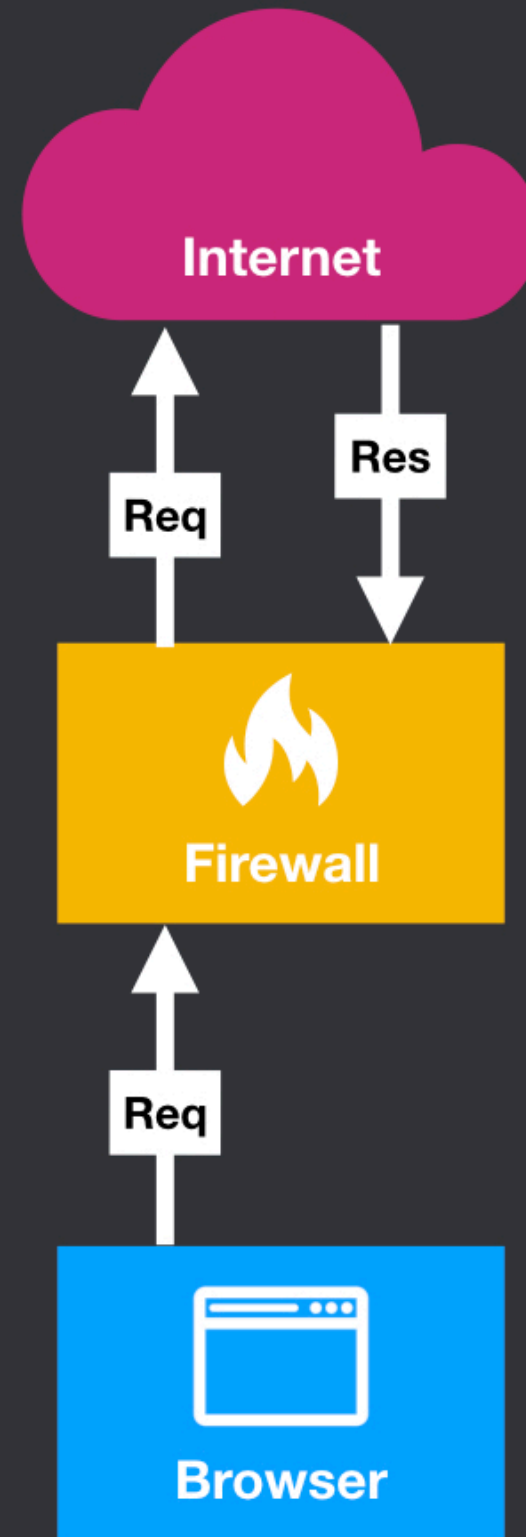
Using the browser as a proxy to make requests to local HTTP servers

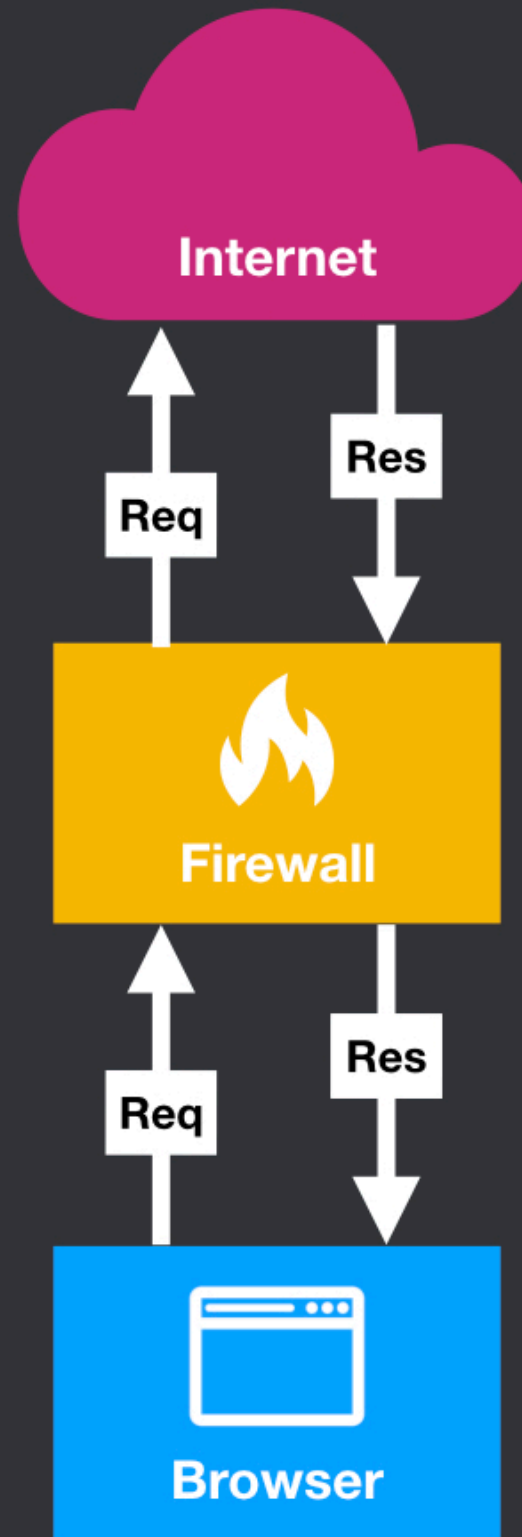


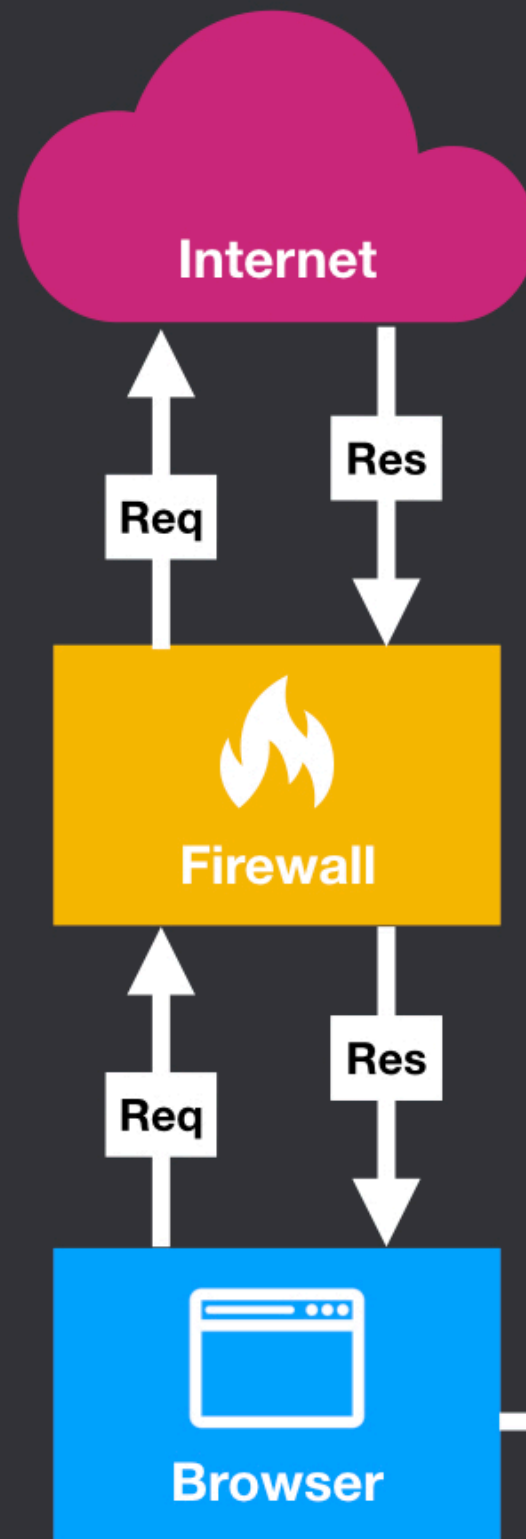






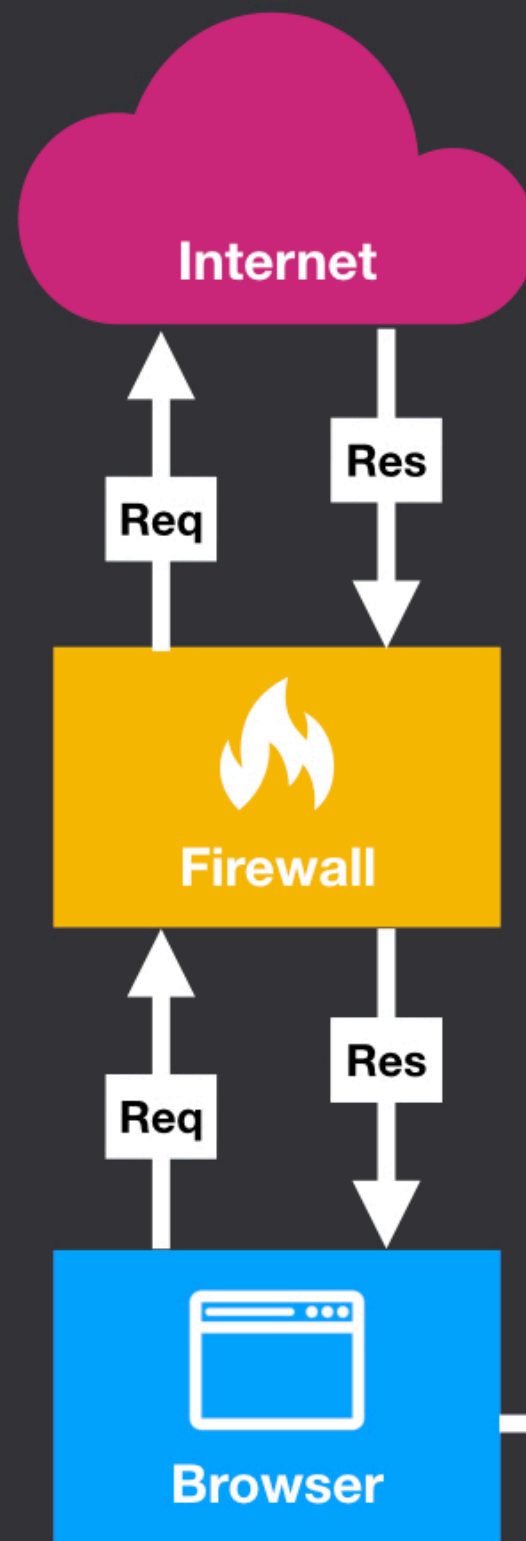






Request to 192.168.1.100





Request to 192.168.1.100



One more thing...

- Introducing the **DNS rebinding attack** ✨
 - **Idea:** trick the browser into thinking the request doesn't need to be preflighted (i.e. can skip the **OPTIONS** request)
- This means **any origin** can *STILL* send requests to our local HTTP server! 😡

DNS rebinding attack

- **DNS rebinding** allows a remote attacker to:
 - Bypass cross-origin resource sharing (CORS) rules
 - Bypass a victim's network firewall and use their web browser as a proxy to communicate directly with vulnerable servers on the local network
- **DNS rebinding** exploits limitations in the same origin policy:
 - Origin is just defined as protocol + hostname + port
 - The actual IP address that the hostname resolves to is not included

DNS rebinding attack

- Very common attack
- Lots of instances of it happening in the news
- We'll cover an example where real-world code I wrote was vulnerable to it

Demo: DNS rebinding attack

Demo: DNS rebinding attack

```
victimApp.put('/', (req, res) => {
  exec(COMMAND, err => {
    if (err) res.status(500).send(err)
    else res.status(200).send('Success')
  })
})

attackerApp.get('/', (req, res) => {
  res.send(`
    <!doctype html>
    <html>
      <body>
        <h3>Welcome to attacker.com!</h3>
        <button>Send PUT request to http://attacker.com:8080</button>
        <script>
          document.querySelector('button').addEventListener('click', () => {
            fetch('http://attacker.com:8080', { method: 'PUT' })
              .then(res => res.text())
              .then(text => document.body.innerHTML += '<br />' + text)
              .catch(err => document.body.innerHTML += '<br />' + err)
          })
        </script>
      </body>
    </html>
  `)
})
```

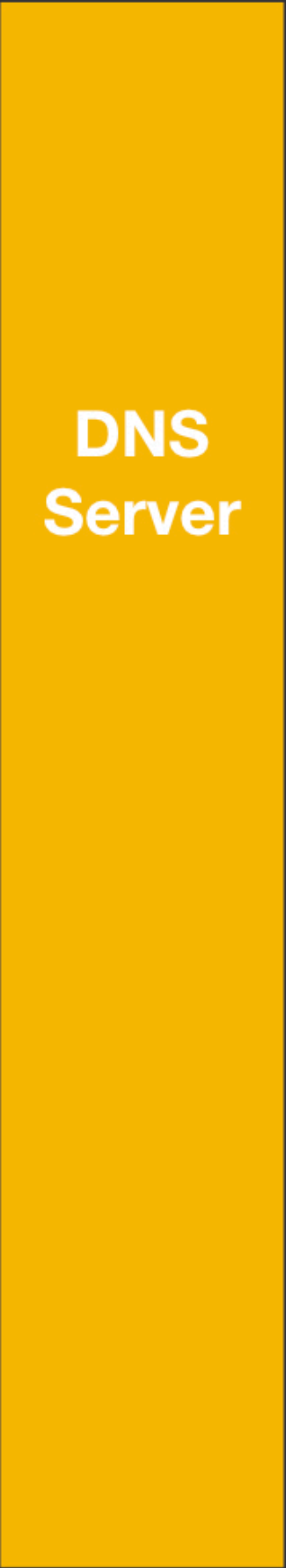
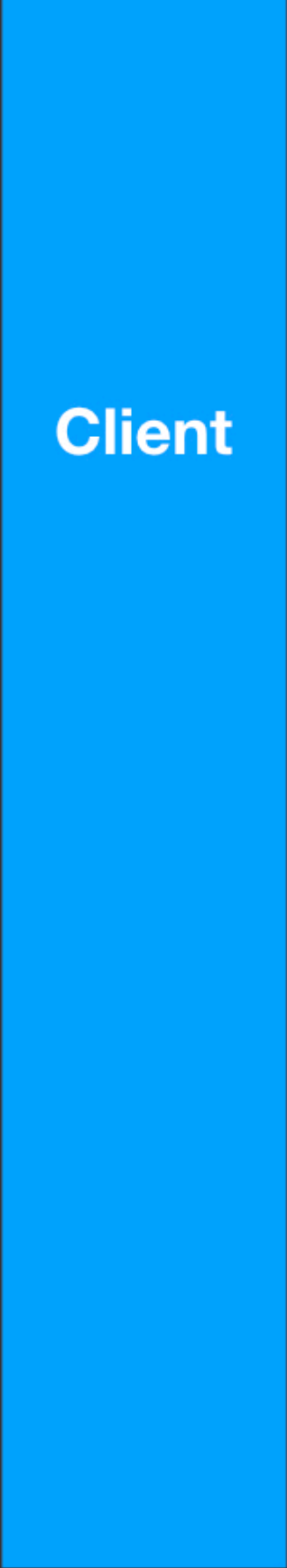
DNS rebinding attack

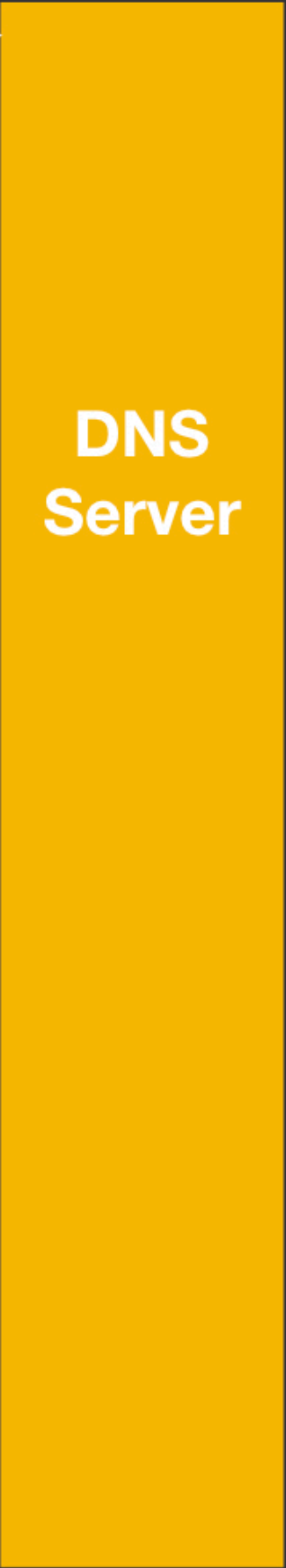
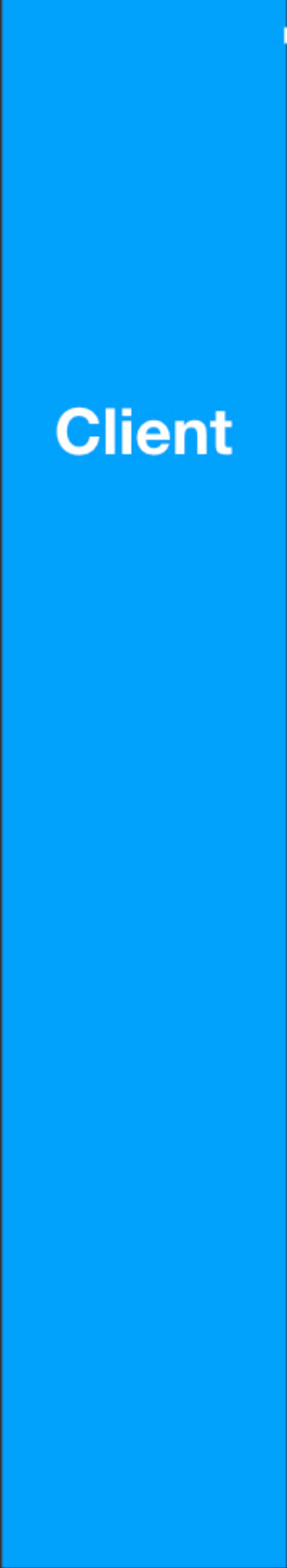
Local
Server

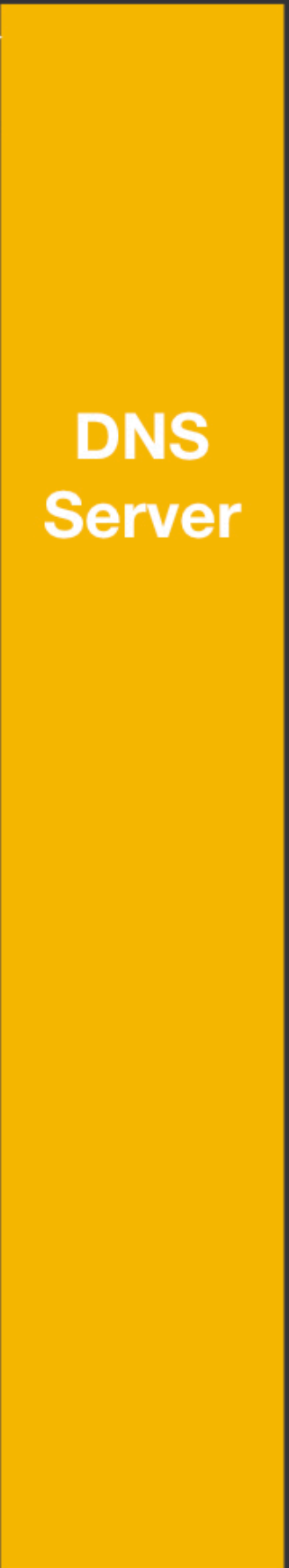
localhost:1234

Client

DNS
Server

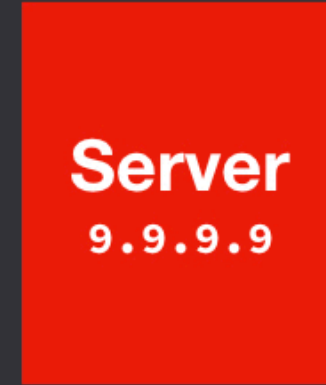


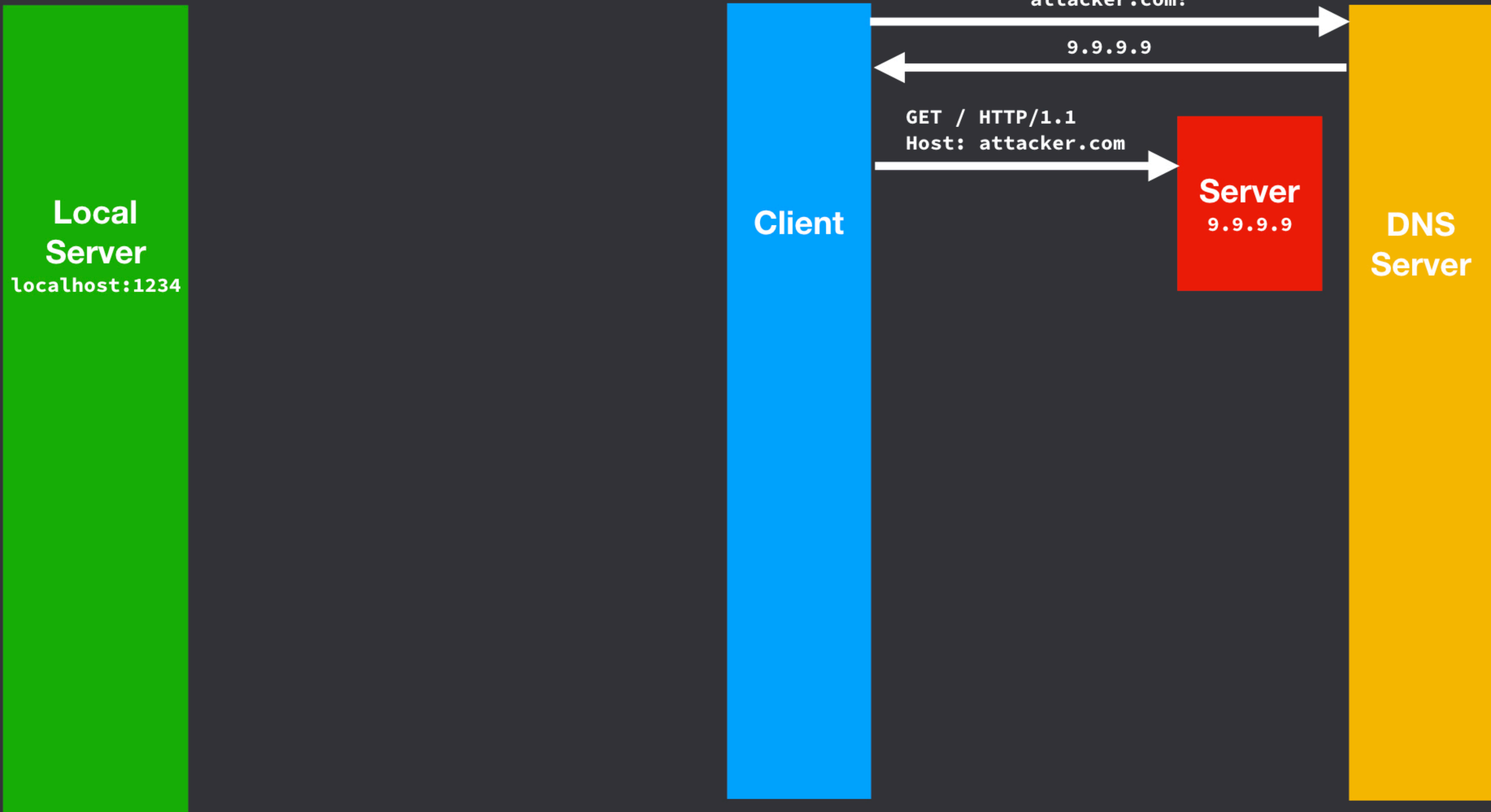


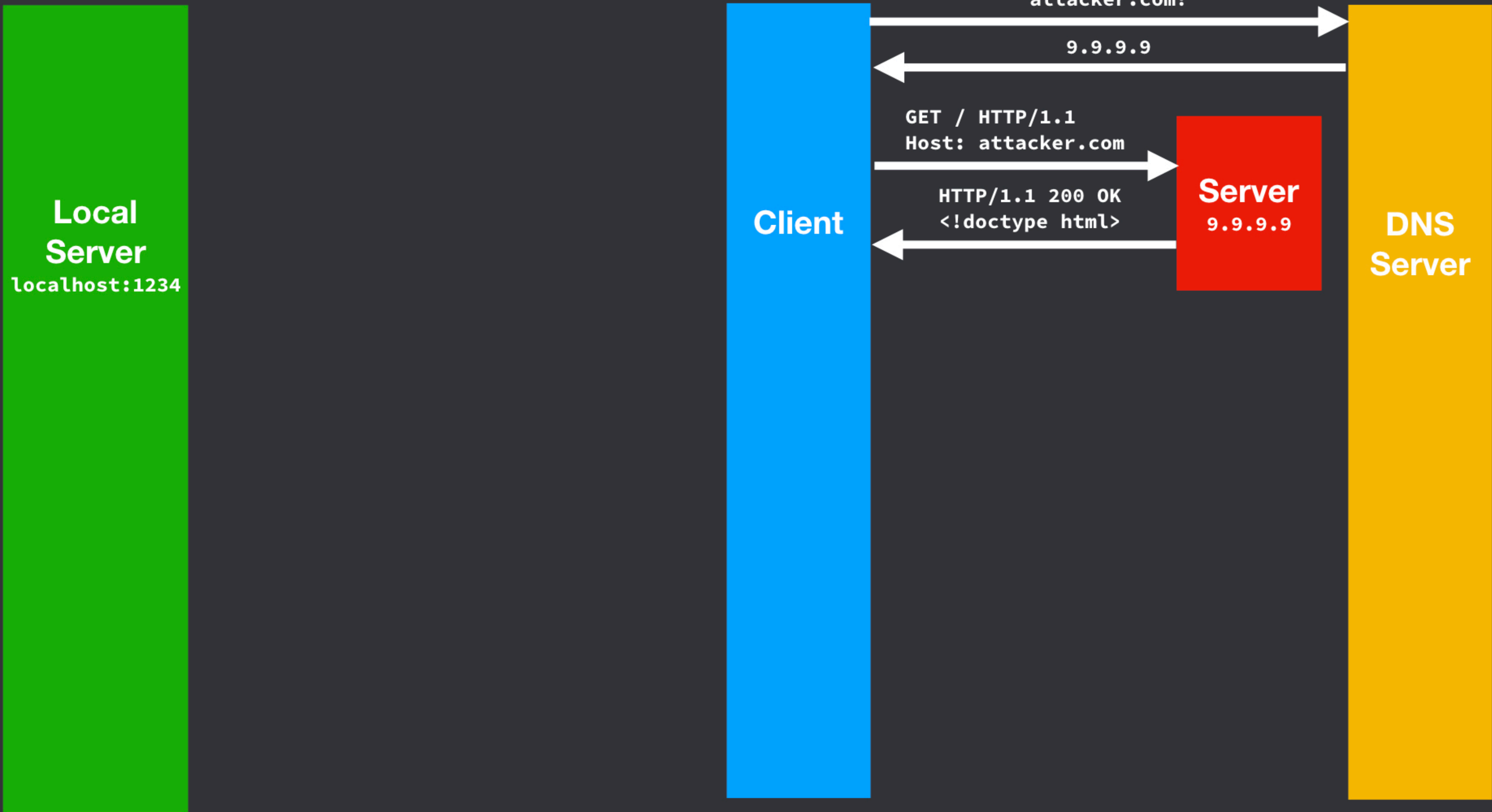


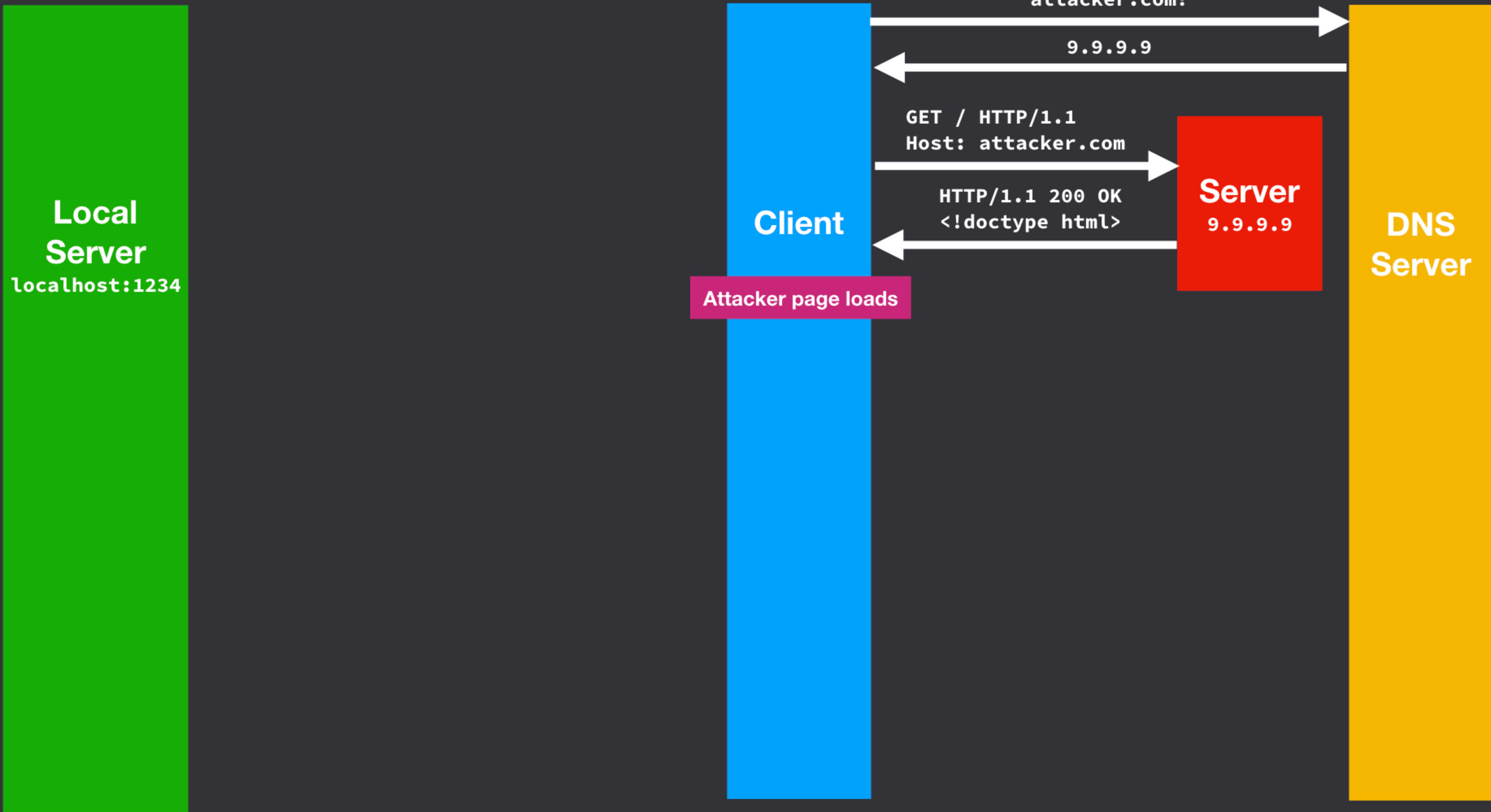
attacker.com?

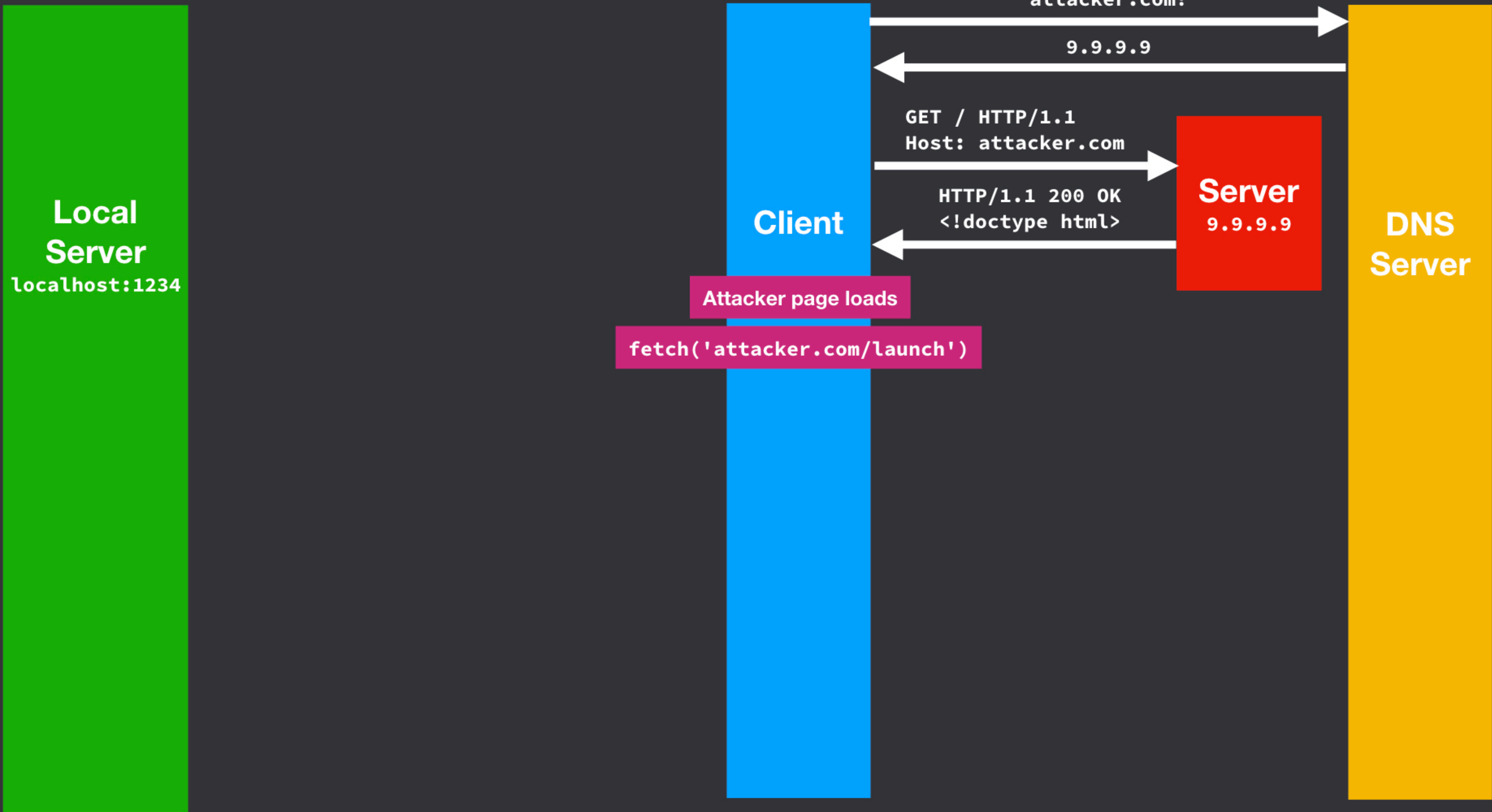
9.9.9.9

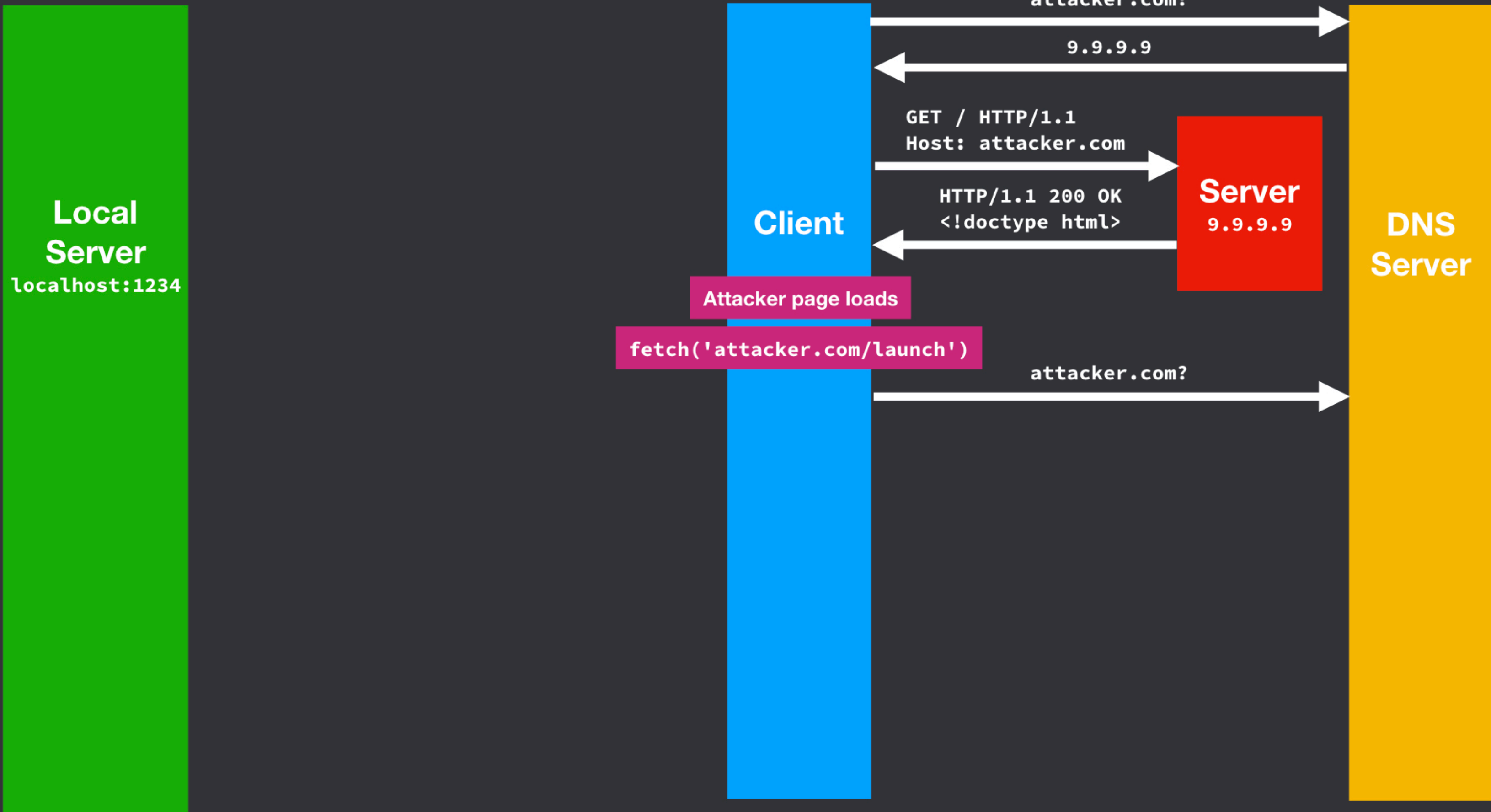


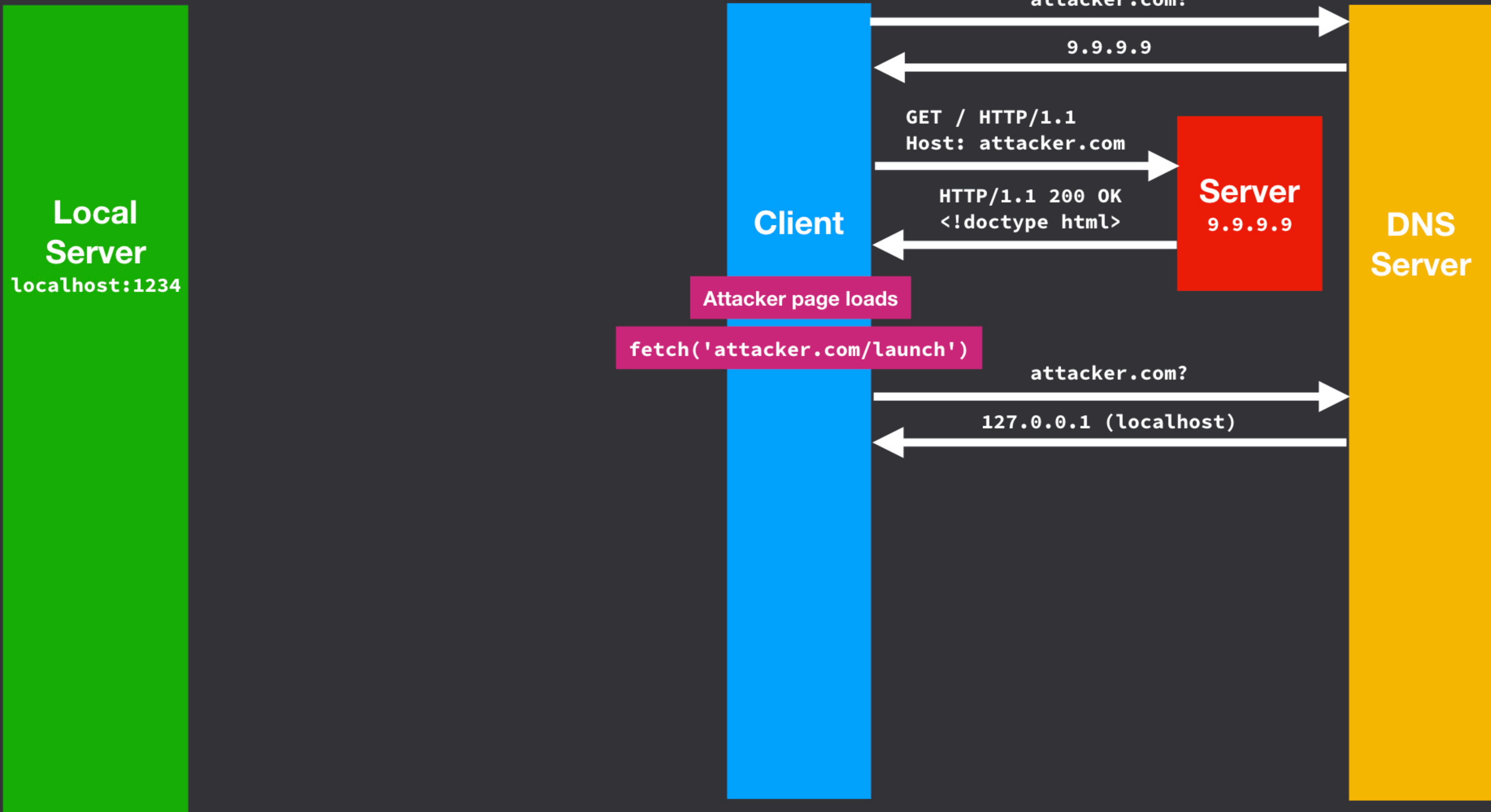


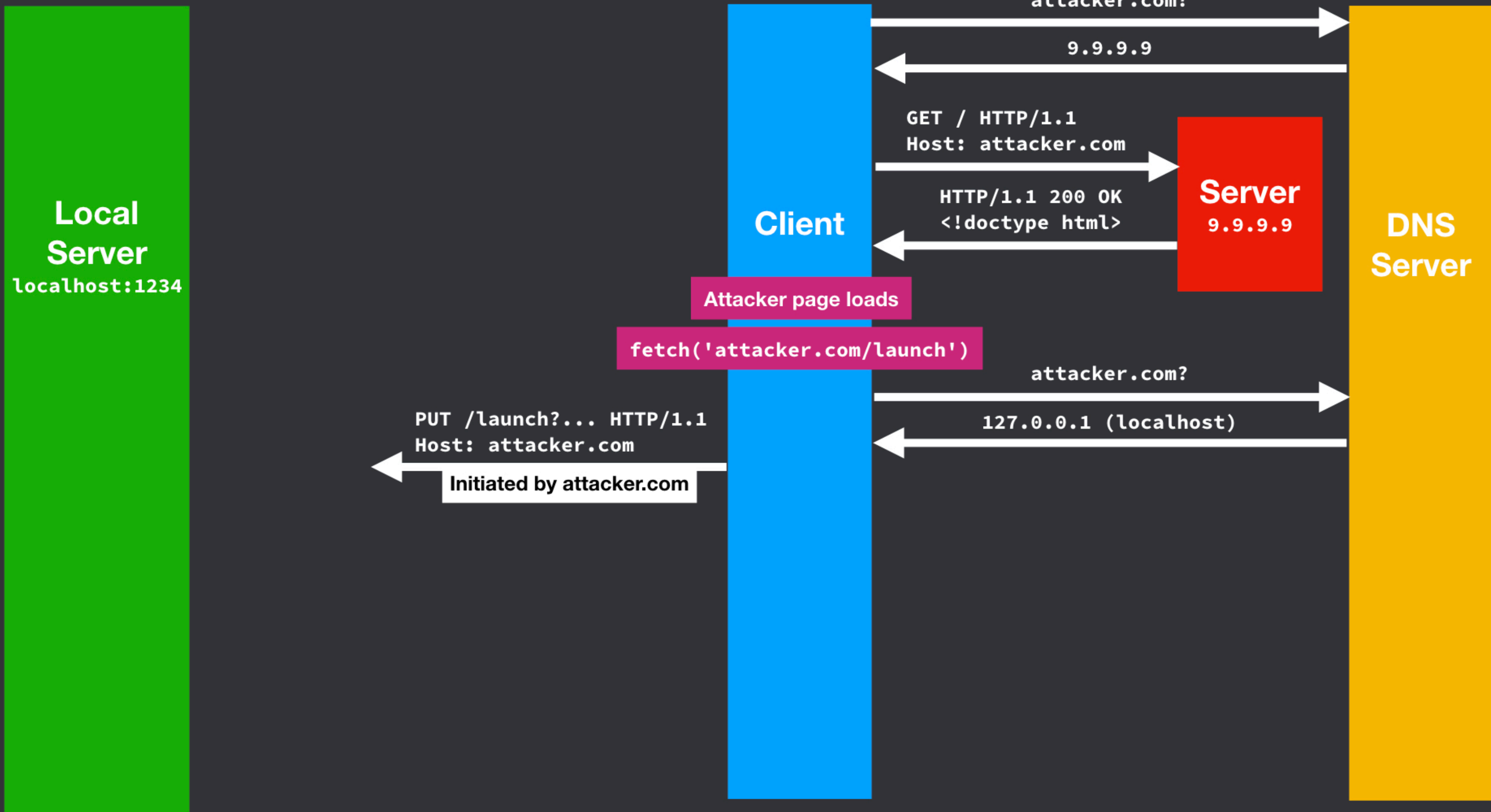


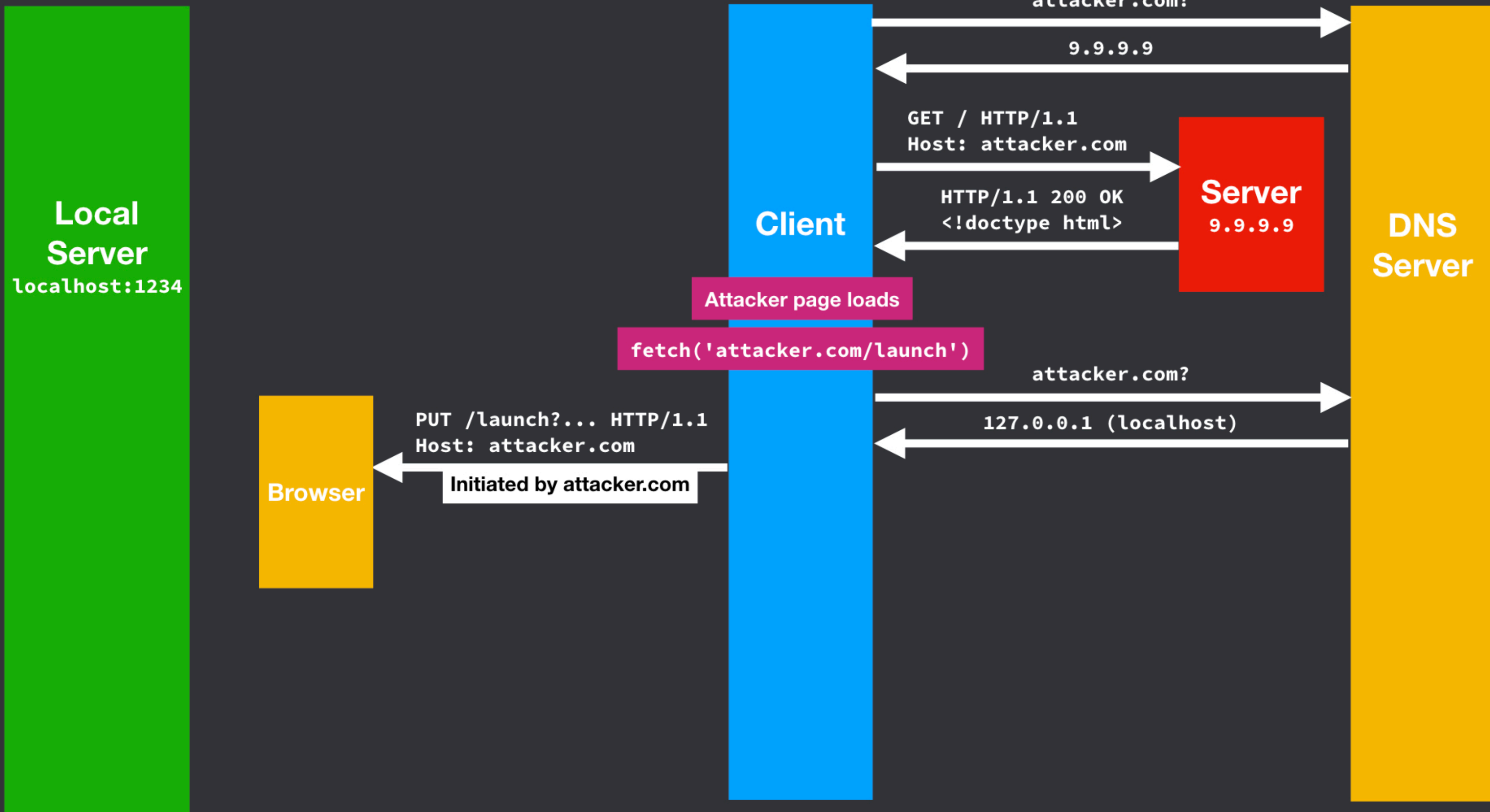


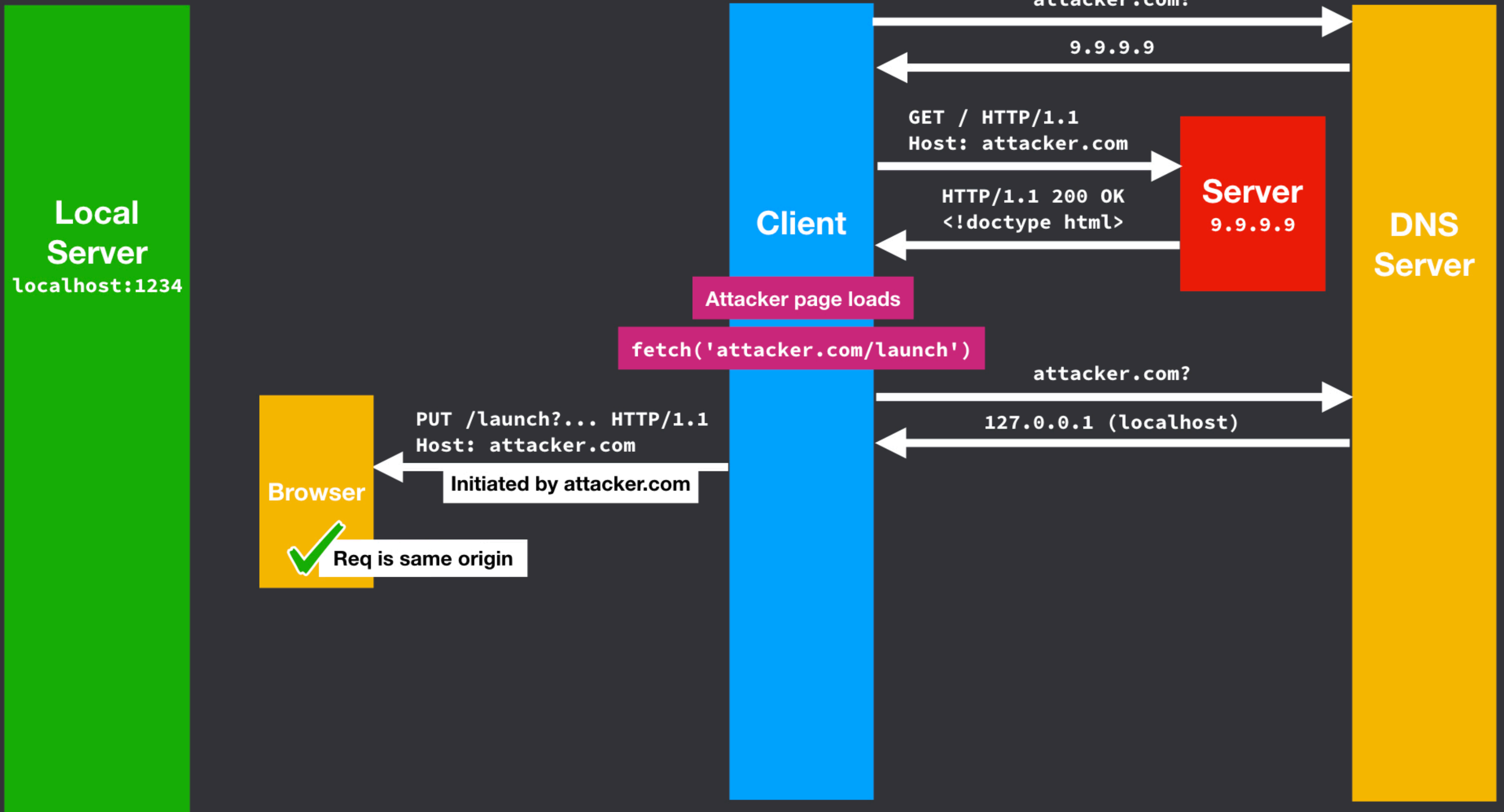


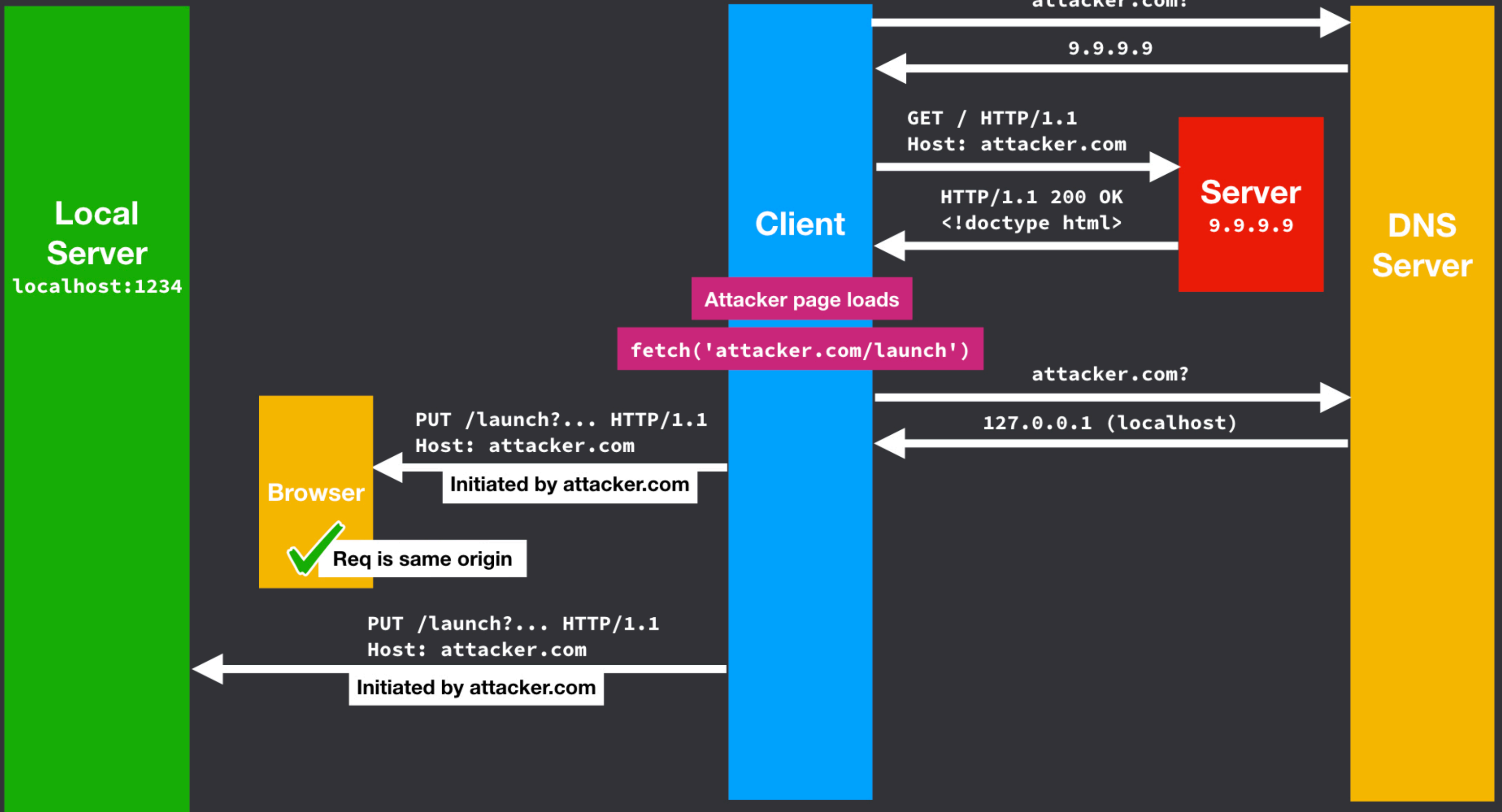


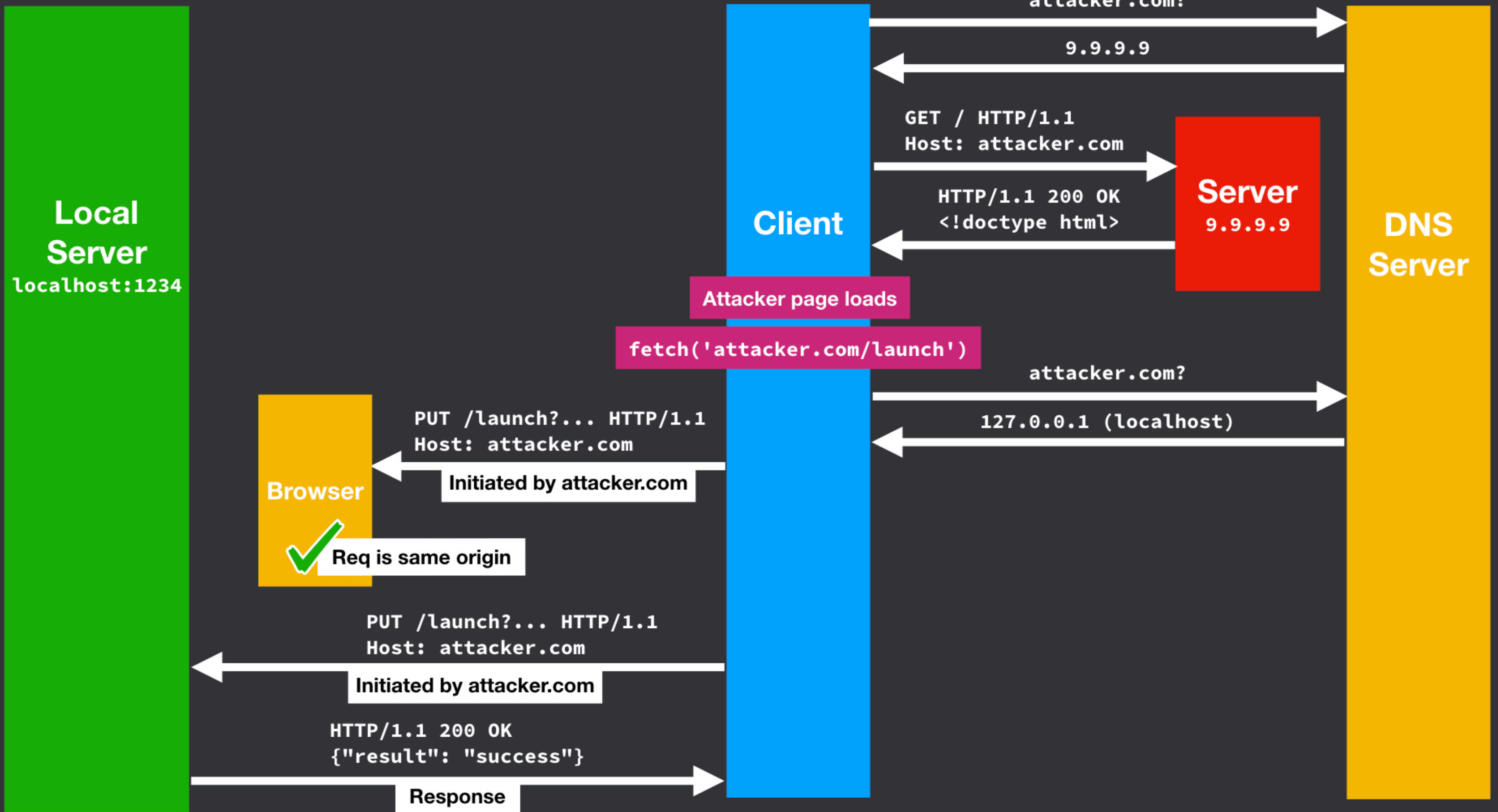












Prevent DNS rebinding attacks

- How can the server protect itself?
- How can the browser protect the server and/or user?

Demo: Prevent DNS rebinding attacks

Demo: Prevent DNS rebinding attacks

```
victimApp.use((req, res, next) => {  
  if (req.headers.host === 'localhost:8080') next()  
  else throw new Error(`Block DNS rebinding from ${req.headers.host}`)  
})
```

```
victimApp.put('/', (req, res) => {  
  exec(COMMAND, err => {  
    if (err) res.status(500).send(err)  
    else res.status(200).send('Success')  
  })  
})
```

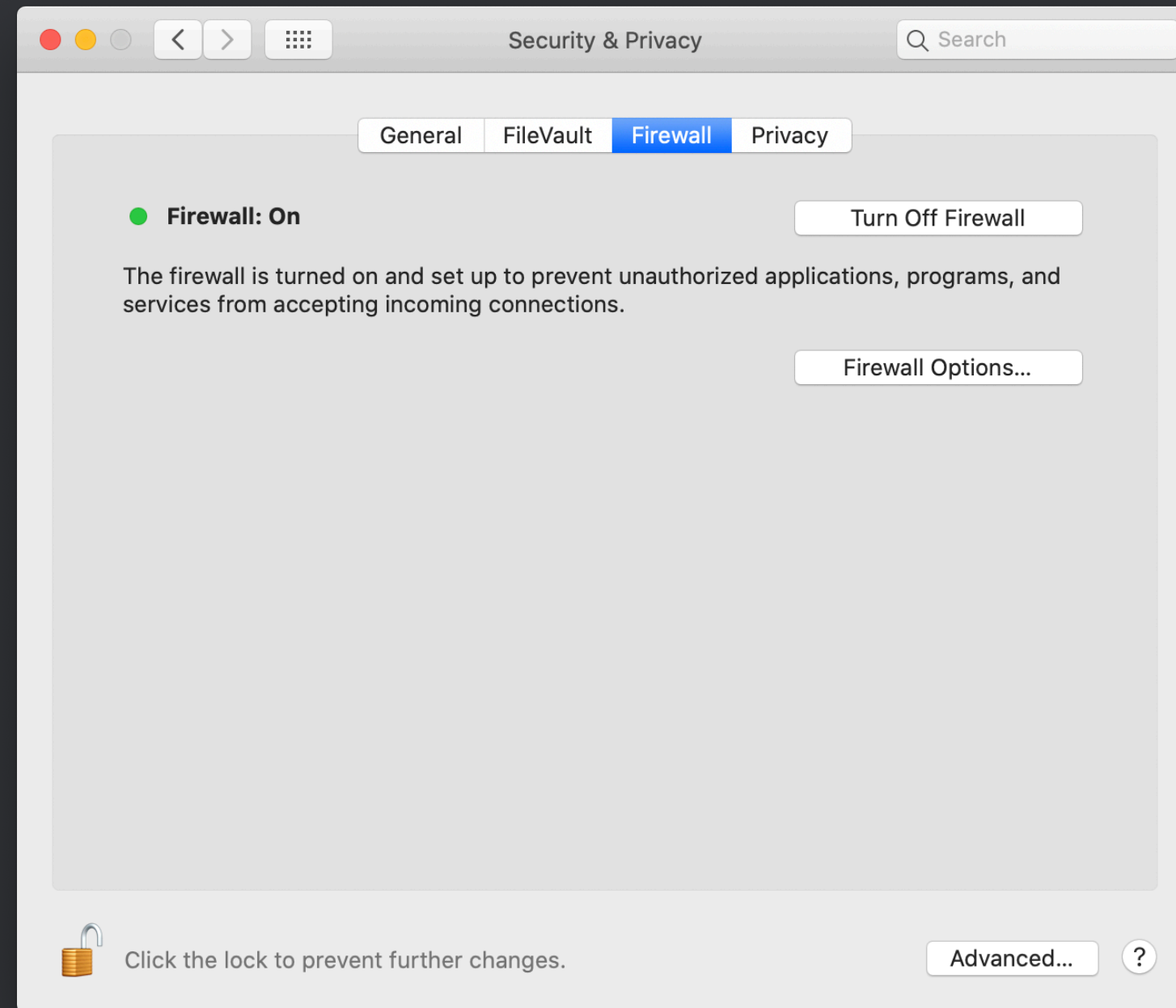
```
attackerApp.get('/', (req, res) => {  
  res.send(`  
    <!doctype html>  
    <html>  
      <body>  
        <h3>Welcome to attacker.com!</h3>  
        <button>Send PUT request to http://attacker.com:8080</button>  
        <script>  
          document.querySelector('button').addEventListener('click', () => {  
            fetch('http://attacker.com:8080', { method: 'PUT' })  
              .then(res => res.text())  
              .then(text => document.body.innerHTML += '<br />' + text)  
              .catch(err => document.body.innerHTML += '<br />' + err)  
          })  
        </script>  
      </body>  
    </html>  
  `)  
})
```

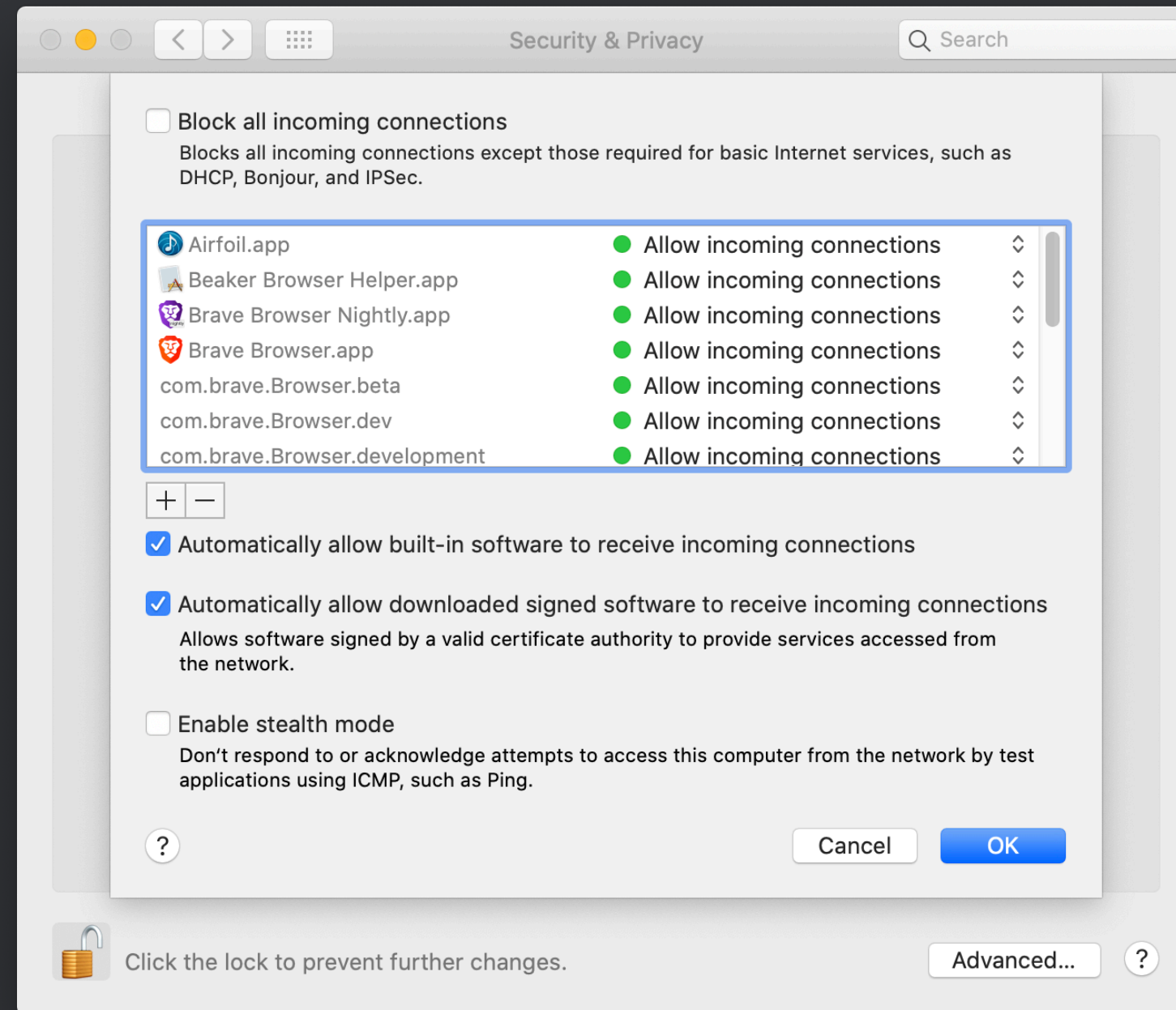
Prevent DNS rebinding

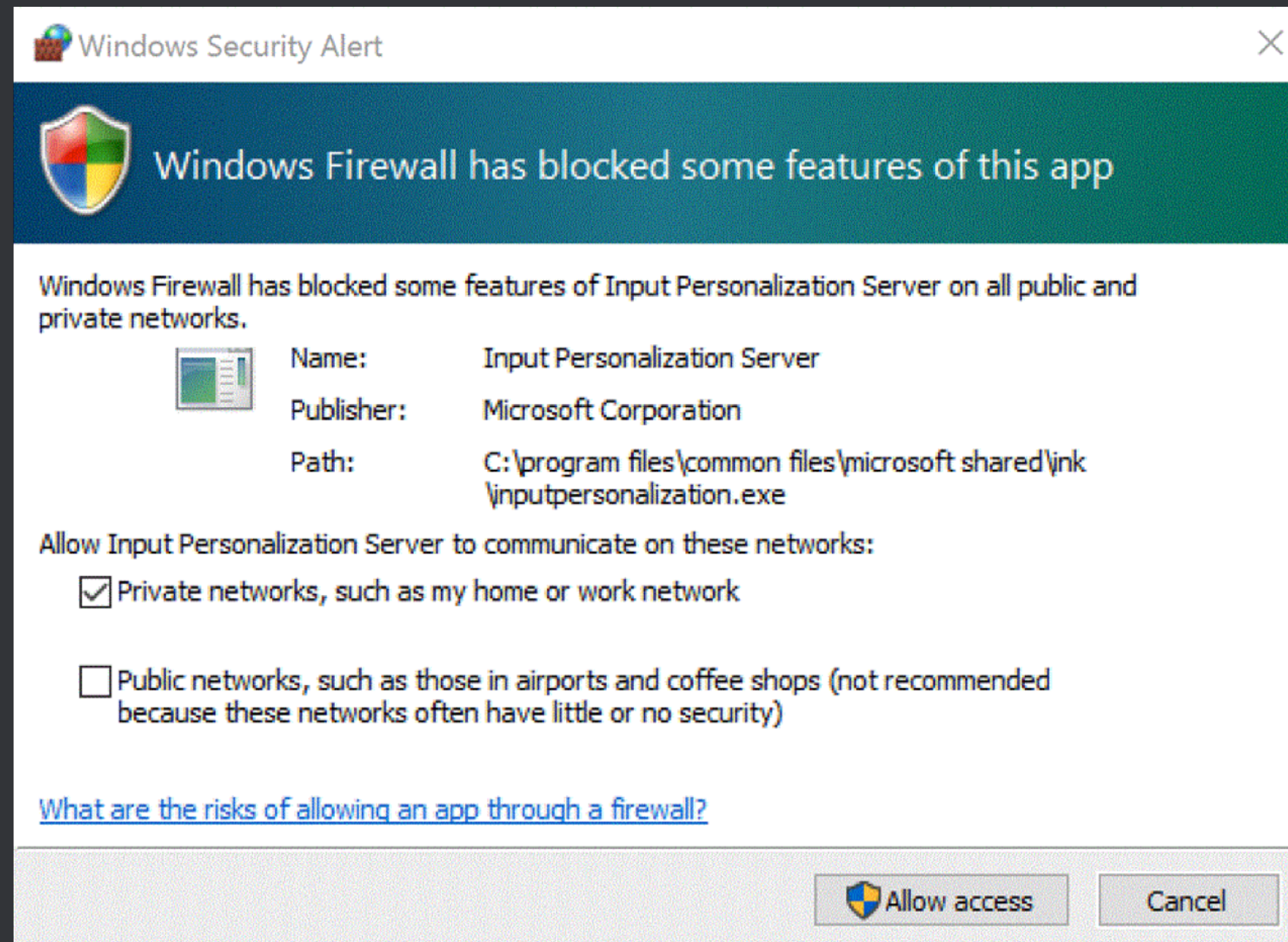
- **Idea:** DNS rebinding allows the attacker to trick the browser into thinking the local victim server is same origin to **attacker.com**
- **Solution:** Ensure that the **Host** header is not for a random origin, but instead for **localhost** or equivalent
- **Remember:** Cannot rely on **Origin** header
 - **Origin** header is not sent for same origin requests

Attack surface of local servers

- Usually bound to localhost with `server.listen(4000, '127.0.0.1')` so other devices on the network cannot connect
- When bound to all interfaces, incoming connections are usually still blocked by the operating system firewall
- Vulnerable to DNS rebinding unless proactive measures are taken (check the **Host** header!)







Attack surface of IoT devices

- Usually bound to all interfaces since they expect connections from other devices on the network
- Usually no operating system firewall or configured to allow incoming connections
- Vulnerable to DNS rebinding unless proactive measures are taken (check the **Host** header!)
- Hard or impossible to update after shipped to the customer, cheaply produced, price pressure, consumers don't care about IoT security

[Home](#) > [Security](#)

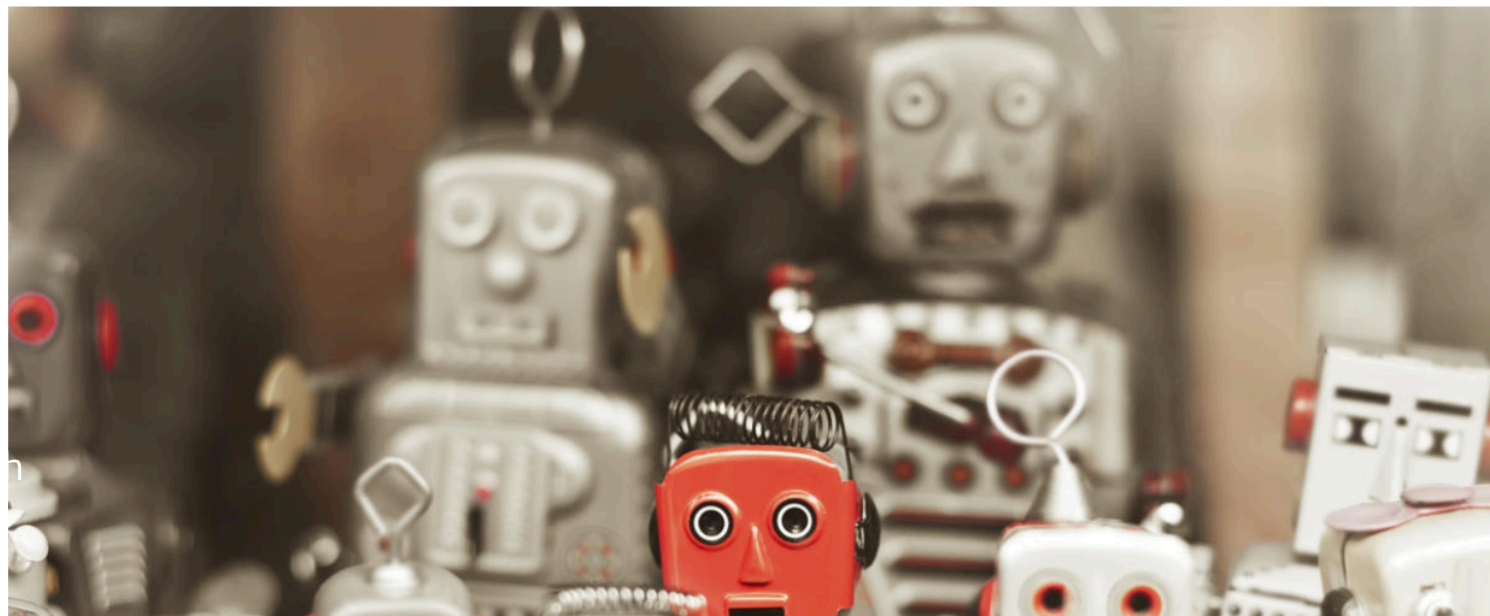
FEATURE

The Mirai botnet explained: How teen scammers and CCTV cameras almost brought down the internet

Mirai took advantage of insecure IoT devices in a simple but clever way. It scanned big blocks of the internet for open Telnet ports, then attempted to log in default passwords. In this way, it was able to amass a botnet army.

**By Josh Fruhlinger**

CSO | MAR 9, 2018 3:00 AM PST



IoT devices botnet (Mirai)

- Minecraft game server hosts launch DDoS attacks against their rivals, hoping to knock their servers offline and attract their business
- Many IoT devices use weak default passwords
- Mirai can launch both HTTP flood and network-level attacks
- Mirai is hard-wired to avoid IPs owned by GE, Hewlett-Packard, and the U.S. DoD
- Contains Russian-language strings which were a red herring

DNS rebinding attack: Blizzard games

- All Blizzard games install a tool called "Blizzard Update Agent"
- The tool starts an HTTP server which accepts commands to install, uninstall, change settings, update and other maintenance related options
- Used a scheme where the HTTP request sender has to prove they can read response data (which is not allowed cross-origin)
- Server was vulnerable to DNS rebinding allowing remote code execution (RCE) from JavaScript on any origin
- See: <https://bugs.chromium.org/p/project-zero/issues/detail?id=1471>

DNS rebinding attack: Transmission

- Transmission is a BitTorrent app, installed locally on the user's device
- Uses a client/server architecture where the UI is a client to the app logic which resides in the server
- Used a scheme where the HTTP request sender has to prove they can read response data (which is not allowed cross-origin)
- Server was vulnerable to DNS rebinding allowing remote code execution (RCE) from JavaScript on any origin
- See: <https://bugs.chromium.org/p/project-zero/issues/detail?id=1447>

DNS rebinding attack: WebTorrent

- WebTorrent has a feature that allows users to serve torrent contents via a local HTTP server
- Useful for streaming video to apps like VLC, allows dynamic piece prioritization
- Server was vulnerable to DNS rebinding allowing attacker to read torrent content (i.e. see the what torrent the user is viewing)
- See: <https://github.com/webtorrent/webtorrent/commit/eea73a38ed8552c6a99cdd0dea5c9619dc955a21#diff-c945a46d13b34fc9ff544d966c9fcbab>
- See: <https://github.com/webtorrent/webtorrent/commit/30adf6a19b50b6e013c8ad9532c7e59d349df461#diff-c945a46d13b34fc9ff544d966c9fcbab>

Why isn't DNS rebinding an effective attack against remote servers?

What is the difference between the Origin and Host headers?

Final thoughts

- Don't ship a local HTTP server with your software
- If you do, it better be for a really good reason
- If you do, you better understand DNS rebinding attacks
- Check the value of the **Host** header!

END

Credits:

<https://www.csoononline.com/article/3258748/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html>