

# Introduction to Cryptography and Security

# Course Description

- Name: Software and Cybersecurity (3-0-2-4)
- Course Code: CS445/ IT445
- Lectures: Tuesday (9:15am) and Friday (11:00am)
- Lab: Monday 2:00pm-4:00pm

# Major division of the course

- Software security
  - C/C++/java/python
- Cybersecurity
  - Computer network/DBMS/Linux
- Cybersecurity professionals for industry
  - tools

# Course evaluation

- Mid semester – 20%
- End semester – 30%
- Assignments – 20%
- Quizzes – 30%

# Security issues

**The world before computers:** much simpler

- Signing, legalizing a paper would authenticate it
  - One can recognize each other's face, voice, hand signature, etc.
- Photocopying easily detected
- Erasing, inserting, modifying words on a paper document easily detectable
- Secure transmission of a document: seal it and use a reasonable mail carrier (hoping the mail train does not get robbed)

# Security issues

**Electronic world:** the ability to copy and alter information has changed dramatically

- No difference between an “original” file and copies of it
- Removing a word from a file or inserting others is undetectable
- Adding a signature to the end of a file/email: one can impersonate it – add it to other files as well, modify it, etc.
- Electronic traffic can be monitored, altered, often without noticing
- How to authenticate the person electronically communicating with you

# Possible adversaries

- **Student:** to have fun snooping on other people's email
- **Businessman:** to discover a competitor's strategic marketing plan
- **Ex-employee:** to get revenge for being fired
- **Accountant:** to withdraw money from a company
- **Stockbroker:** to deny a promise made to a customer by email
- **Convict:** to steal credit card numbers for sale
- **Spy:** to learn an enemy's military or industrial secrets
- **Terrorist:** to steal secret information
- Point to make: making a network or a communication secure involves more than just keeping it free of programming errors
- It involves intelligent, dedicated and often well-funded adversaries

# Security issues: some practical situations

- **A** sends a file to **B**: **E** intercepts it and reads it
  - How to send a file that looks unintelligible to all but the intended receiver?
- **A** sends a file to **B** : **E** intercepts it, modifies it, and then forwards it to **B**
  - How to make sure that the document has been received in exactly the form it has been sent?
- **E** sends a file to **B** pretending it is from **A**
  - How to make sure your communication partner is really who(s) he claims to be?
- **A** sends a message to **B** : **E** is able to delay the message for a while
  - How to detect old messages?
- **A** sends a message to **B**. Later **A** (or B) denies having sent (received) the message
  - How to deal with electronic contracts?
- **E** learns which user accesses which information although the information itself remains secure
- **E** prevents communication between **A** and **B** : **B** will reject any message from **A** because they look unauthentic

# Information Security

- Information Security is the practice of protecting information by mitigating information risks
- It involves the protection of information systems and the information processed, stored and transmitted by these systems from unauthorized access, use, disclosure, disruption, modification or destruction.
- Types: Application Security, Internet Security, Cloud security, Cryptography, etc.

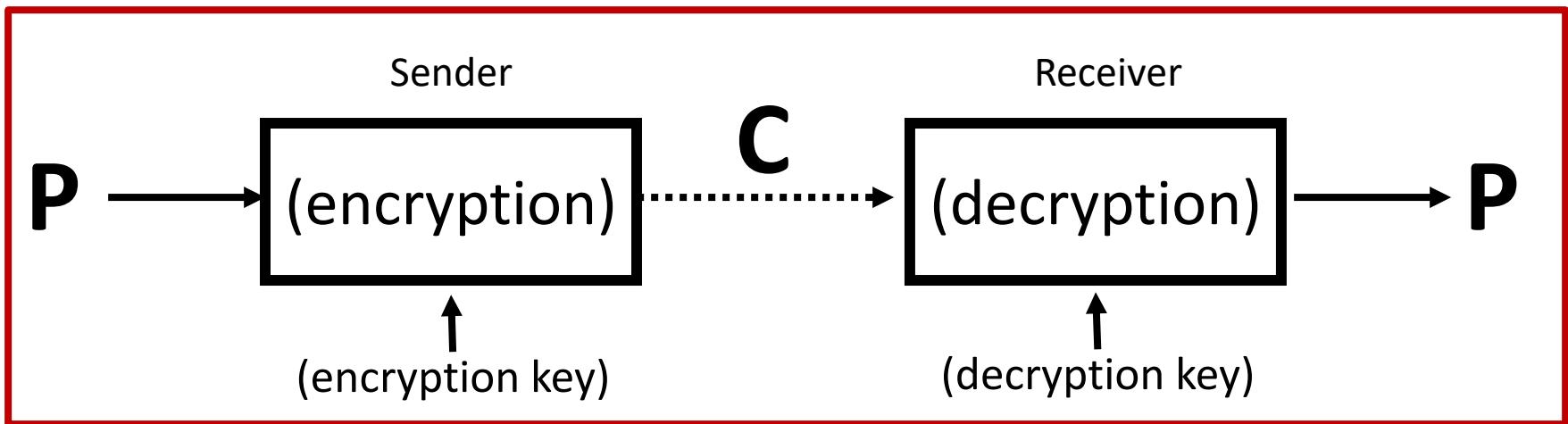
# Classes of network security problems

- Confidentiality (or secrecy)
  - Keep the information out of the hands of unauthorized users, even if it has to travel over insecure links
  - Privacy defines the ability to secure personally identifiable data
- Authentication
  - Determine whom you are talking to before revealing sensitive information
- Data integrity (or message authentication)
  - Make sure that the message received was exactly the message you sent (not necessarily interested here in the confidentiality of the document)
- Non-repudiation (or signatures)
  - the assurance that someone cannot deny the validity of something

# What is Cryptography

- **Cryptography** is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, authentication, and non-repudiation.
- **Cryptology** = Cryptography + Cryptanalysis
  - Cryptography --- code designing
    - study of **secret** (crypto-) **writing** (-graphy)
  - Cryptanalysis --- code breaking

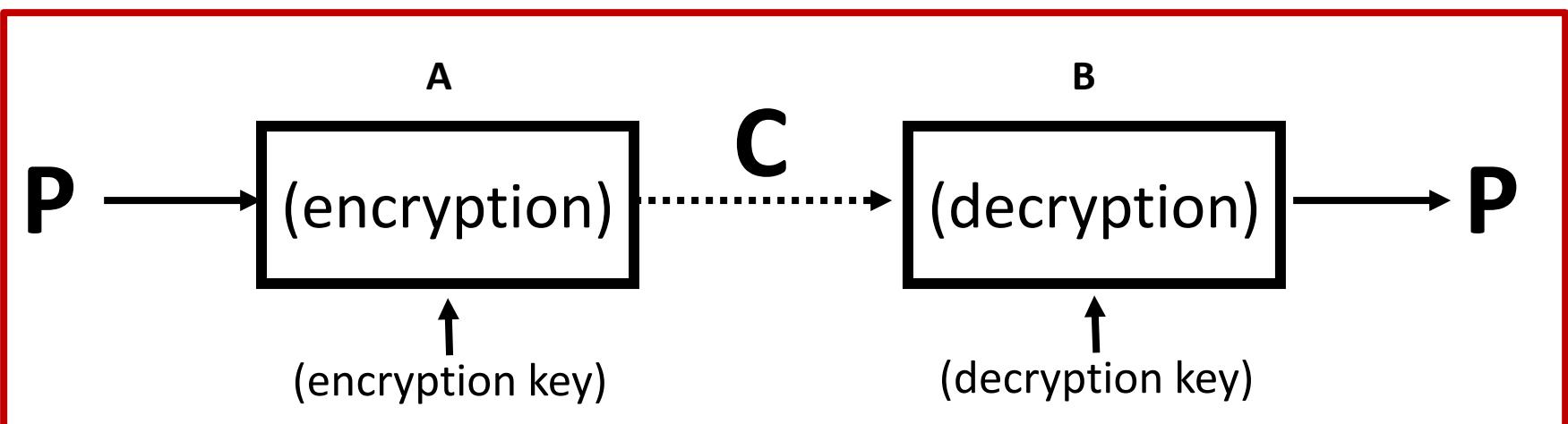
# What is a cryptographic system composed of?



- **Plaintext (P)**: original message or data (also called cleartext)
- **Encryption**: transforming the plaintext, under the control of the key
- **Ciphertext (C)**: encrypted plaintext
- **Decryption**: transforming the ciphertext back to the original plaintext
- **Cryptographic key**: used with an algorithm to determine the transformation from plaintext to ciphertext, and v.v.

# Basic situation in cryptography

- **A**(lice) sends a message (or file) to **B**(ob) through an open channel (say, Internet), where **E**(vil, nemy) tries to read or change the message
- **A** will encrypt the plaintext using a key transforming it into a “unreadable” cryptotext or ciphertext
  - This operation must be computationally easy



# Basic situation in cryptography

- **B** also has a key (say, the same key) and decrypts the cryptotext to get the plaintext
  - This operation must be computationally easy
- **E** tries to cryptanalyze: deduce the plaintext (and the key) knowing only the ciphertext
  - This operation should be computationally difficult
- We will use cryptography to cover both the design of secure systems and their cryptanalysis
  - Do not think in terms of good guys do cryptography and bad guys do cryptanalysis?

# Cryptanalysis – types of attacks

- **Fundamental rule:** one must always assume that the attacker knows the methods for encryption and decryption; he is only looking for the keys
  - Creating a new cryptographic method is a very complex process involving many people – difficult to keep it confidential
  - Bonus for publishing the methods: people will try to break it for you (for free!)
- **Passive attack:** the attacker only monitors the traffic attacking the confidentiality of the data
- **Active attack:** the adversary attempts to alter the transmission attacking data integrity, confidentiality, and authentication.
- **Cryptanalysis:** rely on the details of the encryption algo. plus perhaps some knowledge about the general characteristics of the plaintext – sometimes the plaintext is known and the key is being looked for
- **Brute-force attack:** try every possible key on the ciphertext until an intelligible translation into a plaintext is obtained

# Average time required for exhaustive key search

Key Size (bits)	Number of Alternative Keys	Time required at 1 encryption/ $\mu$ s	Time required at $10^6$ encryptions/ $\mu$ s
32	$2^{32} = 4.3 \times 10^9$	$2^{31} \mu\text{s} = 35.8 \text{ minutes}$	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	$2^{55} \mu\text{s} = 1142 \text{ years}$	10.01 hours
128	$2^{128} = 3.4 \times 10^{38}$	$2^{127} \mu\text{s} = 5.4 \times 10^{24} \text{ years}$	$5.4 \times 10^{18} \text{ years}$
168	$2^{168} = 3.7 \times 10^{50}$	$2^{167} \mu\text{s} = 5.9 \times 10^{36} \text{ years}$	$5.9 \times 10^{30} \text{ years}$
26 characters (permutation)	$26! = 4 \times 10^{26}$	$2 \times 10^{26} \mu\text{s} = 6.4 \times 10^{12} \text{ years}$	$6.4 \times 10^6 \text{ years}$

# Attacks on encryption schemes

Type of attack	Known to cryptanalyst
Ciphertext only	<ul style="list-style-type: none"><li>■ <i>Encryption algorithm</i></li><li>■ <i>Ciphertext</i></li></ul>
Known plaintext	<ul style="list-style-type: none"><li>■ <i>Encryption algorithm</i></li><li>■ <i>One or more pairs plaintext-ciphertext</i></li></ul>
Chosen plaintext	<ul style="list-style-type: none"><li>■ <i>Encryption algorithm</i></li><li>■ <i>One or more pairs plaintext-ciphertext, with the plaintext chosen by the attacker</i></li></ul>
Chosen ciphertext	<ul style="list-style-type: none"><li>■ <i>Encryption algorithm</i></li><li>■ <i>Several pairs plaintext-ciphertext, ciphertext chosen by the attacker</i></li></ul>

# How secure is secure?

Evaluating the security of a system is a crucial and most difficult task

- **Unconditionally secure system**
  - If the ciphertext does not contain enough information to determine uniquely the corresponding plaintext: any plaintext may be mapped into that ciphertext with a suitable key
  - Consequently, the attacker cannot find the plaintext regardless of how much time and computational power he has because the information is not there!
  - **Bad news:** only one known system has this property: one-time pad
- **Conditional or Complexity-theoretic security**
  - Consider a model of computation (e.g., Turing machine) and adversaries modeled as having polynomial computational power
  - Consider the weakest possible assumptions and the strongest possible attacker and do worst-case or at least average-case analysis

# One-Time pad

- **Idea:** use a (truly) random key as long as the plaintext – change the key for every plaintext
- It is unbreakable since the ciphertext bears no statistical relationship to the plaintext
- Moreover, for any plaintext & any ciphertext there exists a key mapping
  - Thus, a ciphertext can be decrypted to any plaintext of the same length
  - The cryptanalyst is in an impossible situation

# Security of the one-time pad

- The security is entirely given by the randomness of the key
  - If the key is truly random, then the ciphertext is random
  - A key can only be used once if the cryptanalyst is to be kept in the “dark”
- Problems with this “perfect” cryptosystem
  - Making large quantities of truly random characters is a significant task
- Key distribution is enormously difficult: for any message to be sent, a key of equal length must be available to both parties

# How secure is secure?

Evaluating the security of a system is a crucial and most difficult task

- **Unconditionally secure system**
  - If the ciphertext does not contain enough information to determine uniquely the corresponding plaintext: any plaintext may be mapped into that ciphertext with a suitable key
  - Consequently, the attacker cannot find the plaintext regardless of how much time and computational power he has because the information is not there!
  - Bad news: only one known system has this property: one-time pad
- **Conditional or Complexity-theoretic security**
  - Consider a model of computation (e.g., Turing machine) and adversaries modeled as having polynomial computational power
  - Consider the weakest possible assumptions and the strongest possible attacker and do worst-case or at least average-case analysis

# How secure is secure?

- **Provable security**
  - Prove that breaking the system is equivalent with solving a supposedly difficult (math) problem (e.g., from Number Theory)
- **Computationally secure**
  - The cost of breaking the system exceeds the value of the encrypted information
  - The time required to break the system exceeds the useful lifetime of the information

# Cryptography – some notations

- Notation for relating the plaintext ( $P$ ), ciphertext ( $C$ ), the key ( $K$ ), encryption algo.  $E()$  and decryption algo.  $D()$ 
  - $C=E_K(P)$  denotes that  $C$  is the encryption of the plaintext  $P$  using the key  $K$
  - $P=D_K(C)$  denotes that  $P$  is the decryption of the ciphertext  $C$  using the key  $K$
  - Then  $D_K(E_K(P))=P$

# Symmetric Key Algorithms

- Historic ciphers – Caesar, shift, mono alphabetic, Playfair, Hill, Autokey, polyalphabetic, Rail fence, Affine
- Stream Ciphers and Block Ciphers
- DES, Double DES, Triple DES,
- AES
- RC4, RC6
- RSA, Deffie-Hellman, ECC
- Hash functions....

# Software Security

## Introduction

## Goals of this course

- **How** does security fails in software?
- **Why** does software often fail?  
What are the underlying root causes?
- **What** are ways to make software more secure?

Focus more on defence than on offense

# Practicalities: prerequisites

- Introductory security course
  - CIA (Confidentiality, Integrity, Availability), Authentication, ...
- Basic programming skills, in particular
  - C(++) or assembly/machine code
    - eg., `malloc()`, `free()`, `*(p++)`, `&x`, strings in C using `char*`
  - Java or some other typed OO language
    - eg. `public`, `final`, `private`, `protected`, `Exceptions`
  - bits of PHP and JavaScript

## The kind of C++ code you will see later

```
char* copy_and_print(char* string)  {
    char* b = malloc(strlen(string));
    strcpy(b, string); // copy string to b
    printf("The string is %s.", b);
    free(b);
    return b;
}

int sum_using_pointer_arithmetic(int a[])  {
    int sum = 0;
    int *pointer = a;
    for (int i=0; i<4; i++) {
        sum = sum + *pointer;
        pointer++;
    }
    return sum;
}
```

## The kind of Java code you will see

```
public int sumOfArray(int[] pin)
    throws NullPointerException,
           ArrayIndexOutOfBoundsException
{
    int sum = 0;
    for (int i=0; i<4; i++ )
    {
        sum = sum + a[i];
    }
    return sum;
}
```

## The kind of OO Java code you will see

```
final class A implements Serializable {  
    public final static int SOME_CONSTANT = 2;  
    private B b1;  
    public B b2;  
  
    protected A ShallowClone(Object o)  
        throws ClassCastException {  
        a = new A();  
        x.b1 = (A) o.b1; // cast o to class A  
        x.b2 = (A) o.b2;  
        return a;  
    }  
}
```

# Will Discuss the following

- What is "software security"?
- The problem of software insecurity
- The causes of the problem
- The solution to the problem
- Security concepts

# Motivation

# Quiz

*Why can websites, servers, browsers, laptops, mobile phones, wifi access points, network routers, cars, pacemakers, the electricity grid, uranium enrichment facilities, ... be hacked?*

Because they contain

**Software**

# Why a course on software security?

- Software is a MAJOR source of security problems and plays MAJOR role in providing security
- Software security does not get much attention
  - in other security courses, or
  - in programming courses,or indeed, in much of the security literature!

# How do computer systems get hacked?

By attacking

- software
- humans



Or: the interaction between software & humans

# Fairy tales

Many discussions about security begin with Alice and Bob



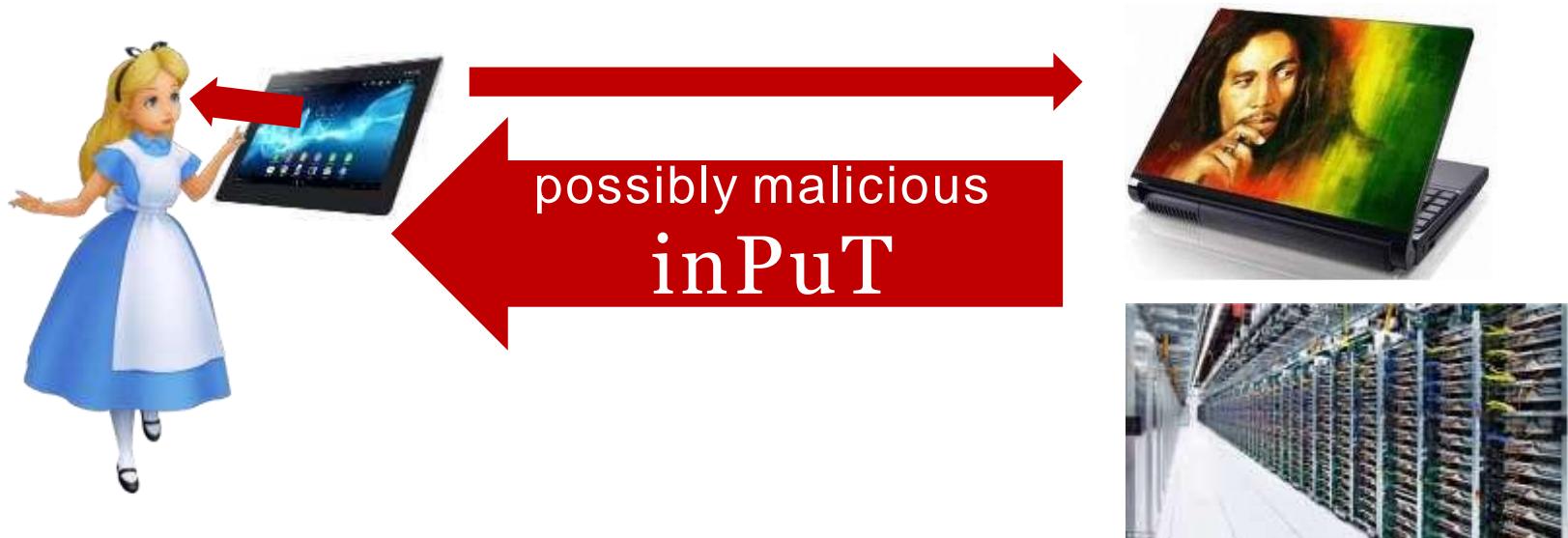
How can Alice communicate securely with Bob,  
when Eve can modify or eavesdrop on the communication?

**Solution?**

This is an interesting  
problem,  
**but it is not the biggest problem**

# The *really big* problem

Alice & her computer are communicating with *another computer*



How to prevent Alice's computer from getting *hacked*?

Or how to detect this? And then react ?

Solving earlier problem (securing the communication) does not help!

# The problem

25<sup>th</sup> January 2003, 5:29 AM

Map Source : [www.visualroute.com](http://www.visualroute.com)



Sat Jan 25 05:29:00 2003 (UTC)

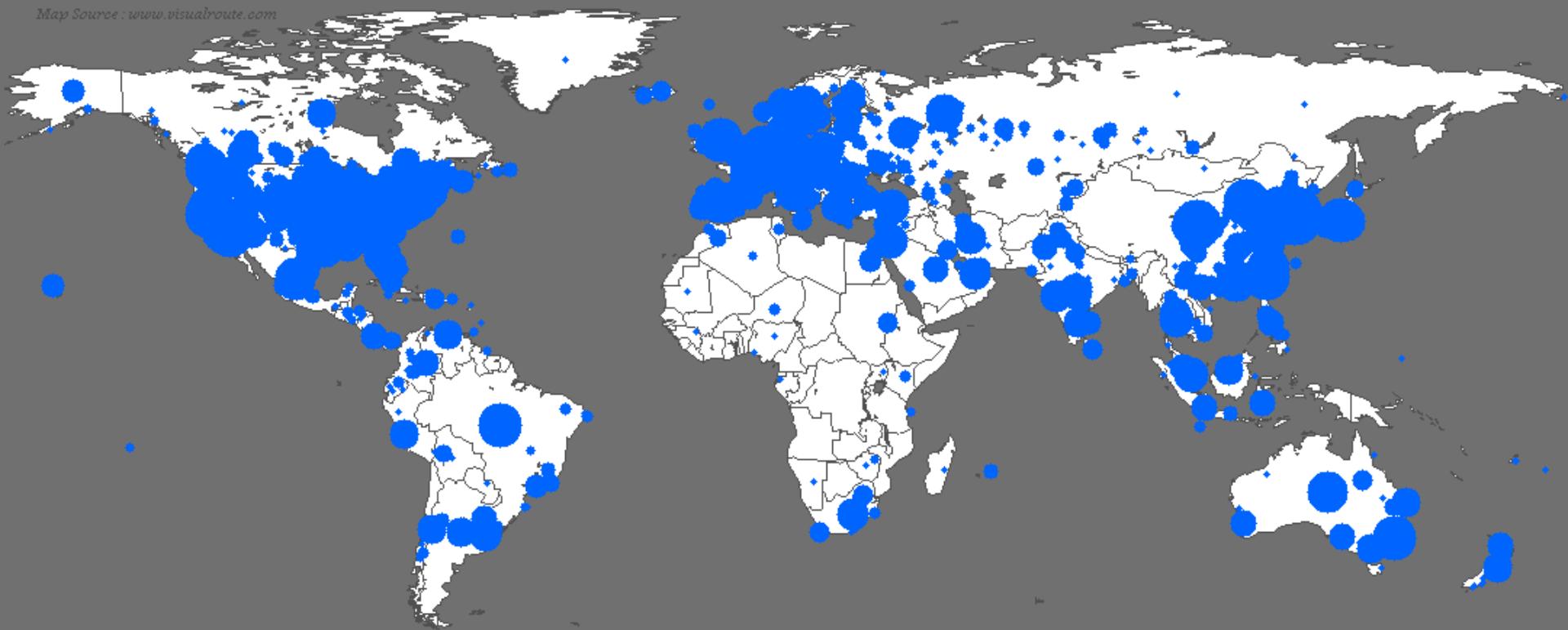
Number of hosts infected with Sapphire: 0

<http://www.caida.org>

Copyright (C) 2003 UC Regents

# 25<sup>th</sup> January 2003, 6:00 AM

Map Source : [www.visualroute.com](http://www.visualroute.com)



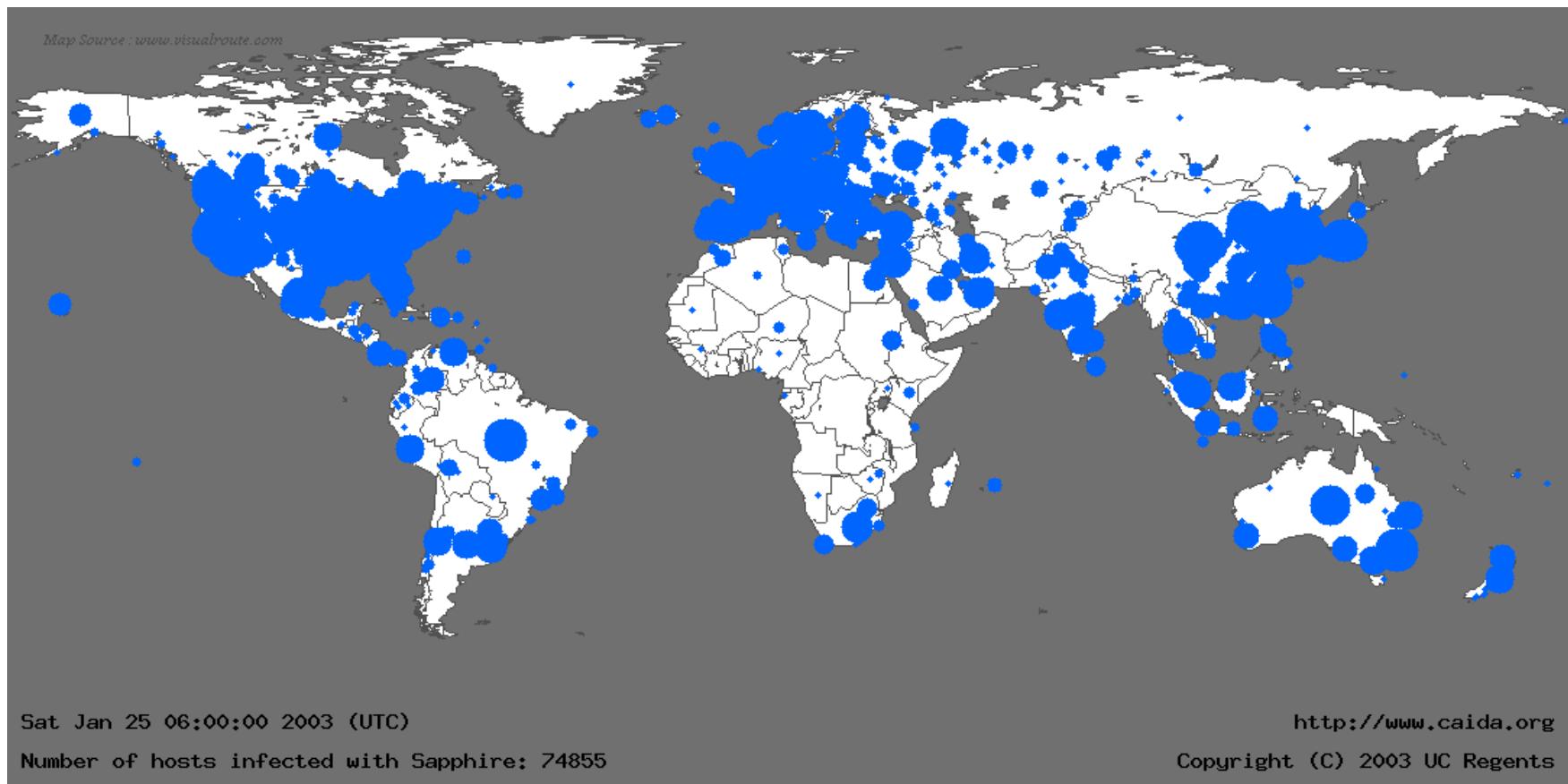
Sat Jan 25 06:00:00 2003 (UTC)

Number of hosts infected with Sapphire: 74855

<http://www.caida.org>

Copyright (C) 2003 UC Regents

# Slammer Worm



From *The Spread of the Sapphire/Slammer Worm*, by David Moore et al.

# Security problems nowadays

To get an impression of the problem, have a look at

US-CERT bulletins

<https://us-cert.cisa.gov/ncas/bulletins>

CVE (Common Vulnerability Enumeration)

<https://cve.mitre.org/cve/>

NIST's vulnerability database

<https://nvd.nist.gov/vuln/search>

# Changing nature of attackers

Traditionally, hackers were **motivated by ‘fun’**

- by **script kiddies & more skilled hobbyists**

Nowadays hackers are **professional:**

- **cyber criminals**

with lots of money & (hired) expertise

Important game changers: **ransomware & bitcoin**

## What motivates the cyber criminals?



# 6 Most Expensive Cyber Attacks in History

1. ExPetr / NotPetya (2017) : \$10 Billion
2. Epsilon (2011) : \$4 Billion
3. Mafiaboy Attack (2000) : \$1 Billion
4. Veterans Administration (2006) : \$500 Million
5. Hannaford Bros (2007) : \$252 Million
6. Sony PlayStation (2011) : \$171 Million

Most prominent way to avoid Cyber Attacks?

Hire them or motivate them to work for you  
or motivate good people to replace them

# Zerodium

A famous premium bounties reward company Zerodium is an American information security company founded in 2015, by cybersecurity experts having experience in vulnerability research and zero-day exploits.

A **zero-day exploit** is a software vulnerability discovered by attackers before the vendor has become aware of it.

## How much reward?

# Some new terms: attack surface

[Remote Code Execution \(RCE\)](#) and [Local Privilege Escalation](#): Sometimes servers have internal vulnerabilities- RCE allows an attacker to discover and exploit these vulnerabilities, escalating privileges and gaining access to connected systems.

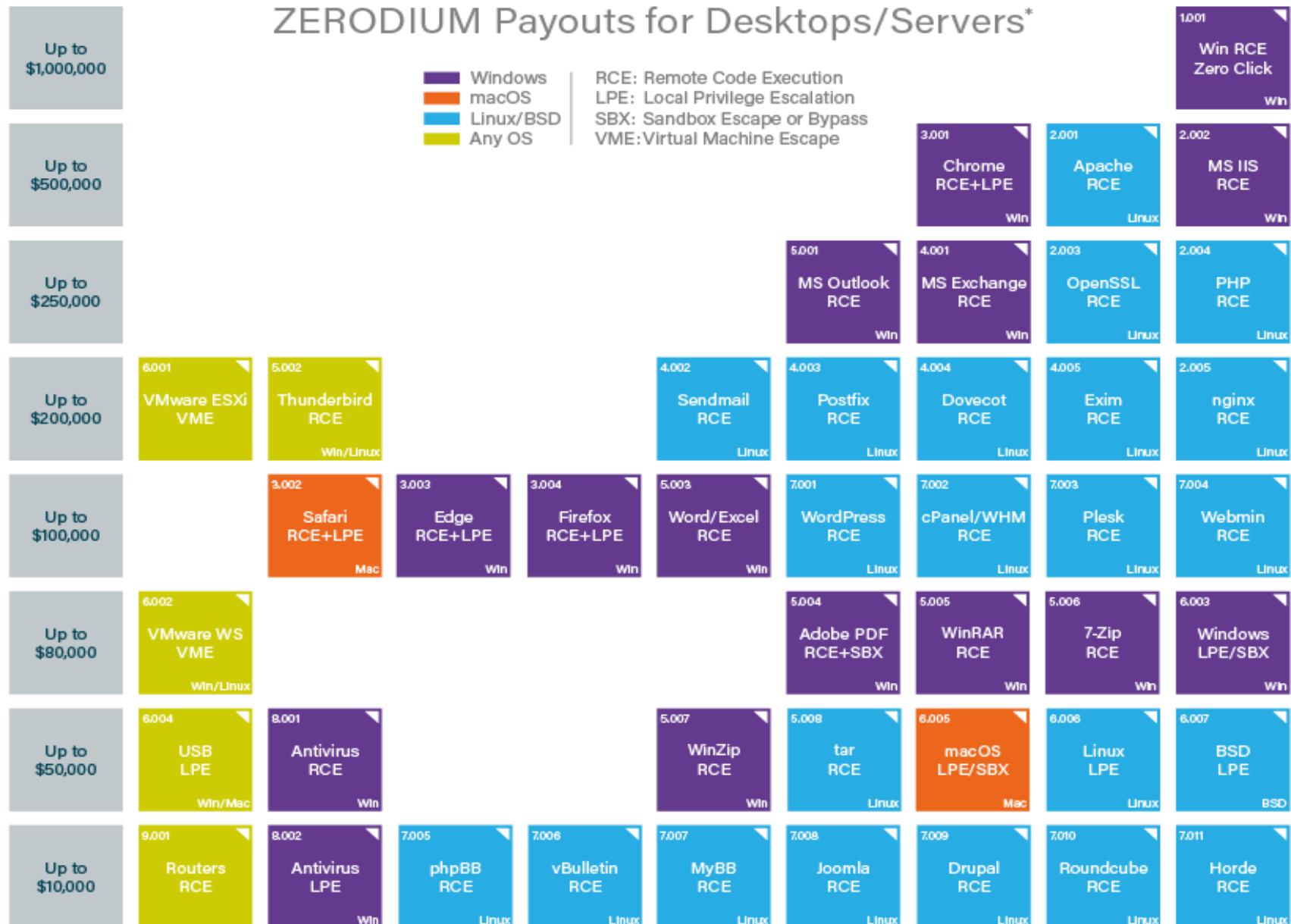
[Virtual machine escape](#) is an exploit in which the attacker runs code on a VM that allows an operating system running within it to break out and interact directly with the hypervisor.

A [hypervisor](#) is a software that you can use to run multiple VMs on a single physical machine

A [sandbox](#) is an isolated testing environment that enables users to run programs or open files without affecting the application, system or platform on which they run

[Sandbox escape](#) refers to the act of exploiting a software vulnerability to break out of a secure or quarantined environment, often called a sandbox.

# Prices for 0days



\* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.

# Prices for 0days

## ZERODIUM Payouts for Mobiles\*

Up to  
\$2,500,000

Up to  
\$2,000,000

Up to  
\$1,500,000

Up to  
\$1,000,000

Up to  
\$500,000

Up to  
\$200,000

Up to  
\$100,000

FCP: Full Chain with Persistence  
RCE: Remote Code Execution  
LPE: Local Privilege Escalation  
SBX: Sandbox Escape or Bypass

iOS  
Android  
Any OS

1.001  
Android FCP  
Zero Click  
Android

1.002  
iOS FCP  
Zero Click  
iOS

2.001  
WhatsApp  
RCE+LPE  
Zero Click  
iOS/Android

2.002  
iMessage  
RCE+LPE  
Zero Click  
iOS

2.003  
WhatsApp  
RCE+LPE  
iOS/Android

2.004  
SMS/MMS  
RCE+LPE  
iOS/Android

3.001  
Persistence  
iOS

2.005  
WeChat  
RCE+LPE  
iOS/Android

2.006  
iMessage  
RCE+LPE  
iOS

2.007  
FB Messenger  
RCE+LPE  
iOS/Android

2.008  
Signal  
RCE+LPE  
iOS/Android

2.009  
Telegram  
RCE+LPE  
iOS/Android

2.010  
Email App  
RCE+LPE  
iOS/Android

4.001  
Chrome  
RCE+LPE  
Android

4.002  
Safari  
RCE+LPE  
iOS

5.001  
Baseband  
RCE+LPE  
iOS/Android

6.001  
LPE to  
Kernel /Root  
iOS/Android

2.011  
Media Files  
RCE+LPE  
iOS/Android

2.012  
Documents  
RCE+LPE  
iOS/Android

4.003  
SBX  
for Chrome  
Android

4.004  
Chrome RCE  
w/o SBX  
Android

4.005  
SBX  
for Safari  
iOS

4.006  
Safari RCE  
w/o SBX  
iOS

7.001  
Code Signing  
Bypass  
iOS/Android

5.002  
WiFi  
RCE  
iOS/Android

5.003  
RCE  
via MitM  
iOS/Android

6.002  
LPE to  
System  
Android

8.001  
Information  
Disclosure  
iOS/Android

8.002  
[k]ASLR  
Bypass  
iOS/Android

9.001  
PIN  
Bypass  
Android

9.002  
Passcode  
Bypass  
iOS

9.003  
Touch ID  
Bypass  
iOS

\* All payouts are subject to change or cancellation without notice. All trademarks are the property of their respective owners.

# Apple & Google payouts

## Google Offers \$1.5M Bug Bounty for Android 13 Beta

The security vulnerability payout set bug hunters rejoicing, but claiming the reward is much, much easier said than done.



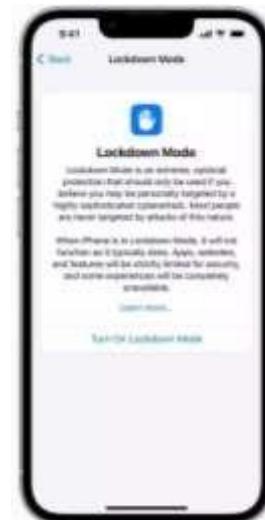
Tara Seals

Managing Editor, News, Dark Reading

May 02, 2022

## Apple will pay you \$2 million if you can break its new 'Lockdown Mode'

By Joe Wituschek published July 07, 2022



# Software security: crucial facts

- *There are no silver bullets!*

Firewalls, crypto, or special security features do not magically solve all problems

- “if you think your problem can be solved by cryptography, you do not understand cryptography and you do not understand your problem” [Bruce Schneier]

- Security is emergent property of entire system

- like quality

- Security should be - but hardly ever is - integral part of the design, right from the start

# security software ≠ software security

Adding **security software** can make a system more secure,  
i.e., **software specifically for security**, such as

- **SSL/TLS, IPSEC, firewall, VPN, ...**
- **AV (AntiVirus), WAF (Web Application Firewall)**
- **access control**, with eg **2FA, logging, monitoring, ...**
- **NIDS (Network Intrusion Detection System)**
- **EDR (Endpoint Detection and Response)**
- **RASP (Runtime Application Self-Protection)**
- ...

# security software ≠ software security

Adding security software can make a system more secure

i.e. software specifically for security, such as

- TLS, IPSEC, firewall, VPN, ...
- AV (AntiVirus), WAF (Web Application Firewall)
- access control, with eg 2FA, logging, monitoring, ...
- NIDS (Network Intrusion Detection System)
- EDR (Endpoint Detection and Response)
- RASP (Runtime Application Self-Protection)
- ...

But all software must be secure, including security software

- Buffer overflow (in your PDF viewer) can still be exploited...
- Adding security software may *add* software bugs and make things less secure:

Check out <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=firewall>

<https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=VPN>

# Root causes

# More root causes: security vs functionality

Primary goal of software is providing functionality & services  
Managing associated risks is a secondary concern

- There is often a trade-off/conflict between
  - security
  - functionality, convenience, speed , ...where **security typically loses out**
- Users are likely to complain about missing or broken functionality, but not about insecurity

# Functionality vs security: Lost battles?

- Operating systems (OSs)
  - with huge OS, with huge attack surface
- Programming languages
  - with easy to use, efficient, but very insecure and error-prone mechanisms
- Web browsers
  - with JavaScript, and Web APIs to access microphone, web cam, location, ...
- Email clients
  - which handle with all sorts of formats & attachments

# More root causes: Weakness in depth

***complex input languages***, for

*interpretable or executable* input, eg  
pathnames, XML, JSON, jpeg, mpeg, xls, pdf...

MALICIOUS  
INPUT

***programming languages***

INPUT  
web browser  
with plugins

INPUT  
application

INPUT  
middleware

INPUT  
platform  
eg Java, .NET  
or JavaScript VM

INPUT  
libraries

INPUT  
SQL  
data  
base

INPUT  
operating system

INPUT  
system APIs

INPUT  
hardware (incl network card & peripherals)

# Weakness in depth

## Software

- runs on a **huge, complicated infrastructure**
  - HW, OS, platforms, web browser, lots of libraries & APIs, ...
- is built using **complicated languages**
  - *programming* languages  
and *input* languages (SQL, HTML, XML, mp4, ...)
- using various **tools**
  - compilers, IDEs, pre-processors, dynamic code downloads

All of these may have **security holes**, or may **make the introduction of security holes very easy & likely**

# Types of software security problems

# Weaknesses vs vulnerabilities

1. (potential) security weaknesses (an application error or bug)

Things that could go wrong & could be better

2. (real) security vulnerabilities (Weakness may escalate to a vulnerability )

Flaws that can actually be exploited by an attacker

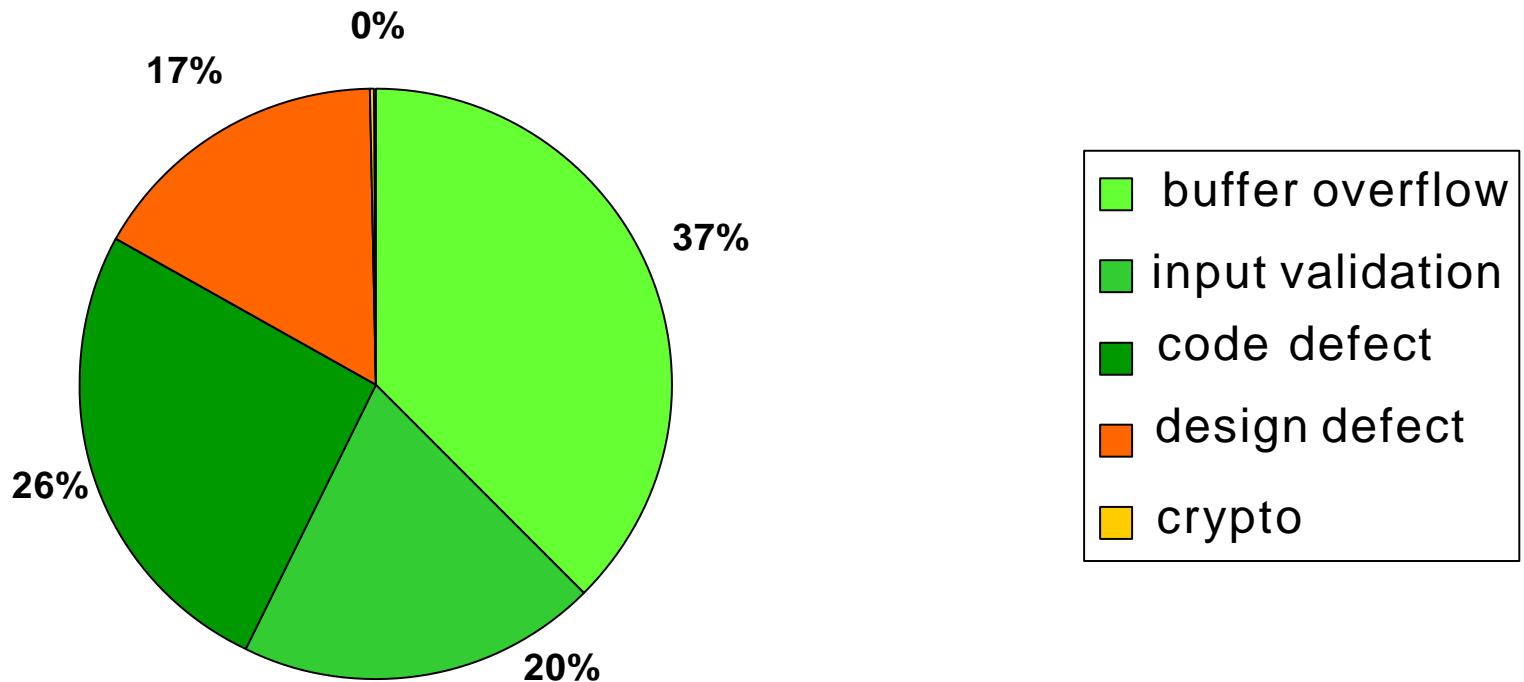
Requires flaw to be

accessible: attacker has to be able to get at it

exploitable: attacker has to be able to do some damage with it

Eg: by turning off Wifi and BlueTooth on my laptop, many vulnerabilities become weaknesses

# Typical software security flaws



SQL Slammer is a 2003 computer worm (exploited a MS SQL Server 2000 vulnerability) that caused a DoS on some Internet hosts and dramatically slowed general Internet traffic and crashing routers all around the world. It spread rapidly, infecting most of its 75,000 victims within 10 minutes.

# 'Levels' at which security flaws can arise

1. Design flaws  
introduced *before* coding
2. Implementation flaws aka bugs aka code-level defects  
introduced *during* coding

*As a rule of thumb, coding & design flaws equally common*

Vulnerabilities can also arise on other levels

3. Configuration flaws
4. Unforeseen consequences of the *intended functionality*

# Types of implementation flaws

## 2a. Flaws that can be understood by looking at program itself

Eg. typos, < instead of <= ..., mistake in the program logic with wrongly nested if-statements, ...

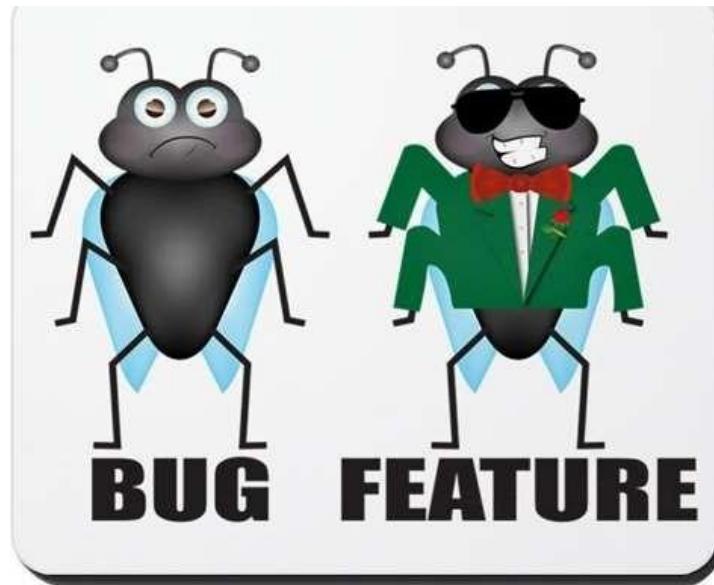
## 2b. Problems in the interaction with the underlying platform or other systems and services, eg

- memory corruption in C(++) code
- SQL injection in program that uses an SQL database
- XSS, CSRF, SSI, XXE, .... in web-applications
- Deserialisation attacks in many programming languages
- ...

# Bug vs features, yet again

Attacks can not only exploit **bugs**, but also **features**

Eg: SQL injection uses a **feature** of the back-end database



A **bug** is an unexpected problem with software or hardware.

# The depressing state of software security

The *bad* news

people keep making the same mistakes

The *good* news

people keep making the same mistakes

..... so we can do something about it!

“Every upside has its downside” [Johan Cruijff]

# Spot the security flaws!

```
int balance;
```

<= should be >=

```
void decreaseBankBalance(int amount)
{ if (balance <= amount)
    { balance = balance - amount; }
else { println("Insufficient funds\n"); }
}
```

what if amount  
is negative?

```
void increaseBankBalance(int amount)
{ balance = balance + amount;
}
```

what if this sum is too  
large for an int?

# Different kinds of implementation flaws

what if amount  
is negative?

## 1. Lack of input validation

Maybe this is a design flaw? We could decide not use signed integers.

Root cause: implicit assumption

<= should be >=

## 2. Logic error

what if sum is too  
large for a 32 bit int?

## 3. Problem in interaction with underlying platform

'Lower level' than the flaws above

Root cause: broken abstraction

# How can we make software secure?

We do *not* know how to do this!

Even if we formally verify software, we may

- miss security properties that need to be verified
- make implicit assumptions
- overlook attack vectors
- ...

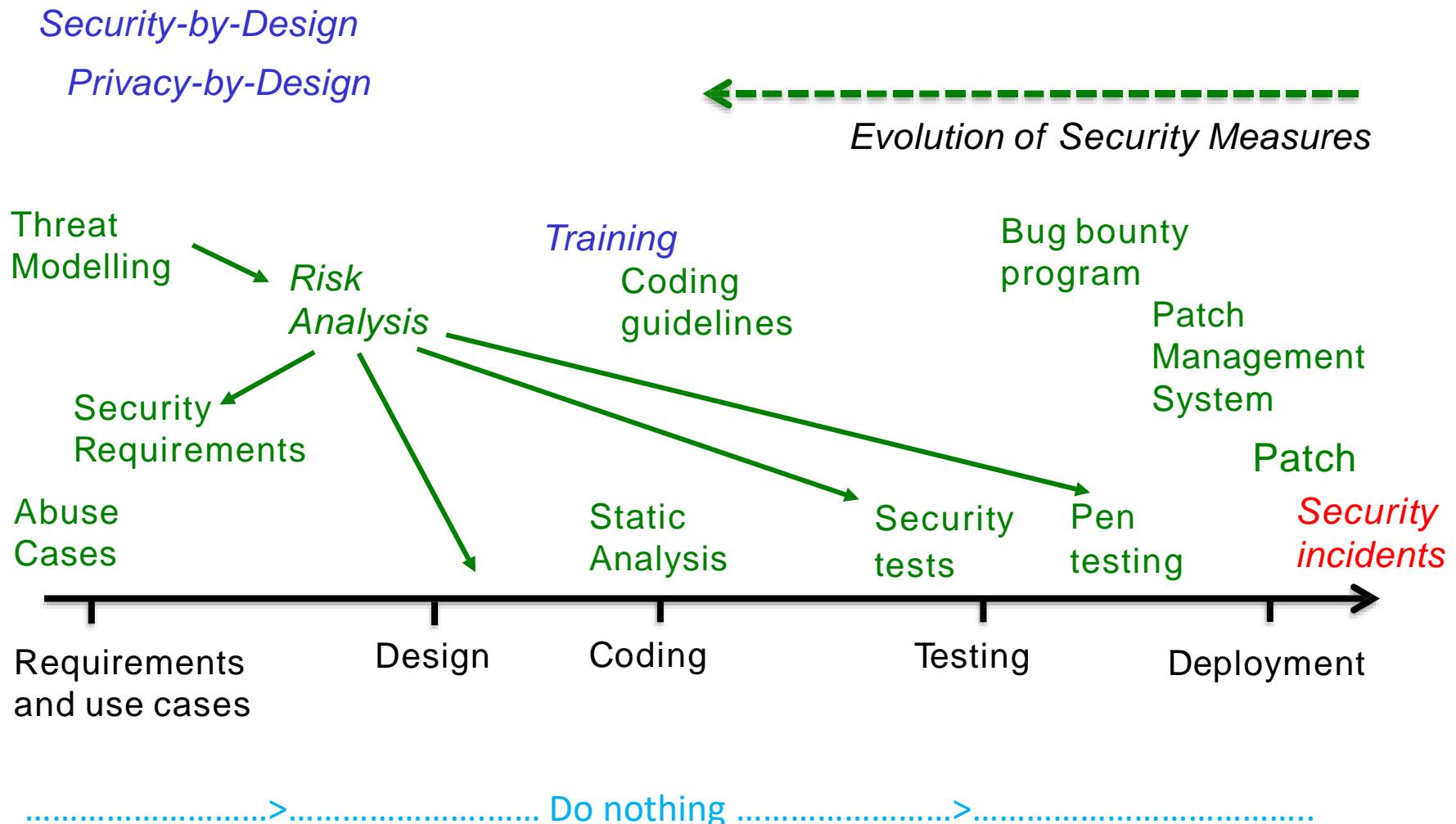
# How can we make software more secure?

We *do* know how to do this!

- Knowledge about standard mistakes is crucial
  - These depends on programming language, “platform”, APIs/technologies used, type of application
  - There is LOTS of information available on this nowadays
- But this is not enough: security to be taken into account from the start, *throughout* the software development life cycle
  - Several ideas, best practices, methodologies to do this

# Security in the Software Development Life Cycle (SDLC)

# Security in Software Development Lifecycle



**pen test:** is an authorized simulated attack

**abuse cases** describe how users **misuse** or exploit the weaknesses

# “Shifting left”

Organisations always begin tackling security at the *end* of the SDLC, and then slowly evolve to tackle it earlier

1. First, **do nothing**
2. Some security issue is discovered:
  - a) Still **do nothing**, if there's no (economic) incentive
  - b) Or: **patch**
3. If this happens often: **update mechanism** for **regular patching**
4. **Do security testing**: eg. **hire pen-testers** or **bug bounty program**
5. **Use static analysis** tools when coding
6. **Give security training** to programmers
7. Think of **abuse cases**, and develop **security tests** for them
8. Think about security *before* you start coding, eg with **security architecture review**
9. ...

# DAST, SAST

Security people keep inventing 4 letter new acronyms

- DAST
  - Dynamic Application Security Testing
  - ie. testing
- SAST
  - Static Application Security Testing
  - ie. static analysis
- IAST
  - Interactive Application Security Testing
  - manual pen-testing
- RASP
  - Run-time Application Security Protection
  - ie. monitoring

# Methodologies for secure software development

- Microsoft SDL
  - with extension for Secure DevOps (DevSecOps)
- BSIMM (Building Security In Maturity Model)
- Open SAMM (Software Assurance Maturity Model)
- Gary McGraw's Touchpoints
- OWASP SAMM (Open Worldwide Application Sec. Proj.)
- Grip op SSD (Secure Software Development)
  - Ongoing initiative by Dutch government organisations
  - <https://www.cip-overheid.nl/en/category/products/secure-software/>
- ...

These come with best practices, checklists, methods for assessments, roadmaps for improvement, ...

# Microsoft's SDL Optimization Model

## The four security maturity levels of the SDL Optimization Model

### Basic

*Security is reactive*

*Customer risk is undefined*

### Standardized

*Security is proactive*

*Customer risk is understood*

### Advanced

*Security is integrated*

*Customer risk is controlled*

### Dynamic

*Security is specialized*

*Customer risk is minimized*

## The five capability areas of the software development process

Training, Policy, and Organizational Capabilities

Requirements and Design

Implementation

Verification

Release and Response



Introduction



Self-assessment guide



Implementer's guide  
Basic→Standardized



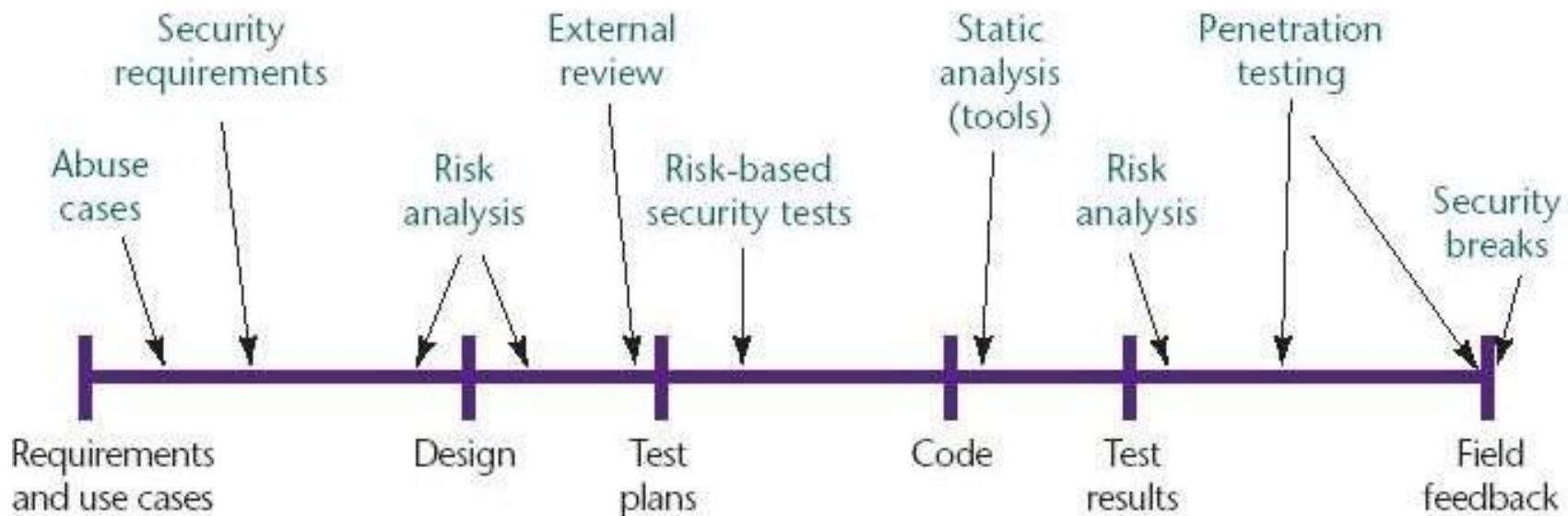
Implementer's guide  
Standardized→Advanced



Implementer's guide  
Advanced→Dynamic

# Security in the software development life cycle

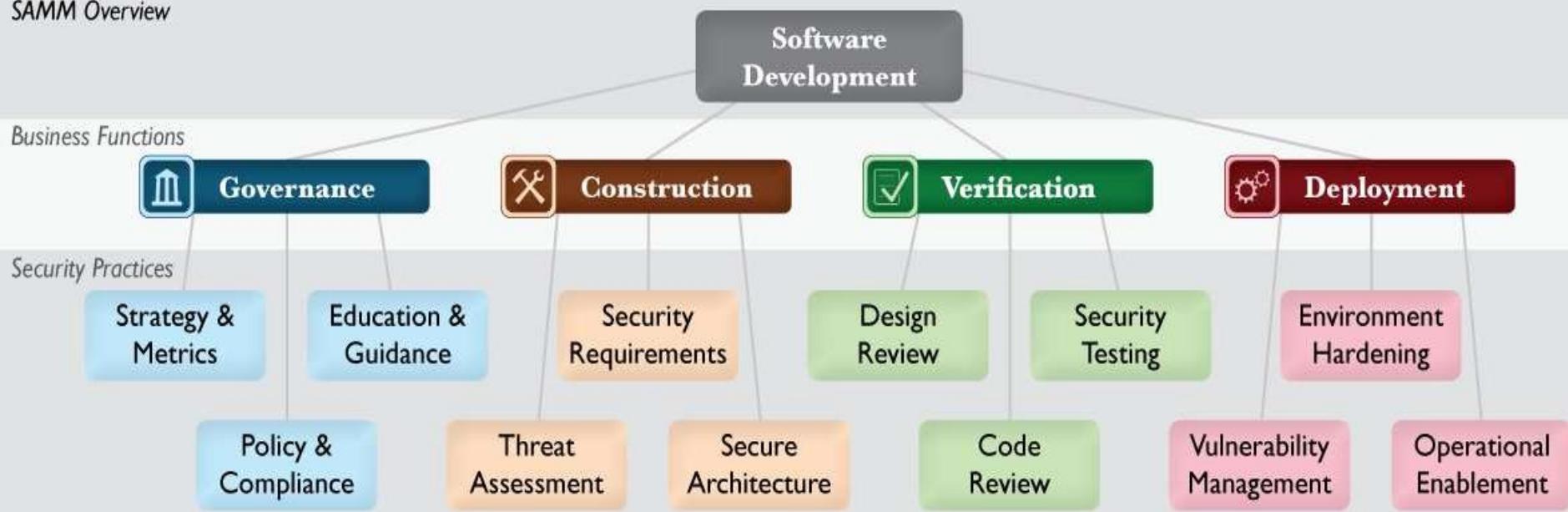
## McGraw's Touchpoints



[Source: Gary McGraw, *Software security*, Security & Privacy Magazine, IEEE, Vol 2, No. 2, pp. 80-83, 2004. ]

With 4 business functions and 12 security practices

## SAMM Overview



# BSIMM (Building Security In Maturity Model)

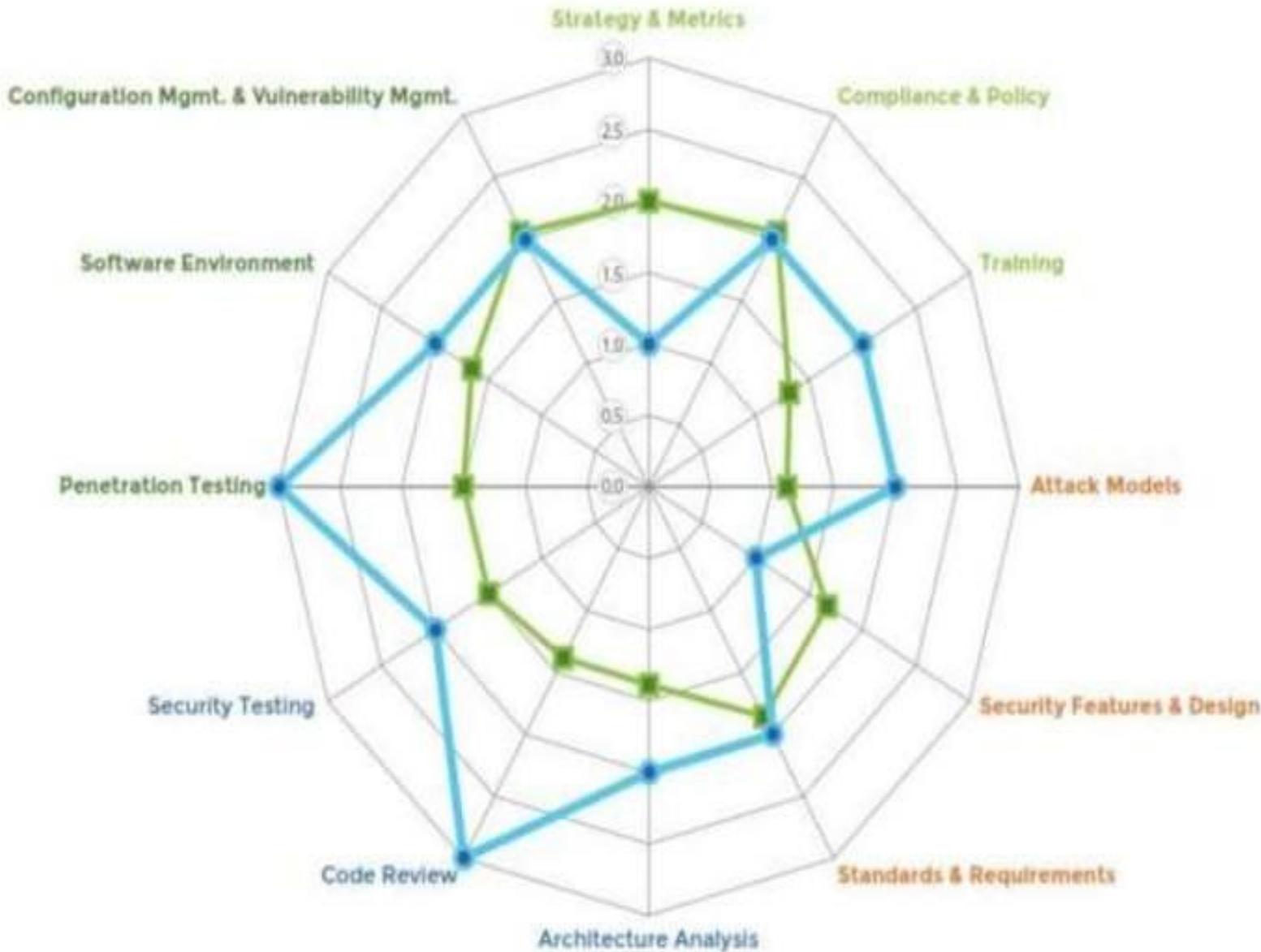
Framework to compare your software security efforts with other organisations

Governance	Intelligence	SSDL Touchpoints	Deployment
Strategy and Metrics	Attack Models	Architecture Analysis	Penetration Testing
Compliance and Policy	Security Features and Design	Code Review	Software Environment
Training	Standards and Requirements	Security Testing	Configuration Management and Vulnerability Management

Based on data collected from large enterprises

<https://www.bsimm.com/framework/>

# BSIMM: comparing your security maturity



But first...

Crucial first step in any security discussion!

1. What are your **security requirements**?

*What does it mean for the system to be secure?*

2. What is your **attacker model**?

*Against what does the system have to be secure?*

- Attack surface / attack vectors
- Attacker's motivations & capabilities
- What are your security assumptions ?

Any discussion of security without answering these questions is *meaningless*

Aka **threat modelling**

# Security requirements

## a) 'This application cannot be hacked'

- generic default requirement 😊
- vague & not actionable 😞
- negative security model

## b) More specific security requirements

- In terms of Confidentiality, Integrity and Availability (CIA)
- Or, usually better, in terms of Access Control
  - (i.e. Authentication & Authorisation)
- Not just Prevention but also Detection & Reaction/Response
- positive security model

# prevention vs detection & reaction



# prevention vs detection & reaction

- Prevention seems to be *the* way to ensure security, but detection & response often more important and effective
  - Eg. breaking into a house with large windows is trivial; despite this absence of prevention, detection & reaction still provides security against burglars
  - Most effective security requirement for most persons and organisations: make good back-ups, so that you can recover after an attack
- *don't ever be tempted into thinking that good prevention makes detection & reaction superfluous.*
- Hence important security requirements include
  - being able to do monitoring
  - having logs for auditing and forensics
  - having someone actually inspecting the logs
  - ...

# Pen Test

# Penetration Testing

## **Definition:**

- A penetration test or pentest is a test evaluating the strengths of all security controls on the computer system. Penetration tests evaluate procedural and operational controls as well as technological controls.

# Who needs Penetration Testing

- Banks/Financial Institutions, Government Organizations, Online Vendors, or any organization processing and storing private information
- Most certifications require or recommend that penetration tests be performed on a regular basis to ensure the security of the system.
- PCI Data Security Standard's (Cardholder's data) Section 11.3 requires organizations to perform application and penetration tests at least once a year.
- HIPAA Security Rule's section 8 of the Administrative Safeguards requires security process audits, periodic vulnerability analysis and penetration testing.

# Penetration Testing Viewpoints

- **External vs. Internal**

**Penetration Testing can be performed from the viewpoint of an external attacker or a malicious employee.**

- **Overt vs. Covert**

**Penetration Testing can be performed with (Overt) or without (Covert) the knowledge of the IT department of the company being tested.**

# Phases of Penetration Testing

- 1. Reconnaissance and Information Gathering**
- 2. Network Enumeration and Scanning**
- 3. Vulnerability Testing and Exploitation**
- 4. Reporting**

# 1. Reconnaissance and Information Gathering

**Purpose:** To discover as much information about a target (individual or organization) as possible without actually making network contact with said target.

## **Methods:**

Google search

Website browsing

Organization info discovery via WHOIS

# WHOIS Results for www.clemson.edu

<https://whois.domaintools.com/>

**Domain Name:** CLEMSON.EDU

**Registrant:**

Clemson University  
340 Computer Ct  
Anderson, SC 29625  
UNITED STATES

**Administrative Contact:**

Network Operations Center  
Clemson University  
340 Computer Court  
Anderson, SC 29625  
UNITED STATES  
(864) 656-4634  
noc@clemson.edu

**Technical Contact:**

Mike S. Marshall  
DNS Admin  
Clemson University  
Clemson University  
340 Computer Court  
Anderson, SC 29625  
UNITED STATES  
(864) 247-5381  
hubcap@clemson.edu

**Name Servers:**

EXTNS1.CLEMSON.EDU	130.127.255.252
EXTNS2.CLEMSON.EDU	130.127.255.253
EXTNS3.CLEMSON.EDU	192.42.3.5

## 2. Network Enumeration and Scanning

**Purpose:** To discover existing networks owned by a target as well as live hosts and services running on those hosts.

### **Methods:**

Scanning programs that identify live hosts, open ports, services, and other info (Nmap, autoscan)

DNS Querying

Route analysis (traceroute)

<https://nmap.online/>

# NMap Results

Nmap is used to discover hosts and services on a computer network by sending packets and analyzing the responses.

**nmap -sS 127.0.0.1**

**1**

**2**

**3 Starting Nmap 4.01 at 2006-07-06 17:23 BST**

**4 Interesting ports on chaos (127.0.0.1):**

**5 (The 1668 ports scanned but not shown below are in state: closed)**

**6 PORT STATE SERVICE**

**7 21/tcp open ftp**

**8 22/tcp open ssh**

**9 631/tcp open ipp**

**10 6000/tcp open X11**

**11**

**12 Nmap finished: 1 IP address (1 host up) scanned in 0.207**

**13 seconds**

# State

Open port - there is a service listening from this port and is not blocked by the firewall.

Closed - there is no service listening for connection on that port.

Filtered - something (firewall, network issue, or filter) blocking connection to that port

# 3. Vulnerability Testing and Exploitation

**Purpose:** To check hosts for known vulnerabilities and to see if they are exploitable, as well as to assess the potential severity of said vulnerabilities.

## Methods:

Remote vulnerability scanning (Nessus, **OpenVAS**)

Active exploitation testing

    Login checking and bruteforcing

    Vulnerability exploitation (Metasploit, Core Impact)

    0day and exploit discovery (Fuzzing, program analysis)

    Post exploitation techniques to assess severity (permission levels, backdoors, rootkits, etc)

<https://hostedscan.com/openvas-vulnerability-scan>

# 4. Reporting

**Purpose:** To organize and document information found during the reconnaissance, network scanning, and vulnerability testing phases of a pentest.

## Methods:

Documentation tools (Dradis)

- Framework helps you manage information security projects.
- Organizes information by hosts, services, identified hazards and risks, recommendations to fix problems

# Website Scanner- PentesterTools

<https://pentest-tools.com/website-vulnerability-scanning/website-scanner>

**View Reports**

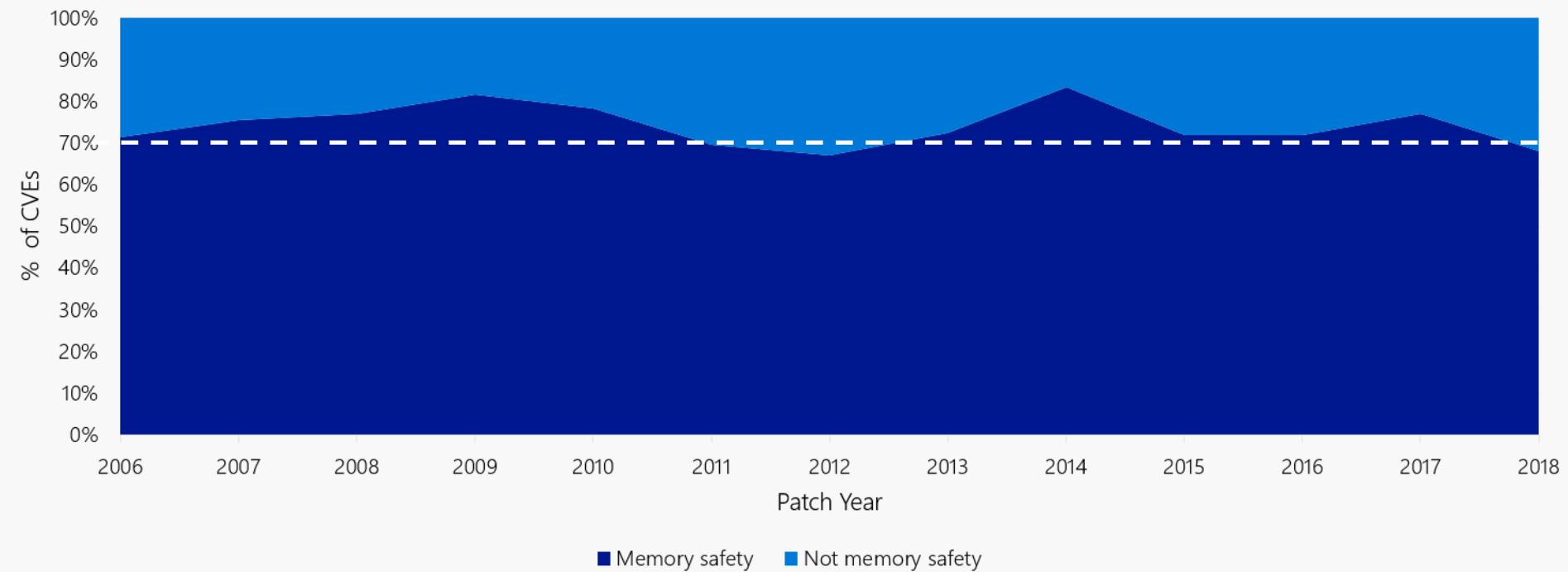
# How to Become a Penetration Tester

- Stay up to date on recent developments in computer security, reading newsletters and security reports are a good way to do this.
- Becoming proficient with C/C++ and a scripting language such as PEARL or Java Script
- Microsoft, Cisco, and Novell certifications
- Penetration Testing Certifications
  - Certified Ethical Hacker (CEH)
  - GIAC Certified Penetration Tester (GPEN)



Software Security  
**Memory corruption**

## Memory corruption bugs vs rest - Microsoft 2006-2018



[Source: <https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code> and “*Trends, challenge, and shifts in software vulnerability mitigation*”, presentation by Matt Miller at BlueHat IL 2019]

70% of high severity & critical security bugs are memory unsafety problems

# Essence of the problem

Suppose in a C program you have an array of length 4

```
char buffer[4];
```

What happens if the statement below is executed?

```
buffer[4] = 'a';
```

We don't know!

This is defined to be **undefined**

*ANYTHING* can happen

undefined behaviour: anything can happen



**undefined** behaviour: nothing may happen

# Anything attackers wants?

```
char buffer[4];  
buffer[4] = 'a';
```

If the attacker controls the value 'a'

then anything that the attacker wants may happen ...

- If we are *lucky* : program crashes with **SEGMENTATION FAULT**
- If we are *unlucky* : program does not crash  
but silently allows **data corruption** or **remote code execution (RCE)**  
and we *won't* know till it's too late

## Nothing may happen

```
char buffer[4];  
buffer[4] = 'a';
```

A compiler could remove the assignment above,  
ie. **do nothing**

- Compilers actually do this (as part of optimisation) and this can cause security problems; examples later.

# Solution to this problem

- Check array bounds at runtime
  - Algol 60 proposed this back in 1960!
- Unfortunately, C and C++ have not adopted this solution.
  - Why?
  - For efficiency  
Regrettably, people often choose performance over security
- As a result, buffer overflows have been the no 1 security problem in software ever since
  - Check out CVEs mentioning buffer (or buffer%20overflow) <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer>
- Fortunately, Perl, Python, Java, C#, PHP, Javascript, and Visual Basic do check array bounds

## Tony Hoare on design principles of ALGOL 60



In his Turing Award lecture in 1980

“The first principle was *security* : ... every subscript was checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished an option to switch off these checks in the interests of efficiency. Unanimously, they urged us not to - they knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous.

I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.”

[C.A.R. Hoare, The Emperor’s Old Clothes, Communications of the ACM, 1980]

## More memory corruption problems

Errors with **pointers** and with **dynamic memory** (aka **the heap**)

- *Have you ever written a C(++) program that uses **pointers**?*
  - *You may encounter a program crashing.*
- *Have you even written a C(++) program that uses **dynamic memory**, ie. **malloc()** and **free()**?*
  - *You may encounter a program crashing.*

In C/C++, the programmer is responsible for **memory management** and this is very error-prone

- Technical term: C and C++ do not offer **memory-safety**

# Spot all (potential) defects

```
1000 ...
1001 void f() {
1002     char* buf, buf1, buf24;
1003     buf = malloc(100);
1004     buf[0] = 'a';           ← null dereference
1005     ...
1006     ...
1007     free(buf24);          ← potential use-after-free
1008     buf[0] = 'b';           ← if buf & buf24 are aliased
1009     ...
1010     ...
1011     free(buf);
1012     buf[0] = 'c';           ← use-after-free; buf[0] points
1013     buf1 = malloc(100);      ← to de-allocated memory
1014     buf[0] = 'd';           ← memory leak; pointer buf1
1015     ...
1016 }
```

Annotations from top to bottom:

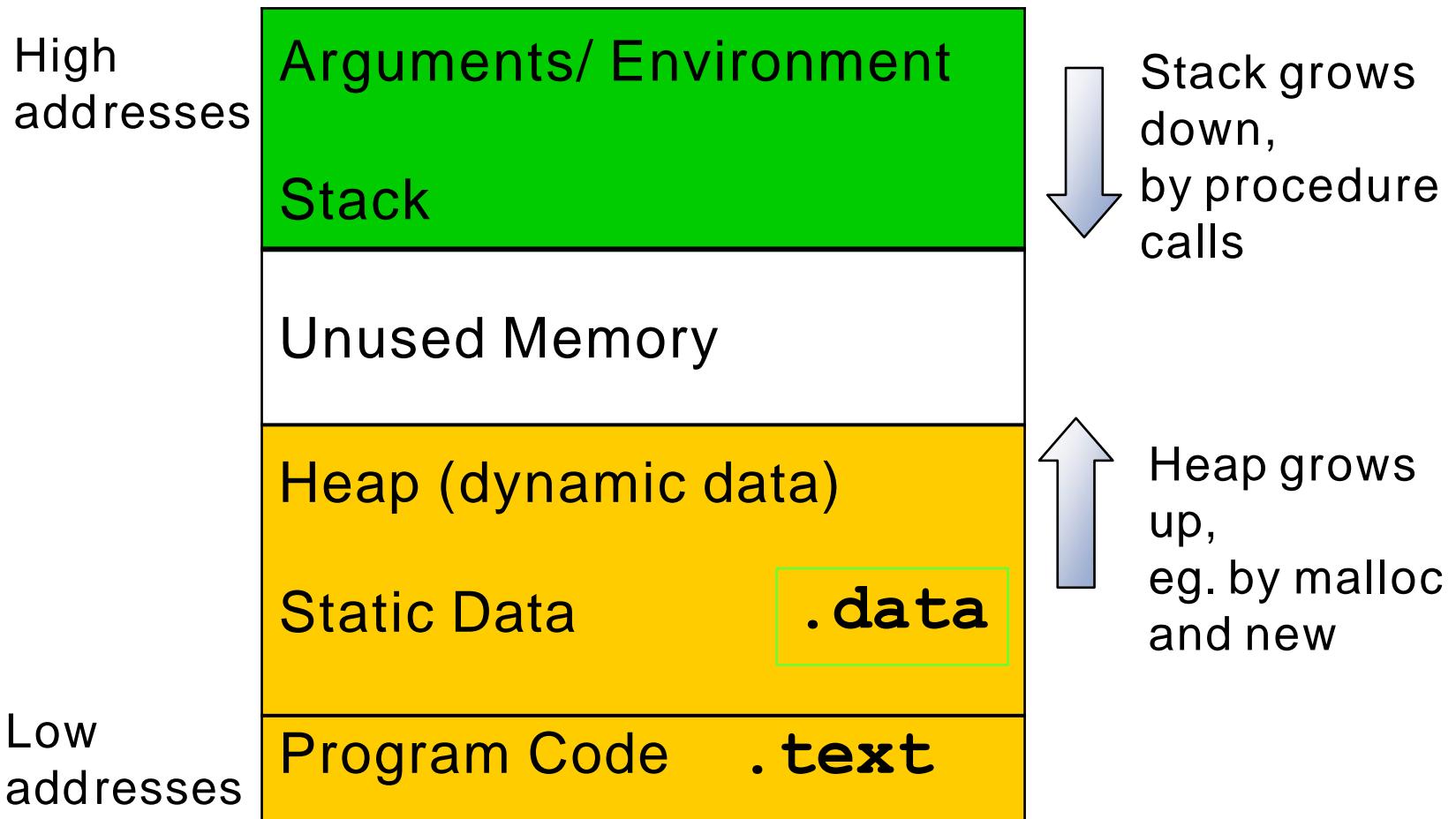
- null dereference if malloc failed
- potential use-after-free* if buf & buf24 are aliased
- use-after-free; buf[0] points to de-allocated memory
- memory leak; pointer buf1 to this memory is lost & memory is never freed
- use-after-free, but now buf[0] may point to memory that has been re-allocated for buf1

# Causes of memory corruption problems

- Access outside array bounds aka buffer overflow
  - overread or overwrite
    - overreads are not a corruption issue, but *confidentiality* issue
- Pointer trouble:
  - buggy pointer arithmetic,
  - dereferencing null pointer,
  - using a dangling pointer aka stale pointer
    - caused by e.g. use-after-free
- Memory management problems:
  - Forgetting to check for failures in allocation
  - Forgetting to de-allocate, aka memory leaks
    - not a corruption issue, but an *availability* issue
- Other ways to break memory abstractions: missing null terminators, too many null terminators, type casts, type confusion, ...

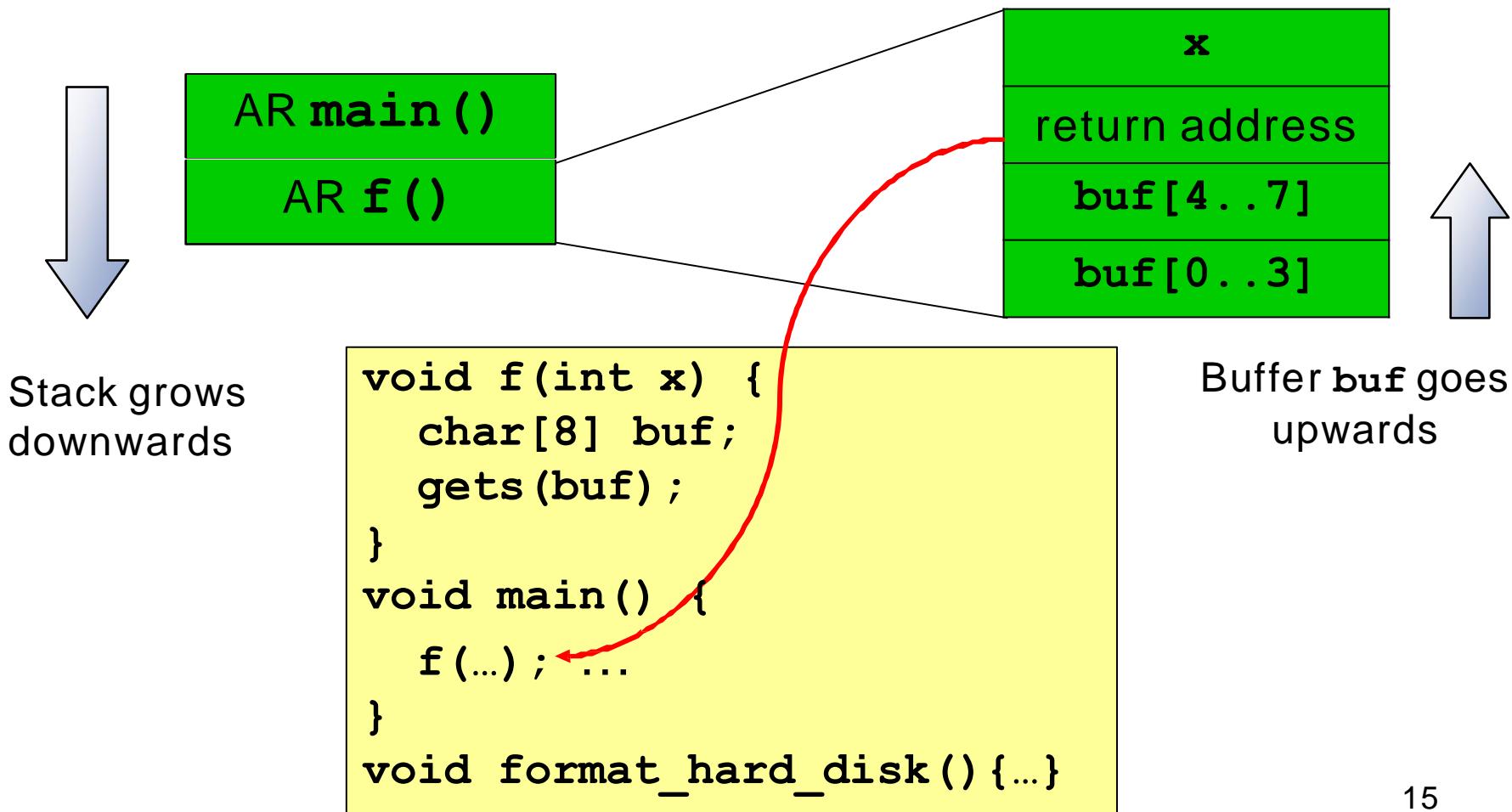
Exploiting this

# Process memory layout



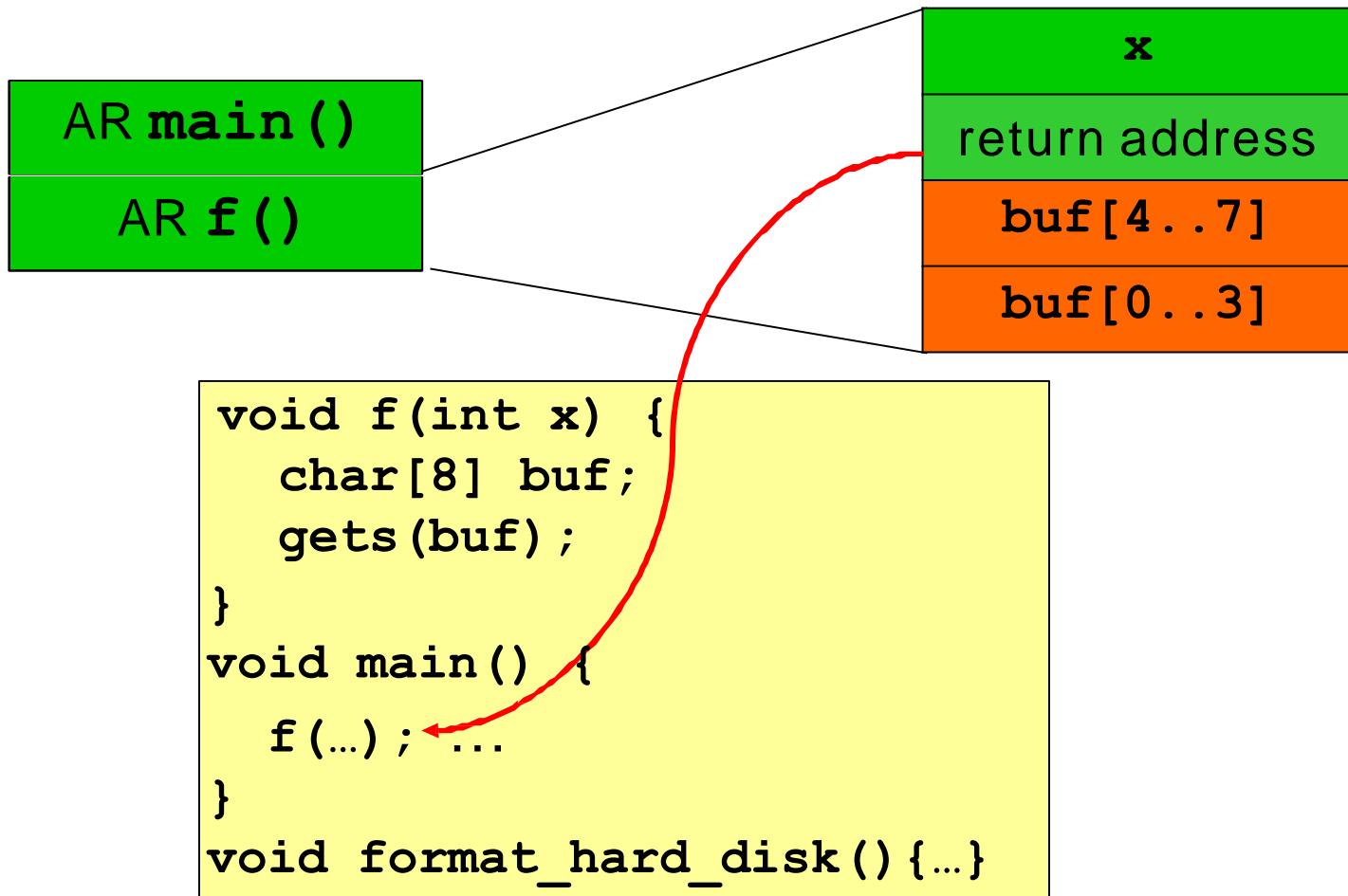
# Stack layout

The stack consists of Activation Records aka **stack frames**:



# Stack overflow attack - case 1

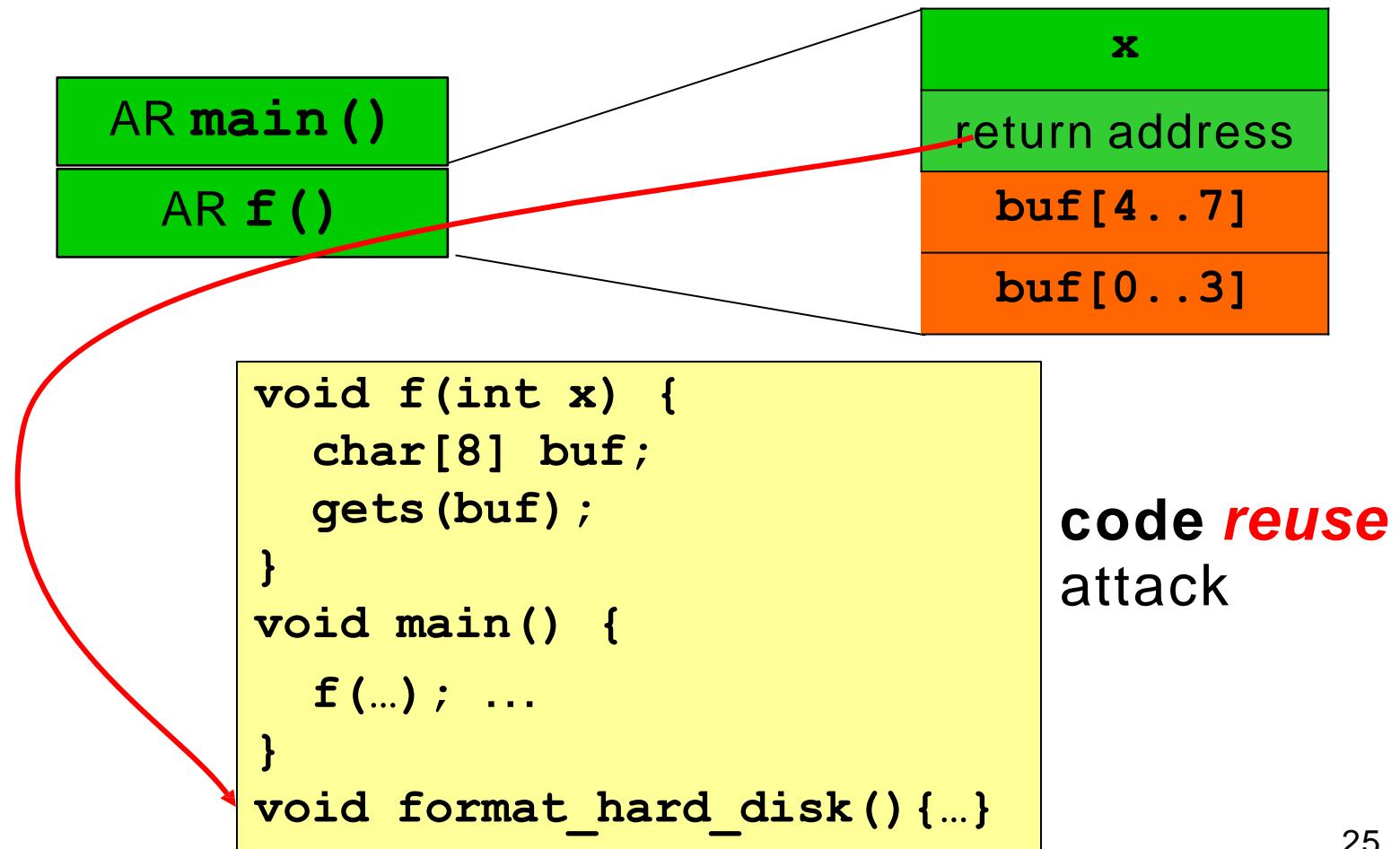
*What if `gets()` reads more than 8 bytes ?*



# Stack overflow attack - case 1

What if `gets()` reads more than 8 bytes ?

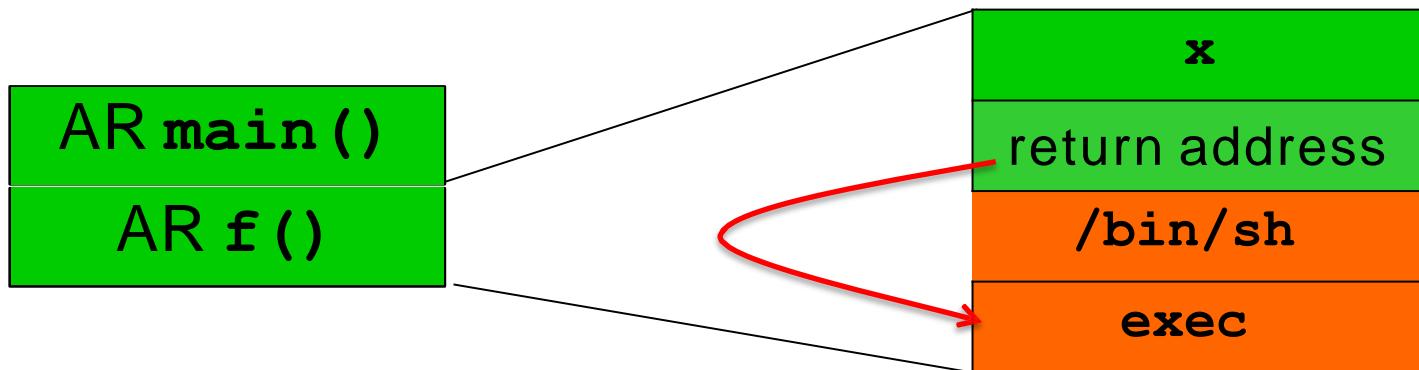
Attacker can jump to arbitrary point in the code!



## Stack overflow attack - case 2

What if `gets()` reads more than 8 bytes ?

Attackers can also jump to their own code (aka shell code)



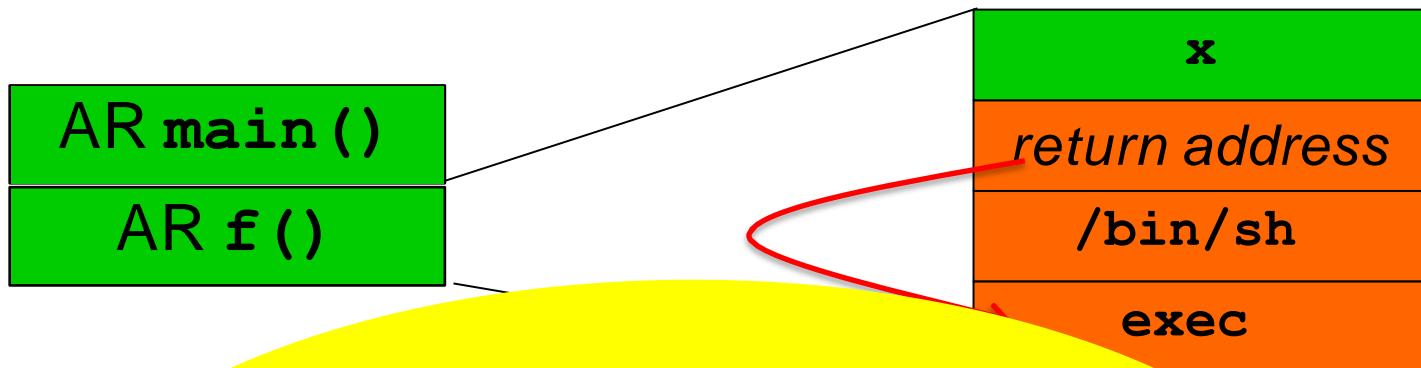
```
void f(int x) {  
    char[8] buf;  
    gets(buf);  
}  
void main() {  
    f(...); ...  
}  
void format_hard_disk() {...}
```

**code injection**  
attack

## Stack overflow attack - case 2

What if `gets()` reads more than 8 bytes ?

Attacker can jump to his own code (aka shell code)



**never use gets !**

Gets has been removed from  
the C standard in 2011

[provides no support to prevent buffer overflow]

# Code *injection* vs code *reuse*

Two types of attacks in these examples

(2) is a code *injection* attack

attackers inject their own shell code in some buffer  
and corrupt return addresss to point to this code

In the example, `exec('/bin/sh')`

This is the classic buffer overflow attack

[Smashing the stack for fun and profit, Aleph One, 1996]

(1) is a code *reuse* attack

attackers corrupt return address to point to existing code

In the example, `format_hard_disk`

# What to attack? Corrupting the stack

```
void f(int x,  
        void(*error_handler)(int) ,  
        bool b) {  
    int diskquota = 200;  
    bool is_superuser = false;  
    char* filename = "/tmp/scratchpad";  
    char[8] username;  
    int j = 12;  
    ...  
}
```

A blue arrow points from the text "function pointer" to the line "void(\*error\_handler)(int) ,".

Suppose attacker can overflow **username**

This can corrupt the return address, but also other data on the stack:

**is\_superuser, diskquota, filename, x, b, error\_handler**

- But not j, unless the compiler chooses to allocate variables in a different order, which the compiler is free to do
- Corruption **function pointers** such as **error\_handler** is particularly interesting!

## What to attack? Corrupting data on the heap

```
struct BankAccount {  
    int number;  
    char username[20];  
    int balance;  
}
```

Suppose attacker can overflow `username`

This can corrupt other fields in the struct

- Which fields depends on the order of the fields in memory.  
The compiler is free to choose this.

## What to attack? Corrupting vtables on the heap

C++ code uses **late binding** to resolve (so-called virtual) method calls

```
Rectangle r;  
Circle c;  
Shape s;  
  
_surface_area = r.area() + c.area() + s.area();
```

Which code to execute for `s.area()` is determined at runtime.

To do this, a **table of function pointers**, the **vtable**,  
is maintained that tells which code to execute for each method

This provides many function pointers for attackers to mess with!

Spotting the problem

## Reminder: C chars & strings

- A char in C is always exactly one byte
- A string is a sequence of **chars** terminated by a **NUL byte**
- String variables are **pointers** of type **char\***

```
char* str = "hello"; // a string str
```



Here **strlen(str)** will be 5

## Example: `gets`

```
char buf[20];
gets(buf); // read user input until
            // first EoL or EoF character
```

- *Never use `gets`*
  - `gets` has been removed from the C library so this code will no longer compile
- Use `fgets(buf, size, file)` instead
  - `fgets()` function reads one less than the size value

# Example

Run this code in online compiler

There is an ERROR in ‘char buf[20]=abcdefgh; ‘  
Compiler will give warning/error of fgets

```
#include <stdio.h>
int main() {

    printf("Hello world");
    char buf[20]=abcdefgh;
    gets(buf);
    return 0;}
```

## Example: `strcpy`

```
char dest[20];  
strcpy(dest, src); // copies string src to dest
```

- `strcpy` assumes that 1. `dest` is long enough  
and 2. `src` is null-terminated
- Use `strncpy(dest, src, size)` instead

Beware of difference between `sizeof` and `strlen`

```
sizeof(dest) = 20          // size of an array  
strlen(dest) = number of chars up to first null byte  
                           // length of a string
```

## Spot the defect!

```
char buf[20];
char prefix[] = "http://";
char* path;

...
strcpy(buf, prefix);
// copies the string prefix to buf
strncat(buf, path, sizeof(buf));
// concatenates path to the string buf
```

## Spot the defect! (1 Segmentation fault)

```
char buf[20];
char prefix[] = "http://";
char* path;

...
strcpy(buf, prefix);
// copies the string prefix to buf
strncat(buf, path, sizeof(buf));
// concatenates path to the string buf
```

strncat's 3rd parameter is number of chars to copy

So this here **sizeof(buf) - 7** but path contains null.

# Better libraries?

Keeping track of the space left in buffers when using `strncpy` is error-prone. Better alternatives:

- `strlcpy(dst, src, size)` and `strlcat(dst, src, size)`  
Here `size` is the size of destination array `dst`, not the maximum length copied. These are consistently used in OpenBSD.
- Functions in Microsoft's `Strsafe.h` also always takes destination size as argument. Moreover, they guarantee null-termination.

## Spot the defect! (2)

```
char src[9];
char dest[9];

char* base_url = "www.ru.nl";
strncpy(src, base_url, 9);
    // copies base_url to src
strcpy(dest, src);
    // copies src to dest
```

## Spot the defect! (2)

```
char src[9];
char dest[9];
```

`base_url` is 10 chars long, incl.  
its null terminator, so `src` will not  
be null-terminated

```
char* base_url = "www.ru.nl";
strncpy(src, base_url, 9);
    // copies base_url to src
strcpy(dest, src);
    // copies src to dest
```



## Spot the defect! (2)

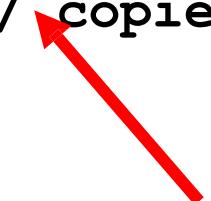
```
char src[9];
char dest[9];
```

`base_url` is 10 chars long, incl.  
its null terminator, so `src` will not  
be null-terminated

```
char* base_url = "www.ru.nl";
```

`strncpy(src, base_url, 9);`   
// copies `base_url` to `src`

```
strcpy(dest, src);
```

`// copies src to dest` 

so `strcpy` will overrun the buffer `dest`,  
because `src` is not null-terminated

## Example: `strcpy` and `strncpy`

Don't replace

`strcpy(dest, src)`

with

`strncpy(dest, src, sizeof(dest))`

but with

`strncpy(dest, src, sizeof(dest) - 1)`

`dest[sizeof(dest) - 1] = '\0';`

if you want `dest` to be null-terminated!

NB: a strongly typed programming language would guarantee that strings are always null-terminated, without the programmer having to worry about this...

## Spot the defect! (3)

```
char *buf;  
int len;  
...
```

```
buf = malloc(MAX(len,1024)); // allocate buffer  
read(fd,buf,len); // read len bytes into buf
```



What happens if `len` is negative?

The length parameter of `read` is unsigned!

So negative `len` is interpreted as a big positive one!

## Spot the defect! (3)

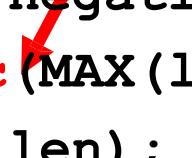
```
char *buf;  
int len;  
  
...  
  
if (len < 0)  
    {error ("negative length"); return; }  
buf = malloc(MAX(len,1024));  
read(fd,buf,len);
```

Note that **buf** is not guaranteed to be null-terminated;  
we ignore this for now.

## Spot the defect! (3)

```
char *buf;  
int len;  
...  
  
if (len < 0)  
    {error ("negative length"); return; }  
buf = malloc(MAX(len,1024));  
read(fd,buf,len);
```

What if the malloc() fails,  
because we ran out of memory ?



## Spot the defect! (3)

```
char *buf;  
int len;  
  
...  
  
if (len < 0)  
    {error ("negative length"); return; }  
buf = malloc(MAX(len,1024));  
if (buf==NULL) { exit(-1);}  
                    // or something a bit more graceful  
read(fd,buf,len);
```

## Better still

```
char *buf;  
int len;  
  
...  
  
if (len < 0)  
    {error ("negative length"); return; }  
buf = calloc(MAX(len,1024));  
    //it initialize allocated memory to 0  
if (buf==NULL) { exit(-1); }  
    // or something a bit more graceful  
read(fd,buf,len);
```

# Spot the defect!

```
#define MAX_BUF 256

void BadCode (char* input)
{
    short len;
    char buf[MAX_BUF];

    len = strlen(input);

    if (len < MAX_BUF) strcpy(buf,input);
}
```

# Spot the defect!

```
#define MAX_BUF 256
```

```
void BadCode (char* input)
{
    short len;
    char buf[MAX_BUF];
    len = strlen(input);
    if (len < MAX_BUF) strcpy(buf, input);
}
```

What if `input` is longer than 32K ?

len may be a negative number,  
due to integer overflow

hence: potential  
buffer overflow

The integer overflow is the root problem,  
the (heap) buffer overflow it causes makes it exploitable

See <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer+overflow>

## Spot the defect!

```
bool CopyStructs(InputFile* f, long count)
{
    structs = new Structs[count];
    for (long i = 0; i < count; i++)
        { if !(ReadFromFile(f, &structs[i])))
            break;
    }
}
```



effectively does a  
`malloc(count*sizeof(type))`  
which may cause integer overflow

And this integer overflow can lead to a (heap) buffer overflow  
Since 2005 Visual Studio C++ compiler adds check to prevent this

# Absence of language-level security

In a **safer** programming language than C/C++,  
the programmer would not have to worry about

- writing past array bounds  
(because you'd get an IndexOutOfBoundsException instead)
- strings not having a null terminator
- implicit conversions from signed to unsigned integers  
(because the type system/compiler would forbid this or warn)
- malloc possibly returning null  
(because you'd get an OutOfMemoryException instead)
- malloc not initialising memory  
(because language could always ensure default initialisation)
- integer overflow  
(because you'd get an IntegerOverflowException instead)
- ...

# Spot the defect!

```
1. void* f(int start) {  
2.     if (start+100 < start) return SOME_ERROR_CODE;  
3.             // checks for overflow  
4.     for (int i=start; i < start+100; i++) {  
5.         . . . // i will not overflow  
6.     } }
```

Integer overflow is **undefined behaviour**! This means

- You cannot assume that overflow produces a negative number; so line 2 is *not* a good check for integer overflow.
  - Worse still, if integer overflow occurs, behaviour is undefined:
    - So compiled code can do *anything* if **start+100** overflows
    - So compiled code can do *nothing* if **start+100** overflows
    - This means the compiler can *remove* line 2
- Modern C compilers are clever enough to know that **x+100 < x** is always false, and optimise code accordingly

## Spot the defect! (code from Linux kernel)

```
1. unsigned int tun_chr_poll( struct file *file,
2.                             poll_table *wait)
3. {
4.     struct sock *sk = tun->sk; // take sk field of tun
5.     if (!tun) return POLLERR; // return if tun is NULL
6.     ...
7. }
```

## Spot the defect! (code from Linux kernel)

```
1. unsigned int tun_chr_poll( struct file *file,
2.                             poll_table *wait)
3. {
4.     struct sock *sk = tun->sk; // take sk field of tun
5.     if (!tun) return POLLERR; // return if tun is NULL
6.     ...
7. }
```

If `tun` is a null pointer, then `tun->sk` is **undefined**

What this function does when `tun` is null is undefined:

**ANYTHING** may happen then.

So compiler **can remove line 5**: the behaviour when `tun` is `NULL` is undefined anyway, so this check is 'redundant'.

Standard compilers (gcc, clang) do this 'optimisation' !

This is code from the Linux kernel where removing line 5 led to a security vulnerability [CVE-2009-1897]

## Spot the defect! (code from Windows kernel)

```
// TCHAR is 1 byte ASCII or multiple byte UNICODE
#ifndef UNICODE
# define TCHAR wchar_t           wide UNICODE character, > 1 byte
# define _tprintf _wprintf       print-function for wide character strings
#else
# define TCHAR char             ASCII character, 1 byte
# define _tprintf _printf       print-function for ASCII character strings
#endif

TCHAR buf[MAX_SIZE];
_tprintf(buf, sizeof(buf), input);
```



`sizeof(buf)` is the size in *bytes*,  
but this parameter should be the  
number of *characters*

Switch from ASCII to UNICODE caused lots of buffer overflows

# Spot the defect!

```
#include <stdio.h>

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

This program is vulnerable to **format string attacks**, where calling the program with strings containing special characters can result in a buffer overflow attack.

# Format string attacks

Type of memory corruption discovered in 2000

- Strings can contain special characters, eg `%s` in  
`printf("Cannot find file %s", filename);`  
Such strings are called **format strings**
- What happens if we execute the code below?  
`printf("Cannot find file %s");`
- What can happen if we execute  
`printf(string)`  
where **string** is user-supplied ?  
Esp. if it contains special characters, eg `%s`, `%x`, `%n`, `%hn`?

# Format string attacks

If attacker can control malicious input `s` to `printf(s)` then this can

- *read the stack*

%x reads and prints bytes from stack

so input `%x%`  
`%x%`  
`x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%`  
`x%x%x%x%x%x%x%x%x%x%x%`...

dumps the stack, including passwords, keys,... stored on the stack

- *corrupt the stack*

%n writes the number of characters printed to the stack

so input `12345678%n` writes the value 8 to the stack

- *read arbitrary memory*

a carefully crafted input string of the form

`\xEF\xCD\xCD\xAB %x%x...%x%s`

print the string at memory address `ABCDCDEF`

# Example

```
#include<stdio.h>

int main()
{
    int e;
    printf("Educative for %nLearning ", &e);
    printf("\nCount: %d", e);

    return 0;
}
```

Output:

e=14 // count the characters before %n

writes the value 14 to the stack

# Preventing format string attacks is EASY

1. Always replace `printf(str)`  
with `printf("%s", str)`
2. Compiler or static analysis (SAST) tool could warn if the number of arguments does not match the format string

As e.g. in `printf ("x is %i and y is %i", x);`

Check <https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string> to see how common format strings still are

## Recap: memory corruption

- #1 weakness in C / C++
  - because these languages are **not memory-safe** and programmer is responsible for memory management
- Tricky to spot
- Typical cause: programming with **arrays, pointers, strings & and dynamic (ie heap-allocated) memory**
- Related attacks
  - **Format string attack**: another way of corrupting stack
  - **Integer overflows**: useful a stepping stone to getting a buffer to overflow, or dangerous in its own right

Security Testing  
especially  
Fuzzing

# Security in the SDLC

Last lecture: static analysis/SAST with PREfast (compile time)

This lecture: dynamic analysis/DAST esp. fuzzing



Focus of this lecture – fuzzing aka fuzz testing of C/C++ code for memory corruption

# The security testing paradox

- Security testing is harder than normal, functional testing
  - We have no idea what we are looking for!  
A peculiar input may trigger a odd bug that is exploitable in some peculiar way, and finding that input with testing is hard
  - Normal users are good testers, as they will complain about functional problems, but they will not complain about many/any security flaws
- Security testing is easier than normal, functional testing
  - We *can* test for some classes of bugs in partly automated way using **fuzzing**
  - Fuzzing is the great success story in (software) security in the past decade

# Overview

1. Testing basics
2. Abuse cases & negative tests
3. Fuzzing

# Testing basics

# SUT, test suite & test oracle

To test a SUT (System Under Test) we need two things

1. test suite, i.e., collection of input data
2. test oracle to decide if response is ok or reveals an error
  - i.e., some way to decide if the SUT behaves as we want

Defining test suites and test oracles can be *a lot of work!*

- In the worst case, a test oracle is a long list which *for every individual test case, specifies exactly what should happen*
- A simple test oracle: just looking if application doesn't crash

*Moral of the story: crashes are good ! (for testing)*

# Code coverage criteria

Code coverage criteria can measure how good a test suite:

- statement coverage
- branch coverage

Statement coverage does not imply branch coverage; e.g.,

```
void f (int x, y) { if (x>0) {y++};  
                      y--; }
```

Statement coverage needs 1 test case, branch coverage needs 2

Code coverage metrics can also be used to guide test case generation

# Possible awkward effect of coverage criteria

High coverage criteria may *discourage* defensive programming, e.g.,

```
void m(File f) {  
    if <security_check_fails> {log (...);  
                             throw (SecurityException);}  
  
    try { <the main part of the method> }  
  
    catch (SomeException) { log(...);  
                           <some corrective action>;  
                           throw (SecurityException); }  
  
}
```

If **defensive code**, i.e., the if- & catch-branches, is hard to trigger in tests, programmers may be tempted (or forced?) to remove this code to improve test coverage...

# Annotations as test oracle

- Annotations, e.g., SAL annotations of C/C++ code, can be used as test oracle by doing **runtime assertion checking**
  - So annotations provide a **test oracle for free!** You can test by sending random data & checking if annotations are violated
- Information flow policies can also be used as test oracles

Annotations in programming languages are, similar to those in linguistics, structural elements containing additional or meta-information in the source code of a program.

source-code annotation language SAL - provides a set of annotations that you can use to describe how a function uses its parameters, the assumptions that it makes about them, and the guarantees that it makes when it finishes. The annotations are defined in the header file <sal.h>

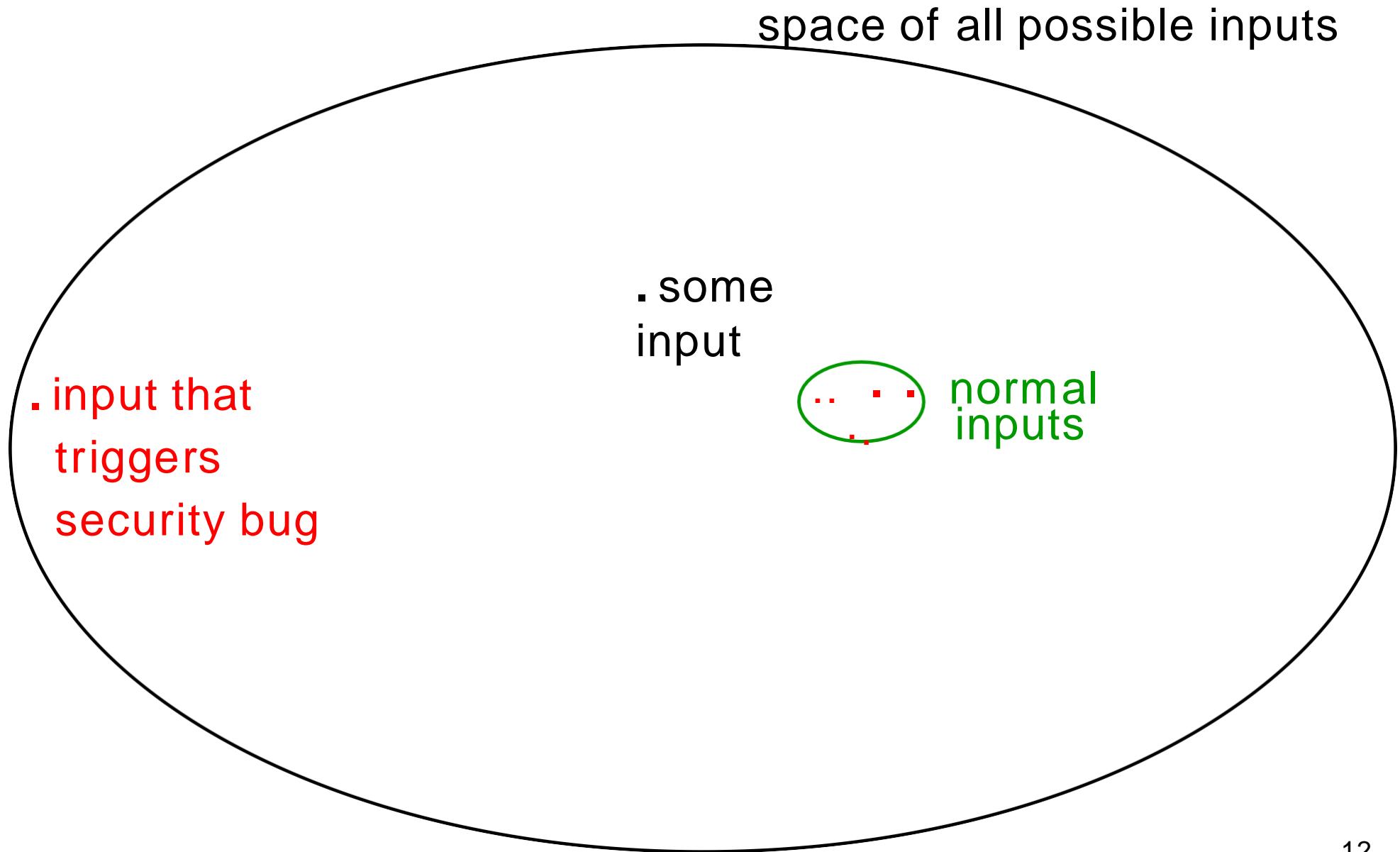
Security testing:  
Abuse cases & Negative test cases

# testing vs security testing

## Difference in focus

- Normal (functional) testing focuses on **correct, desired behaviour** for sensible inputs (aka **the happy flow**), but will include some inputs for borderline conditions
- Security testing (also) – especially – looks for **wrong, undesired behaviour** for really strange inputs
- Normal use of a system is more likely to reveal **functional problems** than **security problems**

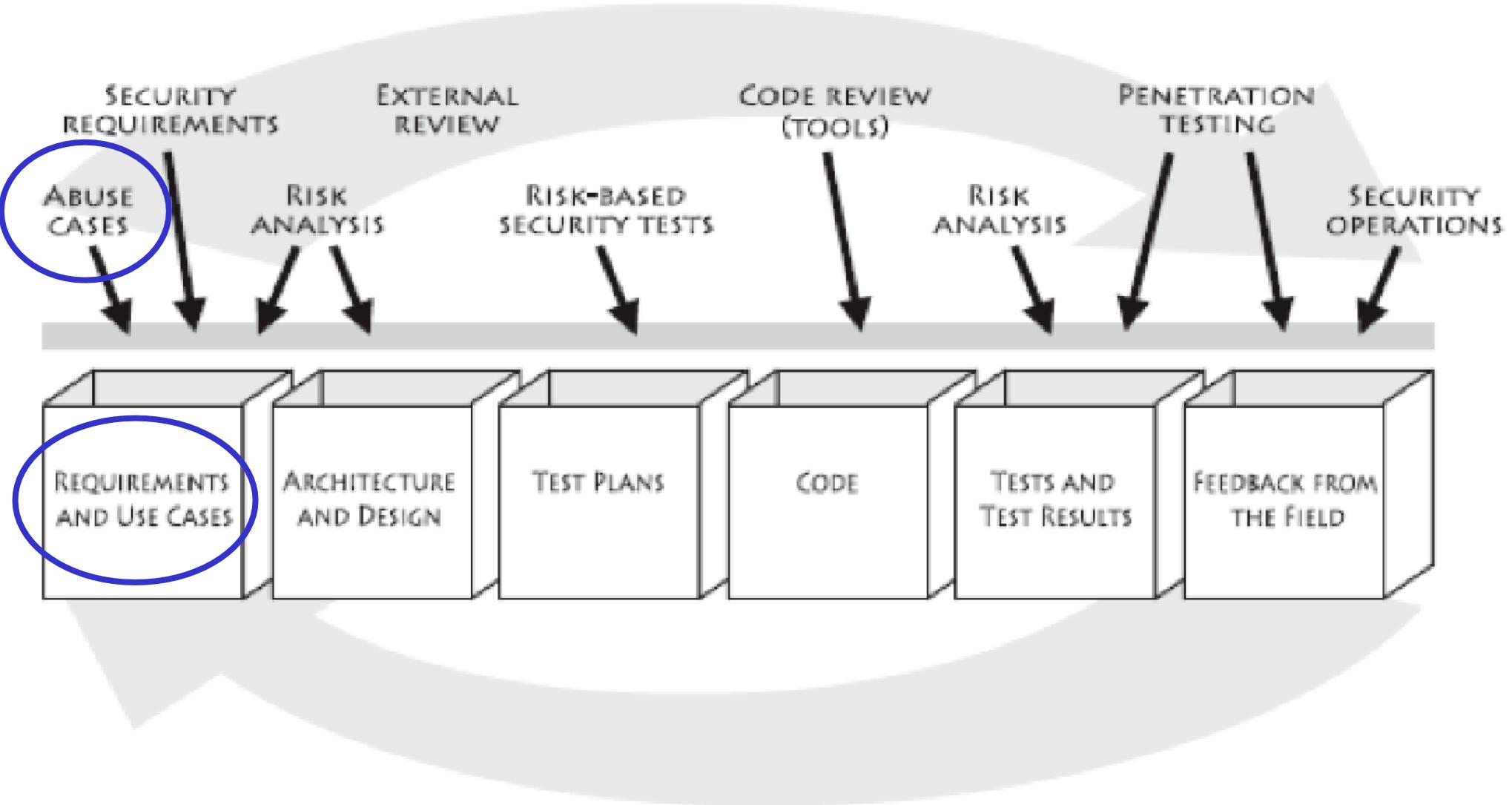
# Security testing is HARD



## Abuse cases → negative test cases

- Thinking about **abuse cases** is a useful way to come up with security tests
  - *what would an attacker try to do?*
  - *where could an implementation slips?*
- This gives rise to **negative test cases**, i.e., test cases which are *supposed to fail* opposed to *positive* test cases, which are meant to succeed

# Abuse cases – early in the SDCL



# APPLE'S iOS goto fail SSL bug (2014)

...

```
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;

if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;

if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;

    goto fail;

if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);

. . .
```

## *Negative test cases* eg. for flawed certificate chains

- David Wheeler's 'The Apple goto fail vulnerability: lessons learned' gives a good discussion of this bug & ways to prevent it, incl. [the need for negative test cases](#)

<http://www.dwheeler.com/essays/apple-goto-fail.html>

- The FrankenCert test suite provides (broken) certificate chains to test for flaws in the program logic for checking certificates.

[Brubaker et al, Using [Frankencerts](#) for Automated [Adversarial Testing](#) of Certificate Validation in SSL/TLS Implementations, Oakland 2014]

- Code coverage requirements on the test suite would also have helped.

**frankencert** generator to auto-generate different **test** certificates involving complex corner cases.

# Fuzzing

# The idea

Suppose some C(++) binary asks for some input

Please enter your username

>

*What would you try?*

1. ridiculous long input, say a few MB

If there is a buffer overflow, a long input is likely to trigger a SEG FAULT

2. %x%x%x%x%x%x%x

To see if there is a format string vulnerability

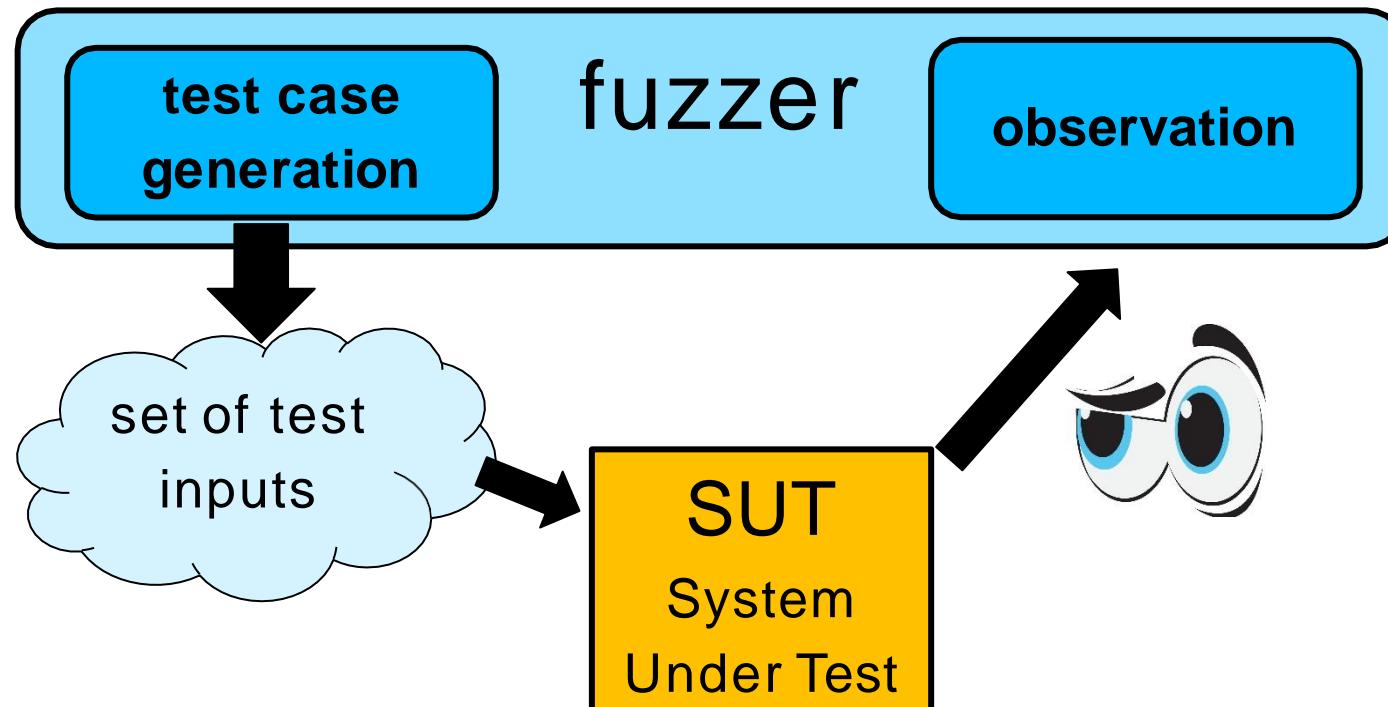
On the command line, we cannot include a null terminator \0 in an input, but in other situations we may be able to

3. Other malicious inputs, depending on back-ends, technologies & APIs used: eg SQL, XML, JSON, Unicode character encodings,...

# Fuzzing

(semi) automatically generate ‘random’ inputs and check if an application crashes or misbehaves in observable way

Great for certain classes of bugs, esp. memory corruption bugs



First tool for this: **fuzz** for UNIX

[Miller et al., An empirical study of the reliability of UNIX utilities, CACM 1990]

# URL Fuzzing

fuzz urls to find hidden directories in a web application.

Before a website can be attacked, having knowledge of the structs, dirs, and files the web server or website uses are very important in order to map out the strategy that will be used to attack.

Read Example Later: -

<https://medium.com/@futaacmcyber/fuzzing-urls-to-find-hidden-web-directories-208e1870f956>

List of tools: <https://blackarch.org/fuzzer.html>

URL Fuzzer: <https://pentest-tools.com/website-vulnerability-scanning/discover-hidden-directories-and-files> (See report of iiitvadodara.ac.in)

# Fuzzing

1. Basic fuzzing with random/long inputs
2. ‘Dumb’ mutational fuzzing
  - example: OCPP
3. Generational fuzzing aka grammar-based fuzzing
  - example: GSM
4. Code-coverage guided evolutionary fuzzing with **afl**
  - aka grey box fuzzing or ‘smart’ mutational fuzzing
5. Whitebox fuzzing with **SAGE**
  - using symbolic execution

Beware: terminology for various forms of fuzzing is messy

The field of fuzzing has been exploding past 10 years!  
See <http://fuzzing-survey.org> for an overview of fuzzing field

# Fuzzing

- Fuzzing aka fuzz testing is a highly effective, largely automated, security testing technique
- Basic idea: (semi) automatically generate random inputs and see if an application crashes
  - So we are NOT testing functional correctness (aka compliance)

# How to fuzz

Depending on input type

- very long inputs, very short inputs, or completely blank input
- min/max values of integers, zero and negative values
- depending on what you are fuzzing, include special values, characters or keywords likely to trigger bugs, eg
  - nulls, newlines, or end-of-file characters
  - format string characters `%s %x %n`
  - semi-colons, slashes and backslashes, quotes
  - application specific keywords `halt`, `DROP TABLES`, ...
  - ....

Good [validation](#) and/or [sanitisation](#) would catch these problems.

# Pros & cons of fuzzing

## Pros

- **Very little effort**: test cases are automatically generated, and test oracle is trivial
  - Fuzzing of a C/C++ binary quickly gives a good indication of robustness of the code

## Cons

- Only finds '**shallow**' bugs and not '**deeper**' bugs
  - If a program takes **complex inputs**, 'smarter' fuzzing is needed to trigger bugs.
- Crashes may be hard to analyse; but a crash is a clear *true positive* that something is wrong!
  - unlike a complaint from a static analysis tool like PREfast

# Improved crash/error detection

Making systems crash on errors is useful for fuzzing!

So when fuzzing C(++) code, all memory safety checks discussed in previous lectures can be deployed (to make crashing in the event of memory corruptions more likely)

Tools for this include

- **ASan** – **AddressSanitizer**
- **MSan** – **MemorySanitizer**
- **valgrind**
  - **MemCheck**

Ideally checks for both **spatial bugs** (e.g. buffer overruns)  
& **temporal bugs** (e.g. malloc/free bugs)

## Improvements to just trying random and/or long inputs

- 1) Mutation-based: apply random mutations to valid inputs
  - Eg observe network traffic, than replay with some modifications
  - More likely to produce interesting invalid inputs than just random input
- 2) Generation-based aka grammar-based aka model-based:  
generate semi-well-formed inputs from scratch, based on description of file format or protocol
  - Either tailor-made fuzzer for a specific input format, eg. FrankenCert, or a generic fuzzer configured with a grammar
  - *Downside?*  
More work to construct this fuzzer or grammar
- 3) Evolutionary/greybox: observe execution to try to learn which mutations are interesting  
  
Eg. **Afl** (**American Fuzzy Lop** is a brute-force fuzzer)
- 4) Whitebox approaches: analyse source code to construct inputs  
  
Eg. **SAGE** (Scalable Automated Guided Execution) first tool to perform dynamic symbolic execution at the x86 binary level

## Example mutational fuzzing

# Example: Fuzzing OCPP [research internship Ivar Derksen]

- OCPP is a protocol for charge points to talk to a back-end server
- OCPP can use XML or JSON messages

Example message in JSON format

```
{ "location": "NijmegenMercator215672",
  "retries": 5,
  "retryInterval": 30,
  "startTime": "2022-10-27T19:10:11",
  "stopTime": "2022-10-27T22:10:11" }
```



# Example: Fuzzing OCPP

Classification of messages into

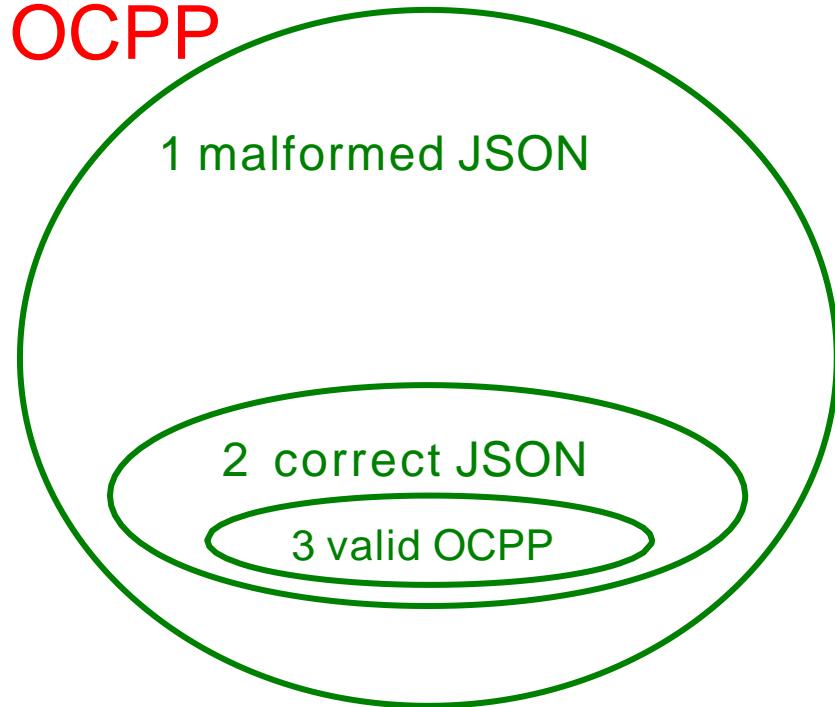
1. **malformed JSON/XML**  
eg missing quote, bracket or comma
2. **well-formed JSON/XML, but not legal OCPP**  
eg with field names not in OCPP specs
3. **well-formed OCPP**

can be used for a simple test oracle:

- The application should never crash
- Malformed messages (type 1 & 2) should generate generic error response
- Well-formed messages (type 3) should not generate errors

Note: this does not require *any* understanding of the protocol semantics!

Figuring out correct responses to type 3 would require that.



# Test results with fuzzing OCPP server

- Mutation fuzzer generated 26,400 variants from 22 example OCPP messages in JSON format
- Problems spotted by this simple test oracle:
  - 945 malformed JSON requests (type 1) resulted in malformed JSON response  
*Server should never emit malformed JSON!*
  - 75 malformed JSON requests (type 1) and 40 malformed OCPP requests (type 2) result in a valid OCPP response that is not an error message.  
*Server should not process malformed requests!*
- One root cause of problems: the Google's **json** library for parsing JSON by default uses **lenient** mode rather than **strict** mode
  - Why does **json** even have a lenient mode, let alone by default?
- Fortunately, **json** is written in Java, not C(++), so these flaws do not result in exploitable buffer overflows

# Postel's Law aka Robustness Principle

“Be conservative in what you send,  
be liberal in what you accept”

[Named after Jon Postel, who wrote early version of TCP]

*Is this good or bad?*

- Good for getting interoperable implementations up & running 😊
- Bad for security, as it leads to implementations with non-standard behavior, deviating from the official specs, in corner cases, which may lead to weird behaviour and bugs 😞 😞

Generational fuzzing  
aka  
Grammar-based fuzzing

# CVEs as inspiration for fuzzing file formats

- Microsoft Security Bulletin MS04-028  
[Buffer Overrun in JPEG Processing \(GDI+\) Could Allow Code Execution](#)  
Impact of Vulnerability: Remote Code Execution  
Maximum Severity Rating: Critical  
Recommendation: Customers should apply the update immediately  
  
Root cause: a zero sized comment field, without content
- CVE-2007-0243  
[Sun Java JRE GIF Image Processing Buffer Overflow Vulnerability](#)  
Critical: Highly critical Impact: System access Where: From remote  
  
Description: A vulnerability has been reported in Sun Java Runtime Environment (JRE). ... The vulnerability is caused due to an error when processing GIF images and can be exploited to cause a **heap-based buffer overflow** via **a specially crafted GIF image with an image width of 0**. Successful exploitation allows execution of arbitrary code.  
  
*Note: a buffer overflow in (native library of) a memory-safe language*

# Generation/grammar/model-based fuzzing

Generate inputs that are malformed or hit corner cases,  
based on knowledge of input format/protocol

Eg using  
[regular expression](#)  
[context free grammar](#), or  
some other description

0	4	8	16	19	24	31
Version	Header Length	Tos	Total length			
identifier		Flags	Fragment offset			
TTL	Protocol	Header checksum				
Source IP address						
Destination IP address						
Options (variable length)						
Data						

Typical things to fuzz:

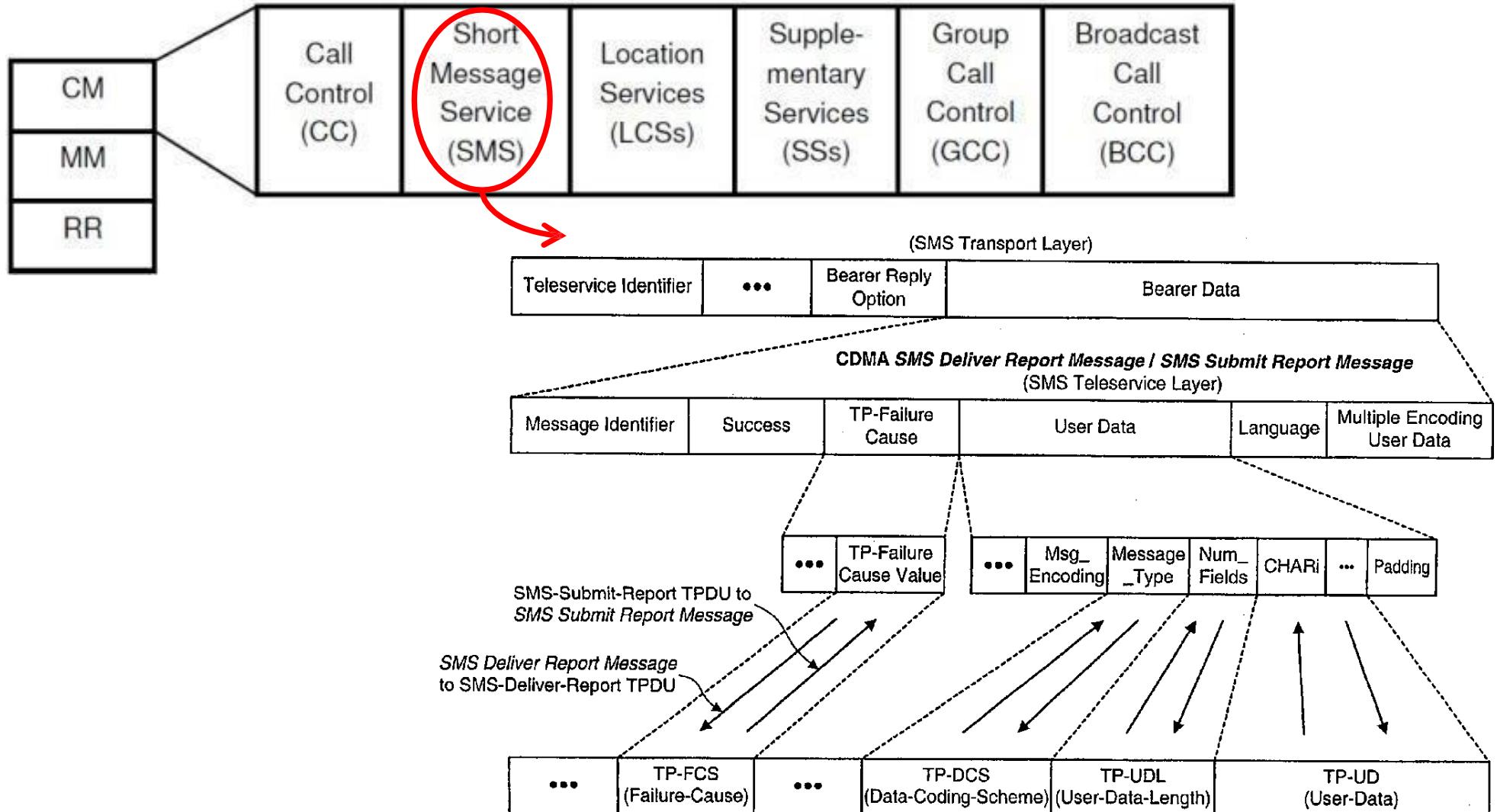
- many/all possible value for specific fields  
esp undefined values, or values Reserved for Future Use (RFU)
- incorrect lengths, lengths that are zero, or payloads that are too short/long

Fuzzing tools: SNOOZE, SPIKE, Peach, Sulley, antiparser, Netzob, ...

# Example: generation based fuzzing of GSM

[Master theses of Brinio Hond and Arturo Cedillo Torres]

GSM is a extremely rich & complicated protocol



# SMS message fields

Field	size
Message Type Indicator	2 bit
Reject Duplicates	1 bit
Validity Period Format	2 bit
User Data Header Indicator	1 bit
Reply Path	1 bit
Message Reference	integer
Destination Address	2-12 byte
Protocol Identifier	1 byte
Data Coding Scheme (CDS)	1 byte
Validity Period	1 byte/7 bytes
User Data Length (UDL)	integer
User Data	depends on CDS and UDL

## Example: GSM protocol fuzzing

Lots of stuff to fuzz!

We can use a [USRP](#)



with open source cell tower software ([OpenBTS](#))

to fuzz any phone



## Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones



## Example: GSM protocol fuzzing

Fuzzing SMS layer of GSM reveals weird functionality in GSM standard and in phones

- eg possibility to receive faxes (!?)

you have a fax!



Only way to get rid if this icon; reboot the phone

## Example: GSM protocol fuzzing

Malformed SMS text messages showing raw memory contents, rather than content of the text message

(a) Showing garbage



(b) Showing the name of a wallpaper and two games



# Our results with GSM fuzzing

- Lots of success to DoS phones:  
phone crashes, disconnects from network, stops accepting calls,...
  - eg requiring reboot or battery removal to restart, to accept calls again, or to remove weird icons
  - after reboot, the network might redeliver the SMS message, if no acknowledgement was sent before crashing, re-crashing phone
    - But: not all these SMS messages could be sent over real network
- There is surprisingly little correlation between problems and phone brands & firmware versions
  - how many implementations of the GSM stack did Nokia have?
- *The scary part: what would happen if we fuzz base stations?*

[Fabian van den Broek, Brinio Hond and Arturo Cedillo Torres, Security Testing of GSM Implementations, ESSOS 2014]

[Mulliner et al., SMS of Death, USENIX 2011]

# Security problem with more complex input formats



iPhone /

## This text message called the ‘Unicode of Death’ will crash your iPhone

By Jacques Coetzee: Staff Reporter on 28 May, 2015

effective.

Power

لُلصِبْلُلصِبْرَرَ  
兀

Example dangerous SMS text message

- This message *can* be sent over the network
- Different character sets & characters encoding are a constant source of problems. Many input formats rely on underlying notion of characters.

## Example: Fuzzing fonts

Google's Project Zero found many Windows kernel vulnerabilities by fuzzing fonts in the Windows kernel

Tracker ID	Memory access type at crash	Crashing function	CVE
<a href="#">1022</a>	Invalid write of $n$ bytes (memcpy)	usp10!otlList::insertAt	CVE-2017-0108
<a href="#">1023</a>	Invalid read / write of 2 bytes	usp10!AssignGlyphTypes	CVE-2017-0084
<a href="#">1025</a>	Invalid write of $n$ bytes (memset)	usp10!otlCacheManager::GlyphsSubstituted	CVE-2017-0086
<a href="#">1026</a>	Invalid write of $n$ bytes (memcpy)	usp10!MergeLigRecords	CVE-2017-0087
<a href="#">1027</a>	Invalid write of 2 bytes	usp10!ttoGetTableData	CVE-2017-0088
<a href="#">1028</a>	Invalid write of 2 bytes	usp10!UpdateGlyphFlags	CVE-2017-0089
<a href="#">1029</a>	Invalid write of $n$ bytes	usp10!BuildFSM and nearby functions	CVE-2017-0090
<a href="#">1030</a>	Invalid write of $n$ bytes	usp10!FillAlternatesList	CVE-2017-0072

<https://googleprojectzero.blogspot.com/2017/04/notes-on-windows-uniscribe-fuzzing.html>

# Even handling simple input languages can go wrong!

Sending an extended length APDU can crash a contactless payment terminal.

APDU Response		
Body	Trailer	
Data Field	SW1	SW2



Found accidentally, without even trying to fuzz,  
when sending legal (albeit non-standard) messages

[Jordi van den Breekel, A security evaluation and proof-of-concept relay attack on Dutch EMV contactless transactions, MSc thesis, 2014]

# Introduction to Cybersecurity

# Dependence on Internet Increasing

- Email, WhatsApp, Snapchat, VoIP, etc.: **Communication**
- business, banking, e-commerce, etc.: **Commerce**
- Google Maps, Google Drive, etc.: **Public utilities**
- Youtube, Netflix+, Disney Videos, etc.: **Entertainment**
- **Significant sensitive data stored on the Internet**
- Organizations have replaced (or replacing) physical/manual processes with Internet-based processes

# Is Internet Secure?

**Internet design priorities:** cost, speed, open architecture, etc.

**No metrics to measure (in)security**

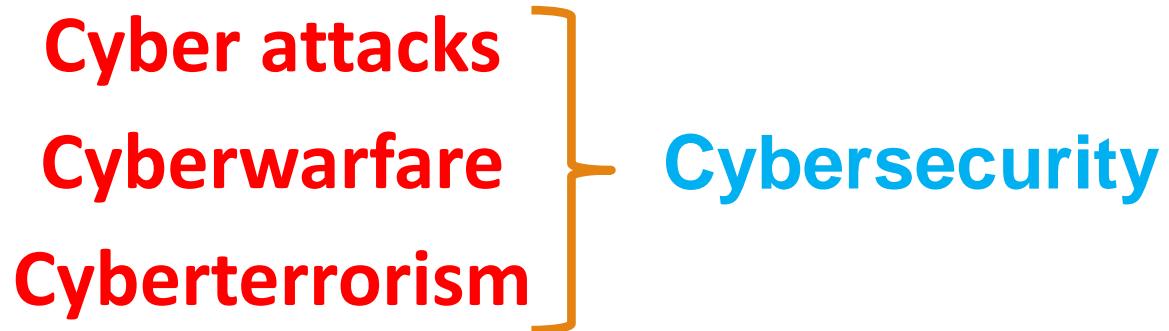
**No single Internet owner**

- Private sector owns most of the infrastructure (IBM, MERIT, NTT, AT&T, British Telecom, Verizon, Sprint, ...)

**Who will ensure Internet security?**

# Security over Internet

The level of dependence we have on  
(hard to secure) internet system  
makes the Internet a target for  
asymmetric attacks (low cost high gain).



# Security over Internet

Internet (or cyberspace) is a weak spot  
for **accidents and failures (Cybercrimes)**

Cybercrimes are crimes committed using computers, phones or the internet. It includes: Illegal interception of data, Illegal data modification, stealing money, leaking privacy, etc.

# Few Well-Known Internet-based (Cyber) Attacks

1. Melissa Virus (1999) by David Lee Smith. He sent users a MSWord file which held a virus. The virus severe damage to hundreds of companies, including Microsoft. Cost approx. \$80 million to repair.
2. NASA Cyber attack (1999) by 15-year-old James Jonathan hacked and shutdown NASA's computers for 21 days! Costs around \$41,000 in repairs.
3. Estonia Cyber attack (2007)- around 58 websites go offline, including government, bank and media websites.
4. Sony's PlayStation Network attack (2011) compromised the personal information of 77 million users.
5. Adobe cyber attack (2013)- the passwords and credit card info, of 2.9 million users were compromised, 35.1 million suffered the loss of their passwords and user IDs.

# Few Well-Known Attacks

6. Yahoo attack (2014): basic information and passwords were stolen of 500 million accounts.
7. Ukraine's power grid attack (2015) first cyberattack on a power grid. around half of the homes in the Ivano-Frankivsk region of the Ukraine were without power for a few hours.
8. WannaCry ransomware attack (2017)- affected around 2 Lakh computers in over 150 countries. Huge impact on several industries with a global cost of around 6 billion pounds to fix!
9. Marriott hotels attack (2018)- undetected for years, an estimated 339 million guests had their data compromised. Consequently, the UK's data privacy watchdog fined the Hotels 18.4 million pounds.
10. RockYou2021 attack- 8.4 billion passwords leaked- use to trick business and employees into clicking on links and documents within emails.

What is “cybersecurity?”

# One way to think about it

**cybersecurity** = security of cyberspace

# One way to think about it

**cybersecurity** = security of **cyberspace**



information systems  
and networks

# One way to think about it

**cybersecurity** = security of information systems and networks

# One way to think about it

**cybersecurity** = security of information systems and networks



+ with the goal of  
protecting operations  
and assets

# One way to think about it

**cybersecurity** = security of information systems and networks  
with the goal of protecting operations and assets

# One way to think about it

**cybersecurity** = **security** of information systems and networks  
with the goal of protecting operations and assets



**security in the face of  
attacks, accidents and  
failures**

# One way to think about it

**cybersecurity** = security of information systems and networks  
in the face of attacks, accidents and failures with the goal of  
protecting operations and assets

# One way to think about it

**cybersecurity** = **security** of information systems and networks  
in the face of attacks, accidents and failures with the goal of  
protecting operations and assets



availability, integrity  
and secrecy

# One way to think about it

**cybersecurity** = availability, integrity and secrecy of information systems and networks in the face of attacks, accidents and failures with the goal of protecting operations and assets

# In Context

**Corporate cybersecurity** = availability, integrity and secrecy of information systems and networks in the face of attacks, accidents and failures with the goal of protecting a **corporate's** operations and assets

**National cybersecurity** = availability, integrity and secrecy of the information systems and networks in the face of attacks, accidents and failures with the goal of protecting a **nation's** operations and assets

# Some more

- Security of cyberspace against cybercrimes
- Cybersecurity is the practice of protecting data, programs, computers, and networks from cyber crimes.

# Web Basics

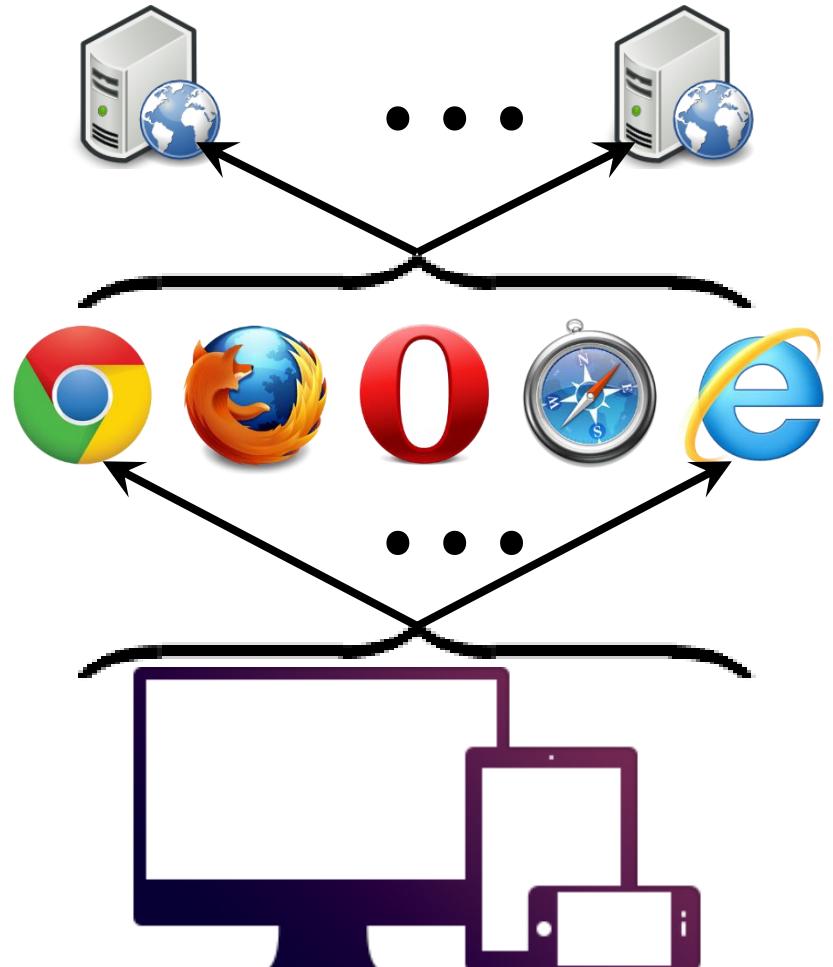
# Introduction

Average user spends 2.5 hrs/day online in India [1]

- People spend much time interacting with Web applications

Before look at web security issues

Let us know few **Web basics**



Source: [2], [3]

# The Web

## Web page:

- Consists of “objects”
- Addressed by a URL

## Most Web pages consist of:

- Base HTML page, and
- Several referenced objects.

## URL has two components:

host name and path name

## User agent for Web is called a browser:

- IE, Firefox, Opera
- Netscape, Apple Safari
- MS Edge

## Server for Web is called Web server:

- Apache (public domain)
- Internet Information Server
- Apache Tomcat
- Node.js

# The Web: the HTTP Protocol

## HTTP: HyperText Transfer Protocol

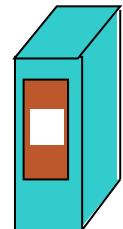
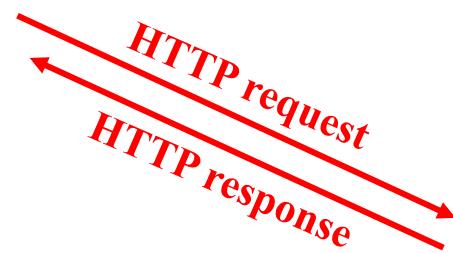
[Web's application layer protocol]

Client/server model

- **Client:** browser that requests, receives, “displays” Web objects
- **Server:** Web server sends objects in response to requests



PC running  
Explorer



Server  
running  
Web server

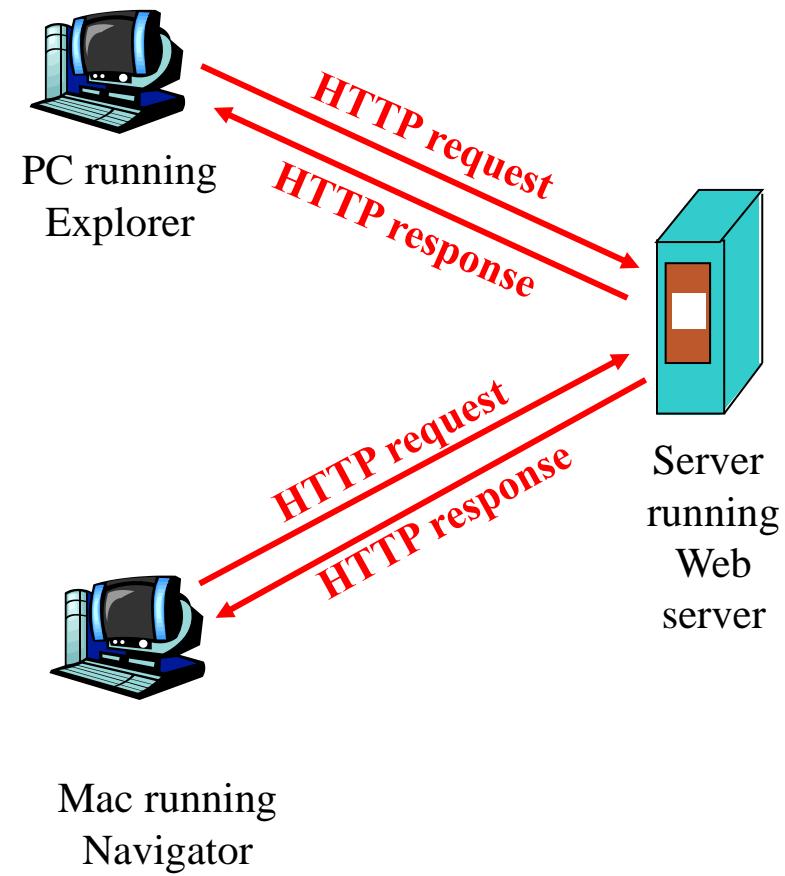
# The HTTP Protocol (Cont.)

Client initiates TCP connection  
(creates socket) to server, port 80

Server accepts TCP connection  
from client

HTTP messages (application-layer  
protocol messages) exchanged  
between browser (HTTP client)  
and Web server (HTTP server)

TCP connection closed



# HTTP Example

(contains text,  
references to 10  
JPEG images)

Suppose user enters URL `http://www.someschool.edu/aDepartment/index.html`

**1a.** HTTP client initiates TCP connection to http server (process) at `www.someschool.edu`. Port 80 is default for HTTP server.

**1b.** HTTP server at host `www.someschool.edu` waiting for TCP connection at port 80. “Accepts” connection, notifies client

**2.** HTTP client sends http *request message* (containing URL) into TCP connection socket

**3.** HTTP server receives request message, forms *response message* containing requested object (`aDepartment/index.html`), sends message into socket

time  
↓

# HTTP Example (Cont.)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing HTML file, displays HTML.  
Parsing HTML file, finds 10 referenced JPEG objects

time

↓  
6. Steps 1-5 repeated for each of 10 JPEG objects

# Non-Persistent and Persistent Connections

## Non-persistent

HTTP/1.0

Client initiates request (conn. + data)

Server parses request, responds, and closes TCP connection

2 RTTs to fetch each object

Each object transfer suffers from slow start

## Persistent

Default for HTTP/1.1

On same TCP connection: server, parses request, responds, parses new request, ...

Client sends requests for all referenced objects as soon as it receives base HTML.

Fewer RTTs and slow start.

**But most browsers use parallel TCP connections.**

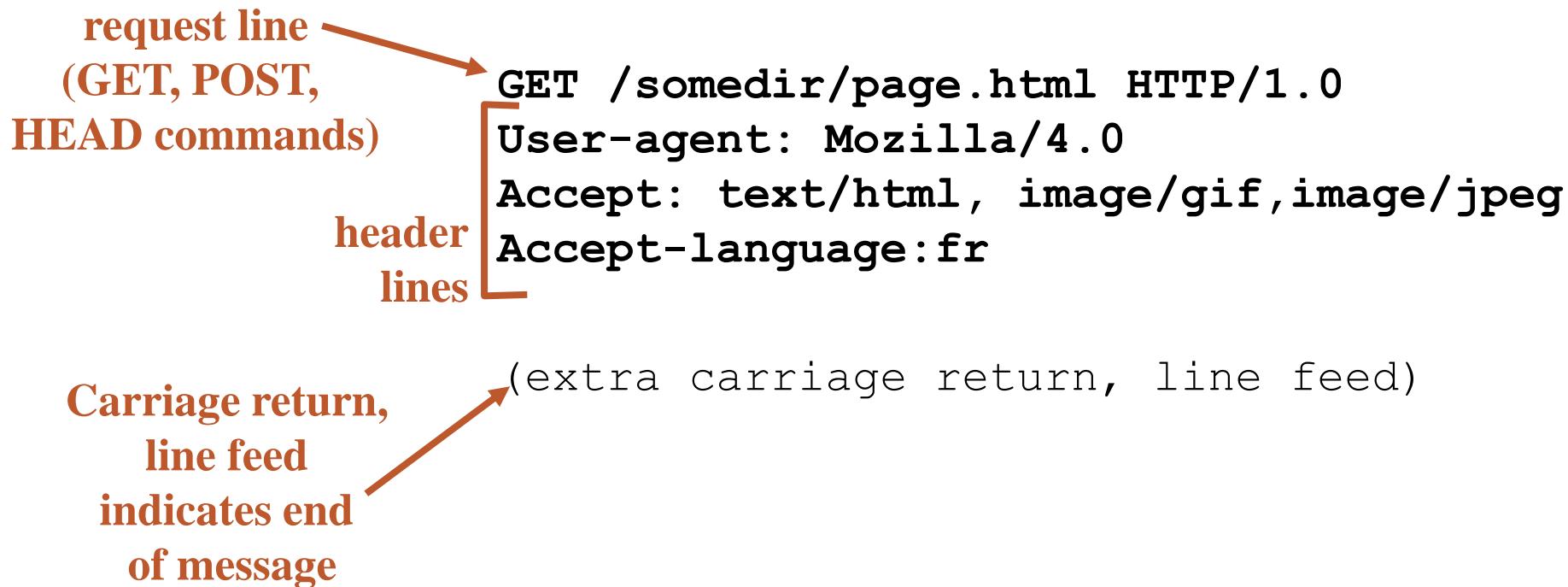


# HTTP Message Format: Request

Two types of HTTP messages: *request, response*

## HTTP request message:

- ASCII (human-readable format)



# HTTP Message Format: Response

status line  
(protocol  
status code  
status phrase)

header  
lines

data, e.g.,  
requested  
html file

HTTP/1.0 200 OK  
Date: Thu, 06 Aug 1998 12:00:15 GMT  
Server: Apache/1.3.0 (Unix)  
Last-Modified: Mon, 22 Jun 1998 ....  
Content-Length: 6821  
Content-Type: text/html

data data data data data ...

# HTTP Response Status Codes

In first line in server→client response message.

A few sample codes:

## 200 OK

- request succeeded, requested object later in this message

## 301 Moved Permanently

- requested object moved, new location specified later in this message (Location:)

## 400 Bad Request

- request message not understood by server

## 404 Not Found

- requested document not found on this server

## 505 HTTP Version Not Supported

# Try HTTP (Client Side) for Yourself

1. Telnet to your favorite Web server:

```
telnet www.iiitvadodara.ac.in 80
```

Opens TCP connection to port 80 (default HTTP server port) at www.iiitvadodara.ac.in  
Anything typed in sent to port 80 at www.iiitvadodara.ac.in

2. Type in a GET HTTP request:

```
GET /index.html HTTP/1.0
```

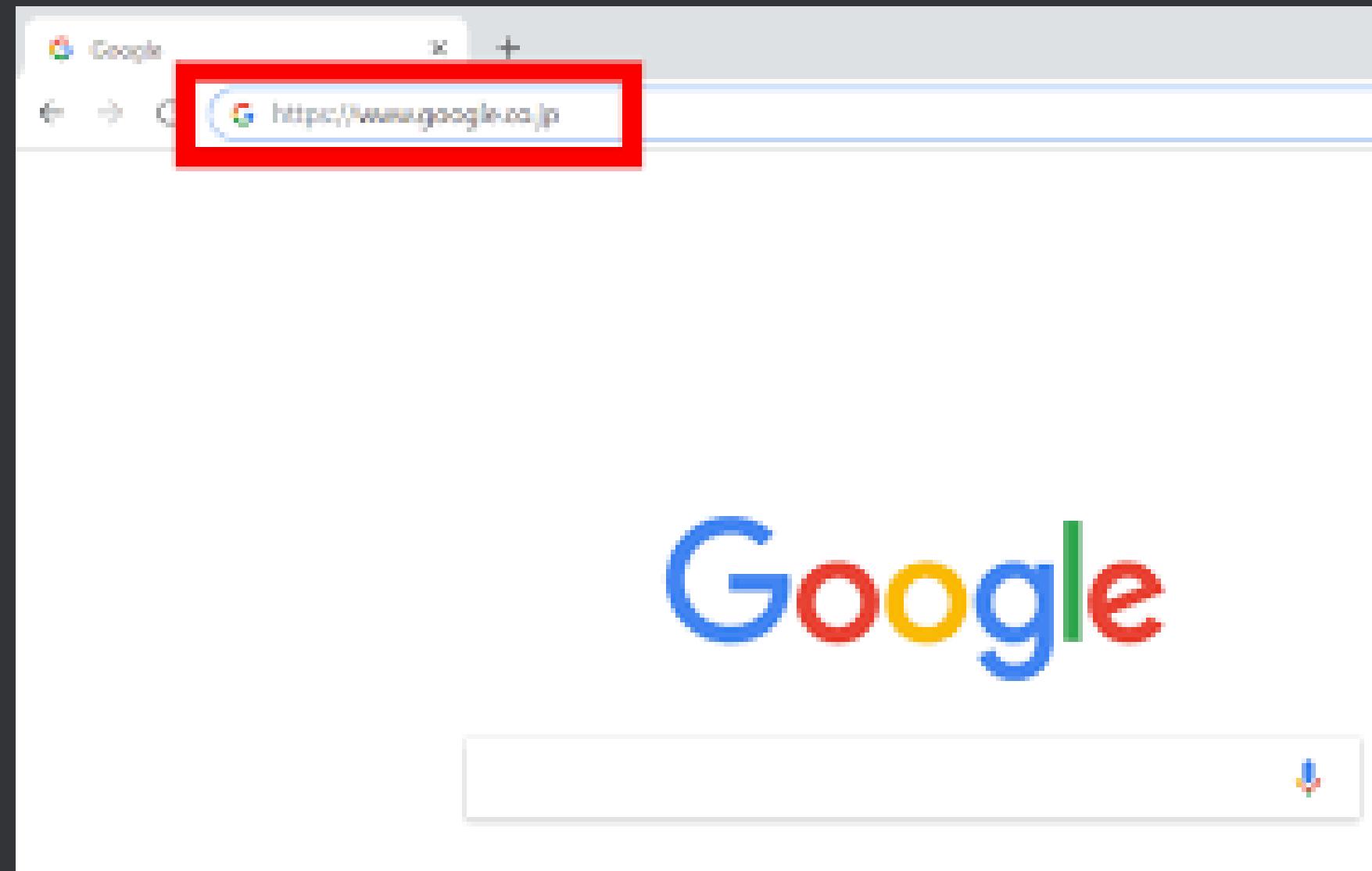
By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to HTTP server

3. Look at response message sent by HTTP server!

# Web Security

## DNS, HTTP

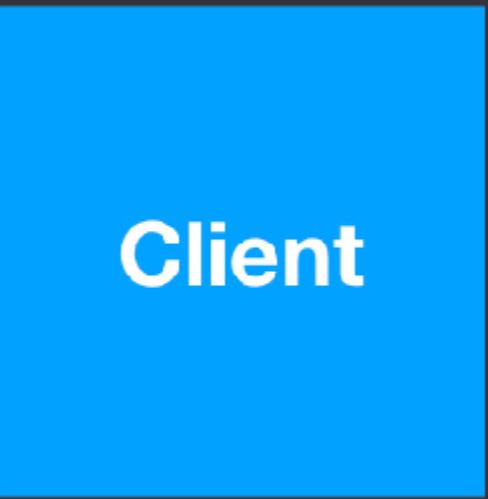
# What happens when you type a URL and press enter?





# Domain Name System (DNS)

# DNS

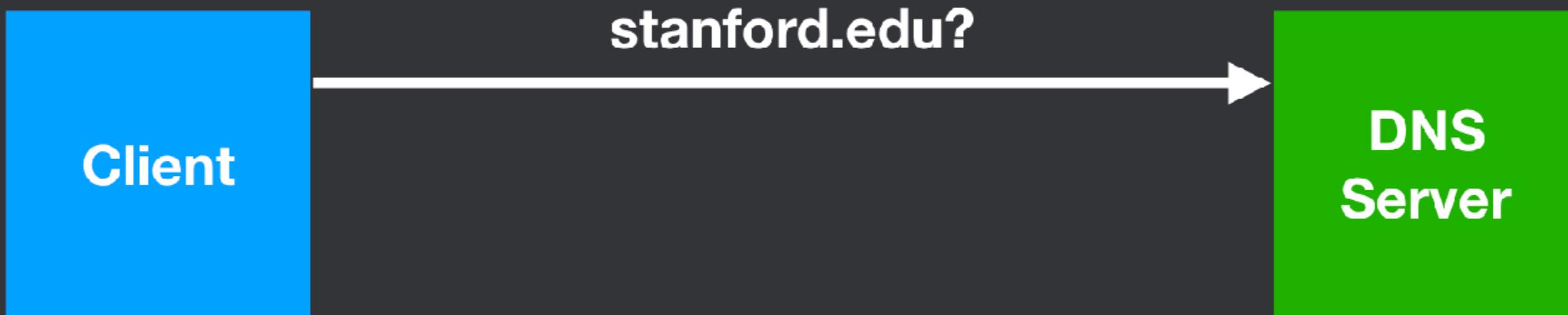


**Client**

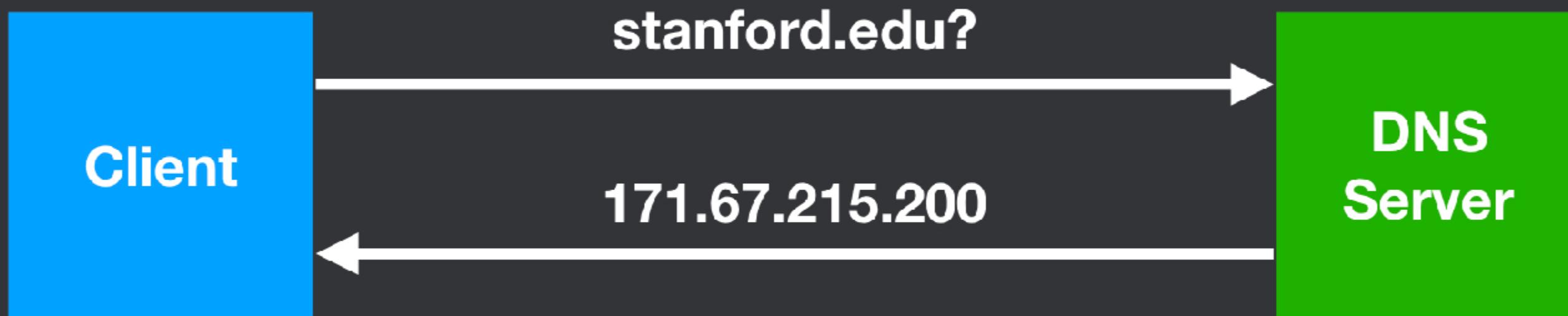


**DNS  
Server**

# DNS



# DNS



# How does the "DNS server" work?

# DNS

Client

DNS  
Recursive  
Resolver

# DNS

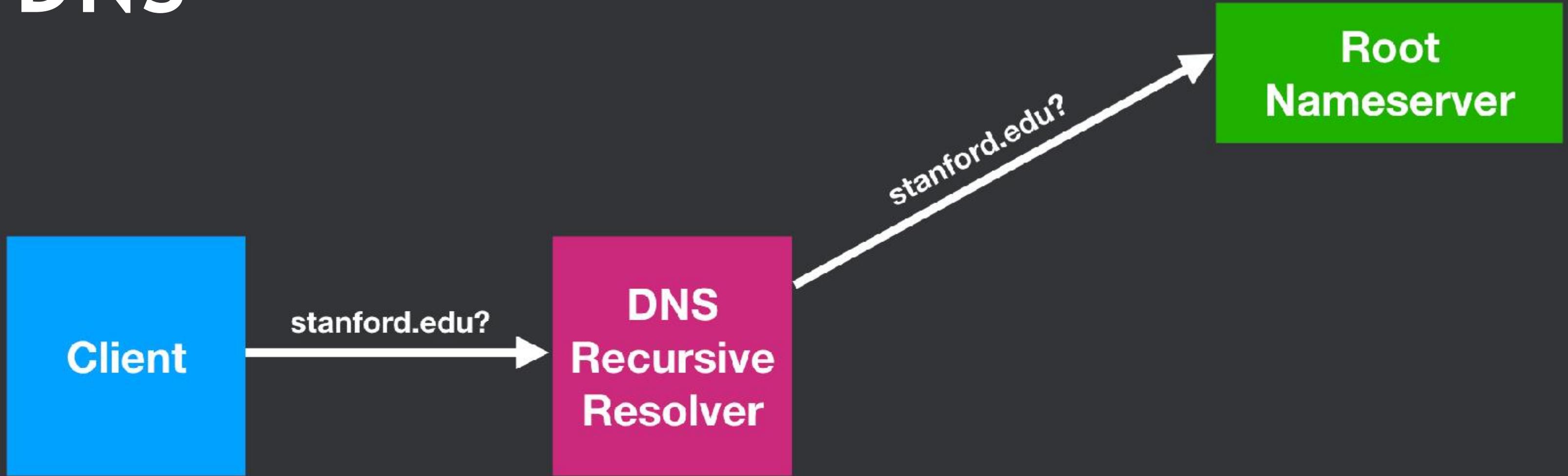


# DNS

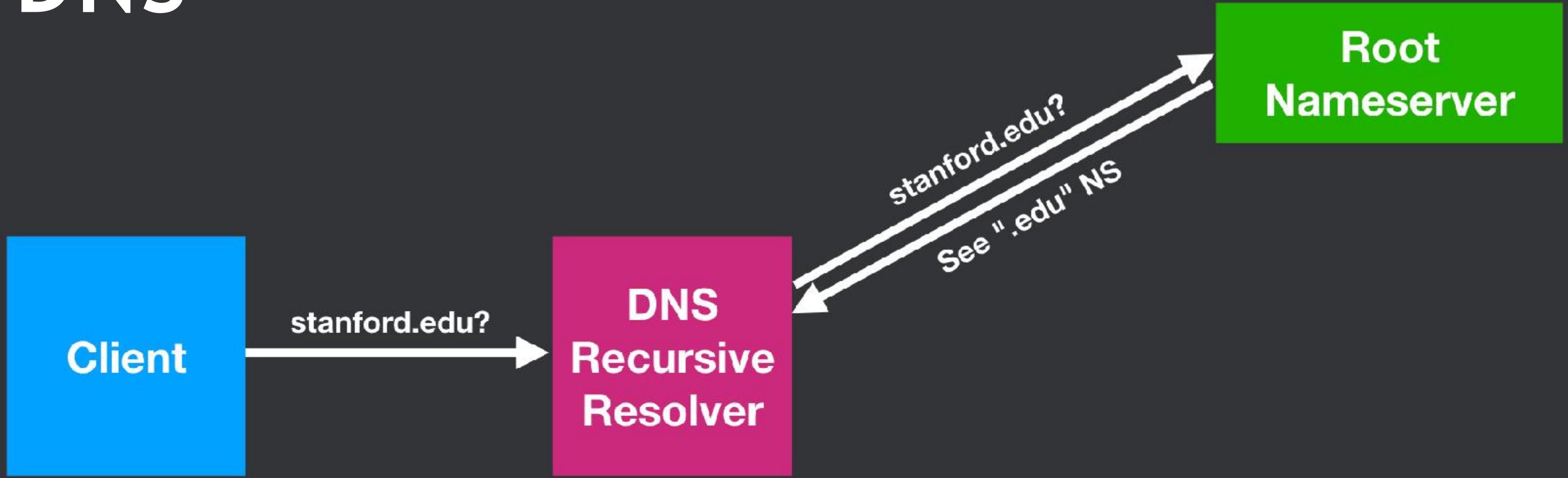
Root  
Nameserver



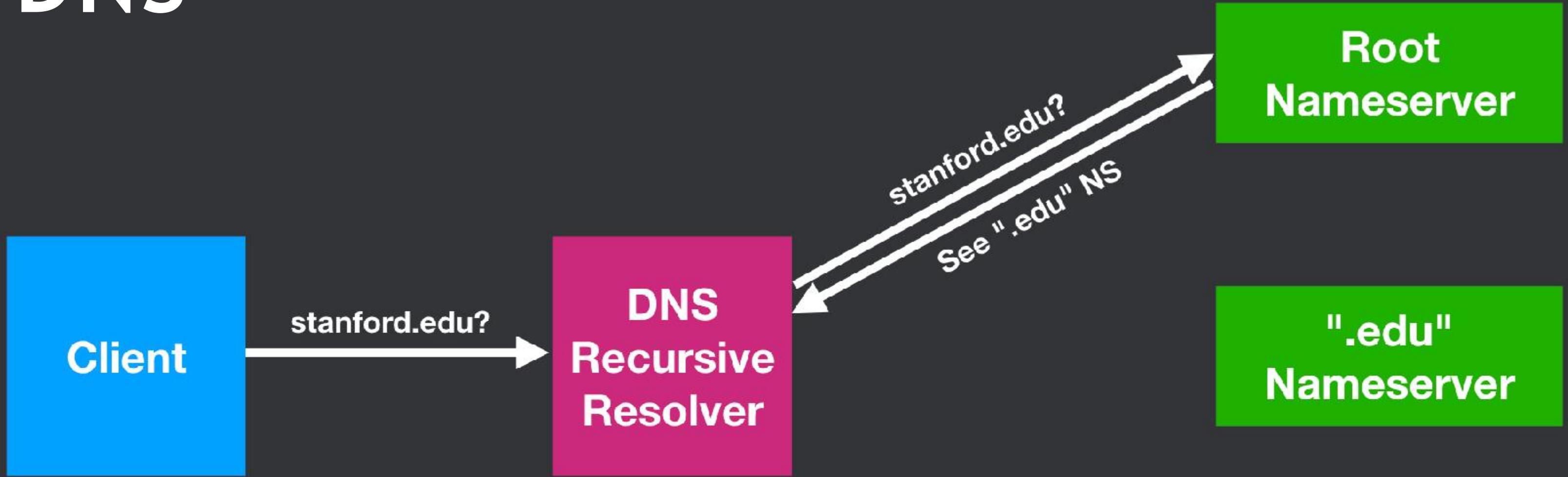
# DNS



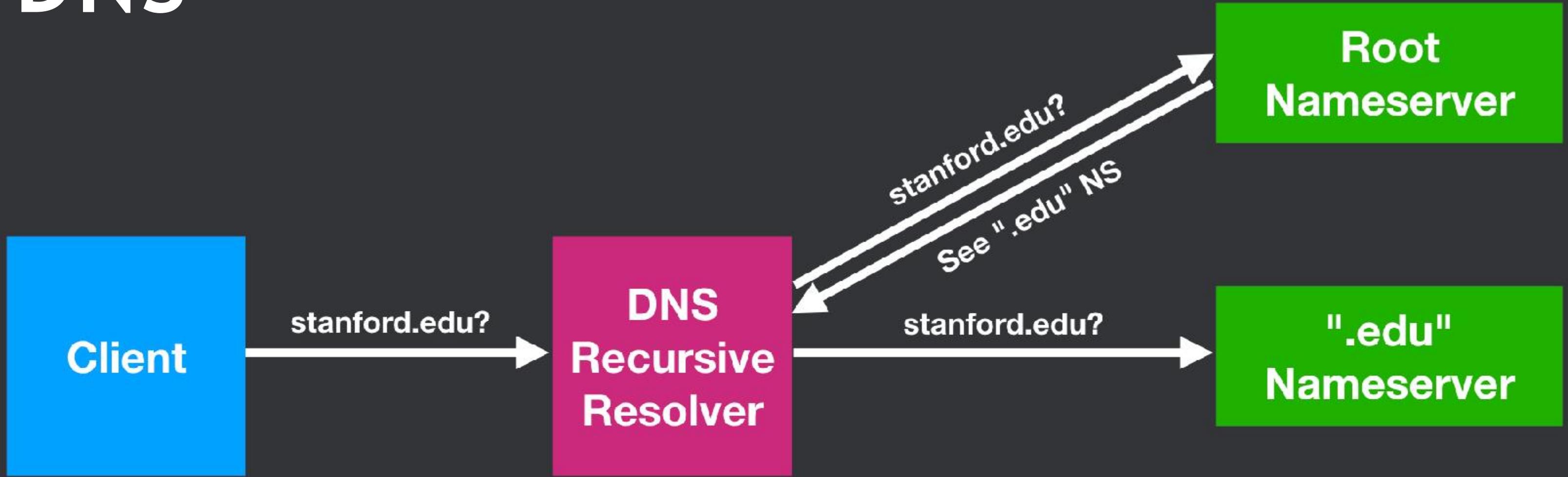
# DNS



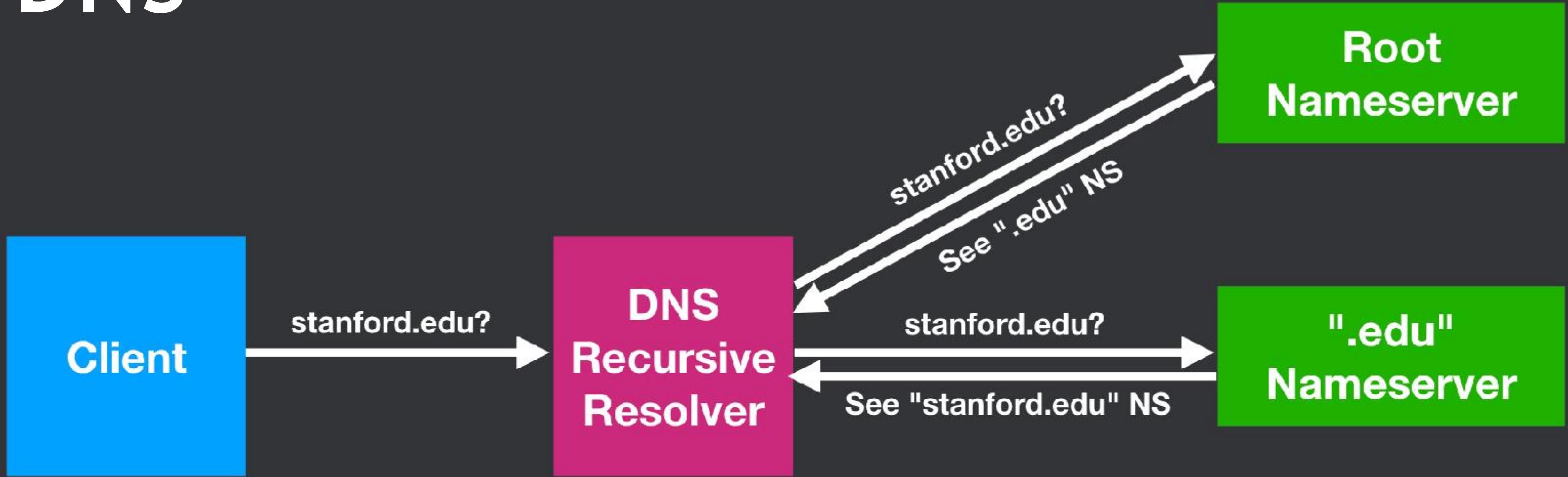
# DNS



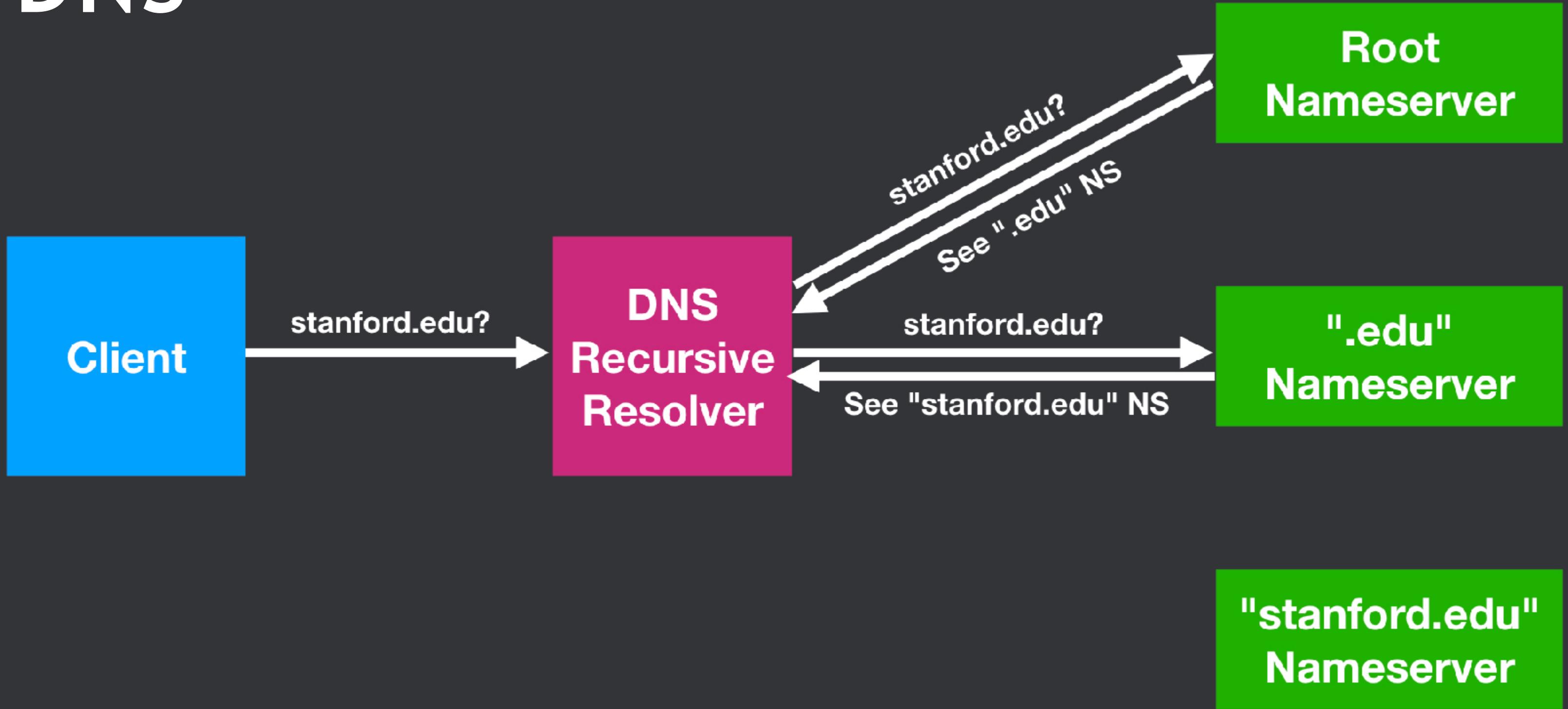
# DNS



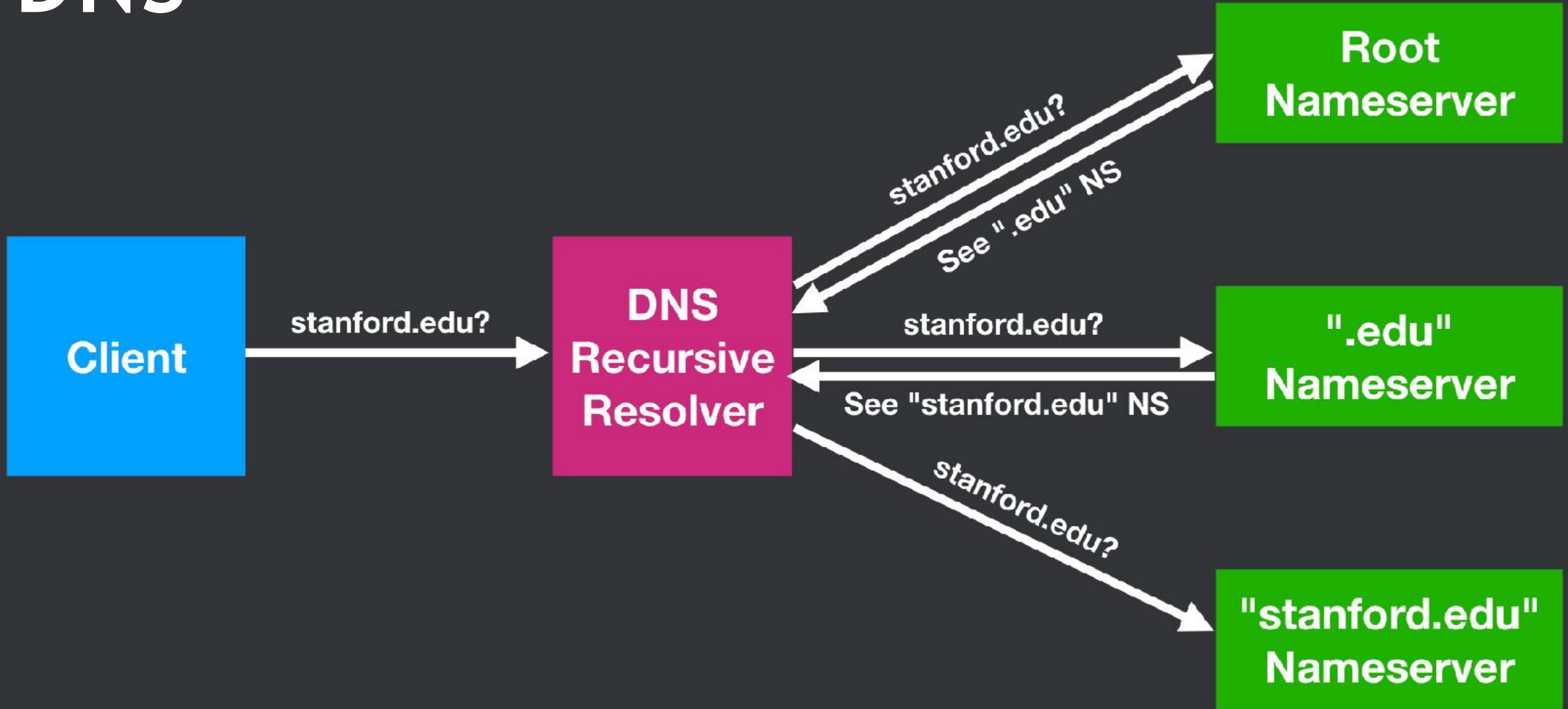
# DNS



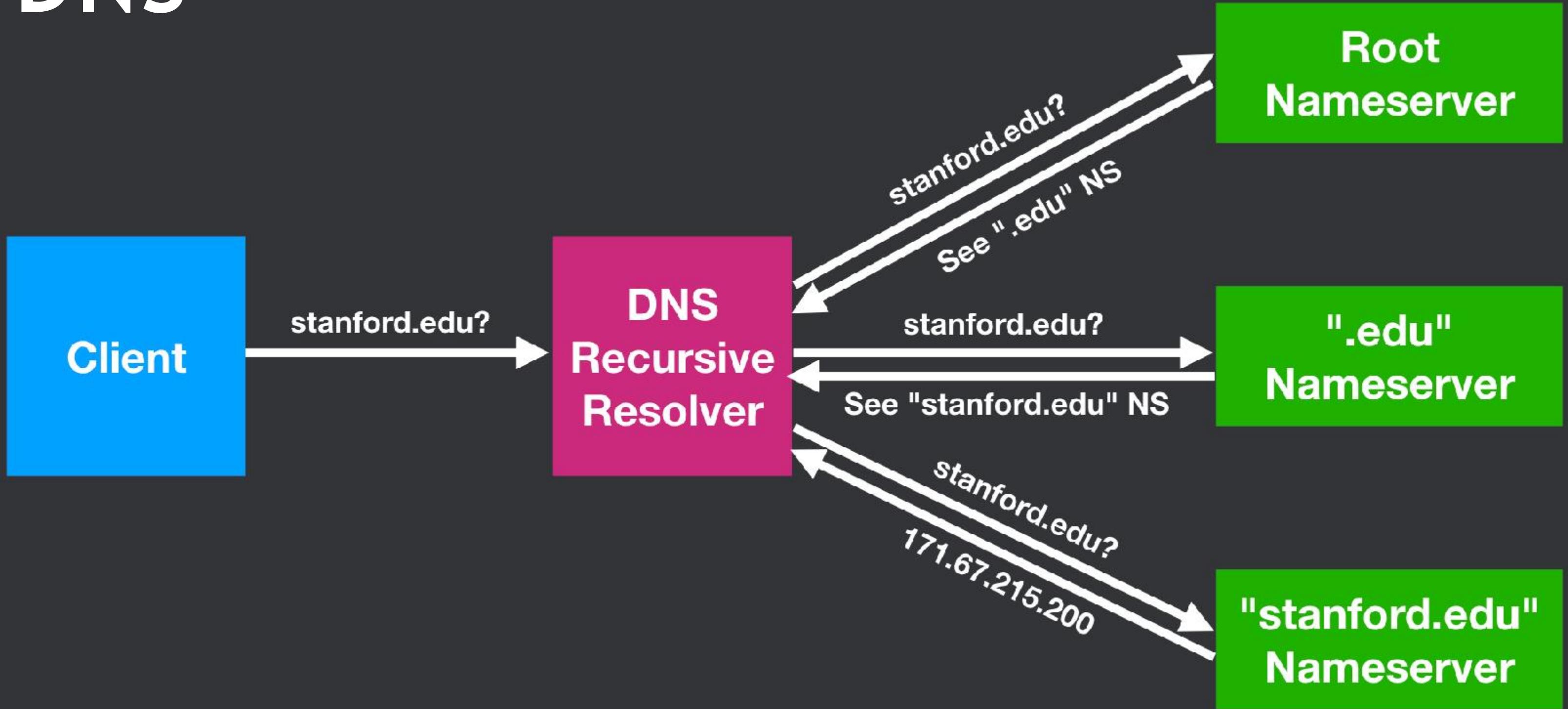
# DNS



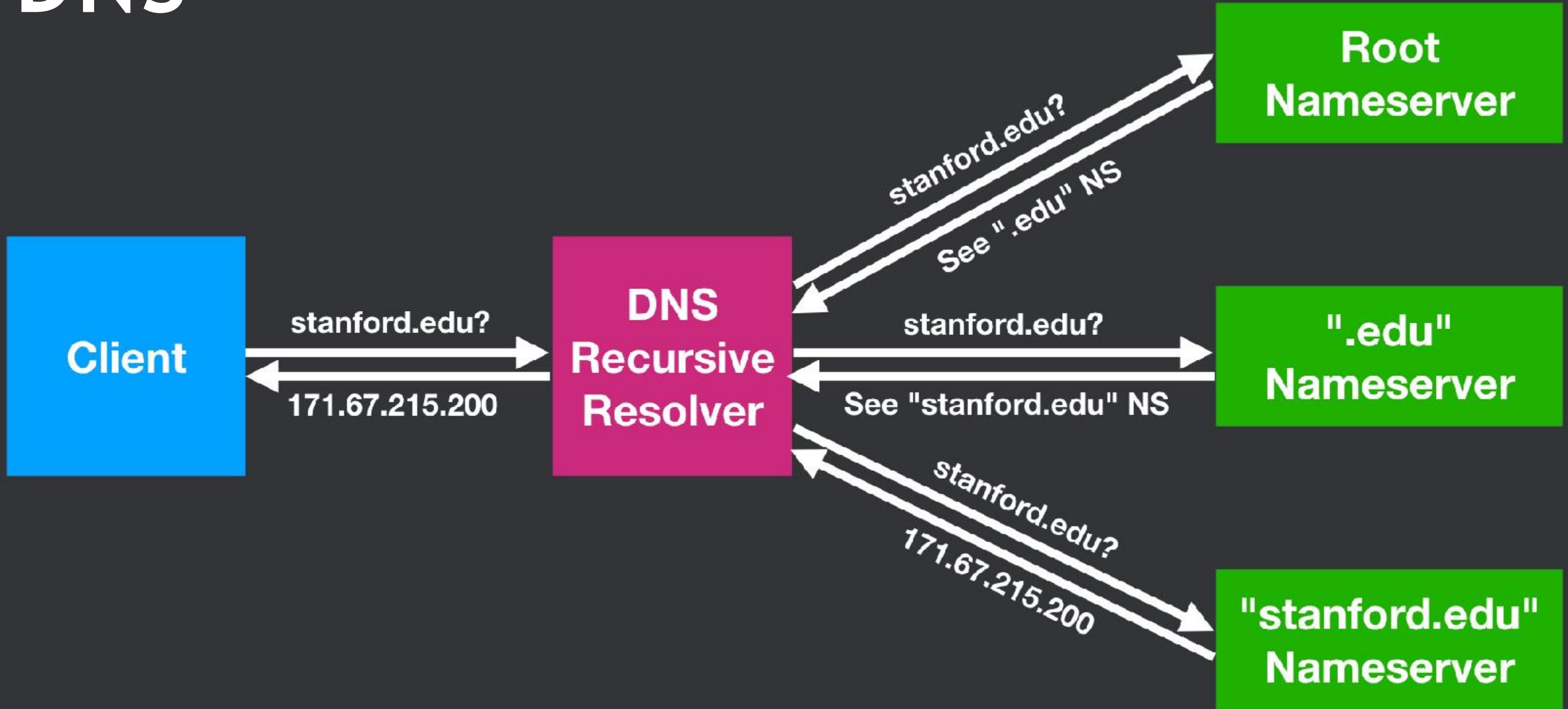
# DNS



# DNS



# DNS



# What happens when you type a URL and press enter?

1. Client asks **DNS Recursive Resolver** to lookup a hostname (**stanford.edu**).
2. **DNS Recursive Resolver** sends DNS query to **Root Nameserver**
  - **Root Nameserver** responds with IP address of **TLD Nameserver** (".**edu**" Nameserver)
3. **DNS Recursive Resolver** sends DNS query to **TLD Nameserver**
  - **TLD Nameserver** responds with IP address of **Domain Nameserver** ("**stanford.edu**" Nameserver)
4. **DNS Recursive Resolver** sends DNS query to **Domain Nameserver**
  - **Domain Nameserver** is authoritative, so replies with server IP address.
5. **DNS Recursive Resolver** finally responds to **Client**, sending server IP address (171.67.215.200)

# DNS + HTTP

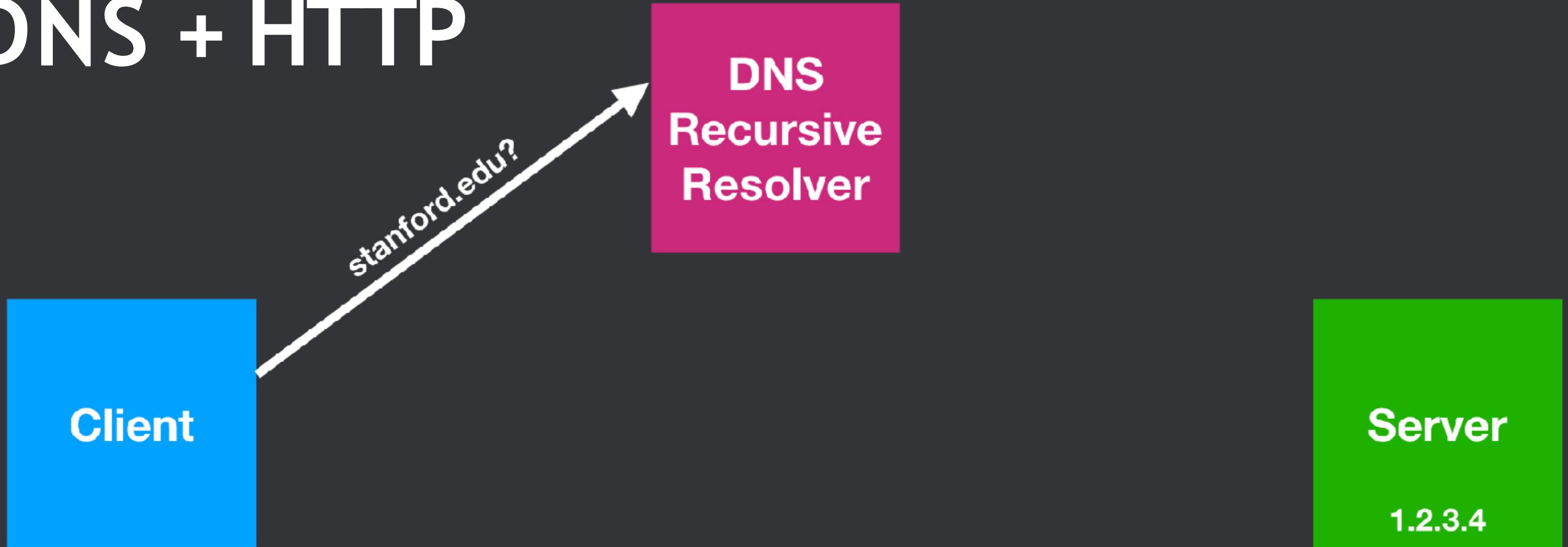
DNS  
Recursive  
Resolver

Client

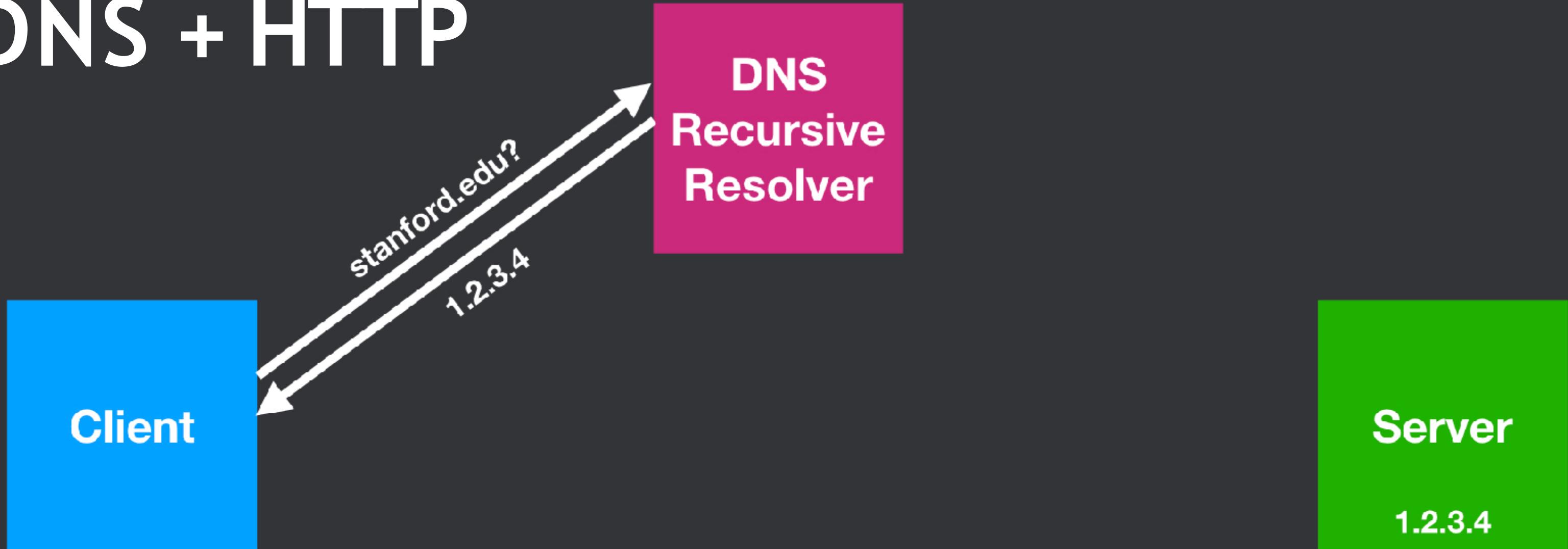
Server

1.2.3.4

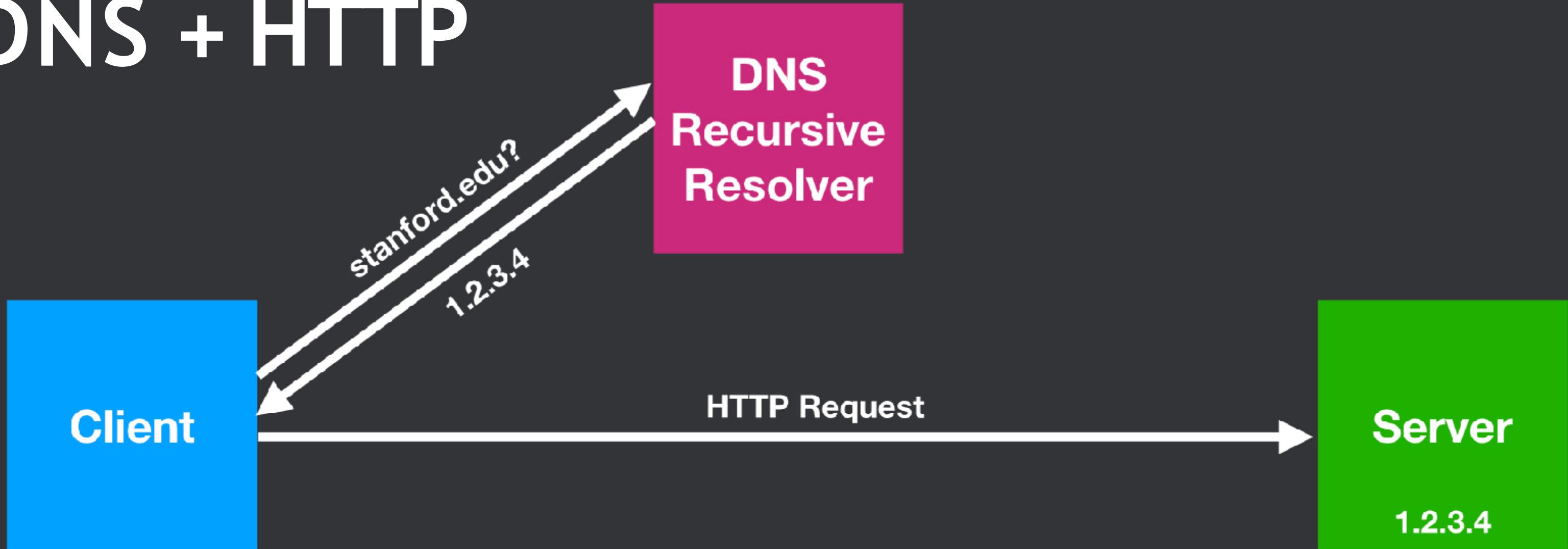
# DNS + HTTP



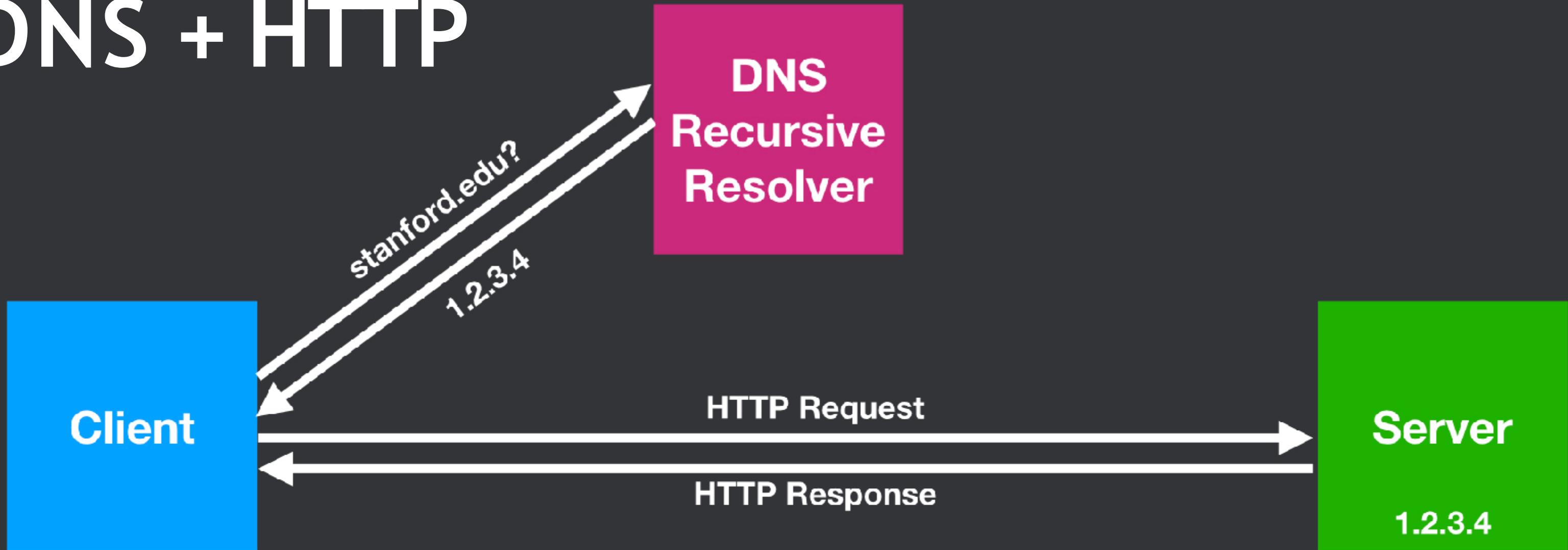
# DNS + HTTP



# DNS + HTTP



# DNS + HTTP

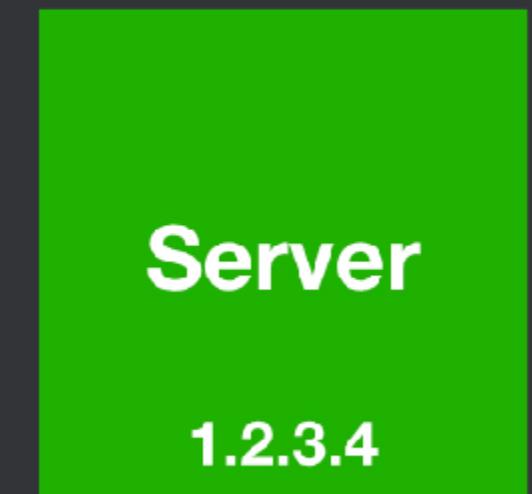
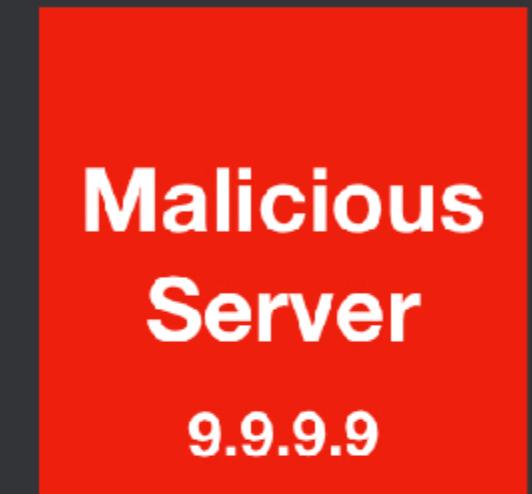
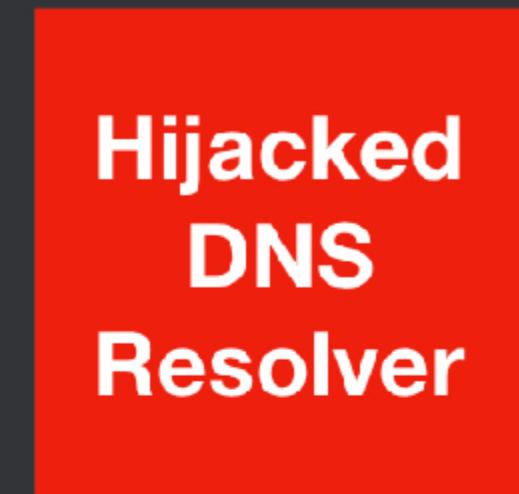
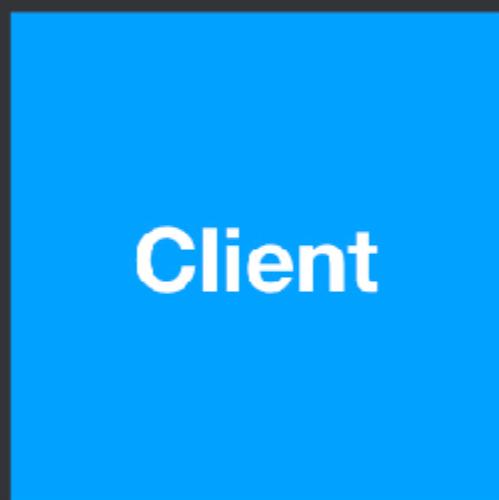


# Attacks on DNS

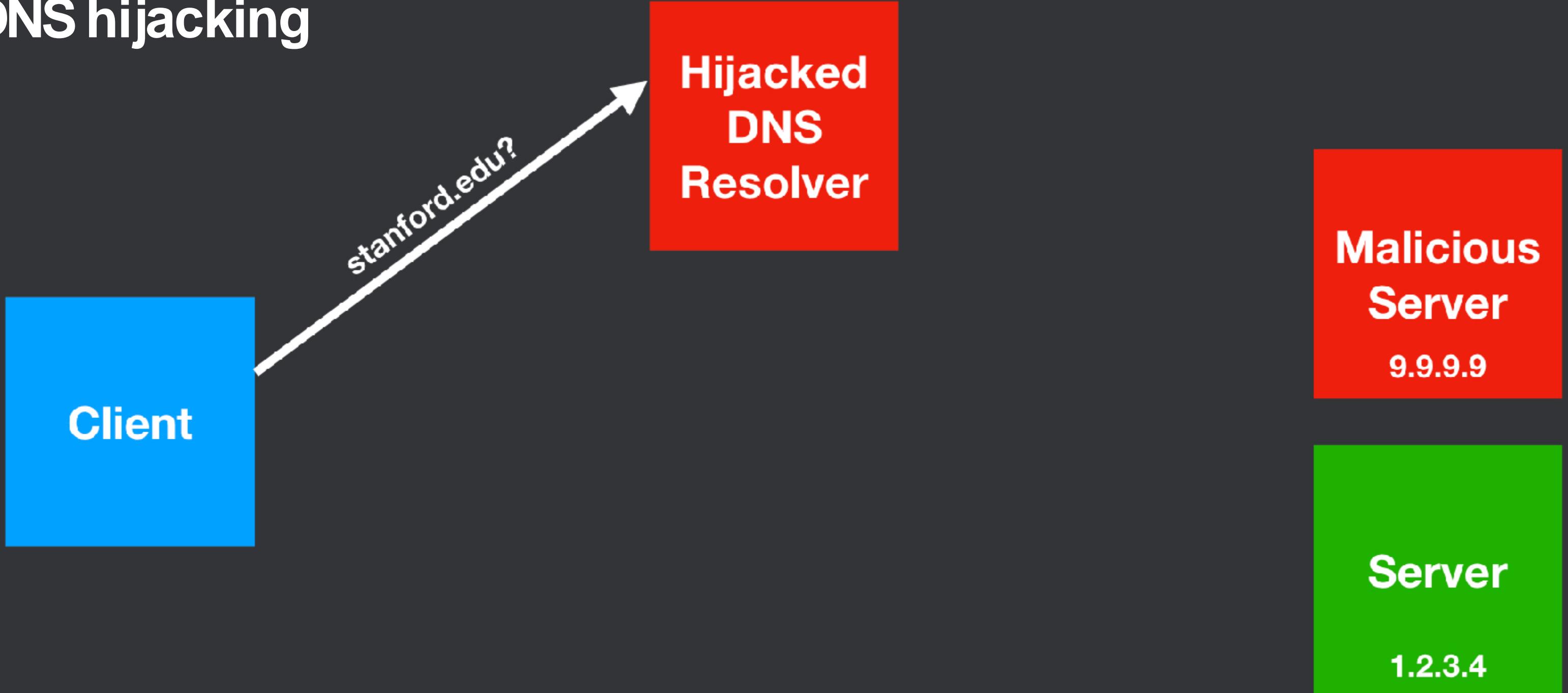
# DNS hijacking

- Attacker changes target DNS record to point to attacker IP address
  - Causes all site visitors to be directed to attacker's web server
- Motivation
  - Phishing
  - Revenue through ads, cryptocurrency mining, etc.
- How do they do it?

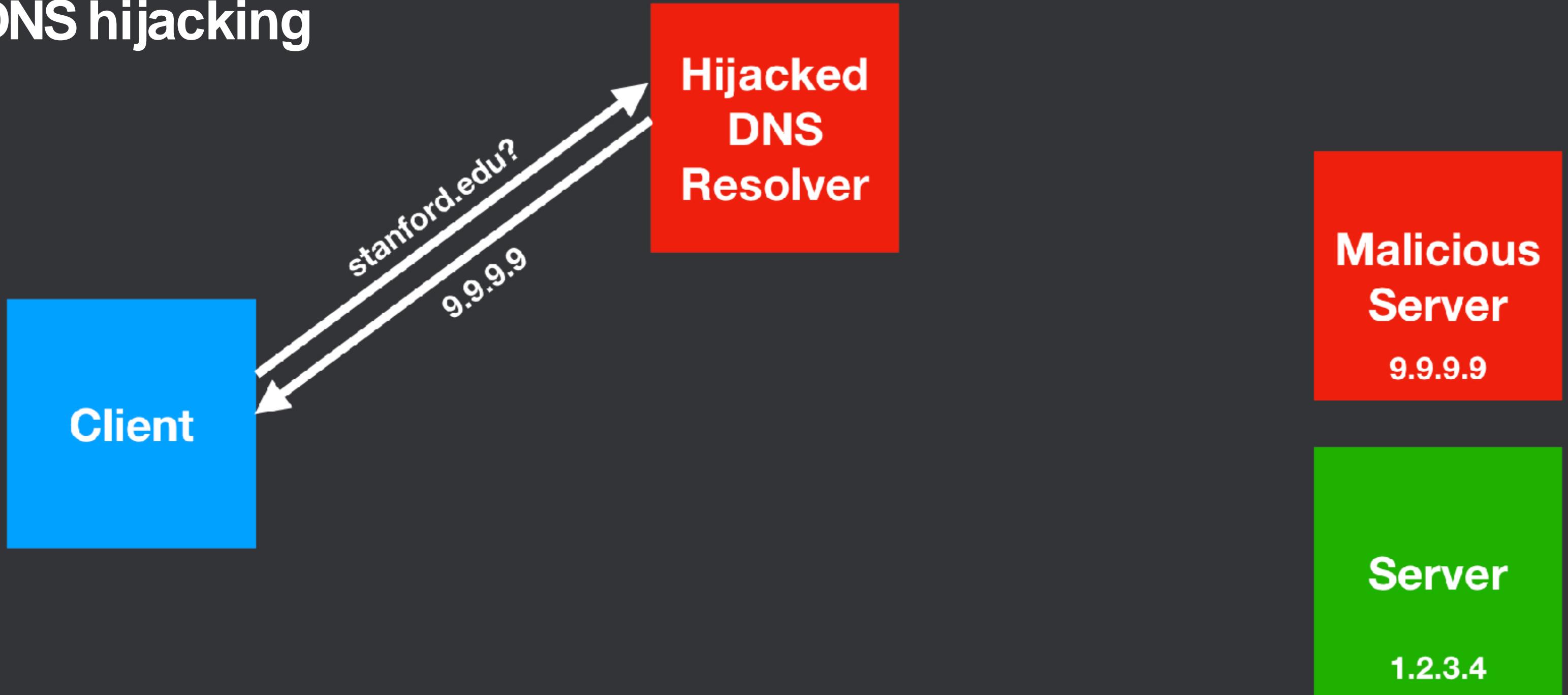
# DNS hijacking



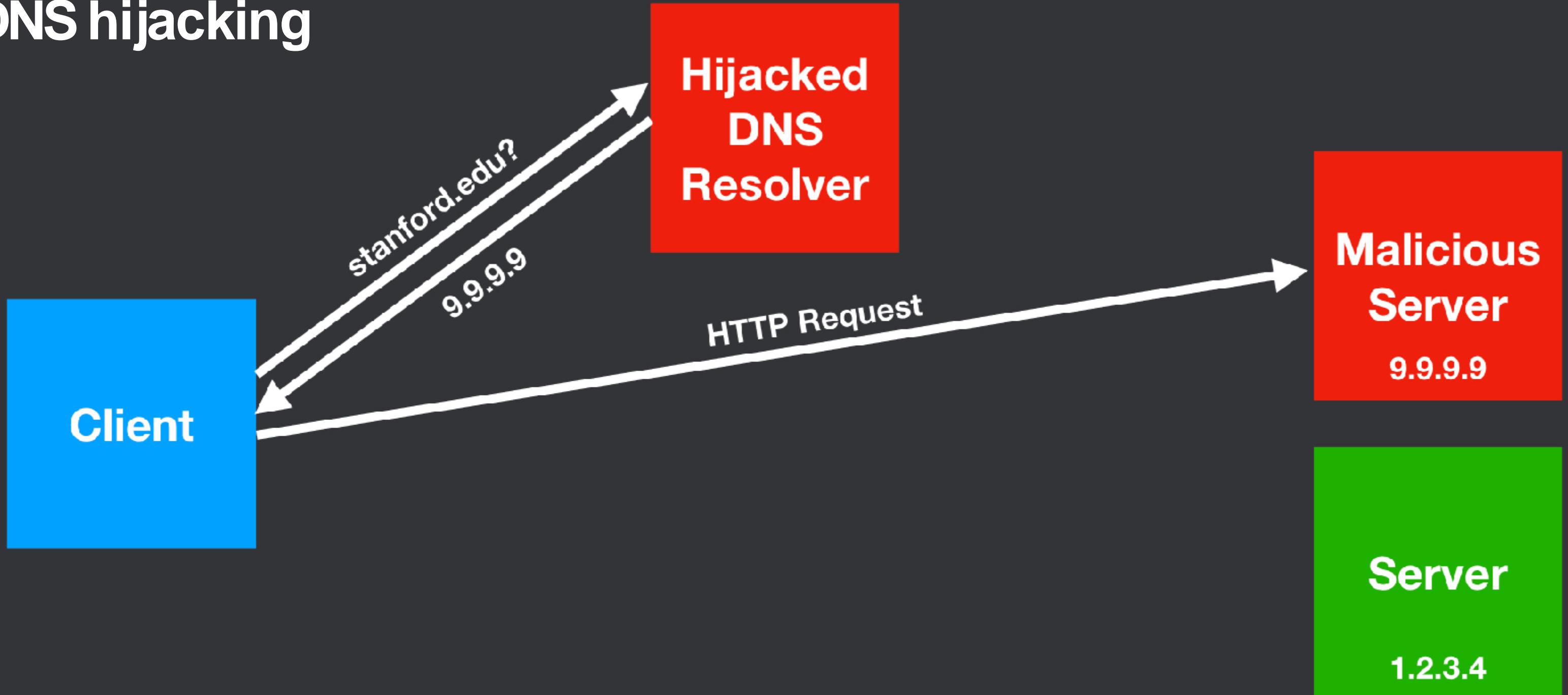
# DNS hijacking



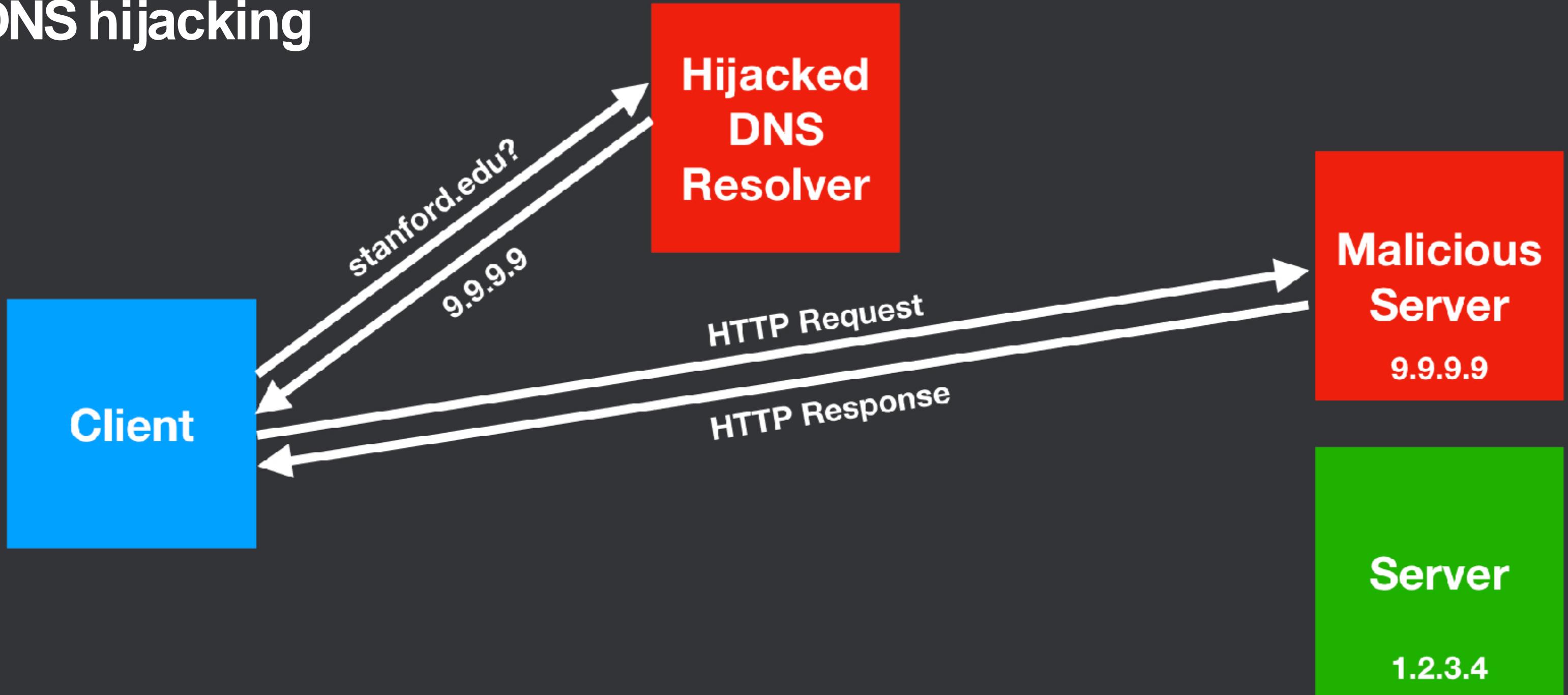
# DNS hijacking



# DNS hijacking



# DNS hijacking



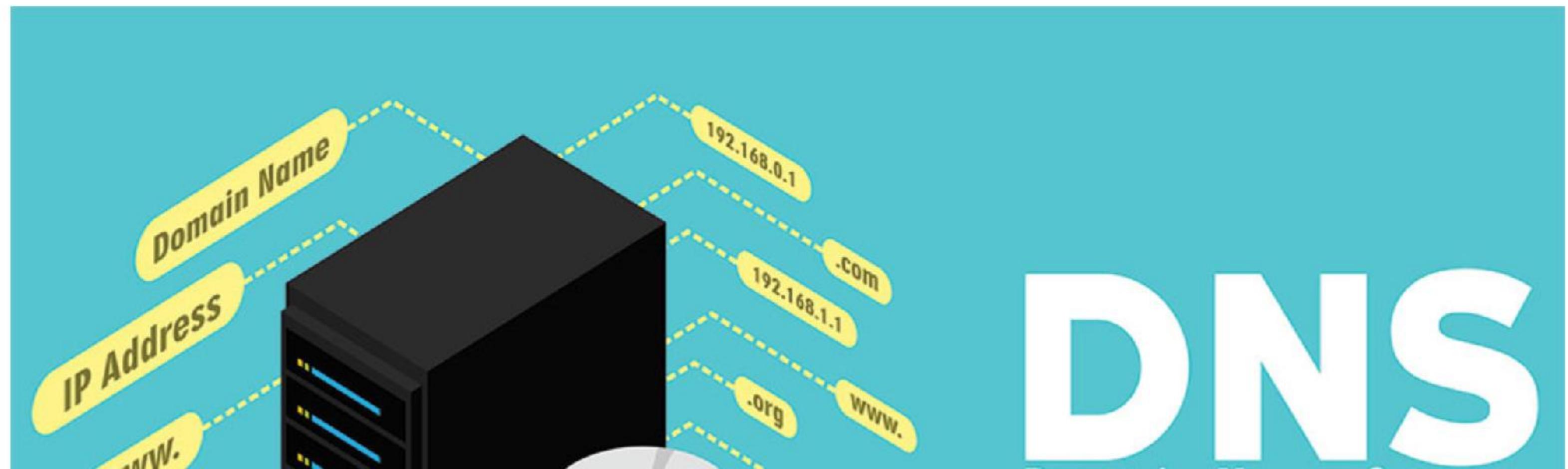
# DNS hijacking vectors

- Hijacked recursive DNS resolver (shown previously)
- Hijacked DNS nameserver
- Compromised user account at DNS provider
- Malware changes user's local DNS settings
- Hijacked router

University Security

# 86% of Education Industry Experienced DNS Attack in Past Year (Sept. 2019)

The education industry also has the lowest adoption of network security policy management automation at only 8%, according to a new report.



# DNS privacy

- Queries are in plaintext
- ISPs have been known to sell this data
- **Tip:** Consider switching your DNS settings to **1.1.1.1** or another provider with a good privacy policy

Cloudflare (public DNS resolver): 1.1.1.1 - 1.0.0.1 (fast due to less users- secure as delete data in 24 hours, will not store your history)

Google: 8888 - 8844



FIREFOX

# What's next in making Encrypted DNS-over-HTTPS the Default

Selena Deckelmann | September 6, 2019

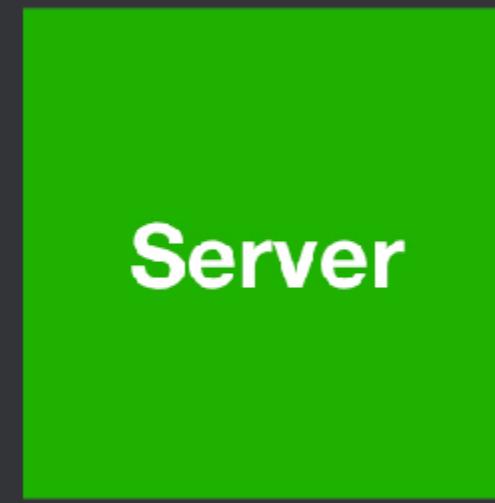
In 2017, Mozilla began working on the DNS-over-  
HTTPS (DoH) protocol, and since [June 2018](#) we've been running experiments in

# What happens when you type a URL and press enter?

# HTTP

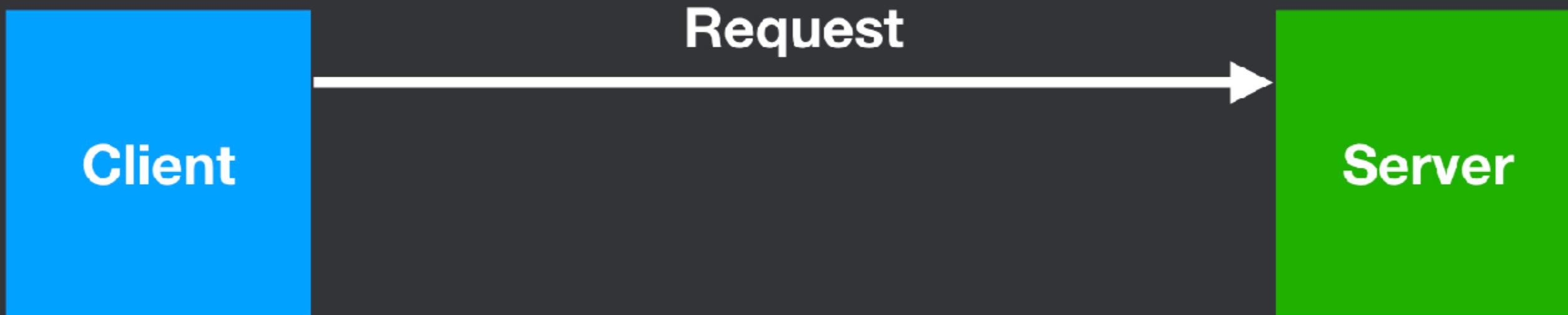


Client

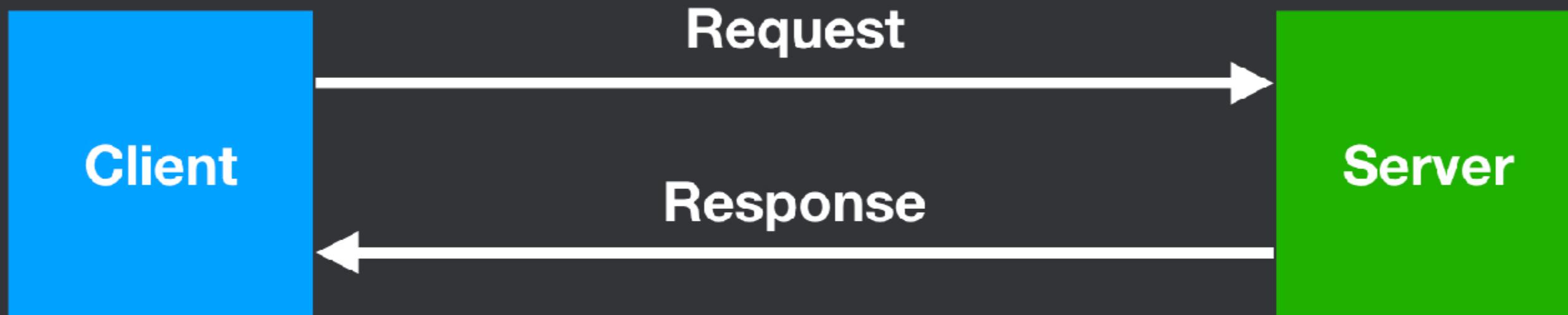


Server

# HTTP



# HTTP



# Demo: Make an HTTP request

# Demo: Make an HTTP request

```
curl https://twitter.com
```

```
curl https://twitter.com > twitter.html
```

```
open twitter.html
```

# HTTP request

GET / HTTP/1.1

Host: twitter.com

User-Agent: Mozilla/5.0 ...

**GET** / **HTTP/1.1**

**Method**

**Path**

**Protocol Version**

# HTTP response

HTTP/1.1 200 OK

**Content-Length:** 9001

**Content-Type:** text/html; charset=UTF-8

**Date:** Tue, 24 Sep 2019 20:30:00 GMT

<!DOCTYPE html ...

**HTTP/1.1 200 OK**

**Protocol Version**

**Status Code**

**Status Message**

# HTTP

- **Client-server model** - Client asks server for resource, server replies
- **Simple** - Human-readable text protocol
- **Extensible** - Just add HTTP headers
- **Transport protocol** - Only requirement is reliability
- **Stateless** - Two requests have no relation to each other

# HTTP is stateless?

- Obviously, we interact with "stateful" servers all the time
- "Stateless" means the HTTP protocol itself does not store state
- If state is desired, is implemented as a layer on top of HTTP

# HTTP Status Codes

- **1xx** - Informational ("Hold on")
- **2xx** - Success ("Here you go")
- **3xx** - Redirection ("Go away")
- **4xx** - Client error ("You messed up")
- **5xx** - Server error ("I messed up")

# HTTP Success Codes

- 200 OK - Request succeeded
- 206 Partial Content - Request for specific byte range succeeded

# Range Request

GET /video.mp4 HTTP/1.1

Range: bytes=1000-1499

# Response

HTTP/1.1 206 Partial Content

Content-Range: bytes 1000-1499/1000000

# HTTP Redirection Codes

- **301 Moved Permanently** - Resource has a new permanent URL
- **302 Found** - Resource temporarily resides at a different URL
- **304 Not Modified** - Resource has not been modified since last cached

# HTTP Client Error Codes

- **400 Bad Request** - Malformed request
- **401 Unauthorized** - Resource is protected, need to authorize
- **403 Forbidden** - Resource is protected, denying access
- **404 Not Found** - We know this one

# HTTP Server Error Codes

- **500 Internal Server Error** - Generic server error
- **502 Bad Gateway** - Server is a proxy; backend server is unreachable
- **503 Service Unavailable** - Server is overloaded or down for maintenance
- **504 Gateway Timeout** - Server is a proxy; backend server responded too slowly

# HTTP with a proxy server



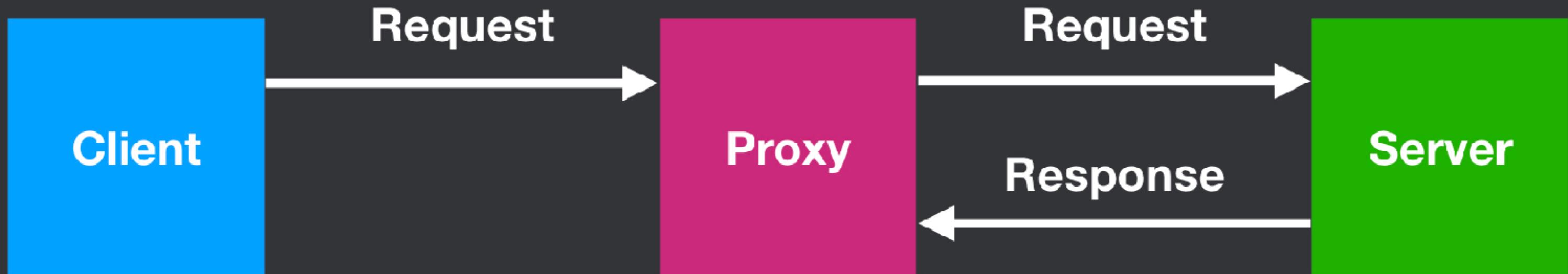
# HTTP with a proxy server



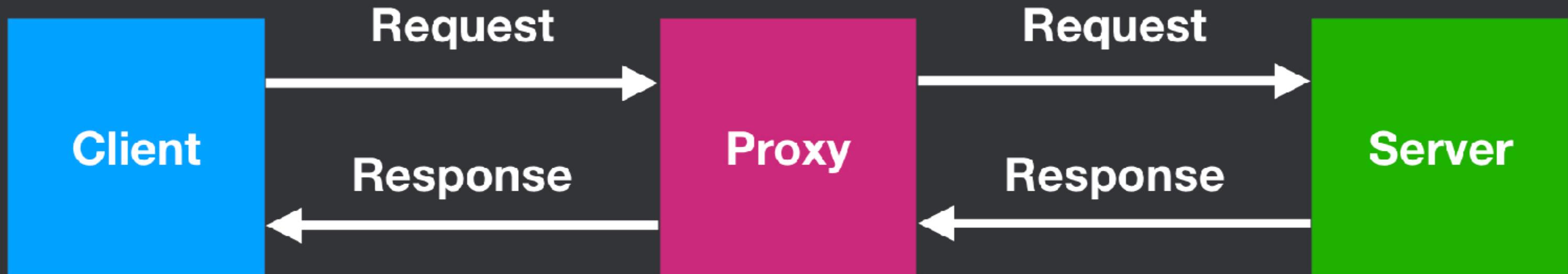
# HTTP with a proxy server



# HTTP with a proxy server



# HTTP with a proxy server



# HTTP proxy servers

- Can cache content
- Can block content (e.g. malware, unauthorized content)
- Can modify content
- Can sit in front of many servers ("reverse proxy")

# HTTP request

GET / HTTP/1.1

Host: example.com

User-Agent: Mozilla/5.0 ...

**Host:** example.com

**Header Name**

**Header Value**

# HTTP headers

- Let the client and the server pass additional information with an HTTP request or response
- Essentially a map of key-value pairs
- Allow experimental extensions to HTTP without requiring protocol changes

# Useful HTTP request headers

- **Host** - The domain name of the server (e.g. `example.com`)
- **User-Agent** - The name of your browser and operating system
- **Referer** - The webpage which led you to this page (misspelled)
- **Cookie** - The cookie server gave you earlier; keeps you logged in
- **Range** - Specifies a subset of bytes to fetch

# Useful HTTP request headers (pt 2)

- **Cache-Control** - Specifies if you want a cached response or not
- **If-Modified-Since** - Only send resource if it changed recently
- **Connection** - Control TCP socket (e.g. **keep-alive** or **close**)
- **Accept** - Which type of content we want (e.g. **text/html**)
- **Accept-Encoding** - Encoding algorithms we understand (e.g. **gzip**)
- **Accept-Language** - What language we want (e.g. **en, fr, es**)

# HTTP response

HTTP/1.1 200 OK

**Content-Length:** 9001

**Content-Type:** text/html; charset=UTF-8

**Date:** Tue, 24 Sep 2019 20:30:00 GMT

<!DOCTYPE html ...

# Useful HTTP response headers

- **Date** - When response was sent
- **Last-Modified** - When content was last modified
- **Cache-Control** - Specifies whether to cache response or not
- **Expires** - Discard response from cache after this date
- **Set-Cookie** - Set a cookie on the client
- **Vary** - List of headers which affect response; used by cache

# Vary on user language

HTTP/1.1 200 OK

**Cache-Control:** public, max-age=31536000 // (secs-1 yr)

**Vary:** Accept-Language

# Useful HTTP response headers (pt 2)

- **Location** - URL to redirect the client to (used with 3xx responses)
- **Connection** - Control TCP socket (e.g. `keep-alive` or `close`)
- **Content-Type** - Type of content in response (e.g. `text/html`)
- **Content-Encoding** - Encoding of the response (e.g. `gzip`)
- **Content-Language** - Language of the response (e.g. `en`)
- **Content-Length** - Length of the response in bytes

**HTML**

**CSS**

**JS**

**HTTP**

**TLS**

**TCP**

**IP**

**HTML**

**CSS**

**JS**

**Hypertext Transfer Protocol**

**Transport Layer Security**

**Transmission Control Protocol**

**Internet Protocol**

# Demo: Implement an HTTP client

- Not magic!
- Steps:
  - Open a TCP socket
  - Send HTTP request text over the socket
  - Read the HTTP response text from the socket

# Implement an HTTP client

```
const net = require('net')
```

JavaScript and Node.js example

```
const socket = net.createConnection({
  host: 'example.com',
  port: 80
})
```

```
const request = `
```

```
GET / HTTP/1.1
```

```
Host: example.com
```

```
`.slice(1)
```

```
socket.write(request)
socket.pipe(process.stdout)
```

write() function writes data from a buffer on a socket. The socket must be a connected socket.

It redirects all the data from the readable stream (server) to the writable stream (client)

# Implement an HTTP client (take 2)

```
const dns = require('dns')
const net = require('net')
```

```
dns.lookup('example.com', (err, address) => { if (err)
    throw err})
```

```
const socket = net.createConnection({
  host: address,
  port: 80
})
```

```
const request = ` GET / HTTP/1.1
Host: example.com
`.slice(1)
```

```
socket.write(request)
socket.pipe(process.stdout)
})
```

# What happens when you type a URL and press enter?

1. Perform a **DNS lookup** on the hostname (**example.com**) to get an IP address (**1.2.3.4**)
2. Open a **TCP socket** to **1.2.3.4** on port **80** (the HTTP port)
3. Send an **HTTP request** that includes the desired path (**/**)
4. Read the **HTTP response** from the socket
5. Parse the HTML into the DOM
6. Render the page based on the DOM
7. Repeat until all external resources are loaded:
  - If there are pending external resources, make HTTP requests for these (run steps 1-4)
  - Render the resources into the page

**Client**

# DNS Recursive Resolver

**Client**

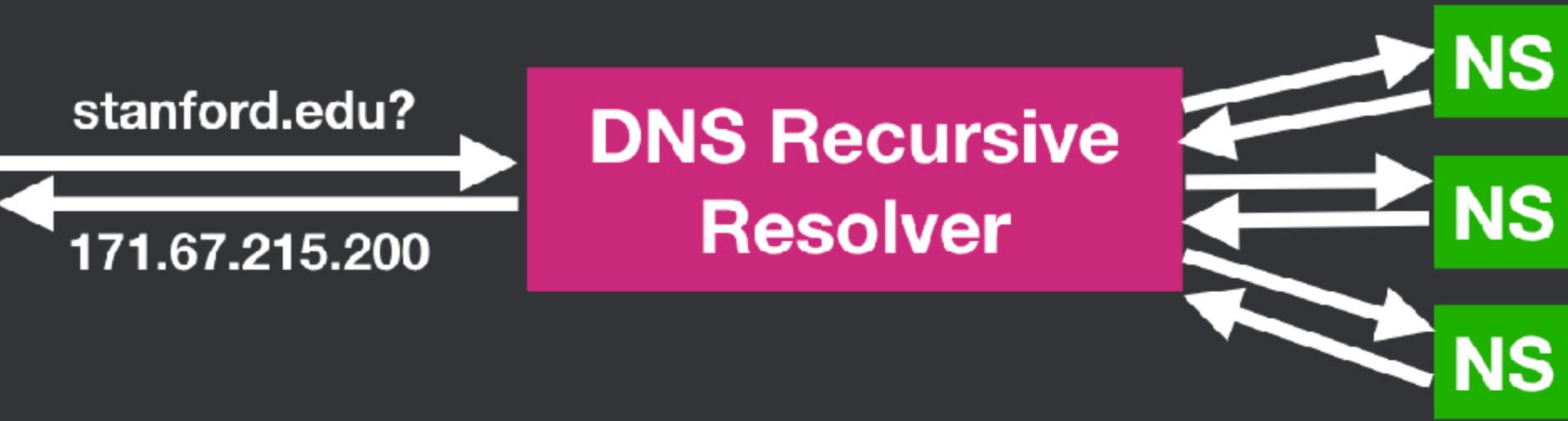
stanford.edu?

**DNS Recursive  
Resolver**

**Client**



**Client**



**Client**



**Server**

171.67.215.200

**Client**

stanford.edu?  
171.67.215.200

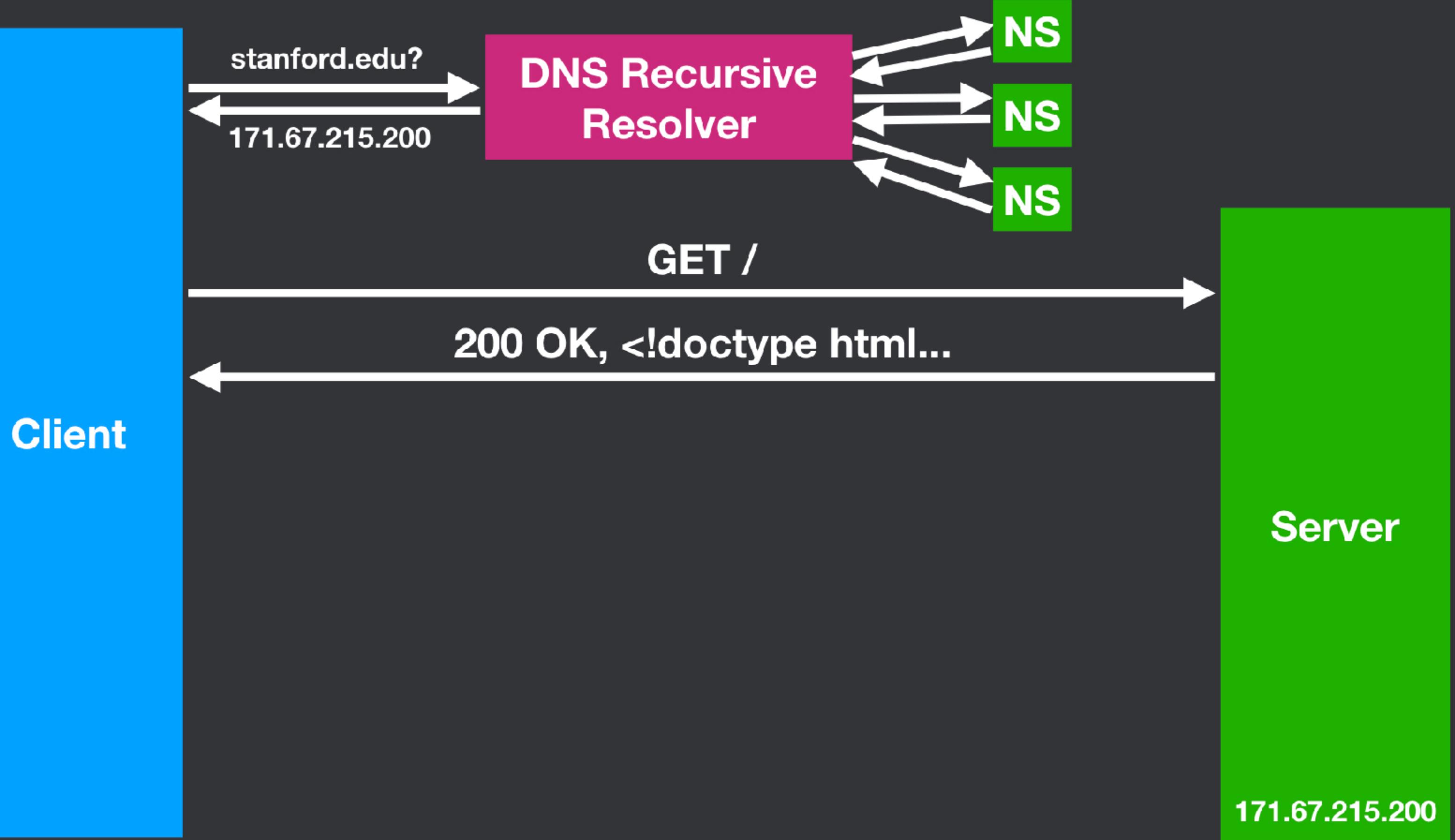
**DNS Recursive  
Resolver**

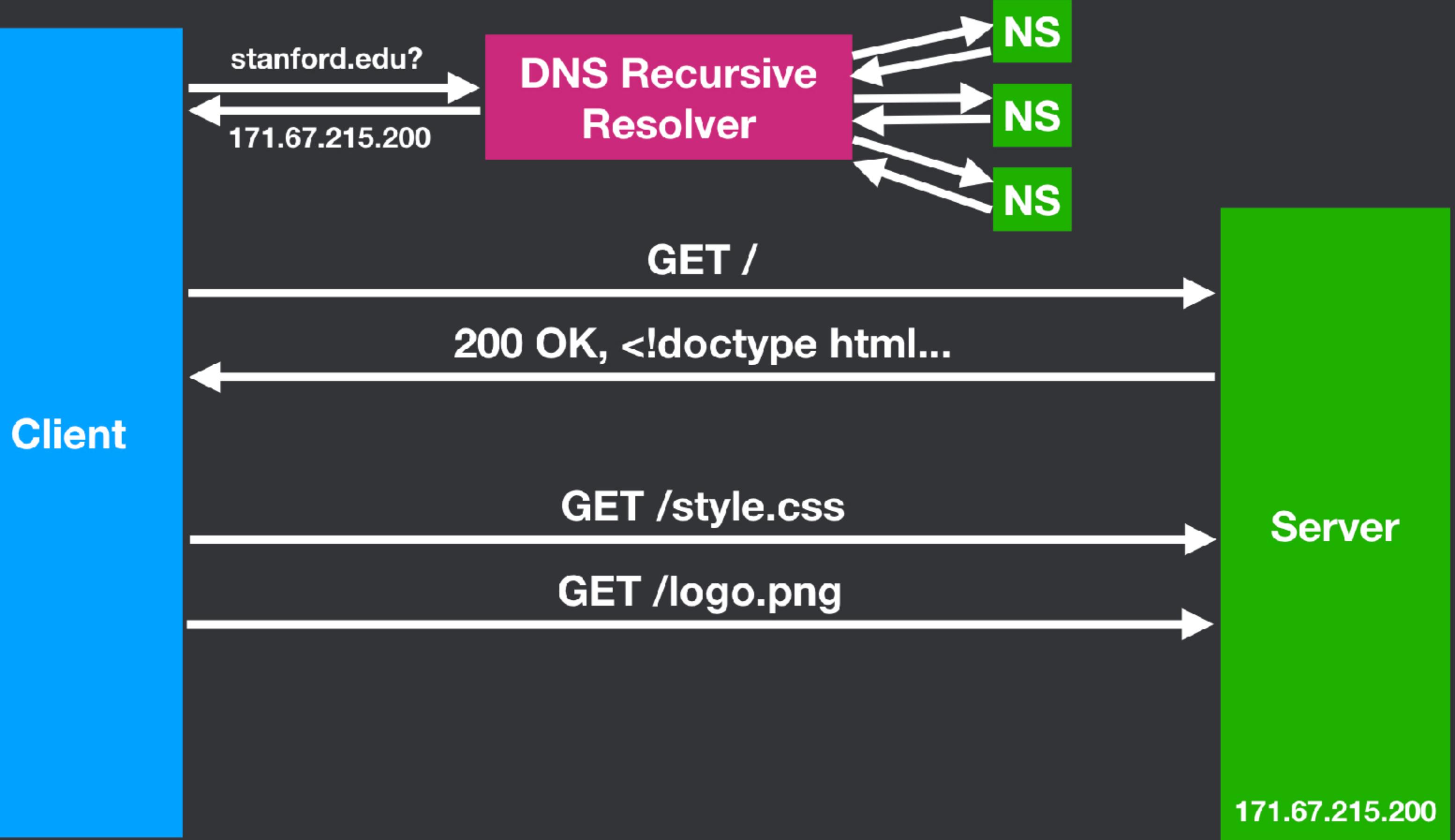
NS  
NS  
NS

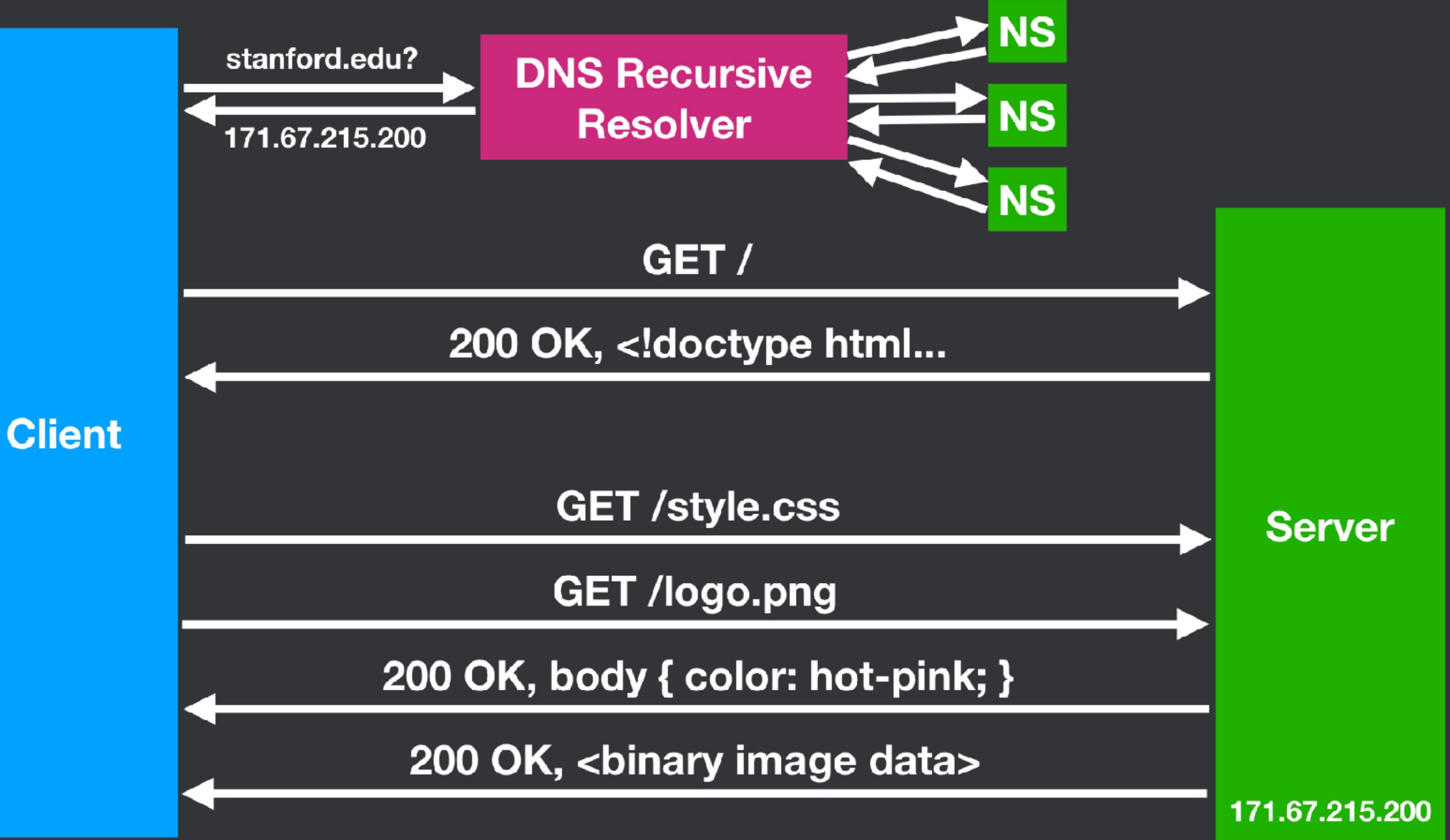
GET /

**Server**

171.67.215.200







# Web Security

## Cookies and Sessions

# Recall: Cookies

# Server sends a cookie with a response

**Set-Cookie: theme=dark;**

**Header Name**

**Cookie Name**

**Cookie Value**

# Client sends a cookie with a request

**Cookie: theme=dark;**

**Header Name**

**Cookie Name**

**Cookie Value**

# Sessions

- Cookies are used by the server to implement sessions
- Goal: Server keeps a set of data related to a user's current "browsing session"
- Examples
  - Logins
  - Shopping carts
  - User tracking

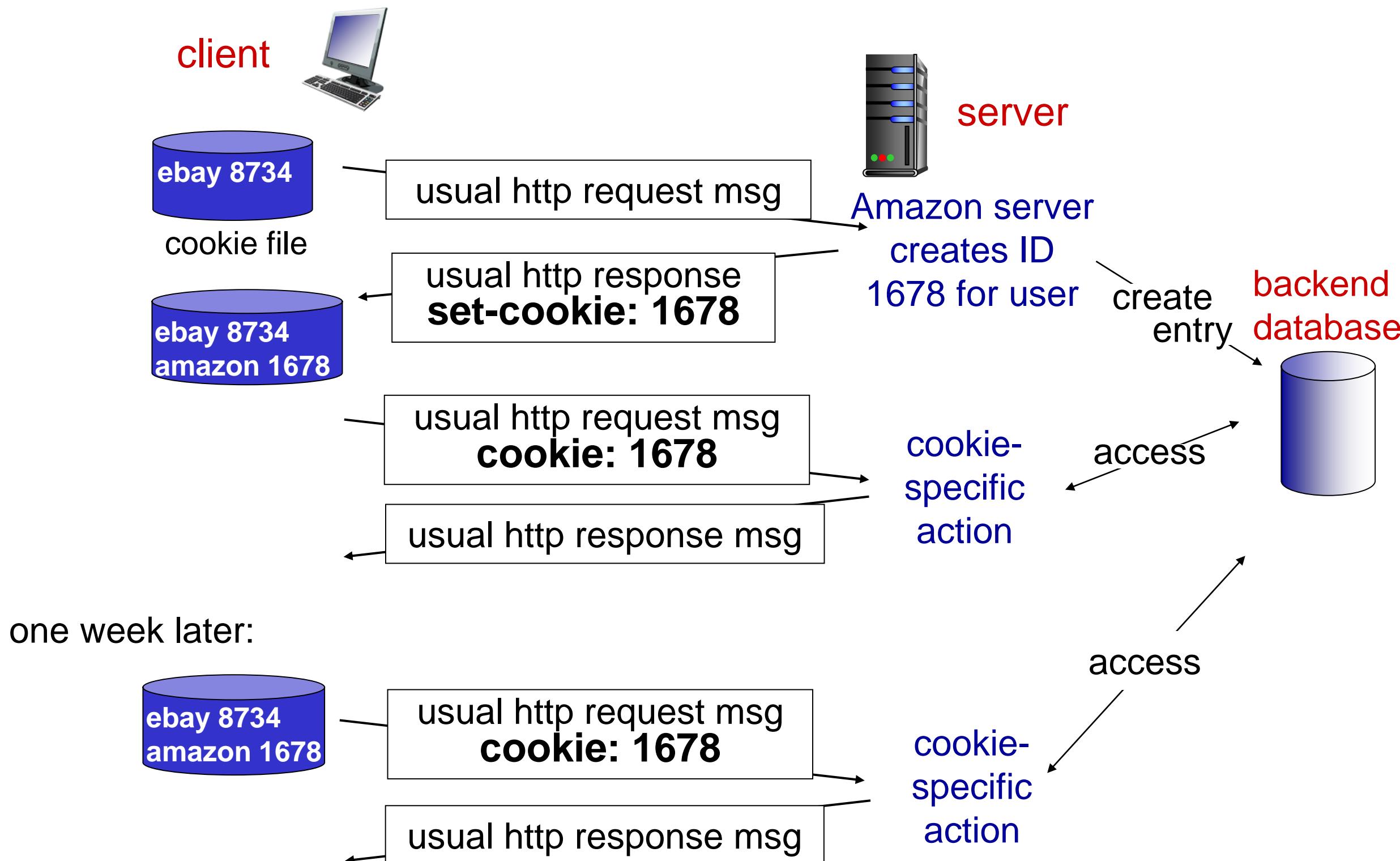
# User-server state: cookies

many Web sites use cookies

example:

- ❖ Ravi always access Internet from PC
- ❖ visits specific e-commerce site for first time
- ❖ when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

# Cookies: keeping “state” (cont.)



## *First HTTP request:*

**POST /login HTTP/1.1**

**Host: example.com**

**username=alice&password=password**

## *HTTP response:*

**HTTP/1.1 200 OK**

**Set-Cookie: username=alice**

**Date: Tue, 24 Sep 2019 20:30:00 GMT**

**<!DOCTYPE html ...**

## *All future HTTP requests:*

**GET /page.html HTTP/1.1**

**Host: example.com**

**Cookie: username=alice;**

# Ambient authority

- Access control - Regulate who can view resources or take actions
- Ambient authority - Access control based on a **global and persistent property** of the requester
  - The alternative is explicit authorization **valid only for a specific action**
- There are four types of ambient authority on the web
  - Cookies - most common, most versatile method
  - IP checking - used at Stanford for library resources
  - Built-in HTTP authentication - rarely used
  - Client certificates - rarely used

# Quick primer: Signature schemes

- Triple of algorithms ( $G$ ,  $S$ ,  $V$ )
  - $G() \rightarrow k$  - generator returns key
  - $S(k, x) \rightarrow t$  - signing returns a tag  $t$  for input  $x$
  - $V(k, x, t) \rightarrow \text{accept|reject}$  - checks validity of tag  $t$  for given input  $x$
- Correctness property
  - $V(k, x, S(k, x)) = \text{accept}$  should always be true
- Security property
  - $V(k, x, t) = \text{accept}$  should almost never be true when  $x$  and  $t$  are chosen by the attacker

**Client**

**Server**

**Client**

$G() \rightarrow k$

**Server**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**Server**

**G() → k**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**G() → k**

**Login info ok?**

**Server**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**G() → k**

**Login info ok?**

**OK!**

**Server**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**G() → k**

**Login info ok?**

**OK!**

**S(k, 'alice') → t**

**Server**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**G() → k**

**Login info ok?**

**OK!**

**S(k, 'alice') → t**

**Server**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**HTTP/1.1 200 OK**  
**Set-Cookie: username=alice;**  
**Set-Cookie: tag=t;**



**GET / HTTP/1.1**  
**Cookie: username=alice; tag=t**



**G() → k**

**Login info ok?**

**OK!**

**S(k, 'alice') → t**

**Server**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**HTTP/1.1 200 OK**  
**Set-Cookie: username=alice;**  
**Set-Cookie: tag=t;**



**GET / HTTP/1.1**  
**Cookie: username=alice; tag=t**



**G() → k**

**Login info ok?**

**OK!**

**S(k, 'alice') → t**

**Server**

**V(k, 'alice', t) → ok?**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**HTTP/1.1 200 OK**  
**Set-Cookie: username=alice;**  
**Set-Cookie: tag=t;**



**GET / HTTP/1.1**  
**Cookie: username=alice; tag=t**



**G() → k**

**Login info ok?**

**OK!**

**S(k, 'alice') → t**

**Server**

**V(k, 'alice', t) → ok?**

**OK!**

**Client**

**POST /login HTTP/1.1**  
**username=alice&password=password**



**HTTP/1.1 200 OK**  
**Set-Cookie: username=alice;**  
**Set-Cookie: tag=t;**



**GET / HTTP/1.1**  
**Cookie: username=alice; tag=t**



**HTTP/1.1 200 OK**  
**Private webpage for Alice!**



**G() → k**

**Login info ok?**

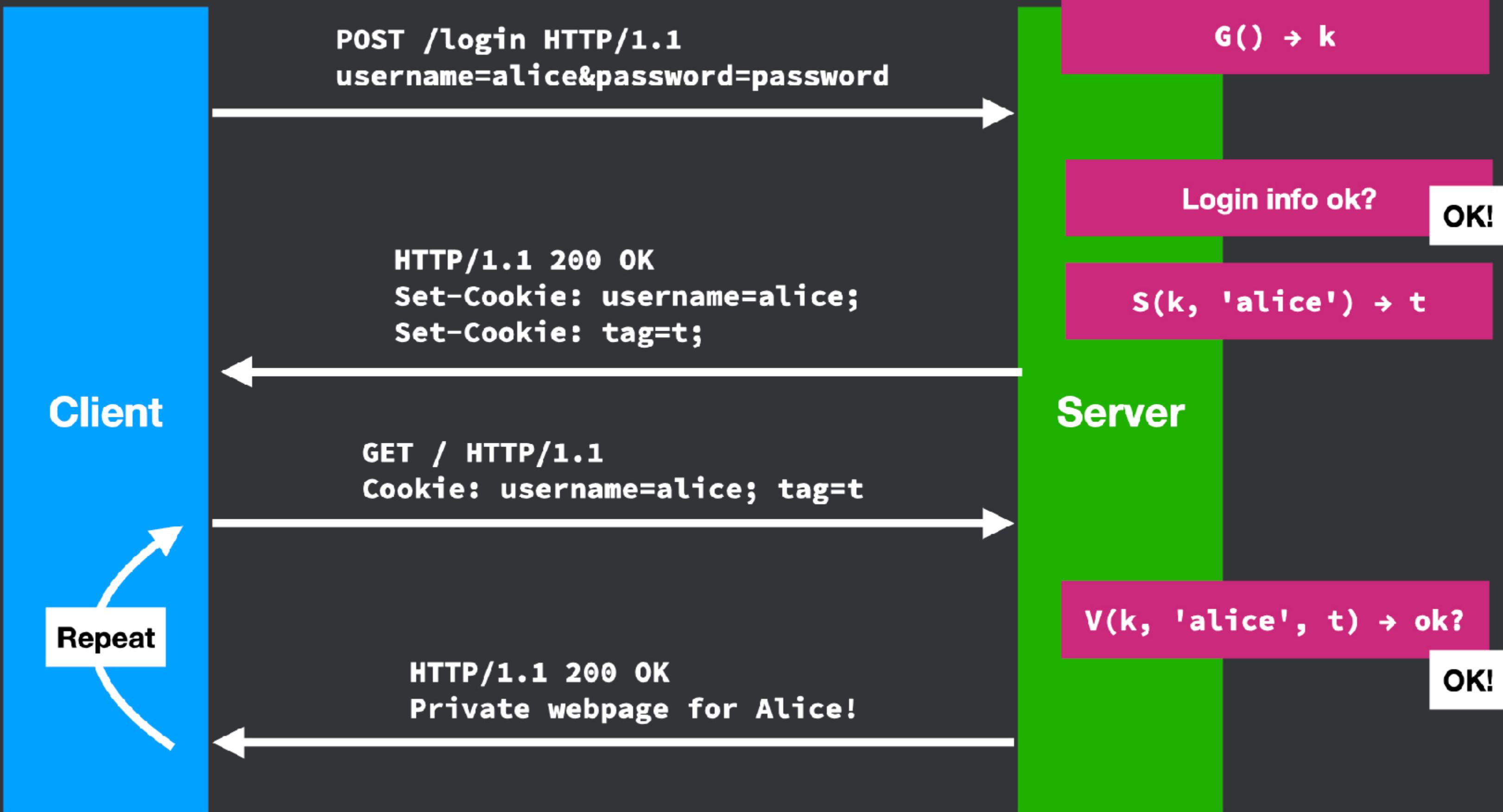
**OK!**

**S(k, 'alice') → t**

**Server**

**V(k, 'alice', t) → ok?**

**OK!**



# Sessions: Desired properties

- Browser remembers user (so user doesn't need to repeatedly log in)
- User cannot modify session cookie to login as another user
- Session cookies are not valid forever
- Sessions can be deleted on the server-side
- Sessions should expire after some time, e.g. 30 days

# History of cookies

- Implemented in 1994 in Netscape and described in 4-page draft
- No spec for 17 years
  - Attempt made in 1997, but made incompatible changes
  - Another attempt in 2000 ("Cookie2"), same problem
  - Around 2011, another effort succeeded (RFC 6265)
- Ad-hoc design has led to *interesting* issues

1997-The specification produced by the group specifies that third-party cookies were either not allowed at all, or at least not enabled by default  
2011 - European law requires that all websites targeting European Union member states gain "informed consent" from users before storing non-essential cookies on their device.