Detecting Attacks

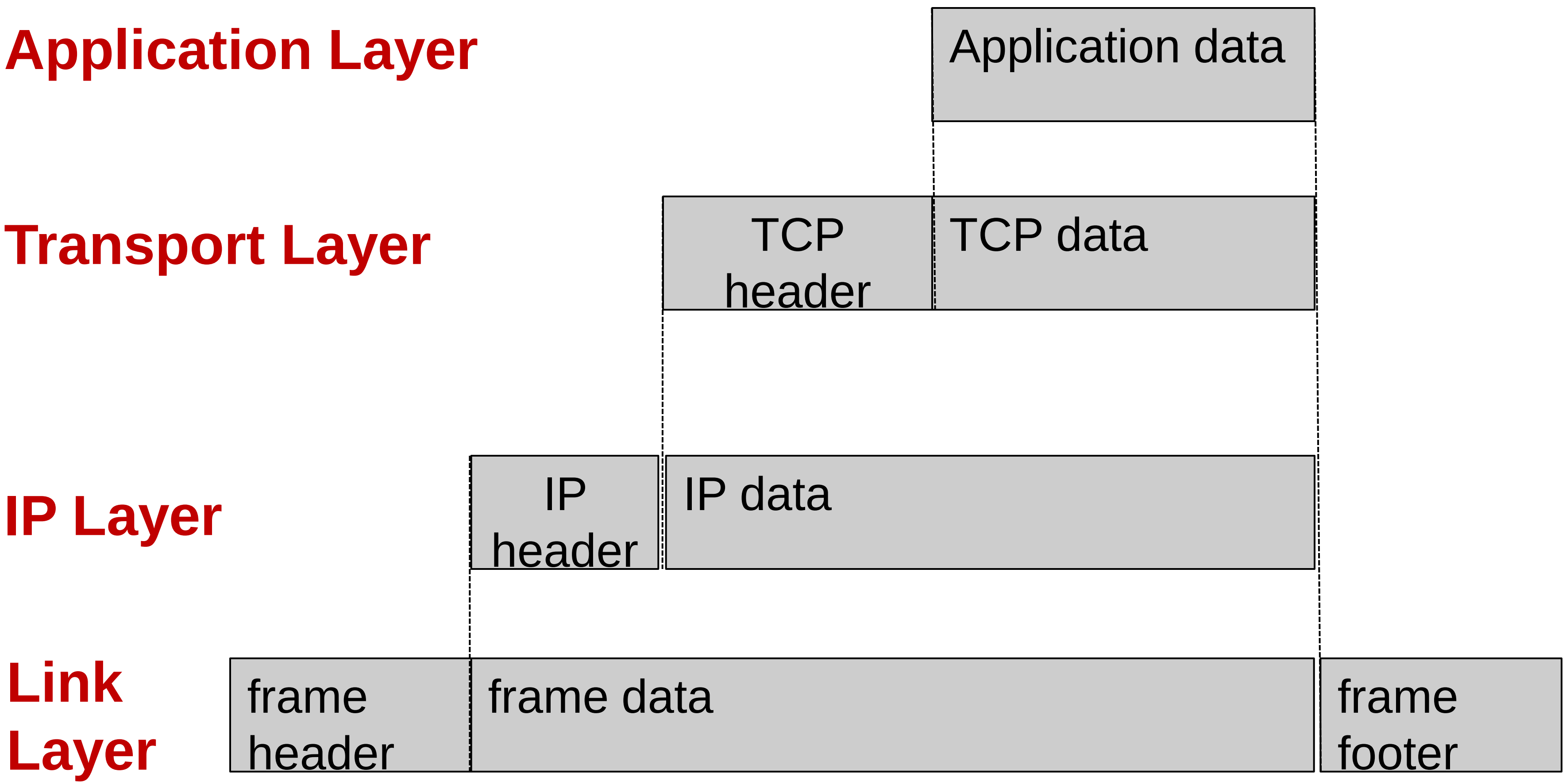
CS 161: Computer Security

Prof. Raluca Ada Popa

March 13, 2018

Quick detour to ARP

Recall the layers stack



Link layer

- Units of transmission are data frames
- Every device that connects to a network has a network interface which has an Identifier called MAC address (Media Access Control)
  - MAC adress is a 48-bit identifier: 01:2B:A3:20:A2:5B
- MAC addresses can be changed by software through the network driver so not considered reliable identification

Link layer

- Data frames at the link layer are sent to MAC addresses not IP addresses

ARP protocol (Address Resolution Protocol)

- When a packet needs to be forwarded at a link layer on a local area network, the sender has a destination IP but needs the MAC address of the destination
- So sender broadcasts an ARP request
- Example:
  - “I have MAC X. Who has IP address 192.100.0.0”?
  - The machine with that IP address sends a reply in a frame addressed to the sender:

    “For X: 192.100.0.0 is at 00:12:B7:93:21:A2”
- Answer is cached in the ARP cache by receiver

ARP spoofing

- It does not have any authentication
- So what can an attacker do?
  - Spoof replies to ARP requests
    "192.100.0.0 is at 01:82:A1:93:21:A2"
- Any machine receiving an ARP reply even without request updates the ARP cache

Man-in-the-middle attack on ARP

How would you do a cache poisoning MITM attack on ARP?

- Eve sends ARP reply to Alice to associate Bob's IP address to Eve's MAC
- Eve sends ARP reply to Bob to associate Alice's IP address with Eve's MAC
- Eve can then observe or modify traffic

How would you do a DoS on ARP?

- Eve sends ARP replies to Alice mapping relevant IP addresses to inexistent or bad MACs

How to address ARP spoofing attacks?

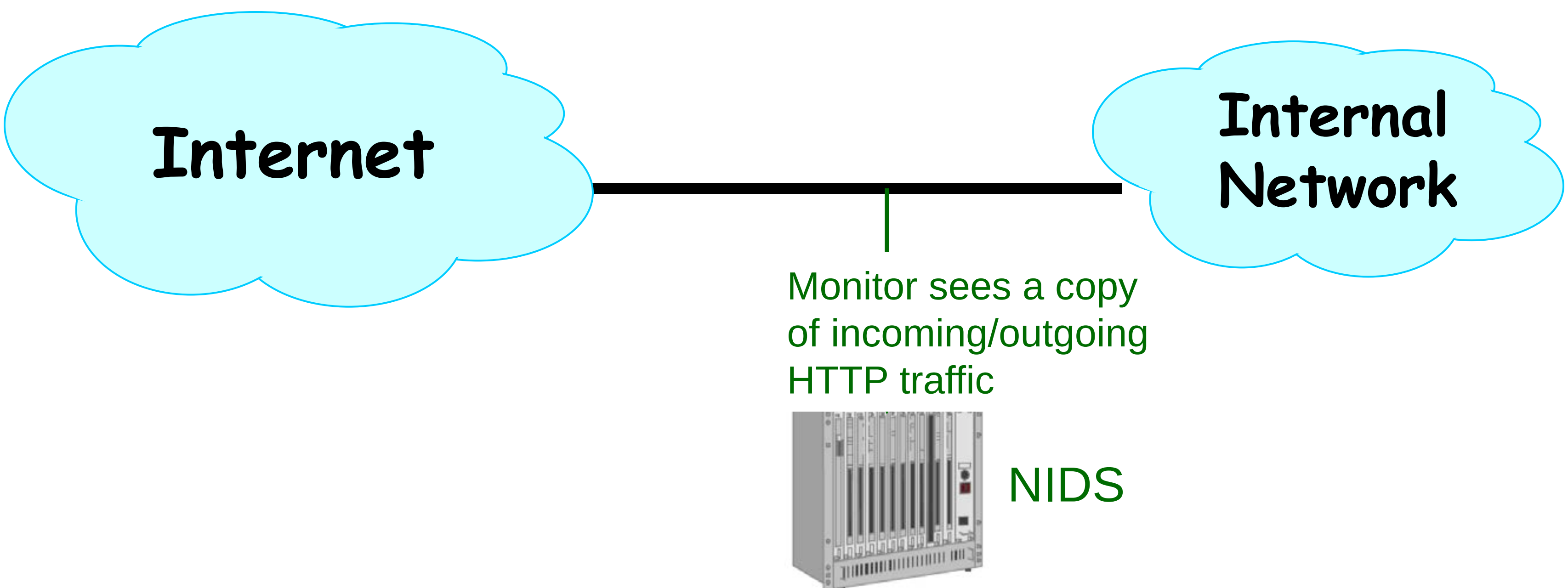
Some ideas:

- Have only trusted users have access to a local network
- Multiple occurrences of the same MAC address on a LAN (local-area network) can be an indication
- Static ARP tables: admin specifies the ARP cache at a device and this does not change (inconvenient)

Back to IDS
(intrusion detection system)

Network Intrusion Detection (NIDS)

- Passively monitor network traffic for signs of attack at perimeter of a network
  - Look for certain rules (e.g., /etc/passwd)
  - Flag a warning to an administrator, do not take preemptive action



NIDS rules set

- A set of rules (string matching, regular expression) that identifies an attack
- Example rule:
  - “any flow containing /etc/password should be flagged”
  - “any flow containing attack.exe should be flagged”

What does a NIDS aim to detect?

Examples:

- Port scans: information gathering intended to determine which ports are open for TCPconnections
- DoS attacks
- Malware (replicating malicious software)
- DNS cache poisoning
- ARP spoofing

Network Intrusion Detection (NIDS)

- NIDS has a table of all active connections, and maintains state for each
  - e.g., has it seen a partial match of /etc/passwd?

- When it sees a new packet not associated with any known connection, it creates a new connection: when NIDS starts it doesn’t know what connections might be existing
  - Meant to be simply added in the network without disrupting

Evasion

Evasion attacks can arise when you have "double parsing"

- *Inconsistency* – interpreted differently
- *Ambiguity* – information needed to interpret is missing

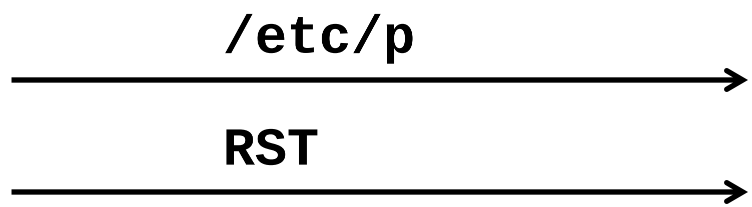
Or when you attack the IDS

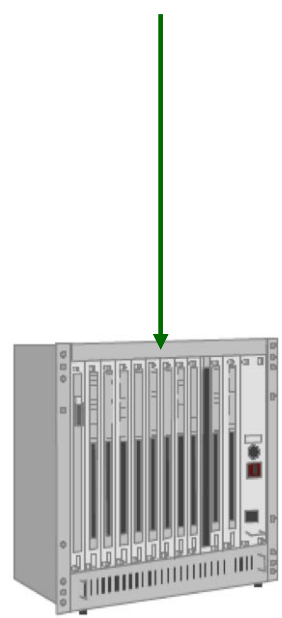
Evasion Attacks (High-Level View)

- Some evasions reflect incomplete analysis
  - In our FooCorp example, hex escapes or "..////.//../" alias
  - In principle, can deal with these with implementation care (make sure we fully understand the spec)
- Some are due to imperfect observability
  - For instance, if what NIDS sees doesn't exactly match what arrives at the destination
- Some are due to attacking the IDS itself

Evasion

- What should NIDS do if it sees a RST packet?





NIDS

(a)Assume RST will be received

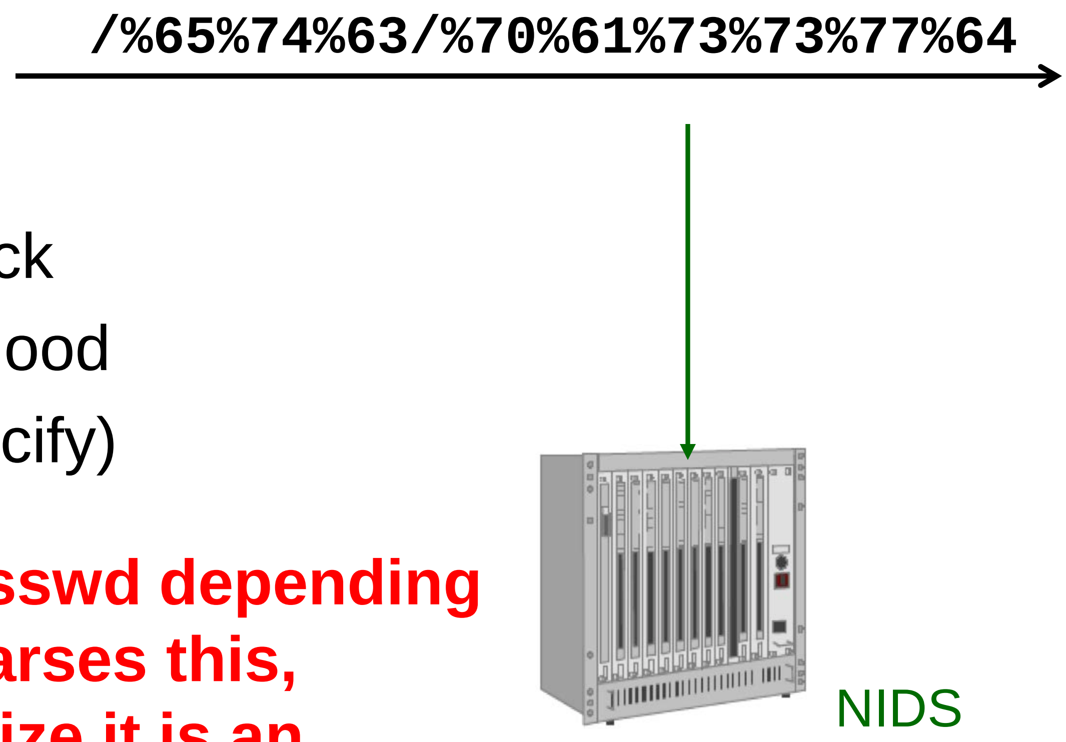
(b)Assume RST won’t be received

(c)Other (please specify)

Safer to consider both possibilities

Evasion

- What should NIDS do if it sees this?



(a)Alert – it's an attack

(b)No alert – it's all good

(c)Other (please specify)

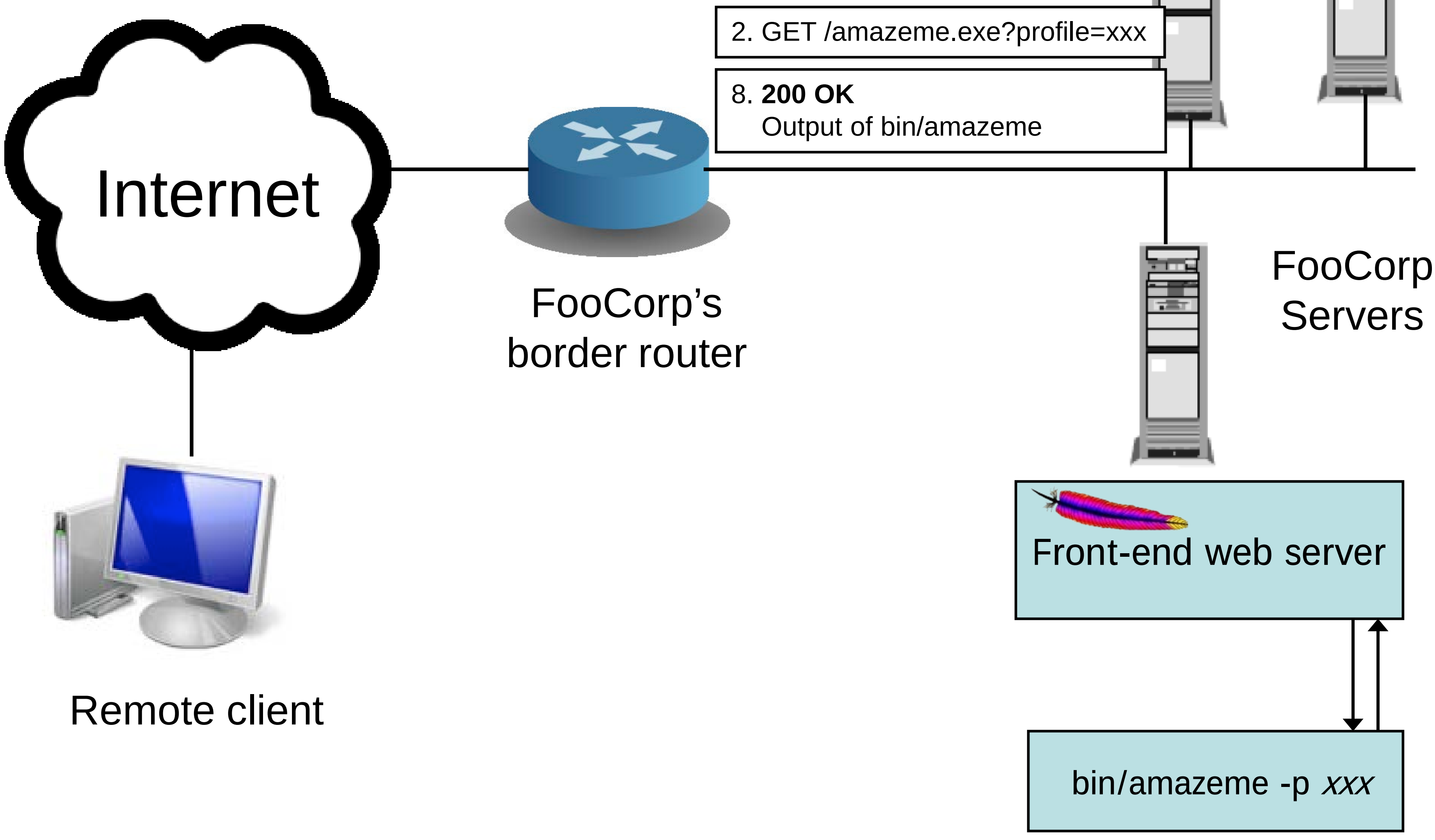
This can be /etc/passwd depending on what protocol parses this, ideally it would realize it is an attack and alert

Evasion

How can you mount a DoS on the IDS?

- Send so many attacks that matches rules to the IDS making the IDS log so much data that it becomes slow or runs out of resources
- Or fake new connections so the IDS creates new state

Structure of FooCorp Web Services



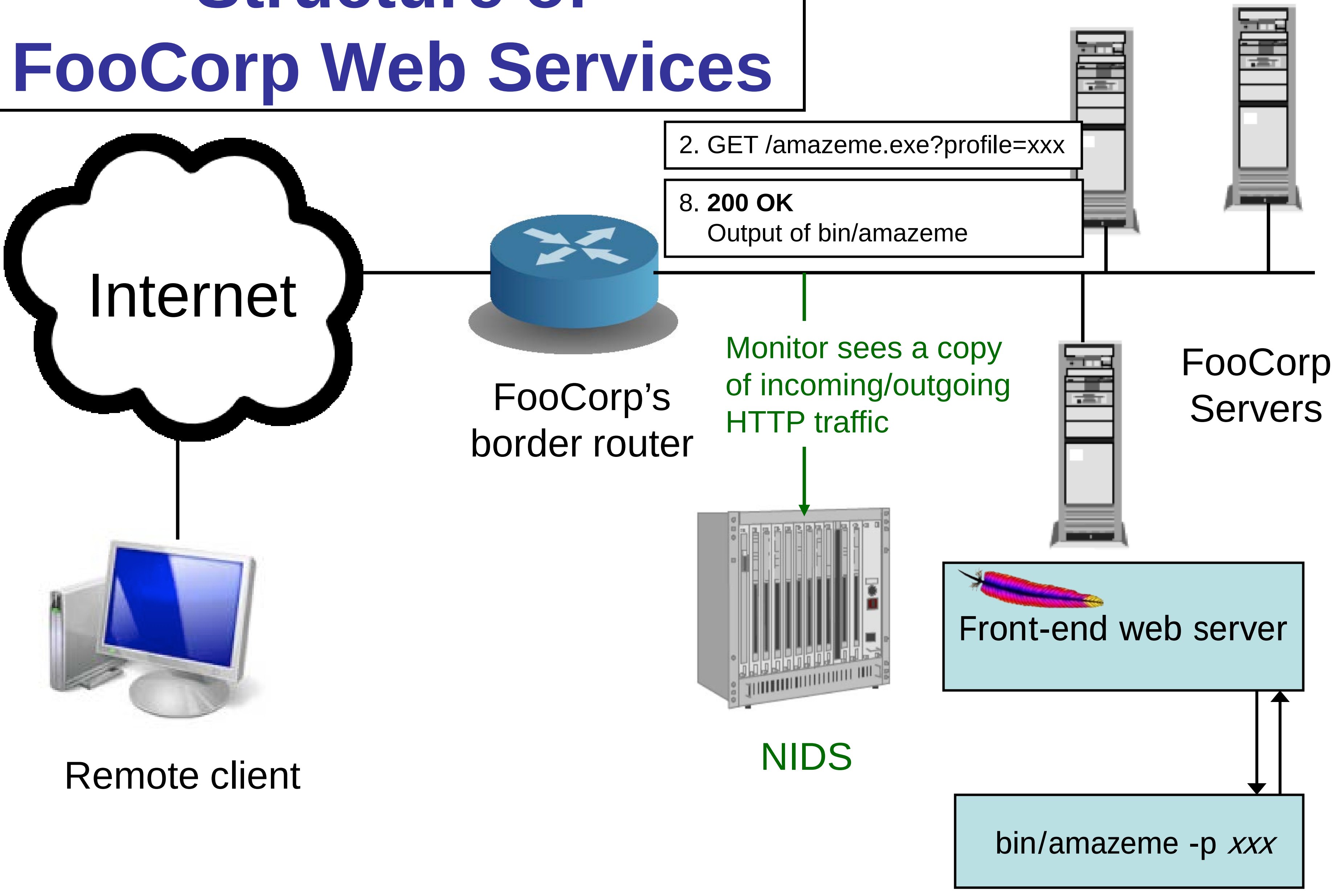
At least two types of IDS

- NIDS: sits in the network
- HIDS: sits at the end host

Network Intrusion Detection

- Approach #1: look at the network traffic
  - (a “NIDS”: rhymes with “kids”)
  - Scan HTTP requests
  - Look for “/etc/passwd” and/or “../../”

Structure of FooCorp Web Services



Network Intrusion Detection

- Approach #1: look at the network traffic
  - (a “NIDS”: rhymes with “kids”)
  - Scan HTTP requests
  - Look for “/etc/passwd” and/or “../../”
- Pros:
  - No need to touch or *trust* end systems
    - Can “bolt on” security
  - Cheap: cover many systems w/ single monitor
  - Cheap: centralized management

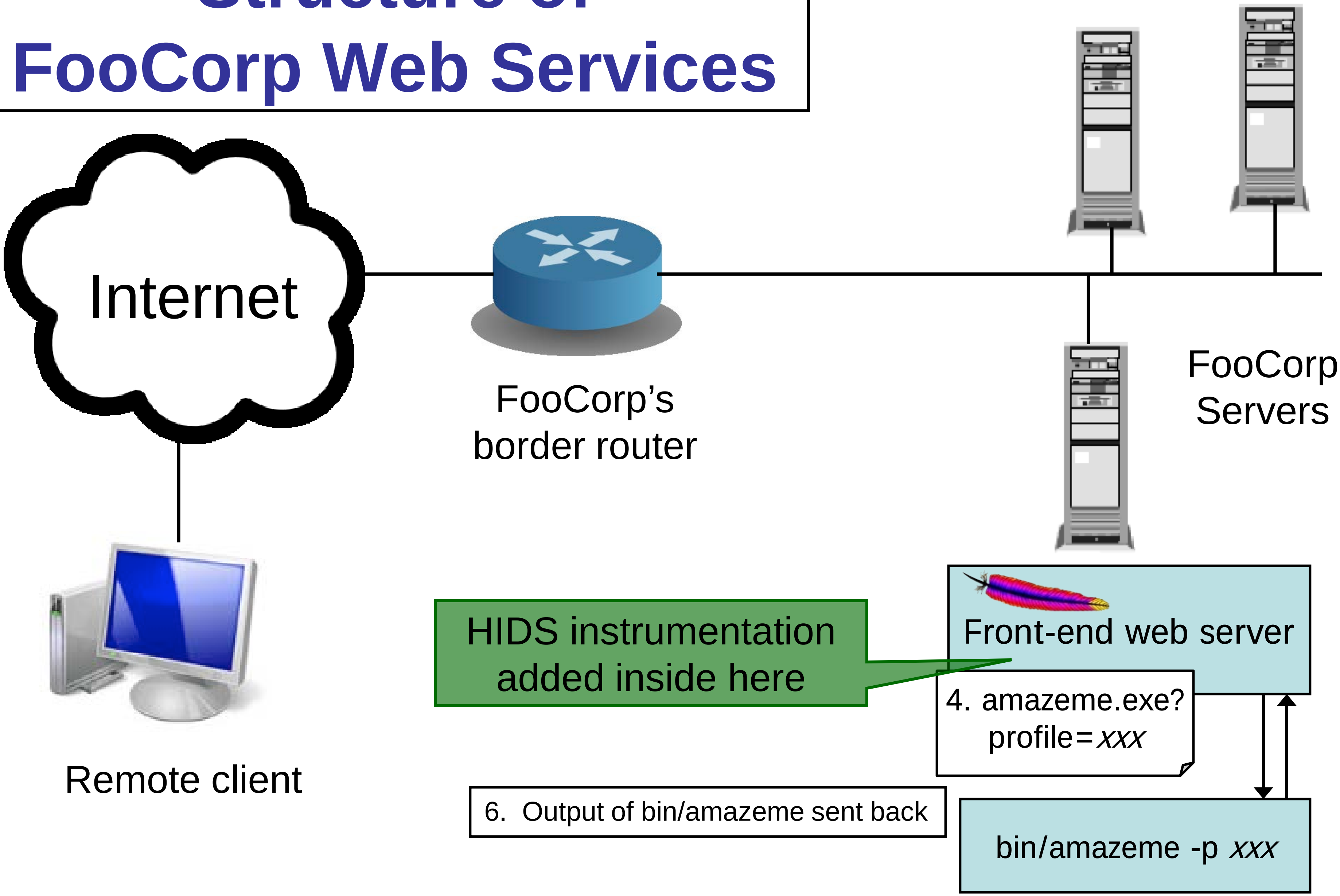
Network-Based Detection

- Issues:
  - Scan for “/etc/passwd”?
    - What about *other* sensitive files?
  - Scan for “../../”?
    - Sometimes seen in legit. requests (= *false positive*)
    - What about “%2e%2e%2f%2e%2e%2f”? (= *evasion*)
      - It needs to do full HTTP parsing
    - What about “..///.///..////”?
      - It needs to understand Unix filename semantics too!
  - What if it’s HTTPS and not HTTP?
    - Need access to decrypted text / session key – yuck!

Host-based Intrusion Detection

- Approach #2: instrument the web server
  - Host-based IDS (sometimes called "HIDS")
  - Resides on a single system and monitors activity on that machine (e.g., OS calls, system logs) and monitors abnormal activity
  - Use heuristics for what is considered to be abnormal activity, e.g., accessing system logs
  - Scan arguments sent to back-end programs
    - Look for "/etc/passwd" and/or "../../"

Structure of FooCorp Web Services



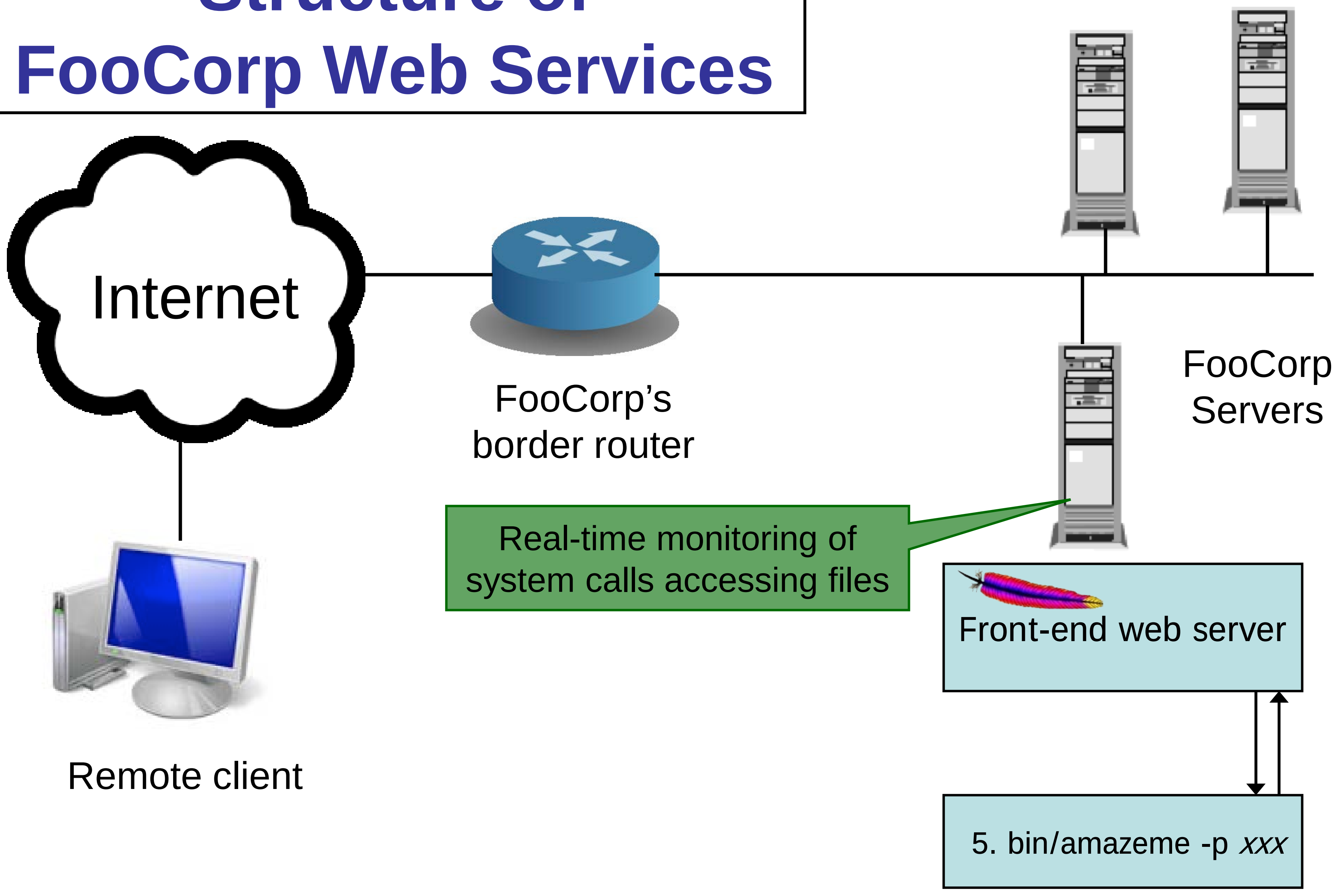
HIDS

- HIDS attempt #1: scan for arguments sent to back-end programs
    - Look for “/etc/passwd” and/or “../../”
- Pros:
  - No problems with HTTP complexities like %-escapes
  - Works for encrypted HTTPS! (because it gets decrypted at endpoint host)
- Issues:
  - Have to add code to each (possibly different) web server
    - And that effort only helps with detecting web server attacks
  - Still have to consider Unix filename semantics (“..////.//”)
  - Still have to consider other sensitive files

Add system Call Monitoring to HIDS

- HIDS attempt #2: monitor system call activity of backend processes
  - Look for access to /etc/passwd which is a sys call

Structure of FooCorp Web Services



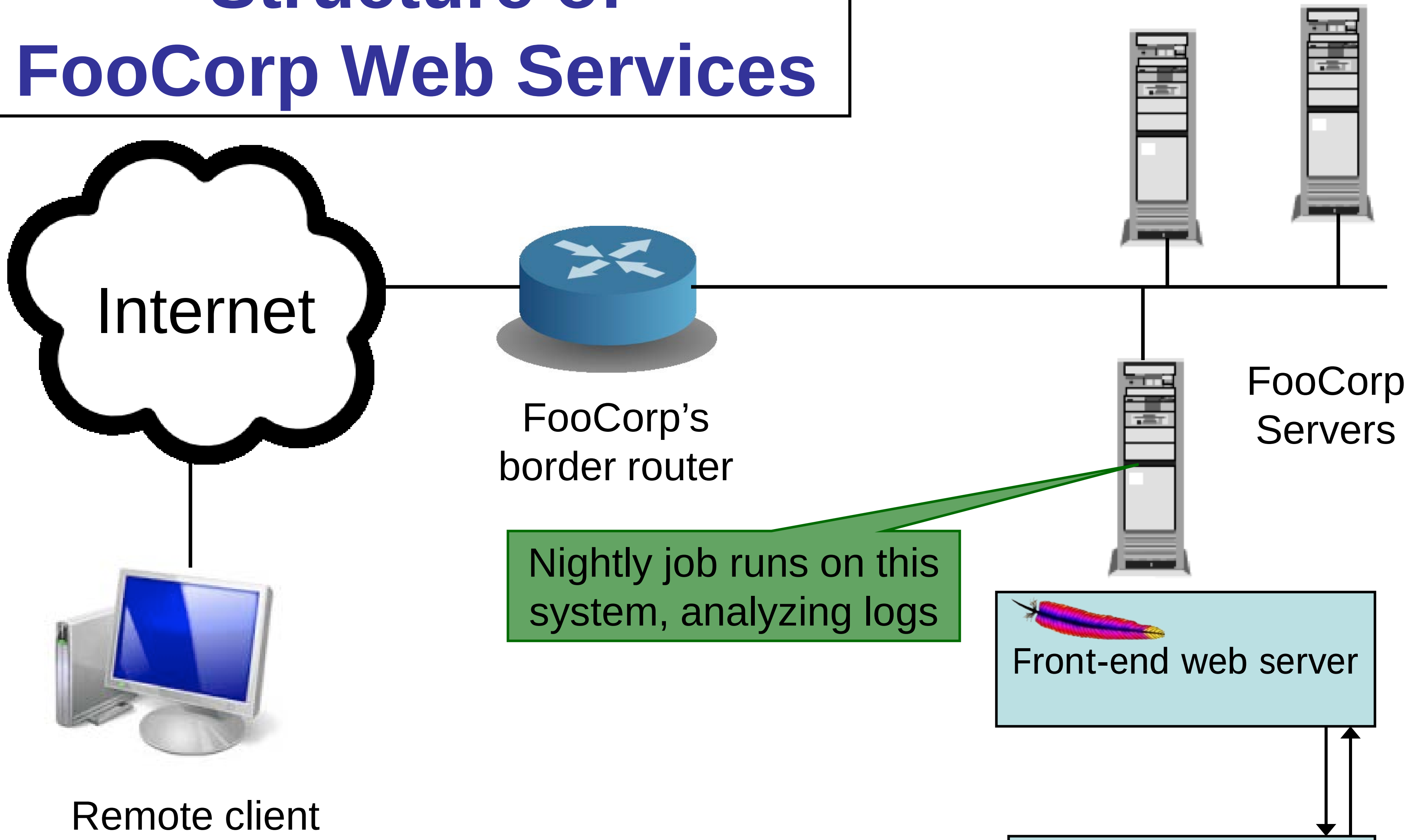
System Call Monitoring (HIDS)

- Approach #4: monitor system call activity of backend processes
  - Look for access to /etc/passwd
- Pros:
  - No issues with any HTTP complexities
  - May avoid issues with filename tricks
  - Attack only leads to an "alert" if attack succeeded
    - Sensitive file was indeed accessed
- Issues:
  - Maybe other processes make legit accesses to the sensitive files (*false positives*)
  - Maybe we'd like to detect attempts even if they fail?
    - "situational awareness"

Log Analysis

- HIDS attempt #3: each night, script runs to analyze log files generated by web servers
  - Again scan arguments sent to back-end programs

Structure of FooCorp Web Services



Log Analysis

- HIDS attempt #3: each night, script runs to analyze log files generated by web servers
  - Again scan arguments sent to back-end programs
- Pros:
  - Cheap: web servers generally already have such logging facilities built into them
  - No problems like %-escapes, encrypted HTTPS since it is at the web application level
- Issues:
  - Again must consider filename tricks, other sensitive files
  - Can’t block attacks & prevent from happening
  - Detection delayed, so attack damage may **compound**
  - If the attack is a compromise, then malware might be able to alter the logs before they’re analyzed
    - (Not a problem for directory traversal information leak example)

Typical HIDS

- A combination of the three attempts, monitor system calls, program inputs and system logs. The more information the better.

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

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;
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
- This example detects any* attempt to exploit a particular buffer overflow in IIS web servers
  – Used by the "Code Red" worm
  * (Note, signature is not quite complete)

Vulnerability Signatures

- What's nice about this approach?
  - Conceptually fairly simple
  - Takes care of known attacks
  - Easy to share signatures, build up libraries
  - Can detect variants of known attacks
  - Much more concise than per-attack signatures

Benefits of attack signatures

- What's problematic?
  - Can't detect novel attacks (new vulnerabilities)
  - Signatures can be hard to write / express
    - Can't just observe an attack that works …
    - … need to delve into how it works

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

- What's problematic about this approach?
  - 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)

Hard to make work well - not widely used today

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` (don't save bash history)

- 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

The Problem of Evasion

- For any detection approach, we need to consider how an adversary might (try to) elude it
  - *Note: even if the approach is evadable, it can still be useful to operate in practice*
  - **But**: if it's very easy to evade, that's especially worrisome (security by obscurity)

The Problem of Evasion

- Imperfect observability is particularly acute for network monitoring
- Consider detecting occurrences of the (arbitrary) string "**`root`**" inside a network connection …
  - We get a copy of each packet, how hard can it be?

Detecting "`root`": Attempt #1

- Method: scan each packet for 'r', 'o', 'o', 't'
  - Perhaps using Boyer-Moore, Aho-Corasick, Bloom filters …

Packet

1 ………..root……………………..

Are we done?

Oops: TCP *doesn't preserve text boundaries*

1 ………..ro

2 ot……………………..

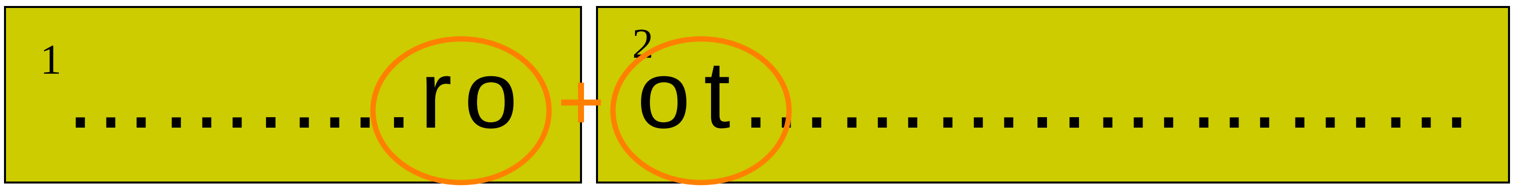
Packet #1

Packet #2

Fix?

Detecting “`root`”: Attempt #2

- Okay: remember match from end of previous packet



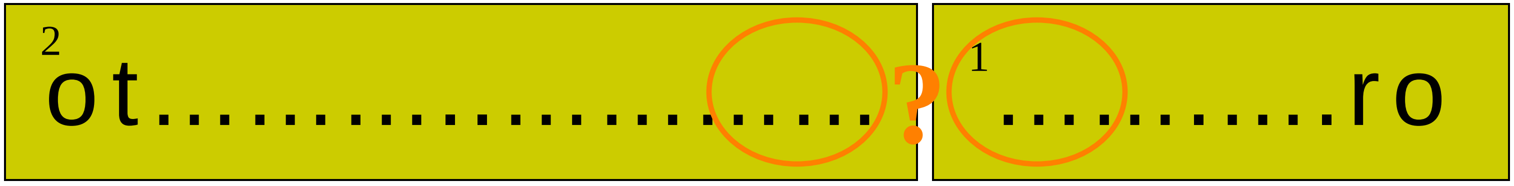
Packet #1 Packet #2

When 2nd packet arrives, continue working on the match

- Now we’re managing state :-(

Are we done?

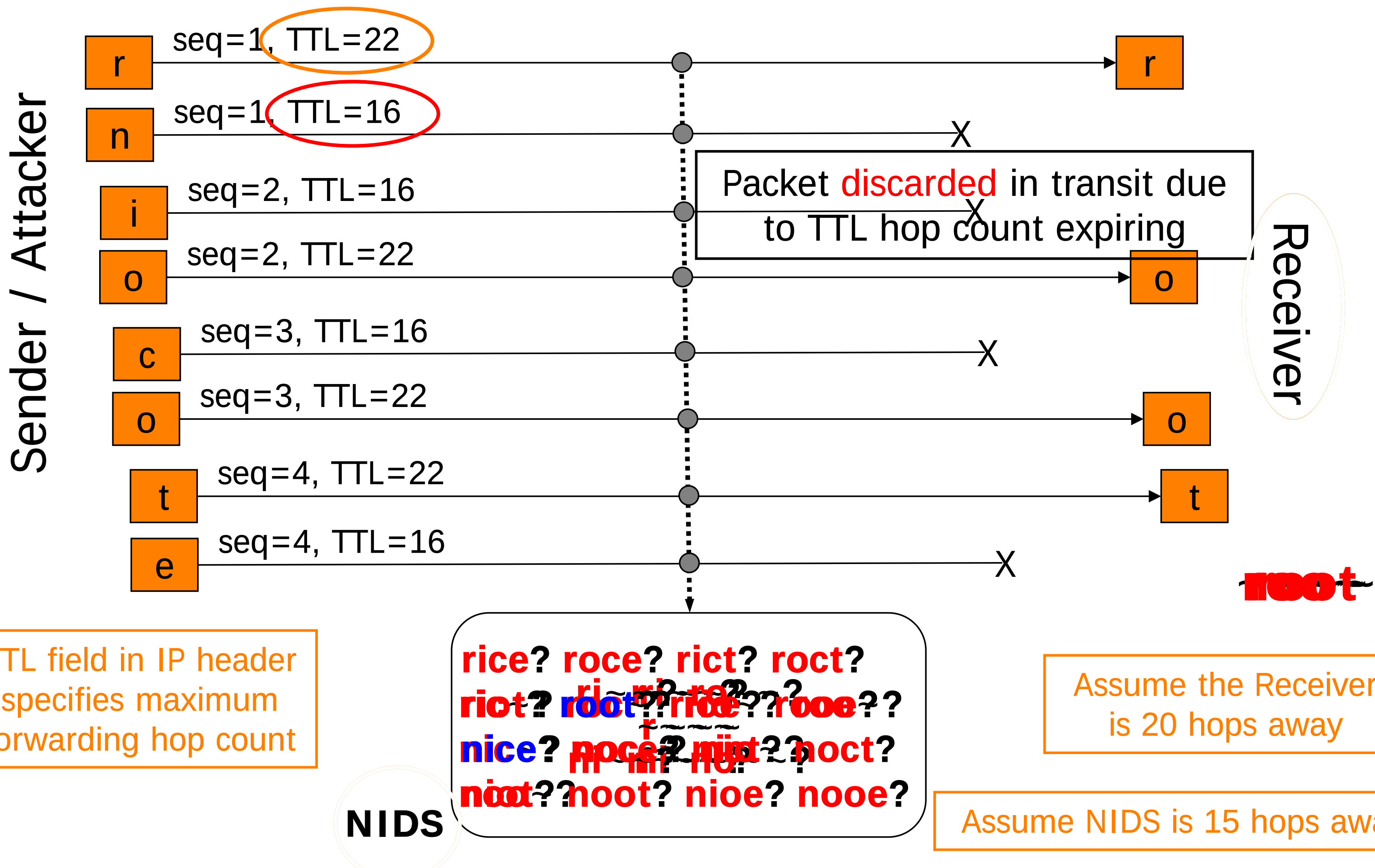
Oops: IP doesn’t guarantee in-order arrival



Detecting “`root`”: Attempt #3

- Fix?
- We need to reassemble the entire TCP bytestream
  - Match sequence numbers
  - Buffer packets with later data (above a sequence “hole”)
- Issues?
  - Potentially requires a lot of state
  - Plus: attacker can cause us to exhaust state by sending lots of data above a sequence hole
- But at least we’re done, right?

Full TCP Reassembly is Not Enough



TTL field in IP header specifies maximum forwarding hop count

Assume the Receiver is 20 hops away

Assume NIDS is 15 hops away

Inconsistent TCP Retransmissions

- Fix?
- Idea: NIDS can alert upon seeing a retransmission inconsistency (two packets for same seqno), as surely it reflects someone up to no good
- This doesn't work well in practice: TCP retransmissions broken in this fashion occur in live traffic
  - Fairly rare (23 times in a day of ICSI traffic)
  - But real evasions much rarer still (Base Rate Fallacy)

  ⇒ This is a *general problem* with alerting on such ambiguities
- Idea: if NIDS sees such a connection, kill it
  - Works for this case, since benign instance is already fatally broken
  - But for other evasions, such actions have collateral damage
- Idea: rewrite traffic to remove ambiguities
  - Works for network- & transport-layer ambiguities
  - But must operate in-line and at line speed

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: fix the basic observation problem
  - E.g., monitor directly at end systems

Inside a Modern HIDS ("AV")

- 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

Summary of 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