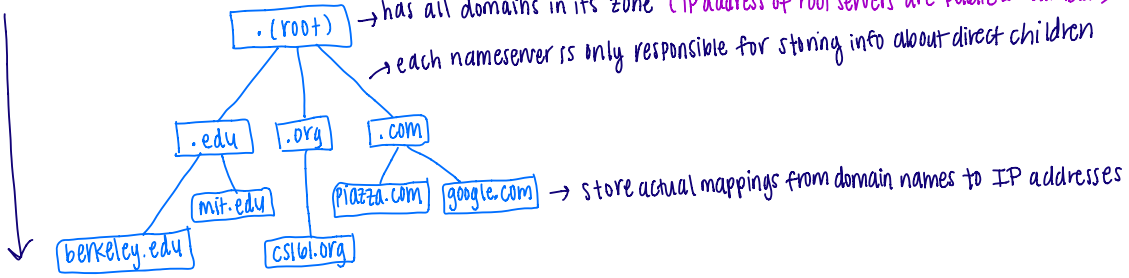
↳ shared PS is result of key exchange: $g^{ab} \bmod p$
- advantage: forward secrecy (even if attacker gets server's private key cannot figure out a or b)

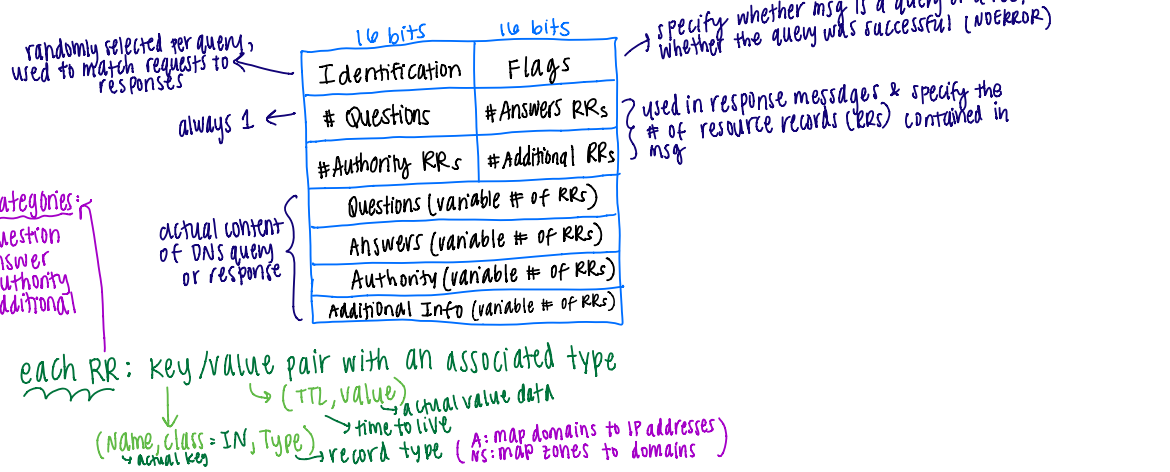
- use PS, R_B, R_S to generate set of 4 shared symmetric keys
  - ↳ encryption key C_B } for client
  - ↳ integrity key I_B }
  - ↳ encryption key C_S } for server
  - ↳ integrity key I_S }
- client & server exchange & verify MACs over all messages sent so far (using I_B & I_S) to ensure no one has tampered w/ msgs sent in handshake so far
- now msgs are encrypted & MACed w/ C & I of sender before being sent
  - ↳ msgs have full confidentiality & integrity = TLS has achieved end-to-end security between client & server
- In practice, msgs sent over TLS usually include some counter/timestep so that an attacker cannot record a TLS msg & send it again within the same connection
- biggest advantage & problem of TLS is certificate authorities
  - ↳ good: delegate trust
  - ↳ bad: all CAs need to be trusted to speak for every site

- DNS (Domain Name System)

↳ protocol to translate between human-readable names and IP addresses
- uses a collection of many name servers (servers dedicated to replying to DNS requests)
  - ↳ each one is responsible for specific zone of domains
  - ↳ they also have their own domain names & IP addresses
- name server hierarchy
  - when you query a name server, instead of always returning the IP address of the domain you queried, the name server can also direct you to another name



- computer can manually perform DNS lookups but in practice, local computer usually delegates the task of DNS lookups to a DNS Recursive Resolver (provided by ISP)
  - ↳ sends queries, processes responses, & maintains internal cache of records
  - ↳ when performing a lookup, DNS stub resolver on computer sends a query to recursive resolver, lets it do all the work, and receives the response
- DNS message format
  - → designed to be very lightweight & fast (uses UDP)
  - → packet header



- DNS security
  - DNS is insecure against a malicious name server
    ↳ to prevent this, resolvers implement bailiwick checking → a nameserver is only allowed to provide records in its zone
  - against on-path attacker, DNS is completely insecure (everything is sent over plaintext)
    ↳ race condition
  - against off-path attacker, Kaminsky attack
    ↳ relies on querying for nonexistant domains
      - legitimate response to nonexistant domain is NXDOMAIN status with no other records (nothing is cached)
        ↳ attacker can include malicious additional records in fake response for nonexistant domain (will be cached)
          ↳ race condition (fake vs. real response)
      - to force victim to make thousands of DNS queries for nonexistant domains, trick victim into visiting a website that tries to load lots of nonexistant domains (img src)
  - Kaminsky attack allows on-path attackers to race until fake response arrives first & off path attackers to race until they successfully guess the ID field
    ↳ UDP source port randomization makes this attack more difficult

- DNSSEC
  ↳ an extension to regular DNS that provides integrity & authentication on all DNS msgs sent
    - every name server generates a public/private key pair and signs every record it sends with its private key
      ↳ when server receives a DNS request, it sends records, signature on records, and public key to resolver; resolver uses public key to verify signature
  - to defend against malicious name servers use trust anchor (instead of inherently trusting all name servers)
    ↳ root server is trust anchor
      ↳ now can delegate trust from trust anchor to somebody else (by signing their public key)
  - new DNSSEC record types (added to DNS design)
    - DNSKEY type record to encode public key
    - RRSIG type record is signature on set of multiple other records in msg (all of same type)
    - DS type record is hash of signer's name & a child's public key (allows trusted server to sign public key of child)
  - each DNSSEC name server generates 2 private/public key pairs
    ↳ KSK is only used to sign zone signing key
    ↳ ZSK is used to sign everything else
    "parent" "child" endorses by signing it
  - for DNS & DNSSEC, start by first querying for root (hardcoded IP address)
  - DNS is meant to be fast, DNSSEC is slow
    ↳ DNSSEC signs records offline & saves signatures w/ records in server
      ↳ to handle nonexistant domains (w/out triggering DoS), name servers precompute signatures on ranges of nonexistant domains (use hashes instead of actual names for better security)

Web security

- every resource on the web is identified by a URL
  - ↳ http://www.example.com/index.html
    - protocol → tells browser how to retrieve the resource (HTTP or HTTPS)
      - ↳ secure version of HTTP using TLS
    - domain name → tells browser which web server to contact to retrieve the resource
    - path → tells browser which page on web server to request
    - (sometimes include port # to distinguish between diff applications running on same web server)

- HTTP (Hypertext Transfer Protocol)
  - language clients use to communicate with servers in order to fetch resources & issue other requests (can only return plain text files)
  - Request Response Model
    - ↳ clients (such as browsers) must actively start a connection to the server and issue a request which the server then responds to
    - Structure of a Request
      - Line 1: method of request (GET / POST), path of request (/), protocol version (HTTP/1.1)
      - → every other line is a request header for GET; POST can include a body as well
    - GET vs. POST
      - ↳ can still pass query parameters via the URL itself
      - GET: intended for "getting" information from the server and generally do not change anything on the server's end
      - POST: intended for sending information to the server that somehow modifies its internal state (ie. adding a comment in a forum or changing your password)

- Elements of a webpage (3 different languages)
  - HTML: lets us create structured documents w/ paragraphs, links, fillable forms, and embedded images, etc.
    - frames pose a security risk since the outer page is now including an inner page that may be from diff (malicious) source
      - ↳ to protect against this, browsers enforce frame isolation (outer page cannot change contents of inner page & inner page cannot change contents of outer page)
  - CSS: lets us modify the appearance of an HTML page by using diff fonts, colors, spacing, etc.
  - Javascript: lets us have dynamic features such as interactive buttons
    - ↳ runs in browser and can arbitrarily modify any HTML or CSS on a webpage
    - modern web browsers typically run Javascript in a sandbox so that any code from a webpage cannot access sensitive data on your computer

- Same origin policy
  - browsing multiple webpages poses a security risk (cuz one may be malicious)
    - ↳ defend against this by enforcing same origin policy (isolates every webpage in your browser except for when two webpages have the same origin)
  - origins (determined by protocol, domain, and port)
    - ↳ to check if two webpages have the same origin, the same-origin policy performs string matching on protocol, domain, and port
      - ↳ same origin if all 3 match (if port is not specified, defaults to 80) (default ports are the one exception to string matching)
    - Exceptions (to origin of webpage is defined by its URL rule)
      - Javascript runs with the origin of the page that loads it
        - ie.) <script src="http://abc.com></script> on http://cs.org
          - ↳ script has origin of http://cs.org
      - Images have origin of page that loads it
      - Frames have origin of URL where the frame is retrieved from, not the origin of the website that loads it
        - ie.) <iframe src="http://google.com"></iframe> on http://cs1601.org
          - ↳ frame has origin of http://google.com

- SQL Injection
  - if web server uses user input as part of the code it runs and the input is not properly checked, an attacker could create a special input that causes unintended code to run on the server
  - trick to force SQL query to always return something
    - ↳ logic that always returns true (ex. 'OR 1=1')

- Defenses
  - escape inputs
    ↳ tell SQL to treat every character that could be used in an attack as part of the string (not actual SQL syntax)
      ↳ but building a good escaper can be tricky
  - parameterized SQL / prepared statements
    ↳ compiles query first and then plugs in user input after the query has already been interpreted by SQL parser
      ↳ no way for attacker input to be interpreted as SQL code
        ↳ best defense against SQL injection

- XSS (cross-site scripting)
  - attacker injects malicious javascript onto a webpage
    ↳ when victim user loads webpage, user's browser will run malicious javascript
    ↳ subvert same-origin policy
  - stored XSS: attacker finds way to persistently store malicious javascript on web server; when victim loads the webpage, web server will load malicious JS & display it
    ↳ make fb post w/ content: <script> alert("LOL") </script>
  - reflected XSS: attacker finds a vulernable webpage where the server receives user input in an HTTP request and displays user input in response
    ↳ google 'search?<script>...</script>' in URL

  - Defenses
    - sanitize input
      ↳ check for malicious input that might cause javascript to run like '<script>'
        ↳ hard to write good detector that catches all attacks
      ↳ replace potentially dangerous characters with their HTML encoding (< → &lt)
    - CSP (content security policy)
      ↳ specifies a list of allowed domains where scripts can be loaded from
      ↳ defined by a web server and enforced by a browser
        ↳ in HTTP response, the server attaches a CSP header and the browser checks any scripts against the header

- Cookies & Session Management
  - HTTP is a stateless protocol, each request and response is ⊥ from all other requests and responses
  - browsers and servers store HTTP cookies to maintain some form of state
  - cookie → piece of data stored in browser (encode state that should persist across multiple requests and responses)
    ↳ when making a change across states, server sends response with set-cookie header (tells browser to store a new cookie)
      ↳ in future requests, browser will automatically attach relevant cookies to a request & send it to web server so server can customize response
    - every cookie is a name, value pair (ie. darkmode, True)
    - only some cookies in each request (browser uses cookie attributes to determine which to send)
      - Domain & Path attributes tell browser which URLs to send cookie to
        - browser sends a cookie to a given URL if the cookie's domain attribute is a domain suffix of the URL domain & the cookie's path attribute is a prefix of the URL path
          ↳ URL domain should end in cookie domain & URL path begins w/ cookie path
        - when a server sets a cookie, the cookie's domain must be a URL suffix of the server's URL
          ↳ for the cookie to be set, the server's URL must end in the cookie's domain attribute (else, browser will reject cookie)
          ↳ Exception: cookies cannot have domains set to a top-level domain, such as .edu or .com since these are too broad and pose a security risk

          (cookie policy allows server to set Path attribute without any restrictions)

- Secure attribute tells browser to only send cookie over secure HTTPS connection
- HttpOnly attribute prevents JS from accessing & modifying the cookie
- expires attribute tells browser when to stop remembering the cookie

- Session Management
  - ↳ cookies are often used to keep users logged in to a website over many requests and responses
    - ↳ when user sends a login request w/ valid username & password, server will generate new session token and send it to user as a cookie
      - ↳ in future requests, the browser will attach session token cookie & send it to the server (server maintains a mapping of session tokens to users so when it receives a request w/ 🍪, it can look up corresponding user & customize response accordingly)
  - secure session tokens should be random and unpredictable (HttpOnly & secure protect them from XSS vulernabilities and network attackers)
  - session token v. cookie
    - ↳ session token: values that browser sends to the server to associate the request with a logged-in user
    - ↳ cookie: how browser stores and sends session tokens to the server (can also save other state → like username)
  - TLDR: session tokens are a special type of cookie that keeps users logged in over many requests & responses

- CSRF (cross-site Request Forgery)
  - ↳ attacker forces victim to make an unintended request (victim's browser will automatically attach the session token cookie to the unintended request & server will accept request as coming from the victim)
  - most common over HTTP POST requests (embedded in HTML in malicious link)
    - ↳ common example that generates HTTP POST request is HTML forms (<form name=>...</form>)
  - Defenses (use both together ⇒ Defense in Depth)
    - CSRF token (included on webpages)
      - ↳ when legitimate user loads a webpage from the server with a form, the server will randomly generate a CSRF token & include it as an extra field in the form
        - ↳ when user submits form, the form will include CSRF token and the server will check that the CSRF token is valid (if its missing or invalid, server rejects request)
      - need to generate new CSRF token everytime user requests a form
        - ↳ should be random & unpredictable
        - ↳ server needs to maintain mapping of CSRF tokens to session tokens
    - Referer validation
      - HTTP request includes Referer header which indicates which URL request was made from
      - server can check Referer header on each request & reject any requests that have untrusted or suspicious Referer headers
      - good defense if included on every request (but Referer header could be blank)