Lecture 25: Detection, Secure Channels

https://cs161.org

Announcements

- Please turn on video so I can see you
- Campus has just announced default P/NP (with option for letter grade)
- Work hard on Project 2 — note, no staff support over spring break

Detection

Detection Accuracy

- Two types of detector errors:
  - False positive (FP): alerting about a problem when in fact there was no problem
  - False negative (FN): failing to alert about a problem when in fact there was a problem
- Detector accuracy is often assessed in terms of rates at which these occur:
  - Define I to be the event of an instance of intrusive behavior occurring (something we want to detect)
  - Define A to be the event of detector generating alarm
- Define:
  - False positive rate = $P[A|\neg I]$
  - False negative rate = $P[\neg A| I]$

Perfect Detection

- Is it possible to build a detector for our example with a false negative rate of 0%?
- Algorithm to detect bad URLs with 0% FN rate:

```
void my_detector_that_never_misses(char *URL)
{
    printf("yep, it's an attack!\n");
}
```

  - In fact, it works for detecting any bad activity with no false negatives! Woo-hoo!
- Wow, so what about a detector for bad URLs that has no false positives?
  - `printf("nope, not an attack\n");`

Detection Tradeoffs

- The art of a good detector is achieving an effective balance between FPs and FNs
- Suppose our detector has an FP rate of 0.1% and an FN rate of 2%. Is it good enough? Which is better, a very low FP rate or a very low FN rate?
  - Depends on the cost of each type of error …
    - E.g., FP might lead to paging a duty officer and consuming hour of their time; FN might lead to $10K cleaning up compromised system that was missed
  - … but also critically depends on the rate at which actual attacks occur in your environment

Base Rate Fallacy

- Suppose our detector has a FP rate of 0.1% (!) and a FN rate of 2% (not bad!)
- Scenario #1: our server receives 1,000 URLs/day, and 5 of them are attacks
  - Expected # FPs each day = 0.1% * 995 ≈ 1
  - Expected # FNs each day = 2% * 5 = 0.1 (< 1/week)
  - Pretty good!
- Scenario #2: our server receives 10,000,000 URLs/day, and 5 of them are attacks
  - Expected # FPs each day ≈ 10,000 :-(
- Nothing changed about the detector; only our environment changed
  - Accurate detection very challenging when base rate of activity we want to detect is quite low
- This is why new recommendations have fewer mammograms and PSA tests…

Styles of Detection: Signature-Based

- Idea: look for activity that matches the structure of a known attack
- Example (from the freeware Snort NIDS):

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 139
flow:to_server,established
content:"|eb2f 5feb 4a5e 89fb 893e 89f2|"
msg:"EXPLOIT x86 linux samba overflow"
reference:bugtraq,1816
reference:cve,CVE-1999-0811
classtype:attempted-admin
```

- Can be at different semantic layers
  e.g.: IP/TCP header fields; packet payload; URLs

Signature-Based Detection

- E.g. for FooCorp, search for "`../../`" or "**`/etc/passwd`**"
- What's nice about this approach?
  - Conceptually simple
  - Takes care of known attacks (of which there are zillions)
  - Easy to share signatures, build up libraries
- What's problematic about this approach?
  - Blind to novel attacks
  - Might even miss variants of known attacks ("`..///.//../`")
    - Of which there are zillions
  - Simpler versions look at low-level syntax, not semantics
    - Can lead to weak power (either misses variants, or generates lots of false positives)

Vulnerability Signatures

- Idea: don't match on known attacks, match on known problems
- Example (also from Snort):

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS 80
uricontent: ".ida?"; nocase; dsize: > 239; flags:A+
msg:"Web-IIS ISAPI .ida attempt"
reference:bugtraq,1816
reference:cve,CAN-2000-0071
classtype:attempted-admin
```

- That is, match URIs that invoke `*.ida?*`, have more than 239 bytes of payload, and have ACK set (maybe others too)
- This example detects attempts to exploit a particular buffer overflow in IIS web servers
  - Used by the "Code Red" worm
  - (Note, signature is not quite complete: also worked for `*.idb?*`)

Styles of Detection: Anomaly-Based

- Idea: attacks look peculiar.
- High-level approach: develop a model of normal behavior (say based on analyzing historical logs). Flag activity that deviates from it.
- FooCorp example: maybe look at distribution of characters in URL parameters, learn that some are rare and/or don't occur repeatedly
  - If we happen to learn that '.'s have this property, then could detect the attack even without knowing it exists
- Big benefit: potential detection of a wide range of attacks, including novel ones

Anomaly Detection Problems

- Can fail to detect known attacks
- Can fail to detect novel attacks, if don't happen to look peculiar along measured dimension
- What happens if the historical data you train on includes attacks?
- Base Rate Fallacy particularly acute: if prevalence of attacks is low, then you're more often going to see benign outliers
  - High FP rate
  - OR: require such a stringent deviation from "normal" that most attacks are missed (high FN rate)
- Proves great subject for academic papers but not generally used

Specification-Based Detection

- Idea: don't learn what's normal; specify what's allowed
- FooCorp example: decide that all URL parameters sent to foocorp.com servers must have at most one '/' in them
  - Flag any arriving param with > 1 slash as an attack
- What's nice about this approach?
  - Can detect novel attacks
  - Can have low false positives
    - If FooCorp audits its web pages to make sure they comply
- What's problematic about this approach?
  - Expensive: lots of labor to derive specifications
    - And keep them up to date as things change ("churn")

Styles of Detection: Behavioral

- Idea: don’t look for attacks, look for evidence of compromise
- FooCorp example: inspect all output web traffic for any lines that match a passwd file
- Example for monitoring user shell keystrokes:
  `unset HISTFILE`
- Example for catching code injection: look at sequences of system calls, flag any that prior analysis of a given program shows it can’t generate
  - E.g., observe process executing read(), open(), write(), fork(), exec() …
  - … but there’s no code path in the (original) program that calls those in exactly that order!

Behavioral-Based Detection

- What's nice about this approach?
  - Can detect a wide range of novel attacks
  - Can have low false positives
    - Depending on degree to which behavior is distinctive
    - E.g., for system call profiling: no false positives!
  - Can be cheap to implement
    - E.g., system call profiling can be mechanized
- What's problematic about this approach?
  - Post facto detection: discovers that you definitely have a problem, w/ no opportunity to prevent it
  - Brittle: for some behaviors, attacker can maybe avoid it
    - Easy enough to not type "`unset HISTFILE`"
    - How could they evade system call profiling?
      - Mimicry: adapt injected code to comply w/ allowed call sequences (and can be automated!)

Summary of Evasion Issues

- Evasions arise from uncertainty (or incompleteness) because detector must infer behavior/processing it can't directly observe
  - A general problem any time detection separate from potential target
- One general strategy: impose canonical form ("normalize")
  - E.g., rewrite URLs to expand/remove hex escapes
  - E.g., enforce blog comments to only have certain HTML tags
- Another strategy: analyze all possible interpretations rather than assuming one
  - E.g., analyze raw URL, hex-escaped URL, doubly-escaped URL …
- Another strategy: Flag potential evasions
  - So the presence of an ambiguity is at least noted
- Another strategy: fix the basic observation problem
  - E.g., monitor directly at end systems

Inside a Modern HIDS (“Antivirus”)

- URL/Web access blocking
  - Prevent users from going to known bad locations
- Protocol scanning of network traffic (esp. HTTP)
  - Detect & block known attacks
  - Detect & block known malware communication
- Payload scanning
  - Detect & block known malware
  - (Auto-update of signatures for these)
- Cloud queries regarding reputation
  - Who else has run this executable and with what results?
  - What’s known about the remote host / domain / URL?

Inside a Modern HIDS

- Sandbox execution
  - Run selected executables in constrained/monitored environment
  - Analyze:
    - System calls
    - Changes to files / registry
    - Self-modifying code (polymorphism/metamorphism)
- File scanning
  - Look for malware that installs itself on disk
- Memory scanning
  - Look for malware that never appears on disk
- Runtime analysis
  - Apply heuristics/signatures to execution behavior

Inside a Modern NIDS

- Deployment inside network as well as at border
  - Greater visibility, including tracking of user identity
- Full protocol analysis
  - Including extraction of complex embedded objects
  - In some systems, 100s of known protocols
- Signature analysis (also behavioral)
  - Known attacks, malware communication, blacklisted hosts/domains
  - Known malicious payloads
  - Sequences/patterns of activity
- Shadow execution (e.g., Flash, PDF programs)
- Extensive logging (in support of forensics)
- Auto-update of signatures, blacklists

NIDS vs. HIDS

- NIDS benefits:
  - Can cover a lot of systems with single deployment
    - Much simpler management
  - Easy to “bolt on” / no need to touch end systems
  - Doesn’t consume production resources on end systems
  - Harder for an attacker to subvert / less to trust
- HIDS benefits:
  - Can have direct access to semantics of activity
    - Better positioned to block (prevent) attacks
    - Harder to evade
  - Can protect against non-network threats
  - Visibility into encrypted activity
  - Performance scales much more readily (no chokepoint)
    - No issues with “dropped” packets

Key Concepts for Detection

- Signature-based vs anomaly detection (blacklisting vs whitelisting)
- Evasion attacks
- Evaluation metrics: False positive rate, false negative rate
- Base rate problem

Secure Channels

Applying crypto technology in practice

- Two simple abstractions cover 80% of the use cases for crypto:
  - “Sealed blob”: Data that is encrypted and authenticated under a particular key (“object security”)
  - Secure channel: Communication channel that can’t be eavesdropped on or tampered with (“channel security”)
- TLS – a secure channel

Building Secure End-to-End Channels

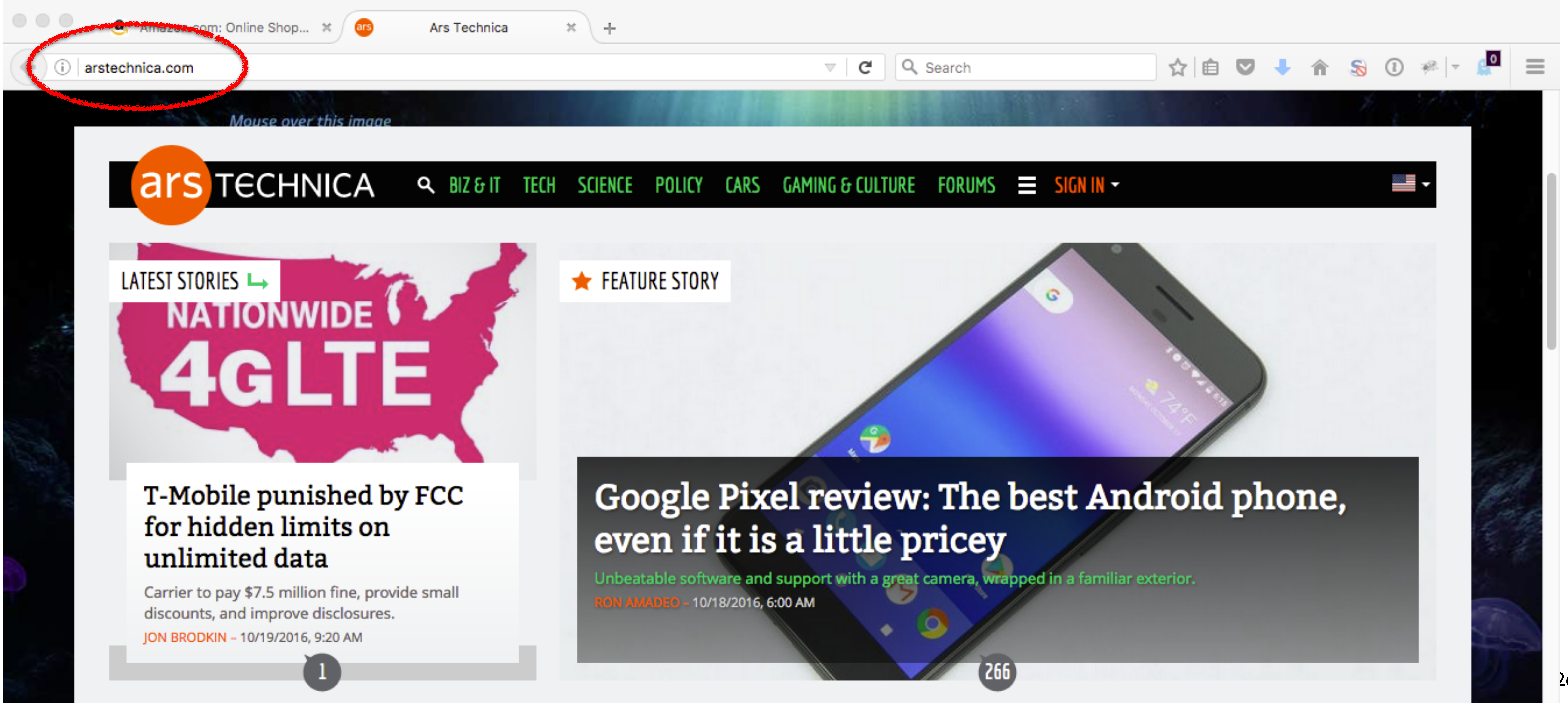
- End-to-end = communication protections achieved all the way from originating client to intended server
  - With no need to trust intermediaries
- Dealing with threats:
  - Eavesdropping: Encryption (including session keys)
  - Manipulation (injection, MITM): Integrity (use of a MAC); replay protection
  - Impersonation: Signatures

(What's missing?
Availability …)

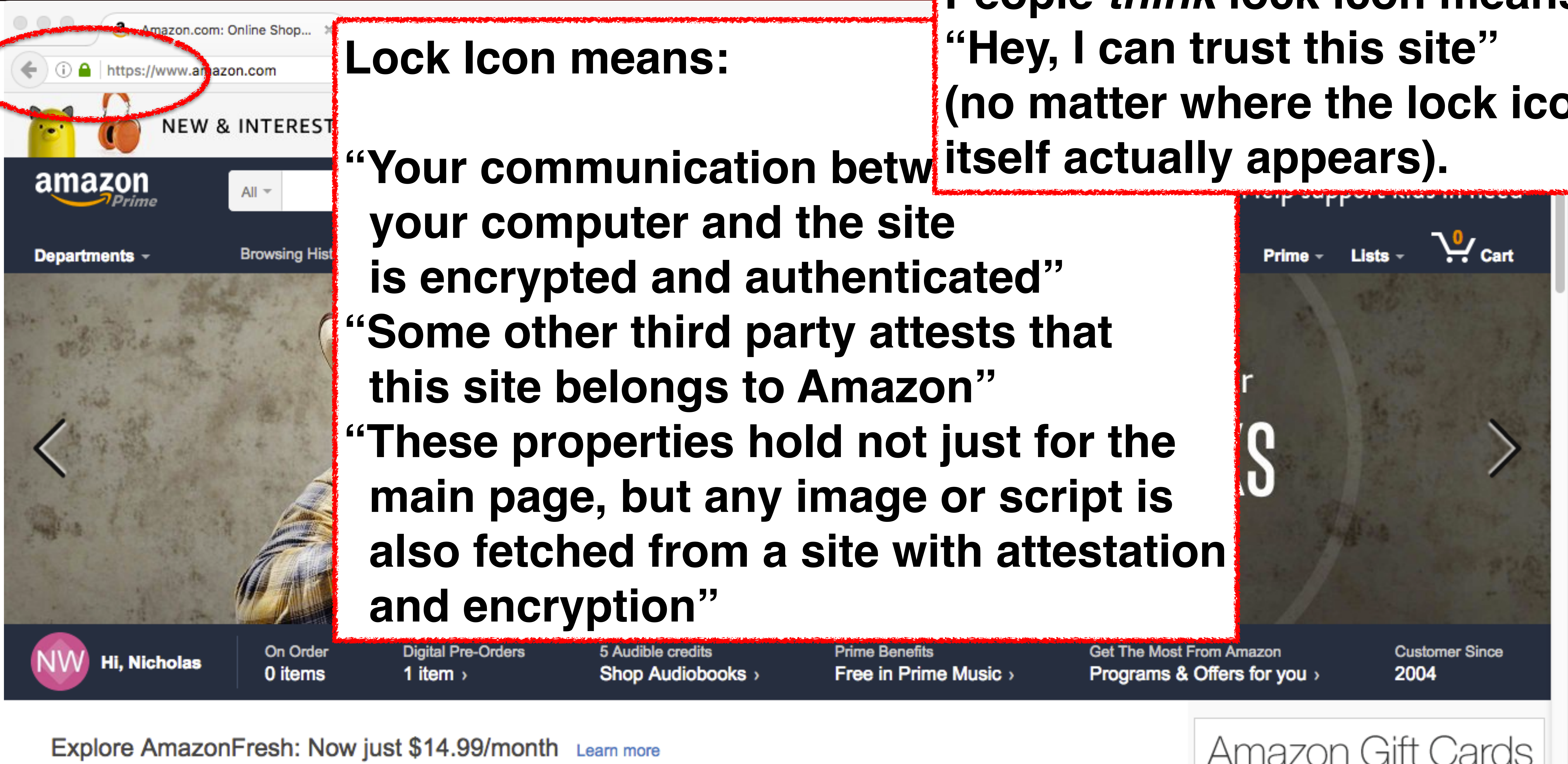
Building A Secure End-to-End Channel: SSL/TLS

- SSL = Secure Sockets Layer (predecessor)
- TLS = Transport Layer Security (standard)
  - Both terms used interchangeably
- Security for any application that uses TCP
  - Secure = encryption/confidentiality + integrity + authentication (of server, but not of client)
- Multiple uses
  - Puts the ‘s’ in “https”
  - Secures mail sent between servers (STARTTLS)
  - Virtual Private Networks

An “Insecure” Web Page



A "Secure" Web Page



Lock Icon means:

"Your communication between your computer and the site is encrypted and authenticated"

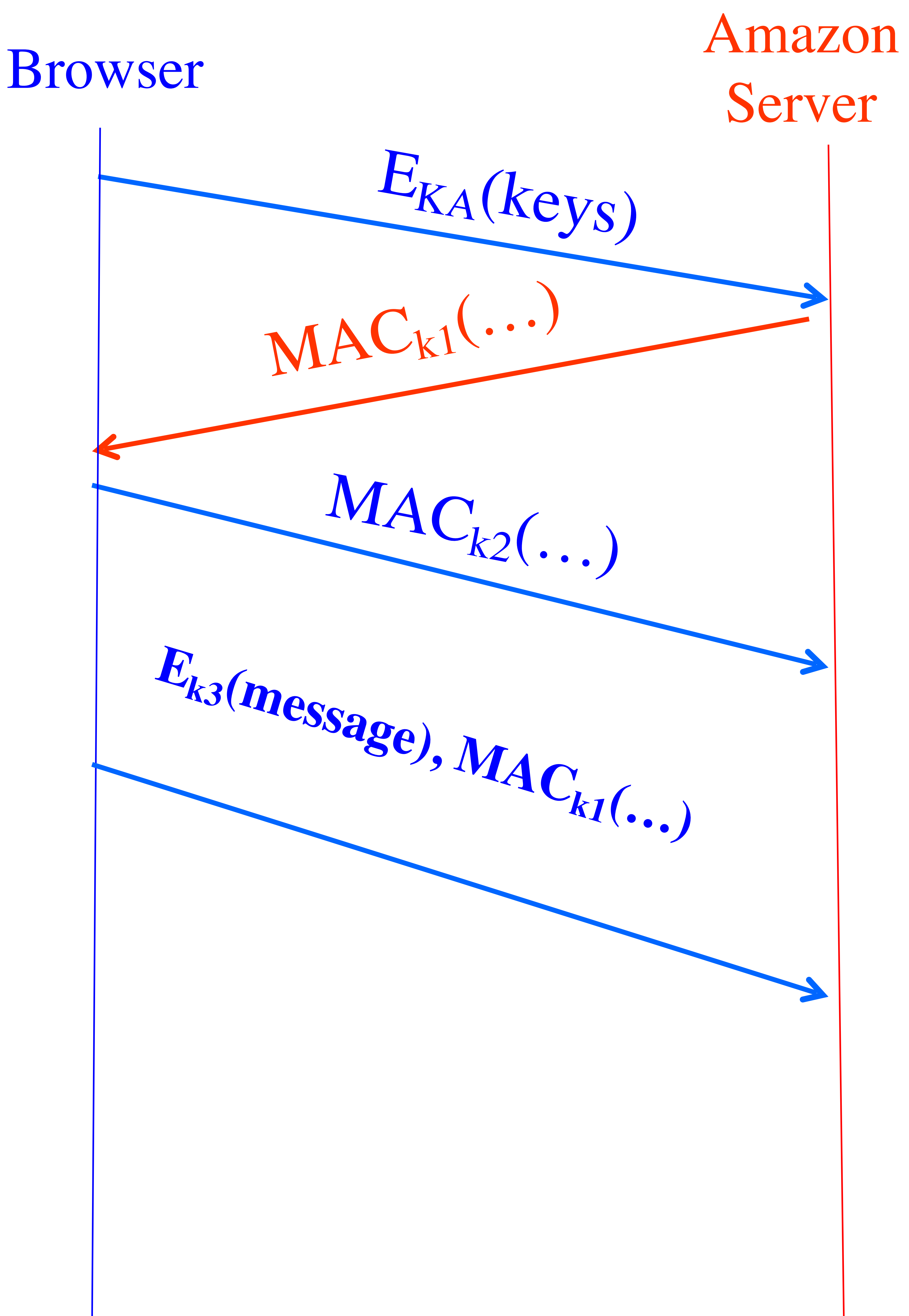
"Some other third party attests that this site belongs to Amazon"

"These properties hold not just for the main page, but any image or script is also fetched from a site with attestation and encryption"

People *think* lock icon means "Hey, I can trust this site" (no matter where the lock icon itself actually appears).

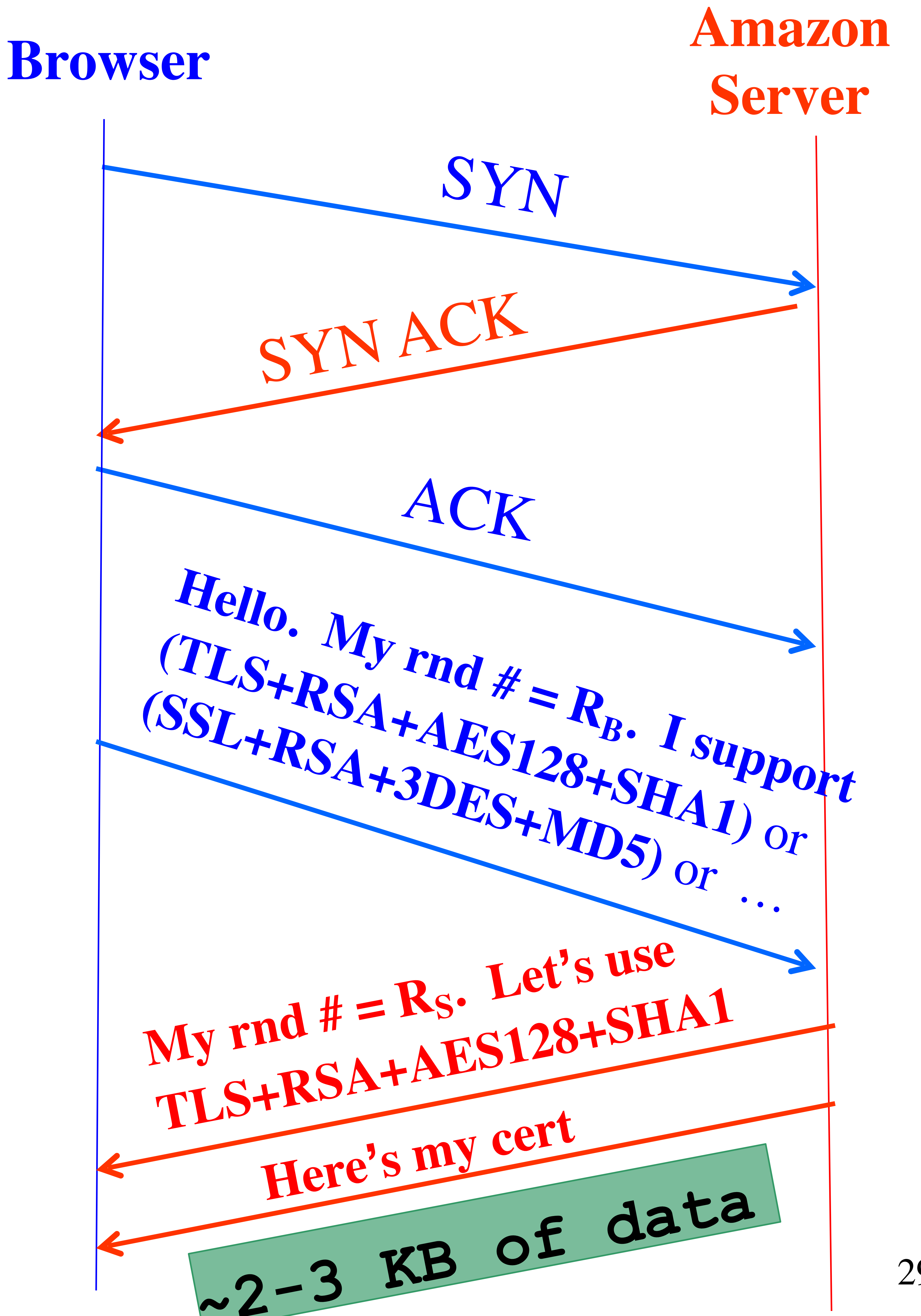
Basic Idea

- Browser (client) picks some symmetric keys for encryption + authentication
- Client sends them to server, encrypted using RSA public-key encryption
- Both sides send MACs
- Now they use these keys to encrypt and authenticate all subsequent messages, using symmetric-key crypto



HTTPS Connection (SSL / TLS)

- Browser (client) connects via TCP to Amazon's HTTPS server
- Client picks 256-bit random number R_B, sends over list of crypto protocols it supports
- Server picks 256-bit random number R_S, selects protocols to use for this session
- Server sends over its certificate
  - (all of this is in the clear)
- Client now ***validates*** cert



HTTPS Connection (SSL / TLS), cont.

- For RSA, browser constructs "Premaster Secret" PS
- Browser sends PS encrypted using Amazon's public RSA key KAmazon
- Using PS, R_B, and R_S, browser & server derive symm. cipher keys (C_B, C_S) & MAC integrity keys (I_B, I_S)
  - One pair to use in each direction
- Browser & server exchange MACs computed over entire dialog so far
- If good MAC, Browser displays 
- All subsequent communication encrypted w/ symmetric cipher (e.g., AES128) cipher keys, MACs
  - Sequence #'s thwart replay attacks