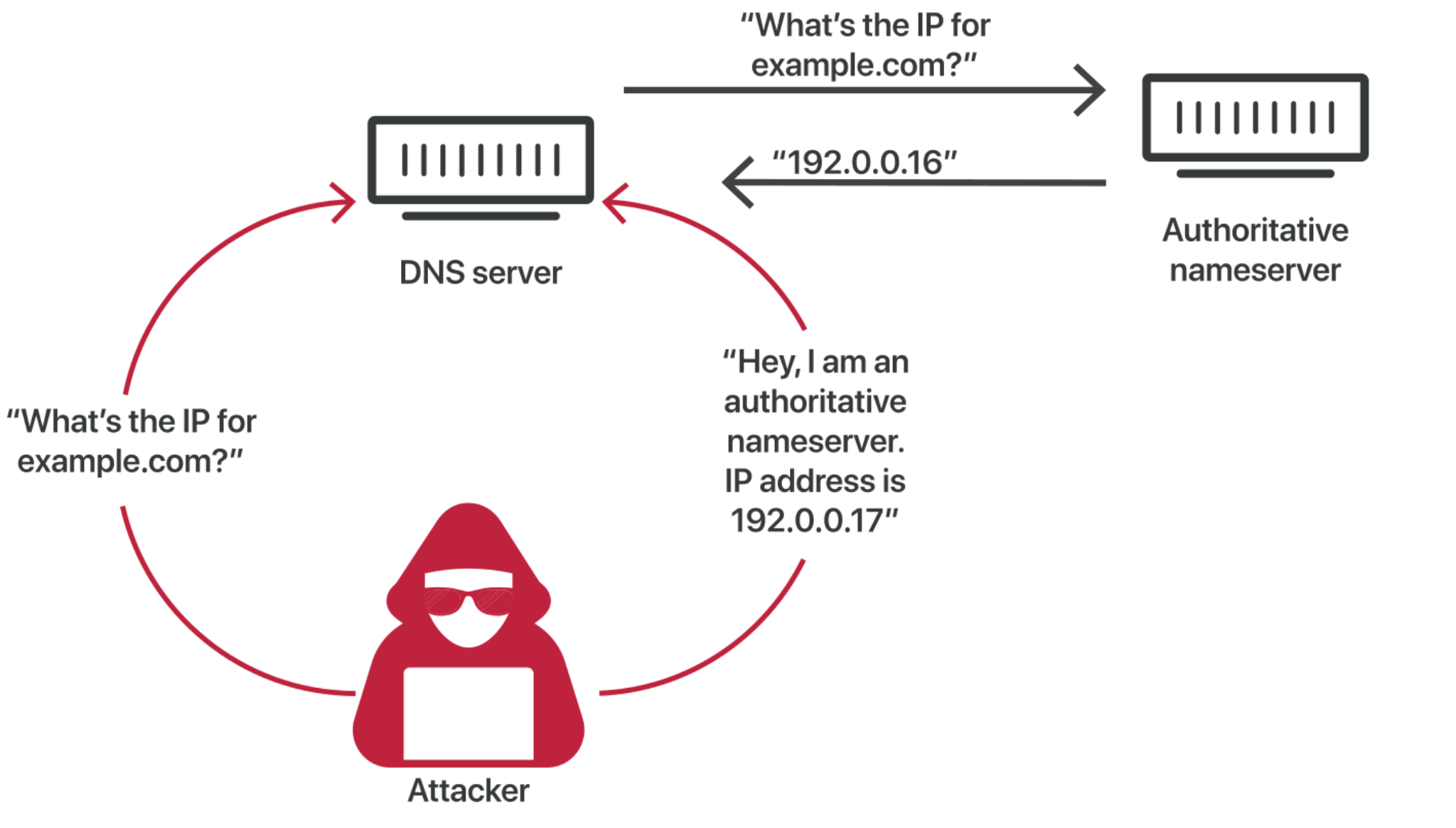
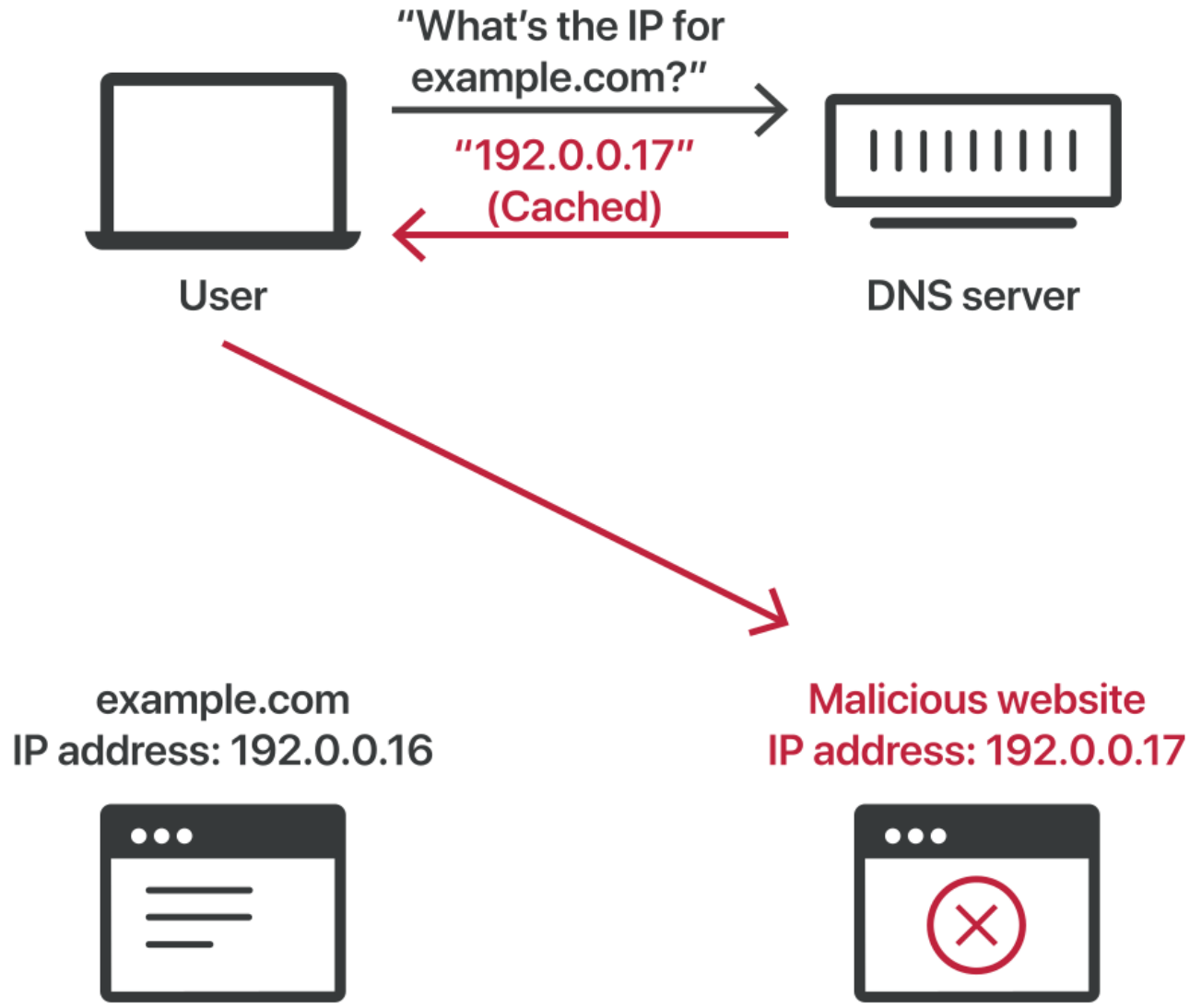
DNS CACHE POISONING ATTACK

DNS cache poisoning attack :



Result:


"What's the IP for example.com?"
"192.0.0.17" (Cached)
User
DNS server
example.com
IP address: 192.0.0.16
Malicious website
IP address: 192.0.0.17

EXPERIMENT SECTION

Architecture Overview:

- - Four containers: Client, DNS Server, Attacker, Root DNS
- - All connected on custom Docker network
- - Controlled IP assignments for deterministic routing
- - Python-based DNS resolver and attack scripts

Docker Compose Structure:

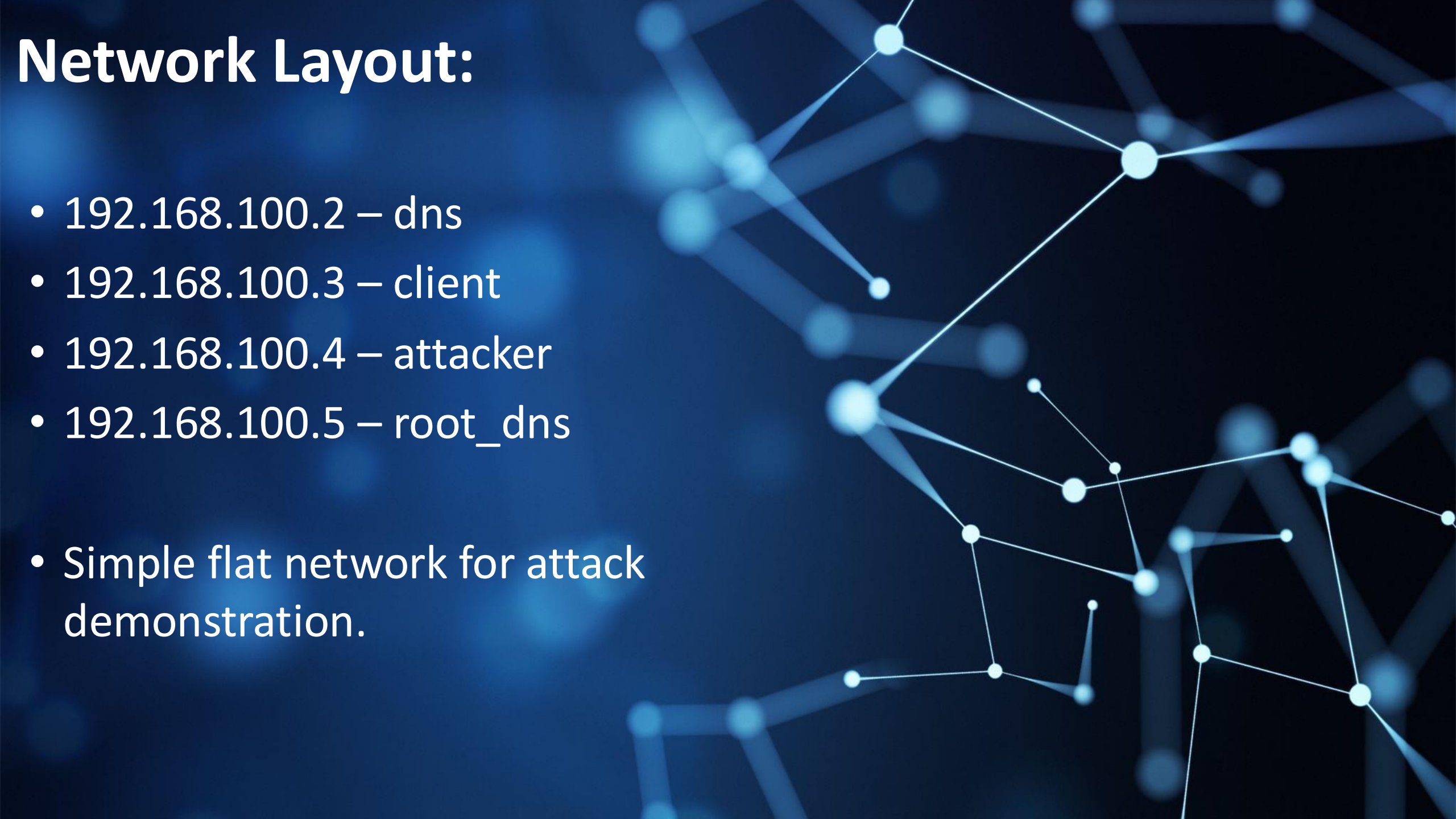
- Services:
  - dns: resolver
  - client: victim DNS requester
  - attacker: sends spoofed replies
  - root_dns: authoritative server

- Network:
  - Subnet 192.168.100.0/28
  - Static IPs to ensure predictable communication

Network Layout:

- 192.168.100.2 – dns
- 192.168.100.3 – client
- 192.168.100.4 – attacker
- 192.168.100.5 – root_dns

- Simple flat network for attack demonstration.



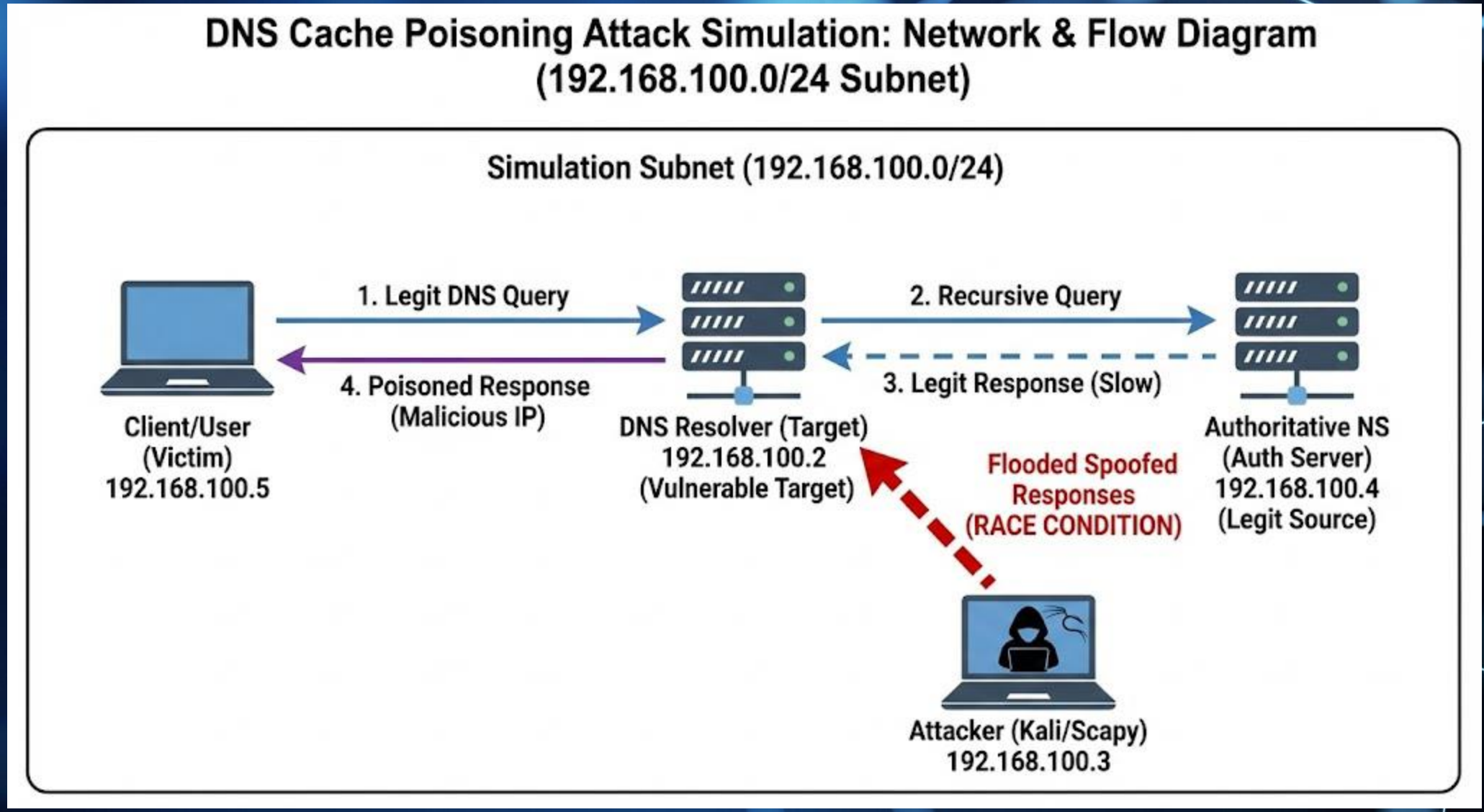
Attack Flow:

- 1. Client sends DNS query to resolver.
- 2. Resolver queries root_dns.
- 3. Attacker floods resolver with forged replies.
- 4. Resolver accepts malicious answer.
- 5. Client receives poisoned DNS record.

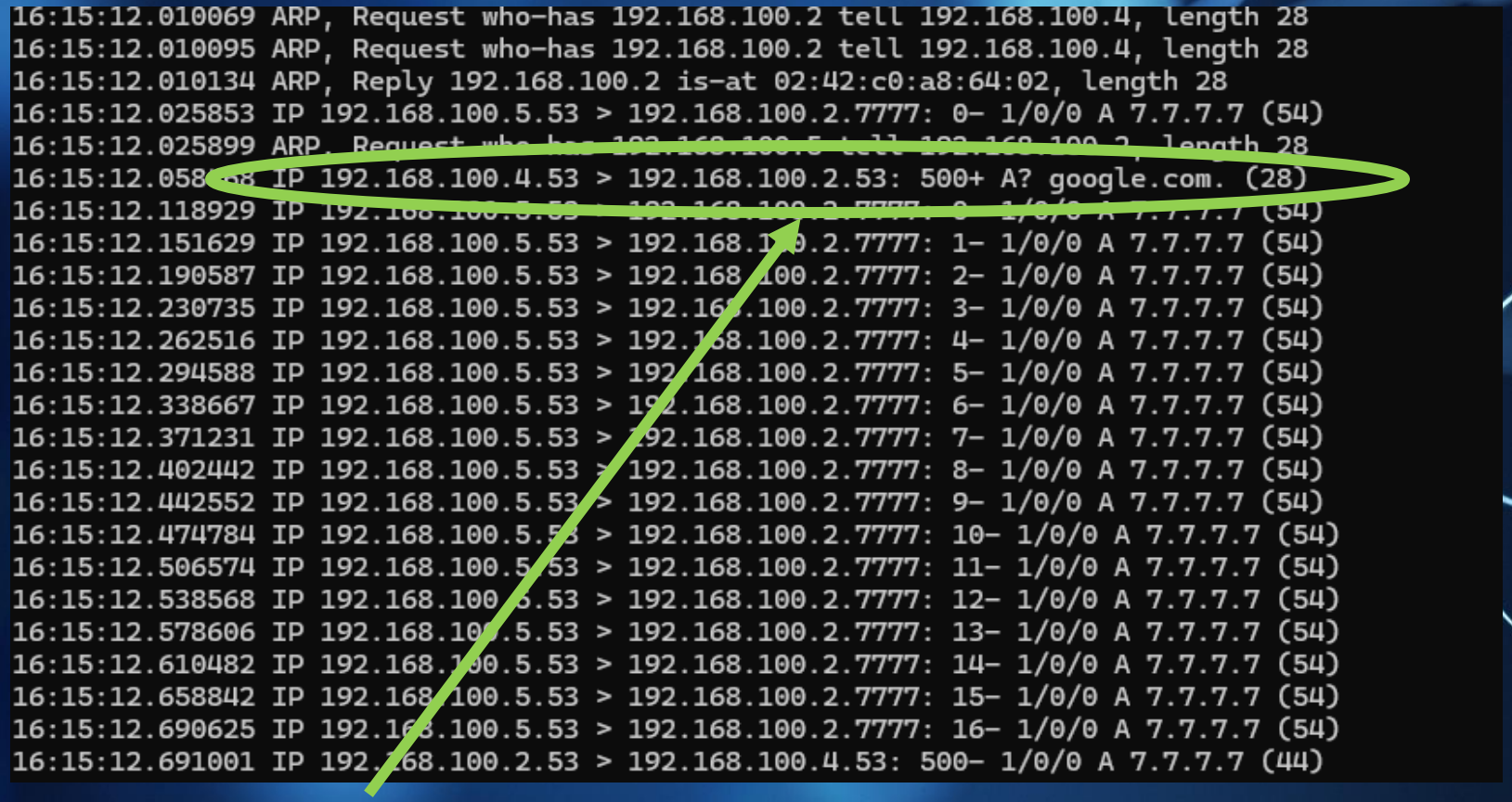
Demonstration Steps:

- 1. docker-compose build
- 2. docker-compose up
- 2. From attacker: Run attacker.py
- 3. From client: dig @dns google.com

Network diagram:



Attack in action:



We trigger an IP lookup for google.com on the target DNS server. We preemptively send a fake reply because we cannot predict exactly when the server will be ready to accept it.

Attack in action:

```
16:15:12.010069 ARP, Request who-has 192.168.100.2 tell 192.168.100.4, length 28
16:15:12.010095 ARP, Request who-has 192.168.100.2 tell 192.168.100.4, length 28
16:15:12.010134 ARP, Reply 192.168.100.2 is-at 02:42:c0:a8:64:02, length 28
16:15:12.025853 IP 192.168.100.5.53 > 192.168.100.2.7777: 0- 1/0/0 A 7.7.7.7 (54)
16:15:12.025899 ARP, Request who-has 192.168.100.5 tell 192.168.100.2, length 28
16:15:12.058468 IP 192.168.100.4.53 > 192.168.100.2.53: 500+ A? google.com. (28)
16:15:12.118929 IP 192.168.100.5.53 > 192.168.100.2.7777: 0- 1/0/0 A 7.7.7.7 (54)
16:15:12.151629 IP 192.168.100.5.53 > 192.168.100.2.7777: 1- 1/0/0 A 7.7.7.7 (54)
16:15:12.190587 IP 192.168.100.5.53 > 192.168.100.2.7777: 2- 1/0/0 A 7.7.7.7 (54)
16:15:12.230735 IP 192.168.100.5.53 > 192.168.100.2.7777: 3- 1/0/0 A 7.7.7.7 (54)
16:15:12.262516 IP 192.168.100.5.53 > 192.168.100.2.7777: 4- 1/0/0 A 7.7.7.7 (54)
16:15:12.294584 IP 192.168.100.5.53 > 192.168.100.2.7777: 5- 1/0/0 A 7.7.7.7 (54)
16:15:12.338657 IP 192.168.100.5.53 > 192.168.100.2.7777: 6- 1/0/0 A 7.7.7.7 (54)
16:15:12.371731 IP 192.168.100.5.53 > 192.168.100.2.7777: 7- 1/0/0 A 7.7.7.7 (54)
16:15:12.402442 IP 192.168.100.5.53 > 192.168.100.2.7777: 8- 1/0/0 A 7.7.7.7 (54)
16:15:12.442552 IP 192.168.100.5.53 > 192.168.100.2.7777: 9- 1/0/0 A 7.7.7.7 (54)
16:15:12.474784 IP 192.168.100.5.53 > 192.168.100.2.7777: 10- 1/0/0 A 7.7.7.7 (54)
16:15:12.506574 IP 192.168.100.5.53 > 192.168.100.2.7777: 11- 1/0/0 A 7.7.7.7 (54)
16:15:12.538568 IP 192.168.100.5.53 > 192.168.100.2.7777: 12- 1/0/0 A 7.7.7.7 (54)
16:15:12.578606 IP 192.168.100.5.53 > 192.168.100.2.7777: 13- 1/0/0 A 7.7.7.7 (54)
16:15:12.610482 IP 192.168.100.5.53 > 192.168.100.2.7777: 14- 1/0/0 A 7.7.7.7 (54)
16:15:12.658842 IP 192.168.100.5.53 > 192.168.100.2.7777: 15- 1/0/0 A 7.7.7.7 (54)
16:15:12.690675 IP 192.168.100.5.53 > 192.168.100.2.7777: 16- 1/0/0 A 7.7.7.7 (54)
16:15:12.691001 IP 192.168.100.2.53 > 192.168.100.4.53: 500- 1/0/0 A 7.7.7.7 (44)
```

We flood the target DNS server with fake responses to beat the upstream DNS server. This successfully poisons the cache, redirecting all users to the malicious IP 7.7.7.7

Result of the attack:

```
root@df708b7e9302:/# dig google.com

; <<>> DiG 9.18.39-0ubuntu0.24.04.2-Ubuntu <<>> google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 12214
;; flags: qr rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             3600    IN      A       7.7.7.7

;; Query time: 0 msec
;; SERVER: 192.168.100.2#53(192.168.100.2) (UDP)
;; WHEN: Sun Nov 23 16:25:51 UTC 2025
;; MSG SIZE  rcvd: 44
```

If the attack was successful, on any machine retrieving the IP from this DNS server, we can see that google.com resolves to the IP 7.7.7.7

Possible countermeasures:

- Switching to DNSSEC (Domain Name System Security Extensions): adds a layer of security by attaching cryptographic signatures to DNS records. This ensures data integrity and origin authentication, allowing the resolver to verify that the response comes from the legitimate source. Consequently, it prevents cache poisoning attacks because any spoofed response (lacking a valid signature) is automatically rejected

- Source Port Randomization: increases the entropy of the DNS transaction. Instead of using a fixed or predictable port for outgoing queries, the resolver selects a random ephemeral port for each request. To succeed, an attacker must now correctly guess both the 16-bit Transaction ID and the random 16-bit source port simultaneously. This expands the search space to over 4 billion combinations, making blind brute-force attacks computationally infeasible

- DNS 0x20: Encoding uses mixed-case randomization in the domain name query (e.g., requesting `GoOgLe.cOm` instead of `google.com`). Since standard authoritative servers copy the query section exactly into their response, the resolver can verify that the casing in the reply matches the query perfectly. An attacker, unable to predict the specific randomization pattern, will likely send a response with incorrect casing (e.g., all lowercase), causing it to be rejected.

Source Port Randomization code example:

```
# --- FIX: PORT RANDOMIZATION ---
# Instead of fixed port 7777, we use 0. The kernel will assign a random high ephemeral port.
# This forces the attacker to guess among ~60,000 ports instead of just 1.
upstream_socket.bind((LOCAL_INTERFACE_IP, 0))
# ---------------------------------
```

DNS 0x20 code example:

```python
def apply_0x20_encoding(domain_str):
    """
    Randomly transforms 'google.com' into mixed-case 'GoOgLe.CoM'.
    This adds entropy to the request to prevent spoofing.
    """
    if domain_str.endswith('.'): domain_str = domain_str[:-1]
    result = []
    for char in domain_str:
        if char.isalpha():
            if randint(0, 1):
                result.append(char.upper())
            else:
                result.append(char.lower())
        else:
            result.append(char)
    return "".join(result) + "."
```

Sources:

- → Che cos'è il DNS cache poisoning? | DNS spoofing | Cloudflare
- → https://youtu.be/pFh1hmdRAoU?si=ayNj7a114_zaEpA2
- → https://youtu.be/7MT1F0O3_Yw?si=NpUvmqdFNvp4iDqV

My github repo:

Zartep/DNS_cache_poisoning_attack_simulation: Academic Project: DNS Cache Poisoning Simulation