Sentinel

Deep Packet Inspection

DEFENDING OUR DIGITAL WAY OF LIFE

Lesson objectives

→ Learn about **deep packet inspection**

→ Learn to identify and reassemble **TCP streams**

Original firewalls

- Permits/denies communication based on packet headers
  - Allow communication with IP 4.2.2.4
  - Deny port 1389
  - etc.
- The data required for making decisions comes from the headers
  - IP header - src, dst IPs
  - TCP header - src, dst ports

What about the payload?

- What could we miss if we don't look into the payload?

Deep packet inspection

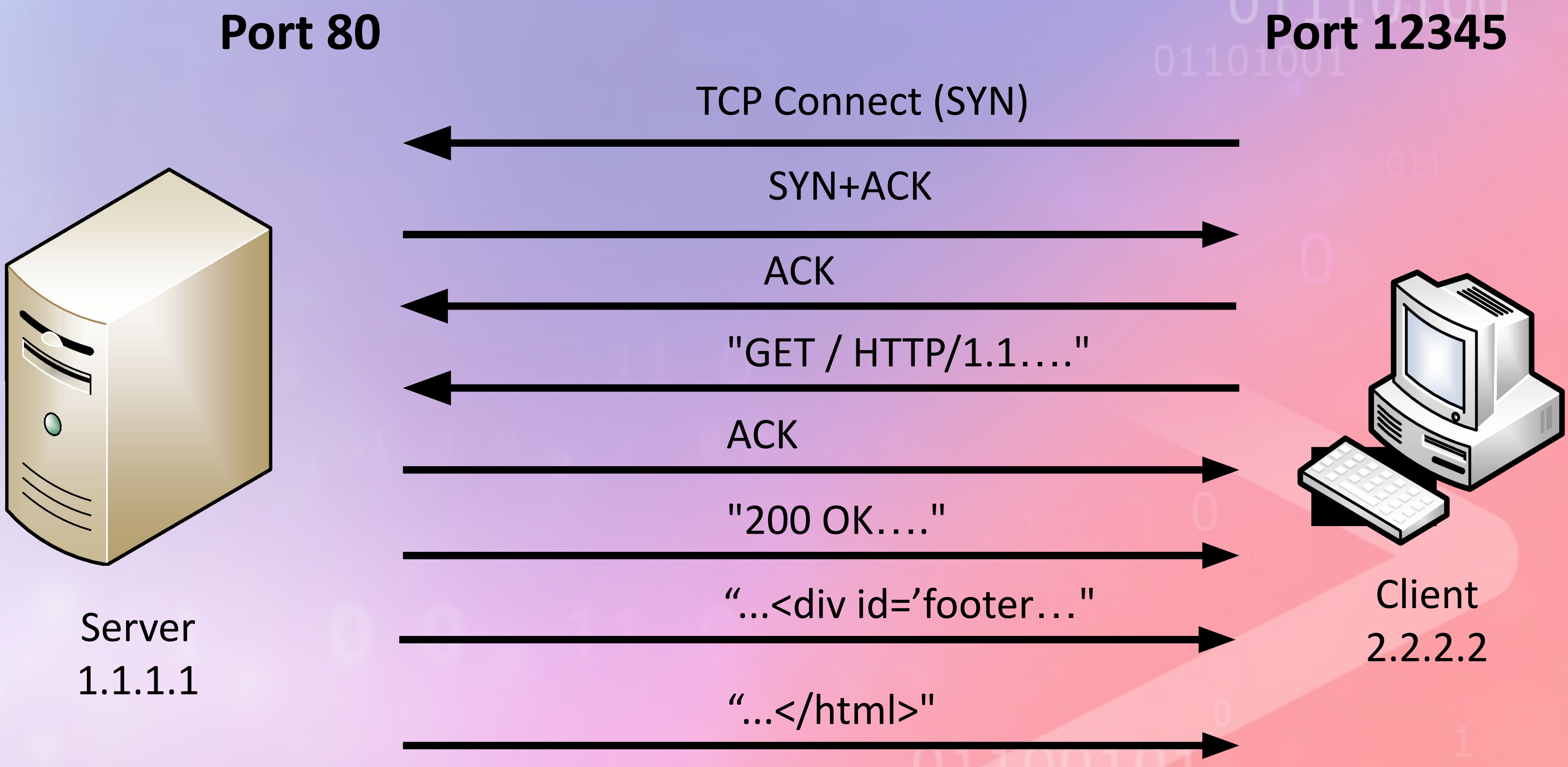
- Simple malicious string search
- Signatures (MD5/SHA256/SHA512)
  - Hash the data and compare with known malware
- Specific detections
  - Varies by application
  - But requires parsing of the TCP payload in question
- Protocol detection
  - Maybe we consider HTTP on port other than 80 as something malicious
    - C2 control channel?

Deep packet inspection

- If we want visibility into TCP payloads in our NIDS
- We have to understand TCP streams first!

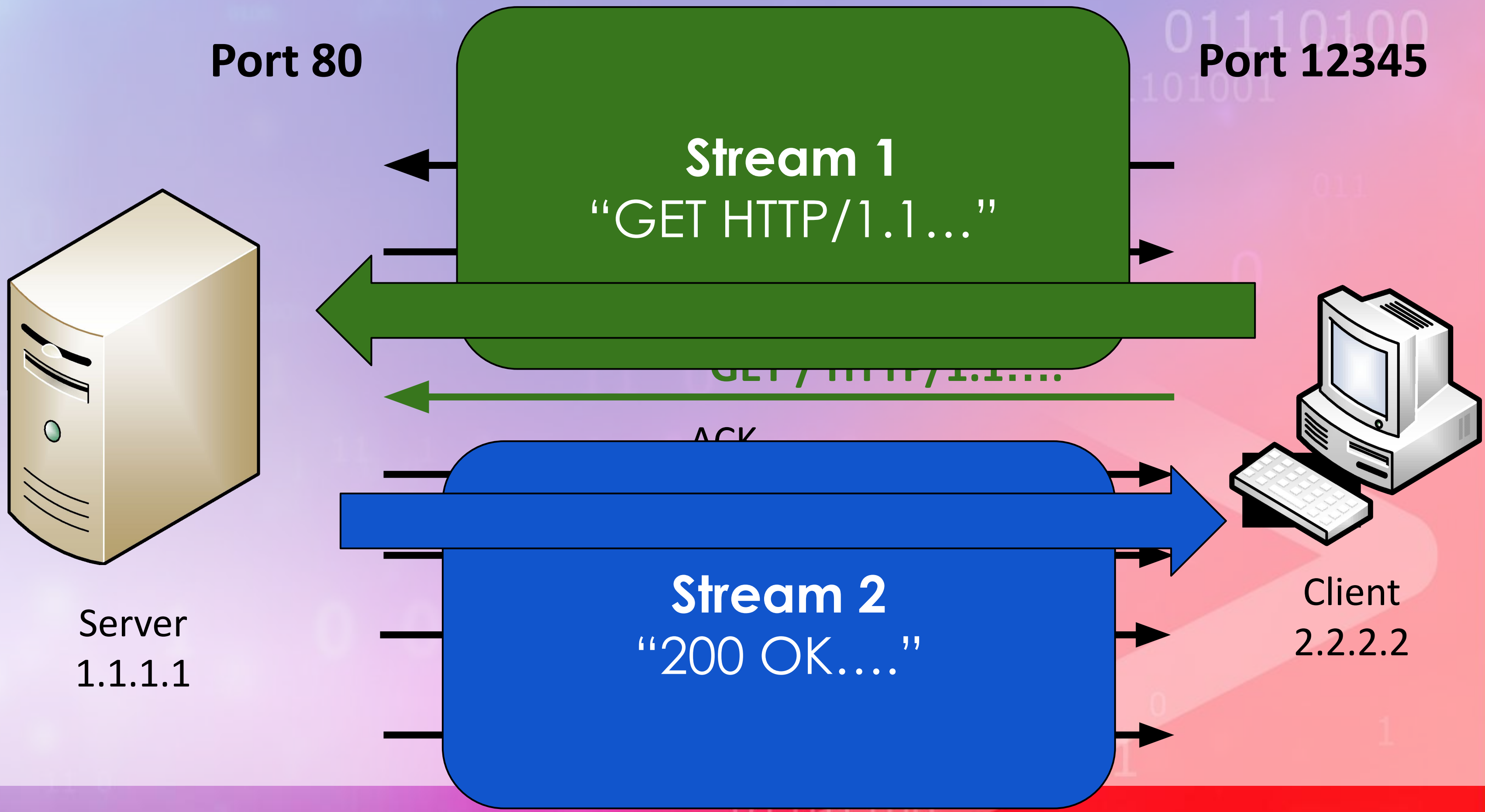
TCP Streams

- How many streams here?



TCP Streams

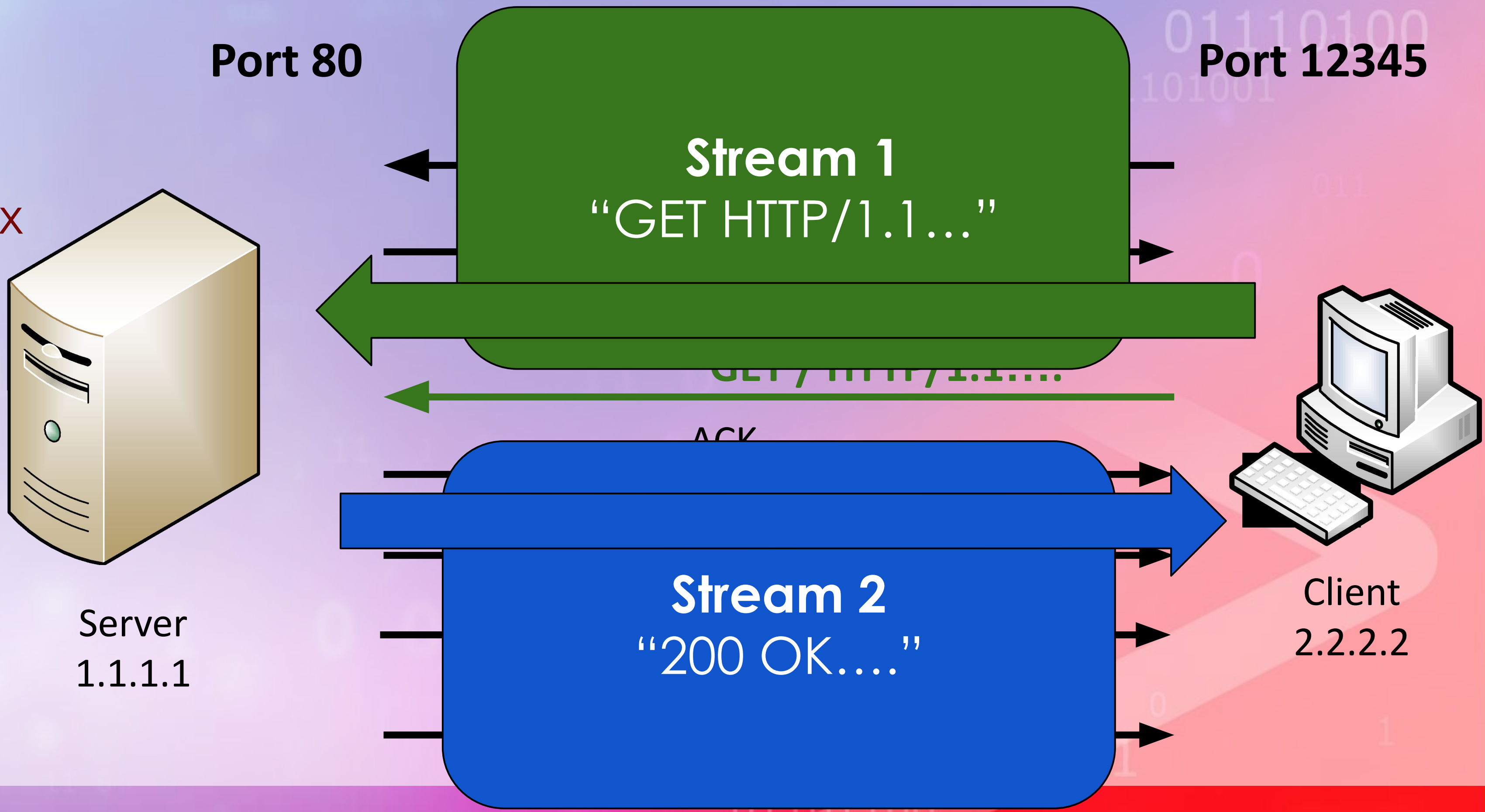
- How many streams here?



TCP Streams

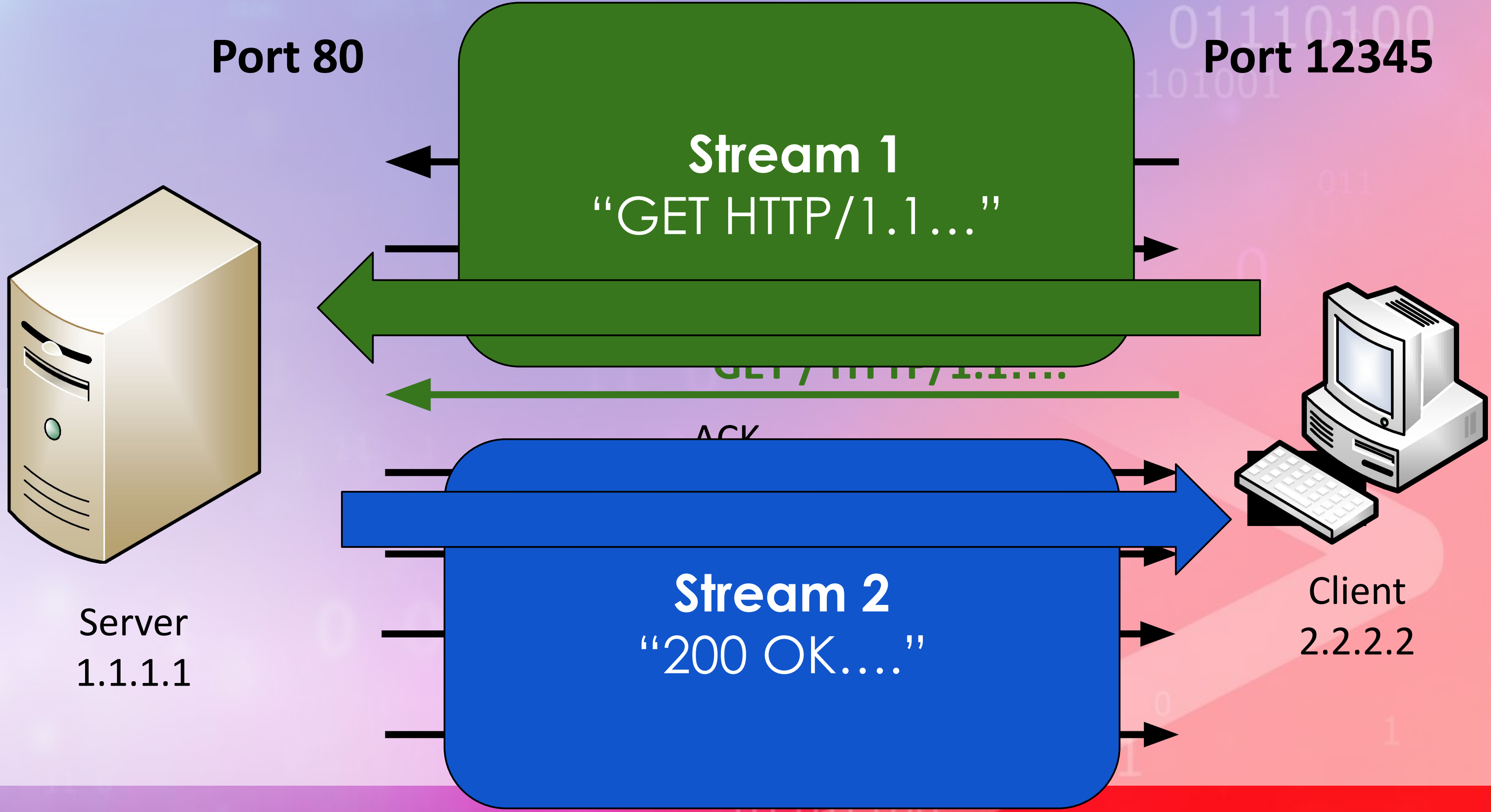
- How many streams here?
- TCP is full-duplex:

  1 stream per direction
- Each stream is half-duplex



TCP Streams

- How do we identify each stream?

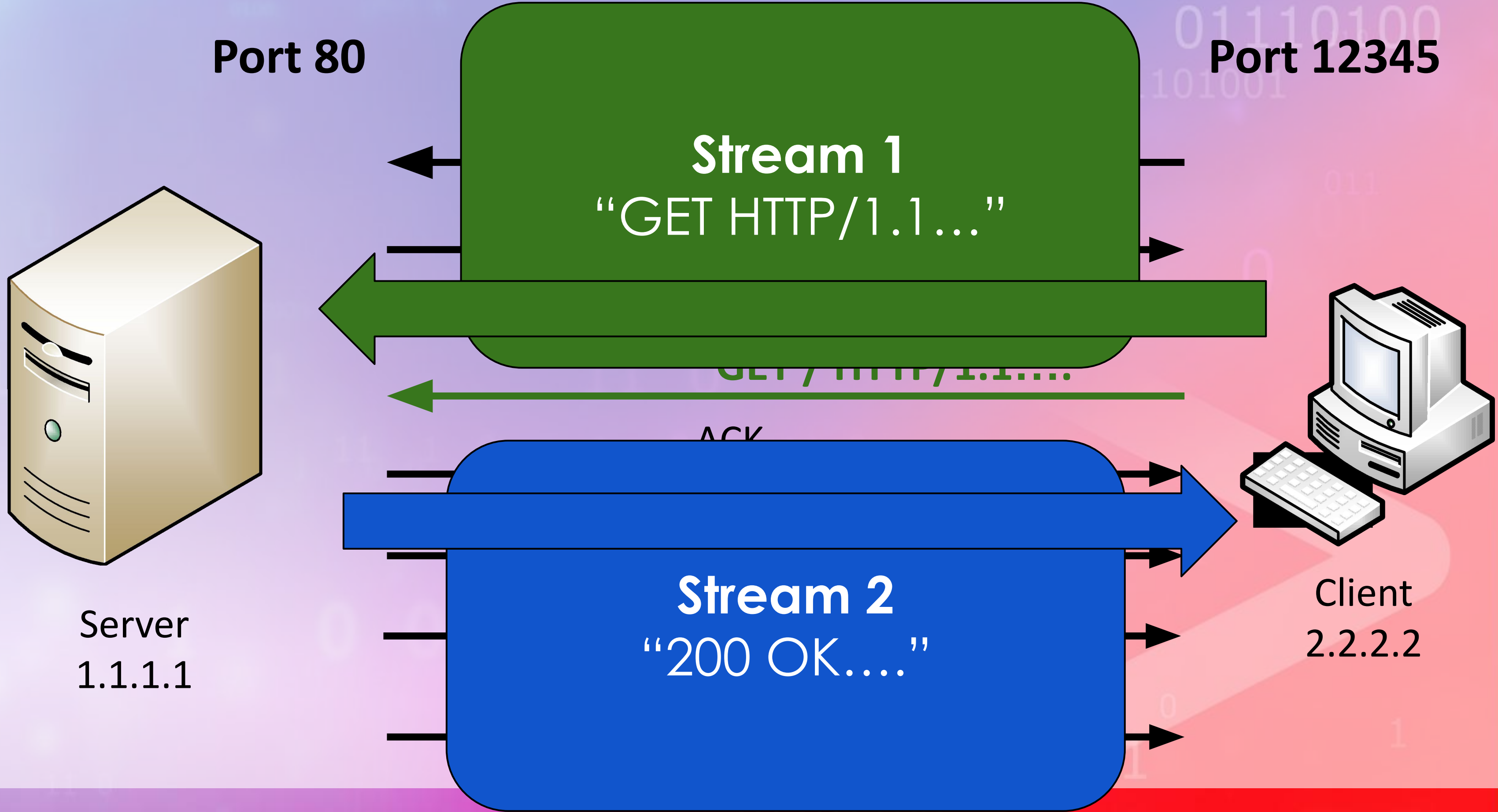


TCP Streams

- How do we identify each stream?

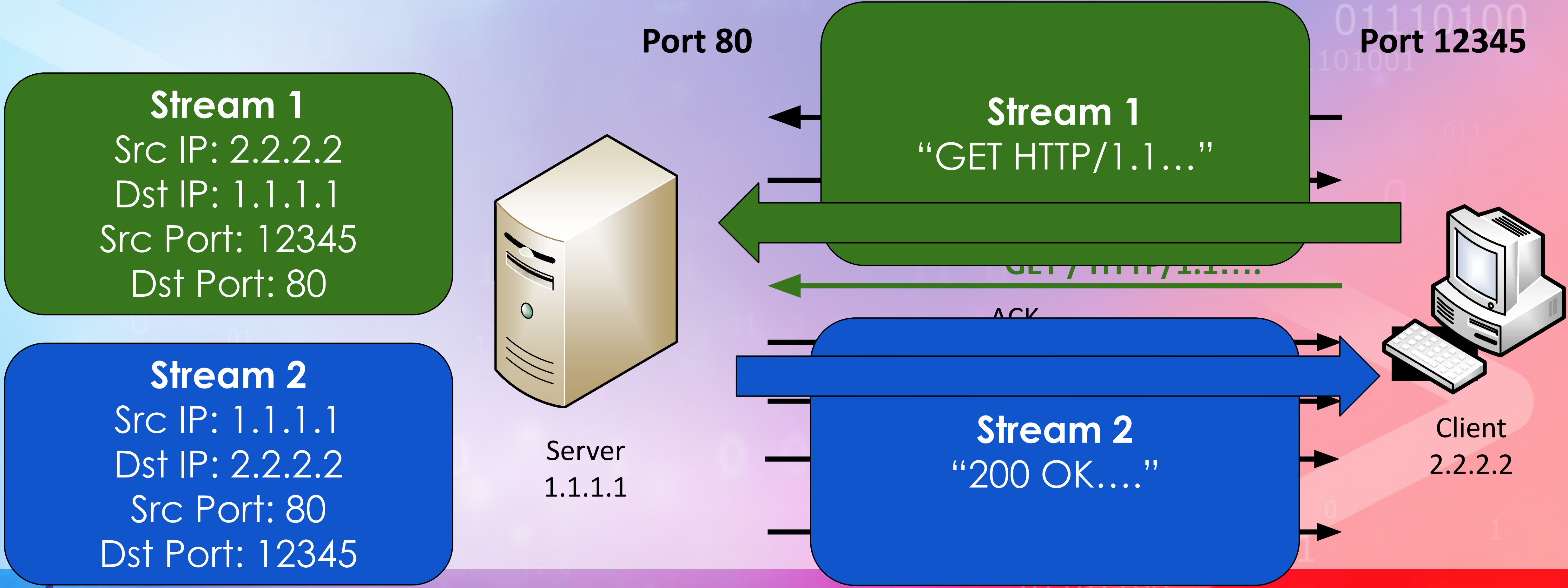
Stream 1
Src IP: 2.2.2.2
Dst IP: 1.1.1.1
Src Port: 12345
Dst Port: 80

Stream 2
Src IP: 1.1.1.1
Dst IP: 2.2.2.2
Src Port: 80
Dst Port: 12345



TCP Streams

- All packets of each stream have the same src/dst IP and src/dst port



Stream reassembly

Packet #1
Src IP: 2.2.2.2
Dst IP: 1.1.1.1
Src Port: 12345
Dst Port: 80

Stream ID	src IP = 1.1.1.1 dst IP = 2.2.2.2 src port = 80 dst port = 12345	src IP = 2.2.2.2 dst IP = 1.1.1.1 src port = 12345 dst port = 80

First packet arrives

Stream reassembly

Packet #2
Src IP: 1.1.1.1
Dst IP: 2.2.2.2
Src Port: 80
Dst Port: 12345

Stream ID	src IP = 1.1.1.1 dst IP = 2.2.2.2 src port = 80 dst port = 12345	src IP = 2.2.2.2 dst IP = 1.1.1.1 src port = 12345 dst port = 80
		Packet #1

Second packet arrives

According to src/dst IP & port,

it belongs to the green stream

Stream reassembly

Packet #3
Src IP: 1.1.1.1
Dst IP: 2.2.2.2
Src Port: 80
Dst Port: 12345

Stream ID	**src IP = 1.1.1.1 dst IP = 2.2.2.2 src port = 80 dst port = 12345**	**src IP = 2.2.2.2 dst IP = 1.1.1.1 src port = 12345 dst port = 80**
	Packet #2	**Packet #1**

According to src/dst IP & port,
it belongs to the blue stream

Stream reassembly

Packet #4
Src IP: 1.1.1.1
Dst IP: 2.2.2.2
Src Port: 80
Dst Port: 12345

Stream ID	src IP = 1.1.1.1 dst IP = 2.2.2.2 src port = 80 dst port = 12345	src IP = 2.2.2.2 dst IP = 1.1.1.1 src port = 12345 dst port = 80
	Packet #3	
	Packet #2	Packet #1

Stream reassembly

All packets have arrived.

Final state:

Stream 1	Packet2.data	Packet3.data	Packet4.data
Stream 2	Packet1.data		

Identified 3 packets to Stream 1, and 1 packet to Stream 2

Stream ID	src IP = 1.1.1.1 dst IP = 2.2.2.2 src port = 80 dst port = 12345	src IP = 2.2.2.2 dst IP = 1.1.1.1 src port = 12345 dst port = 80
	Packet #4	
	Packet #3	
	Packet #2	**Packet #1**

Stream reassembly

Back to our NIDS

- Our code runs on every packet that arrives
- How do we run our detection functions having at once in memory the entire TCP stream?

Stream reassembly

- If we observe each packet's "stream identifier" (src/dst IP/port)
- And group packets based on their "stream identifier"
  - Storing them in list
  - One list per stream
  - In Python?
  - A dict of lists, where the dict key is the stream identifier
- We end up with lists of packets, each list an entire stream
- Concatenate their payload (packet[Raw].load)
- And you get entire streams!

Stream reassembly

- That is true for perfect conditions
- In reality, TCP packets could be retransmitted, arrived out of order…
- We would need to take this into account
  - Remember - we're just reading raw packets
  - We're circumventing the operating system's TCP stack
  - Which takes care of these things when we call socket.recv()

The good news

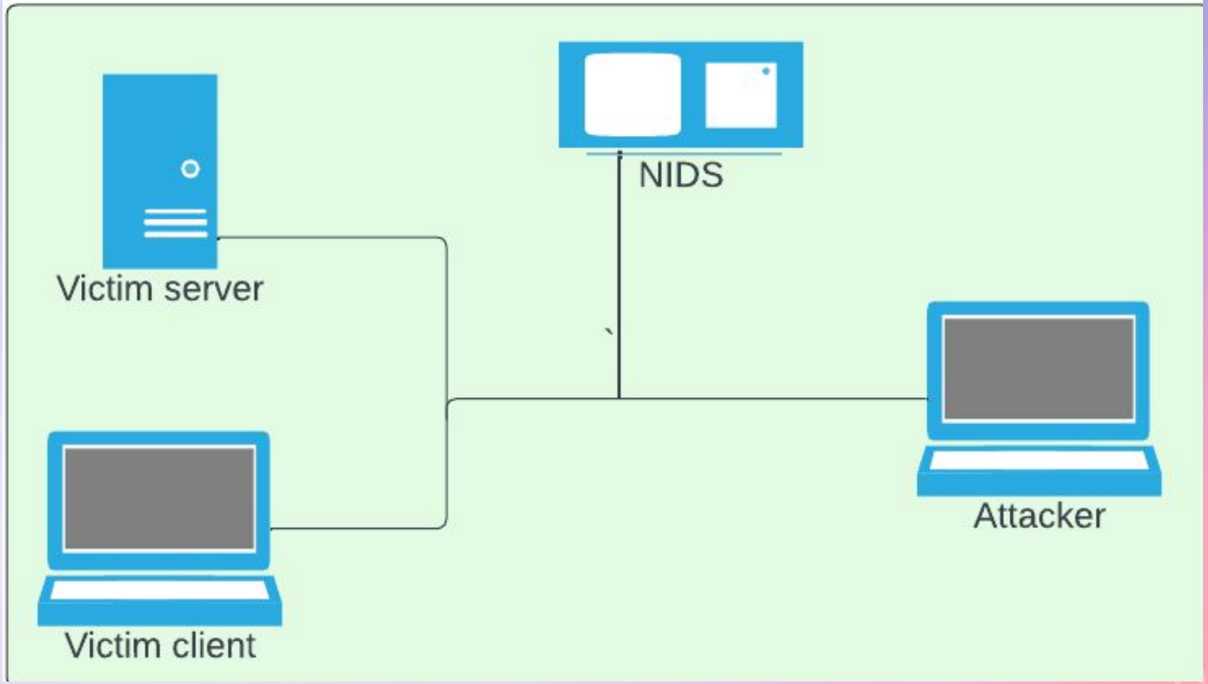
- The conditions are good in the workshop environment 👍
- You can assume there are no retransmissions, out of order packets, etc…
- You will have bugs sometimes
- But if the detection *usually* works, it's fine for today

Reassembling streams is a big deal

- Takes up resources
- Memory could be exhausted
- Important to clean resources
    - When stream has been handled (detection code inspected it)

Stream Insights

- What could we assume a stream is, if its source port is 80?
- What could we assume a stream is, if its destination port is 80?



Reassembled streams

- Now that we have the entire stream data
- And we have a clue what protocol it is
- What detections could we implement?

Basic implementation

- Reassemble streams we want to have detections for
    - e.g. we want to detect malware in HTTP downloads?
    - Reassemble all streams from src port 80
- Once the stream is finished
    - Try to detect the malware
    - And cleanup the stream data (***del var*** in python)

How do we know a stream is finished?

- FIN

127.0.0.1	127.0.0.1	TCP	56 62122 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=2
127.0.0.1	127.0.0.1	TCP	56 8080 → 62122 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS
127.0.0.1	127.0.0.1	TCP	44 62122 → 8080 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
127.0.0.1	127.0.0.1	TCP	44 62122 → 8080 [FIN, ACK] Seq=1 Ack=1 Win=2161152 Len=0
127.0.0.1	127.0.0.1	TCP	44 8080 → 62122 [ACK] Seq=1 Ack=2 Win=2161152 Len=0
127.0.0.1	127.0.0.1	TCP	44 8080 → 62122 [FIN, ACK] Seq=1 Ack=2 Win=2161152 Len=0

- RST

127.0.0.1	127.0.0.1	TCP	56 62142 → 8080 [SYN] Seq=0 Win=65535 Len=0 MSS=65495
127.0.0.1	127.0.0.1	TCP	56 8080 → 62142 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0
127.0.0.1	127.0.0.1	TCP	44 62142 → 8080 [ACK] Seq=1 Ack=1 Win=2161152 Len=0
127.0.0.1	127.0.0.1	TCP	68 62142 → 8080 [PSH, ACK] Seq=1 Ack=1 Win=2161152 Len
127.0.0.1	127.0.0.1	TCP	44 8080 → 62142 [ACK] Seq=1 Ack=25 Win=2161152 Len=0
127.0.0.1	127.0.0.1	TCP	44 62142 → 8080 [FIN, ACK] Seq=25 Ack=1 Win=2161152 Le
127.0.0.1	127.0.0.1	TCP	44 8080 → 62142 [ACK] Seq=1 Ack=26 Win=2161152 Len=0
127.0.0.1	127.0.0.1	TCP	44 8080 → 62142 [RST, ACK] Seq=1 Ack=26 Win=0 Len=0

Summary

→ **Visibility** into **TCP streams** is critical for detections

→ We need to understand **TCP** in order to code it correctly

Q&A