

# CS 161 Final Exam Review

Post Midterm 2 Material

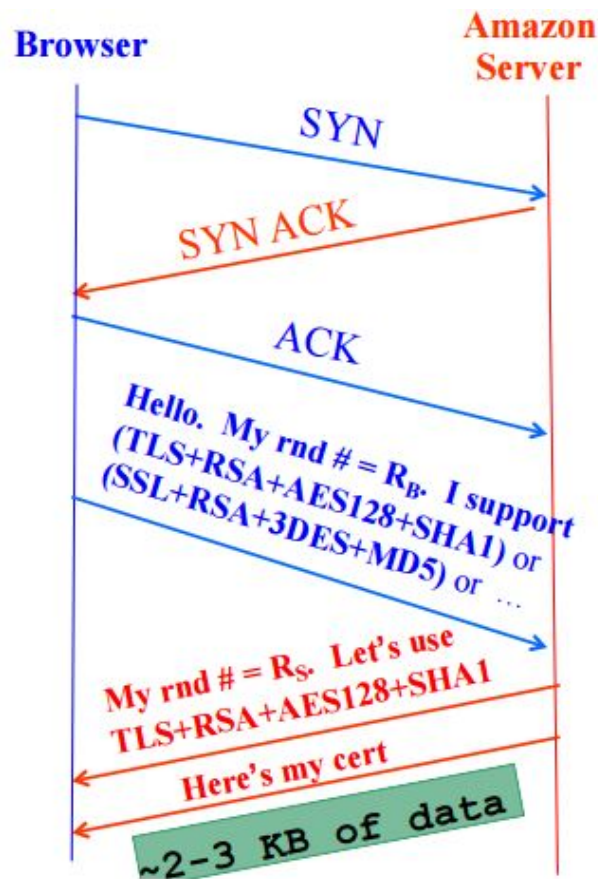
# Agenda

- TLS
- DNSSEC
- Detection
- Malware
- Firewalls
- DoS


TLS

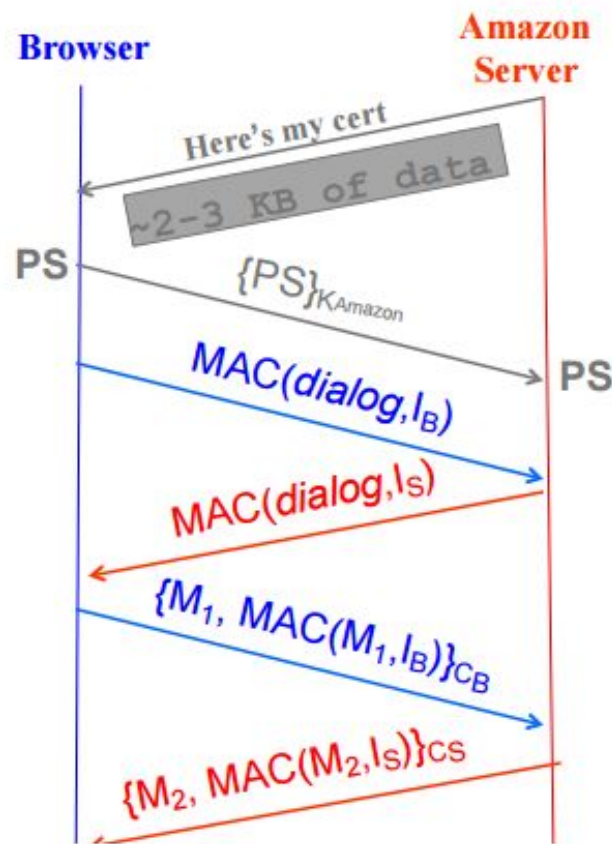
# SSL/TLS

- 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



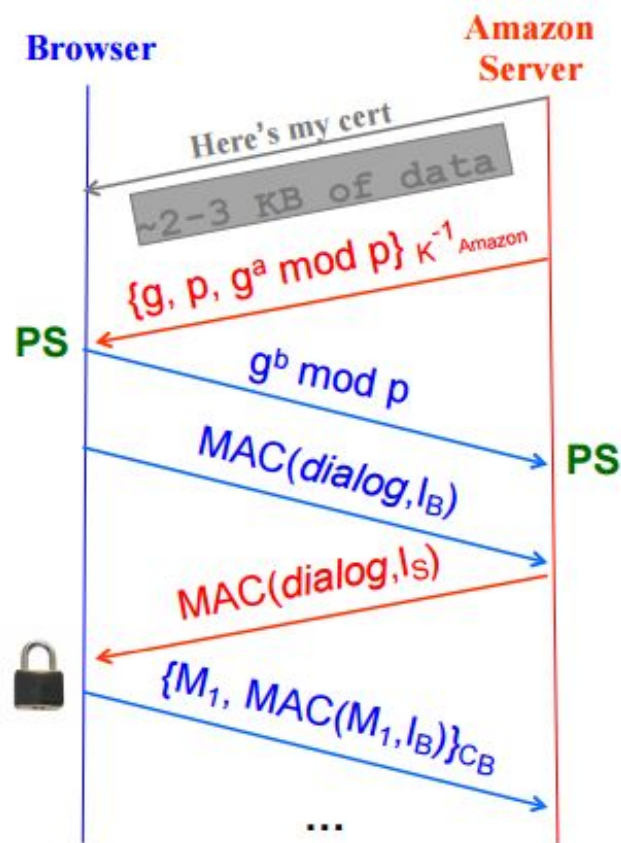
# SSL/TLS (RSA)

- For RSA, browser constructs “Premaster Secret” PS
- Browser sends PS encrypted using Amazon’s public RSA key  $K_{\text{Amazon}}$
- 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



# SSL/TLS (Diffie-Hellman)

- For Diffie-Hellman, server generates random  $a$ , sends public parameters and  $g^a \bmod p$ 
  - Signed with server's private key
- Browser verifies signature
- Browser generates random  $b$ , computes  $PS = g^{ab} \bmod p$ , sends  $g^b \bmod p$  to server
- Server also computes  $PS = g^{ab} \bmod p$
- Remainder is as before: from  $PS$ ,  $R_B$ , and  $R_S$ , browser & server derive symm. cipher keys ( $C_B$ ,  $C_S$ ) and MAC integrity keys ( $I_B$ ,  $I_S$ ), etc...





# SSL/TLS Limitations

- Properly used, SSL / TLS provides powerful end-to-end protections
- So why not use it for *everything*??
- Issues:
  - Cost of public-key crypto
    - Takes non-trivial CPU processing (but today a minor issue)
    - Note: *symmetric* key crypto on modern hardware is non-issue
  - Hassle of buying/maintaining certs (fairly minor)
  - **DoS amplification**
    - Client can force server to undertake public key operations
    - But: requires established TCP connection, and given that, there are often other juicy targets like back-end databases
  - Integrating with other sites that don't use HTTPS
  - **Latency**: extra round trips  $\Rightarrow$  pages take longer to load

## Spring 2014 - Q5

Alice goes to `https://paypal.com/`, logs in by entering her Paypal username and password, adds her credit card number to her account, and makes a payment—all through Paypal's web site. Assume her browser is using the latest and greatest version of TLS and that Paypal uses HTTPS for everything (no HTTP).

Which of the following could an on-path eavesdropper deduce? Circle all that the eavesdropper could deduce.

Assume there are no security bugs or flaws in Alice's browser or Paypal's server, beyond what is implied by the statement of the problem. Assume the attacker can only passively eavesdrop; no active attacks, no man-in-the-middle.



# Spring 2014 - Q5

Which of the following could an on-path eavesdropper deduce?

- A. The approximate size of the HTTPS requests from Alice's browser
- B. The approximate size of the responses from the Paypal server
- C. The approximate number of requests made by Alice's browser
- D. Alice's Paypal username and password
- E. Alice's credit card number
- F. The fact that Alice is visiting Paypal (and not, say, <https://wellsfargo.com/>)
- G. The session cookie for this connection
- H. Any CSRF tokens that the Paypal server uses during this connection
- I. The TCP initial sequence numbers used for this connection

## Spring 2014 - Q5

- A. The approximate size of the HTTPS requests from Alice's browser
- B. The approximate size of the responses from the Paypal server
- C. The approximate number of requests made by Alice's browser
- D. Alice's Paypal username and password
- E. Alice's credit card number
- F. The fact that Alice is visiting Paypal (and not, say, `https://wellsfargo.com/`)
- G. The session cookie for this connection
- H. Any CSRF tokens that the Paypal server uses during this connection
- I. The TCP initial sequence numbers used for this connection

## Spring 2016 - Q8

- (a) Suppose Alice downloads a buggy version of the Chrome browser that implements TLS incorrectly. Instead of picking  $R_B$  randomly, it increments a counter and sends it instead. Which of the following attacks are possible? Circle all that apply.
1. A man-in-the-middle can compromise Alice's confidentiality (e.g., learn the data she sends over TLS).
  2. If Alice visits a HTTPS URL, a man-in-the-middle can successfully replay that HTTP request to the server a second time.
  3. If Alice visits the same HTTPS URL twice, when Alice visits that URL the second time a man-in-the-middle can successfully replay the HTML page that was returned by the server on Alice's first visit.
  4. A man-in-the-middle can learn the symmetric MAC keys that protect data sent over TLS connections initiated by Alice's browser.
  5. None of the above

## Spring 2016 - Q8

- (a) Suppose Alice downloads a buggy version of the Chrome browser that implements TLS incorrectly. Instead of picking  $R_B$  randomly, it increments a counter and sends it instead. Which of the following attacks are possible? Circle all that apply.

**Solution:** None will be possible. Even if  $R_B$  were to repeat, Alice would choose a different pre-master secret each time and thus they'll end up with different symmetric keys for each different connection, so it's not possible for a MITM to replay an HTML page returned on a prior connection.

## Spring 2016 - Q8

(b) Alice downloads a patch for her buggy version of Chrome, which fixes the previous problem but introduces another bug. Now, her client generates the pre-master secret by incrementing a counter. Which of the following attacks are possible? Circle all that apply.

1. A man-in-the-middle can compromise Alice's confidentiality (e.g., learn the data she sends over TLS).
2. If Alice visits a HTTPS URL, a man-in-the-middle can successfully replay that HTTP request to the server a second time.
3. If Alice visits the same HTTPS URL twice, when Alice visits that URL the second time a man-in-the-middle can successfully replay the HTML page that was returned by the server on Alice's first visit.
4. A man-in-the-middle can learn the symmetric MAC keys that protect data sent over TLS connections initiated by Alice's browser.
5. None of the above

# Spring 2016 - Q8

(b) Alice downloads a patch for her buggy version of Chrome, which fixes the previous problem but introduces another bug. Now, her client generates the pre-master secret by incrementing a counter. Which of the following attacks are possible? Circle all that apply.

1. A man-in-the-middle can compromise Alice's confidentiality (e.g., learn the data she sends over TLS).
2. If Alice visits a HTTPS URL, a man-in-the-middle can successfully replay that HTTP request to the server a second time.
3. If Alice visits the same HTTPS URL twice, when Alice visits that URL the second time a man-in-the-middle can successfully replay the HTML page that was returned by the server on Alice's first visit.
4. A man-in-the-middle can learn the symmetric MAC keys that protect data sent over TLS connections initiated by Alice's browser.
5. None of the above



## Spring 2016 - Q8

**Solution:** The MITM can recover the pre-master secret by brute-forcing it, infer the symmetric keys, and then all security is lost. For example, an attacker can decrypt all the traffic Alice sent over TLS, and then inject it into a new TLS session from Alice (encrypted and MAC'ed with the proper keys for that connection, which can also be recovered in the same way).

Replay means that the recipient receives the message a second time. (Successfully means that the recipient doesn't reject/detect the message when it is replayed, i.e., that the second copy is accepted as if it were intended to be sent by Alice.)

## Spring 2016 - Q8

(c) Alice downloads an updated version of Chrome that is finally correct. With her fixed browser, she visits `https://lazycodrz.com/`. That server's TLS implementation has a bug: instead of picking  $R_S$  randomly, it always sends all zeros. Which of the following attacks are possible? Circle all that apply.

1. A man-in-the-middle can compromise Alice's confidentiality (e.g., learn the data she sends over TLS).
2. If Alice visits a HTTPS URL, a man-in-the-middle can successfully replay that HTTP request to the server a second time.
3. If Alice visits the same HTTPS URL twice, when Alice visits that URL the second time a man-in-the-middle can successfully replay the HTML page that was returned by the server on Alice's first visit.
4. A man-in-the-middle can learn the symmetric MAC keys that protect data sent over TLS connections initiated by Alice's browser.
5. None of the above

## Spring 2016 - Q8

(c) Alice downloads an updated version of Chrome that is finally correct. With her fixed browser, she visits `https://lazycodrz.com/`. That server's TLS implementation has a bug: instead of picking  $R_S$  randomly, it always sends all zeros. Which of the following attacks are possible? Circle all that apply.

2. If Alice visits a HTTPS URL, a man-in-the-middle can successfully replay that HTTP request to the server a second time.

**Solution:** Attack 2 is possible because the symmetric keys are derived as a function of the pre-master secret,  $R_B$ , and  $R_S$ . If the MITM replays the entire handshake to the server, the pre-master secret and  $R_B$  will automatically be identical; and due to the bug in the server,  $R_S$  will be identical as well, so the same symmetric keys will be derived, and encrypted and MAC'ed data from the prior connection can be replayed a second time.

DNSSEC

(Switch to PPTX Slide Deck)

Detection  
(Switch to PPTX Slide Deck)

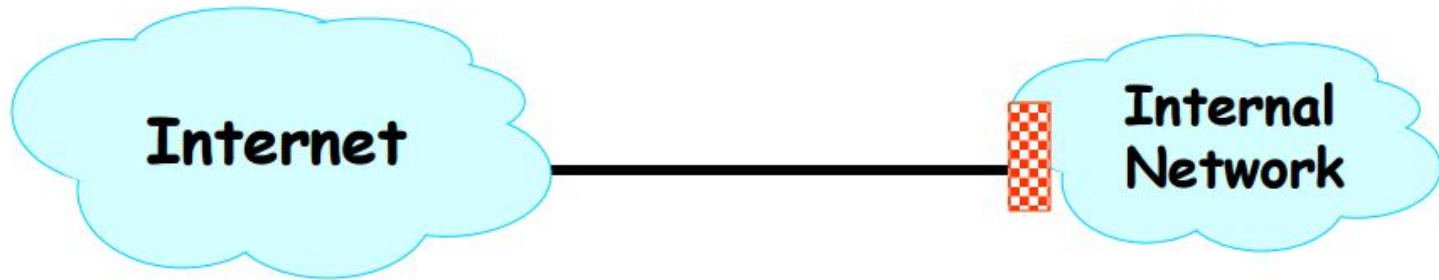
Malware  
(Switch to PPTX Slide Deck)



# Firewalls

# What is a Firewall?

A firewall is a network security device that monitors incoming and outgoing network traffic and decides whether to allow or block specific traffic based on a defined set of security rules.



# Selecting a Security Policy

- Effectiveness of firewall relies on deciding what **policy** it should implement:
  - *Who is allowed to talk to whom, accessing what service?*
- Distinguish between **inbound** & **outbound** connections
  - **Inbound**: attempts by external users to connect to services on internal machines
  - **Outbound**: internal users to external services
  - Why? Because fits with a common **threat model**
- Conceptually simple **access control policy**:
  - Permit inside users to connect to any service
  - External users restricted:
    - **Permit** connections to services meant to be externally visible
    - **Deny** connections to services not meant for external access

# Packet Filters

- Most basic kind of firewall is a *packet filter*
  - Router with list of *access control rules*
  - Router checks each received packet against security rules to decide to *forward* or *drop* it
  - Each rule specifies which packets it applies to based on a packet's header fields (*stateless*)
    - Specify source and destination IP addresses, port numbers, and protocol names, or *wild cards*
    - Each rule specifies the *action* for matching packets: **ALLOW** or **DROP** (aka DENY)  
*<ACTION> <PROTO> <SRC:PORT> -> <DST:PORT>*
  - First listed rule has *precedence*

# Examples of Packet Filter Rules

```
allow tcp 4.5.5.4:1025 -> 3.1.1.2:80
```

- States that the firewall should **permit** any TCP packet that's:
  - from Internet address 4.5.5.4 **and**
  - using a source port of 1025 **and**
  - destined to port 80 of Internet address 3.1.1.2

```
deny tcp 4.5.5.4:* -> 3.1.1.2:80
```

- States that the firewall should **drop** any TCP packet like the above, regardless of source port



# Network Control & Tunneling

- *Tunneling* = embedding one protocol inside another
  - Sender and receiver at each side of the tunnel **both cooperate** (so it's **not useful for initial attacks**)
- Traffic takes on properties of outer protocol
  - Including for **firewall inspection**, which generally can't analyze inner protocol (due to complexity)
- Tunneling has **legitimate** uses
  - E.g., Virtual Private Networks (VPNs)
    - Tunnel server relays remote client's packets
    - Makes remote machine look like it's **local** to its home network
    - Tunnel **encrypts** traffic for privacy & to prevent meddling



DoS

# Availability Attacks

- Denial-of-Service (**DoS**, or “doss”): *keeping someone from using a computing service*
- How broad is this sort of threat?
  - Very: **huge** attack surface
- We do though need to consider our **threat model** ...
  - What might motivate a DoS attack?
- Two basic approaches available to an attacker:
  - Deny service via a **program flaw** (“\*NULL”)
    - E.g., supply an input that crashes a server
    - E.g., fool a system into shutting down
  - Deny service via **resource exhaustion** (“while(1);”)
    - E.g., consume CPU, memory, disk, network

# DoS Defenses (general)

- Defending against **program flaws** requires:
  - Careful coding/testing/review
  - Careful *authentication*
    - Don't obey shut-down orders from imposters
  - Consideration of *behavior of defense mechanisms*
    - E.g. buffer overflow detector that when triggered halts execution to prevent code injection ⇒ **denial-of-service**
- Defending resources from **exhaustion** can be **really** hard. Requires:
  - *Isolation mechanisms*
    - Keep adversary's consumption from affecting others
  - *Reliable identification* of different users
    - Know who the adversary is in the first place!

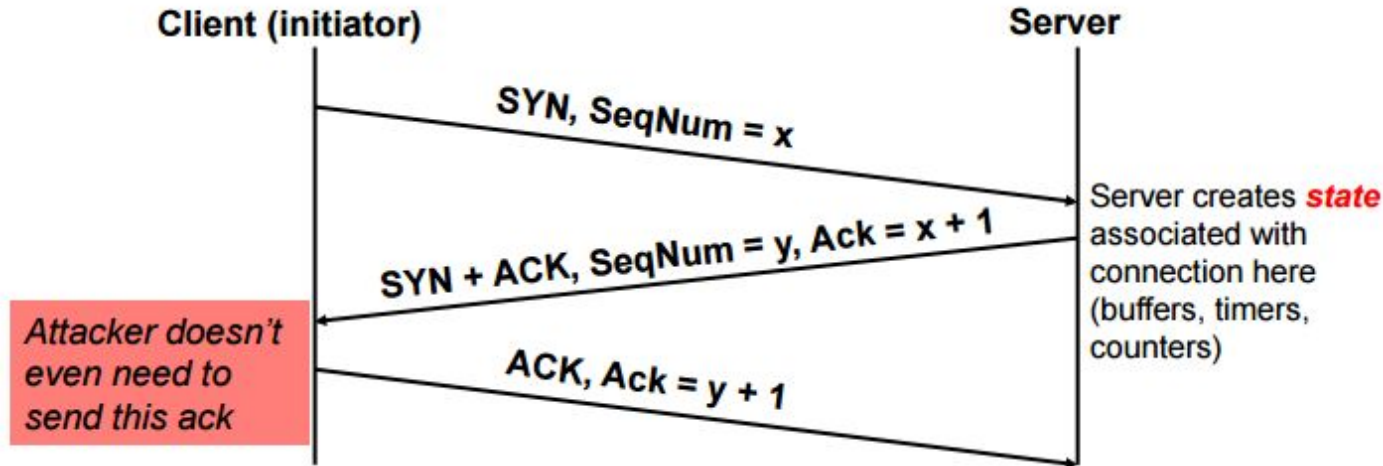
# Amplification Vector: DNS/UDP

- Consider DNS lookups:
  - *Reply is generally much bigger than request*
    - Since it includes a copy of the reply, plus answers etc.
  - ⇒ Attacker spoofs request **seemingly from the target**
    - Small attacker packet yields **large** flooding packet
    - Doesn't increase # of packets, but **total byte volume**
  - Works for other request/response protocols too
- Note #1: attacks involve **blind spoofing**
  - So for network-layer flooding, generally only works for UDP-based protocols (can't establish TCP conn.)
- Note #2: victim doesn't see spoofed source addresses
  - Addresses are those of actual intermediary systems



# Transport-Layer Denial of Service

- Recall TCP's 3-way connection establishment handshake
  - Goal: agree on initial sequence numbers
- So a **single** SYN from an attacker suffices to force the server to **spend some memory**



## Spring 2011 Q5

Consider an ecommerce website that includes the notion of a “shopping cart.” Customers visiting the site put items of interest in their shopping cart. After finishing their browsing and shopping, they click on **Checkout** to pay for the items. At that point, the customer logs into the site to enable the site to retrieve their payment information.

- (a) (8 points) Suppose that the site implements the shopping cart by storing the associated items and prices in files on the server, with one file for each customer. The site identifies customers by their IP addresses.

This design is vulnerable to a DoS attack. Sketch it in a single sentence.



## Spring 2011 Q5

Consider an ecommerce website that includes the notion of a “shopping cart.” Customers visiting the site put items of interest in their shopping cart. After finishing their browsing and shopping, they click on **Checkout** to pay for the items. At that point, the customer logs into the site to enable the site to retrieve their payment information.

- (a) (8 points) Suppose that the site implements the shopping cart by storing the associated items and prices in files on the server, with one file for each customer. The site identifies customers by their IP addresses.

This design is vulnerable to a DoS attack. Sketch it in a single sentence.

**Solution:** An attacker can continually add shopping cart items without end, exhausting the state available on the server for remembering the items.

## Spring 2011 Q5

- (b) (16 points) Suppose that instead the site keeps a list of shopping cart items on the client side. Every time a user clicks on *add-to-cart*, the server sends all of the associated details (item name, price, quantity) in its reply, incorporating them into a hidden HTML form field. Through some Javascript magic, now when the user finally clicks on **Checkout**, all of the previously bought items embedded in the hidden form field are sent to the server. The server then joins them together into a list and presents the user with the corresponding total amount for payment.
1. Is this design vulnerable to the DoS attack you sketched above? Explain why or why not.

# Spring 2011 Q5

(b) (16 points) Suppose that instead the site keeps a list of shopping cart items on the client side. Every time a user clicks on *add-to-cart*, the server sends all of the associated details (item name, price, quantity) in its reply, incorporating them into a hidden HTML form field. Through some Javascript magic, now when the user finally clicks on **Checkout**, all of the previously bought items embedded in the hidden form field are sent to the server. The server then joins them together into a list and presents the user with the corresponding total amount for payment.

1. Is this design vulnerable to the DoS attack you sketched above? Explain why or why not.

**Solution:** This design is not vulnerable to the state-exhaustion attack because now the shopping cart state is all kept at the client. The attacker will exhaust their own memory, rather than the server's.

It's valid to observe that the server might still be vulnerable to a DoS attack depending on what resources it requires for processing the list once the client proceeds to checkout.

Good Luck!