Homework 3

Part 1: Hexademical and Decimal Numbers

Decimal Numbers: Decimal numbers composed of digits ranging from 0 to 9. "Dec-" refers to the fact that each digit can take on one of ten distinct values. Hexadecimal numbers differ from decimal numbers in that each digit ranges from 0-9 and then A-F thereby allowing 16 different possible values for each digit. However, hexadecimal numbers and decimal numbers are otherwise similar. We can add, subtract, multiply, divide and otherwise manipulate hexadecimal numbers in much the same way we would decimal numbers. However, with hexadecimal, when adding, we only carry a number when the summation of the two digits exceeds 15 (i.e., F). Analogous modifications to the standard rules of arithmetic apply to accomodate the expanded range of allowed values for digit when using hexadecimal. Below we provide a table of the first 16 numbers starting from 0 representable using hexadecimal and decimal.

Examples:

Hexadecimal Decimal

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

A 10

B 11

C 12

D 13

E 14

F 15

Often times one can infer that a number is hexadecimal by the fact that the number contains a letter in A-F. In Python, C, C++, and various other languages, we denote a hexadecimal number by preceding the number with the characters "0x". For example, in Python

>>> x = 0xB

>>> x

11

In math we denote whether a number is hexadecimal by appending a subscript 16 to the end of the number, as in

The 16 is called the number's base. For decimal numbers, the base is 10. We could thus rewrite the above as

We can decompose a decimal number into its ones, tens, and hundreds digits and so on as illustrated below.

Consider the decimal number

.

We can expand it into its ones, tens, and hundreds digits as follows:

\begin{align\*} 456\_{10} &= 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0 \ &= 4 \times 100 + 5 \times 10 + 6 \times 1 \ &= 400 + 50 + 6 \end{align\*}

Thus, the number

is composed of: \begin{itemize} \item Hundreds digit: (4 \times 10^2) \item Tens digit: (5 \times 10^1) \item Ones digit: (6 \times 10^0) \end{itemize}

We can do the same with hexadecimal numbers except using powers of 16.

Consider the hexadecimal number

.

We can expand it into its ones, sixteens, and two hundred fifty-sixths digits as follows:

\begin{align\*} 1A3\_{16} &= 1 \times 16^2 + A \times 16^1 + 3 \times 16^0 \ &= 1 \times 256 + 10 \times 16 + 3 \times 1 \ &= 256 + 160 + 3 \end{align\*}

Thus, the number

is composed of: \begin{itemize} \item Two hundred fifty-sixths digit:

\item Sixteens digit:

\item Ones digit:

\end{itemize}

In decimal the summation of these hexadecimal digits becomes

When dealing with decimal numbers it is customary to omit the base 10.

When dealing with hexadecimal numbers often times we use 0x or subscript 16 except when the context is sufficiently clear.

You can break a decimal number into its digits by successively dividing by 10 as in the following example.

Decimal Example: Breaking a Number into its Digits

Let's take the number 365 and break it into its ones, tens, and hundreds digits by successively dividing by 10.

Start with 365.

Divide by 10:

Quotient: 36

Remainder: 5 (ones digit)

Divide the quotient (36) by 10:

Quotient: 3

Remainder: 6 (tens digit)

Divide the quotient (3) by 10:

Quotient: 0

Remainder: 3 (hundreds digit)

So, the ones digit is 5, the tens digit is 6, and the hundreds digit is 3.

Hexadecimal Example: Converting Decimal to Hexadecimal

Let's take the decimal number 365 and convert it to hexadecimal by successively dividing by 16 and converting the remainders to hexadecimal.

Start with 365.

Divide by 16:

Quotient: 22

Remainder: 13 (D in hexadecimal)

Divide the quotient (22) by 16:

Quotient: 1

Remainder: 6 (6 in hexadecimal)

Divide the quotient (1) by 16:

Quotient: 0

Remainder: 1 (1 in hexadecimal)

So, in hexadecimal representation, 365 is equal to 16D.

Convert the following numbers from decimal to hexadecimal:

✅ Problem 1. 27

1B

✅ Problem 2. 45

2D

✅ Problem 3. 81

51

✅ Problem 4. 245

F5

✅ Problem 5. 583

247

Convert the following numbers from hexadecimal to decimal:

✅ Problem 6. 0x1A

26

✅ Problem 7. 0x2F

47

✅ Problem 8. 0x79

121

✅ Problem 9. 0x010A

266

X Problem 10. 0xFFA1

6

IP v6 Addresses

IPv6 addresses are represented as hexadecimal with each 4 hedecimal digits separated by a colon. Each 4 hexadecimal number corresponds to a 2 byte unsigned integer (i.e., an unsigned short). Double colon means that 1 or more consecutive shorts are 0. For all IP addresses, the most significant bits (the leftmost bits) correspond to a portion of the Internet. Often times a range of IP addresses corresponds to an ISP, a large corporation, or a campus. For example, Ole Miss is allocated all IPv6 addresses a range

NetRange: 2620:38:C000:: - 2620:38:C000:FFFF:FFFF:FFFF:FFFF:FFFF

CIDR: 2620:38:C000::/48

NetName: OLEMISS

CIDR stands for Classless Inter-Domain Routing. A CIDR prefix is the first (leftmost) bits in an IP address. The CIDR prefix for Ole Miss is 2620:38:C000. All network nodes on the Ole Miss campus that use IPv6 addresses will share this same prefix. Unfortunately it appears that much of Ole Miss does not use IPv6 yet.

The word "classless" is historical. In the early Internet, the 32-bit IPv4 address space was broken up into class A, class B, and class C networks. Class A networks were identified by a leading bit of 0 in the first octet of the IP address. The remaining 7 bits were used to identify the network. Class B networks were identified by a leading bit pattern of 10 in the first octet. This allowed for a moderate number of networks (up to 16,384) with a moderate number of hosts per network (up to 65,534). Class C networks were identified by a leading bit pattern of 110 in the first octet. This allowed for a large number of networks (up to 2,097,152) with a small number of hosts per network (up to 254). In 1993, the IETF decided to do away with classes to allow the specification of prefixes of any number of bits. /48 in 2620:38:C000::/48 refers to a prefix that spans 48 bits.

We have considered using IPv6 addresses as a side channel. We could use a small part of the Ole Miss address space for this purpose. I arbitrarily chose F1E2:D3C4 to identify the session initiator beacon, hereafter referred to as the beacon. This number is appended to the network prefix for Ole Miss. Thus all beacons begin with 2620:38:C000:F1E2:D3C4. We then use the 128-80 = 48 least significant bits to communicate a session ID between the beacon and the unsandboxed beacon detector (i.e., the detector).

I created an example beacon in the fling repository in examples/ex2\_prototype\_fling\_ipv6. I ran into an issue. It appears that our own school network is not IPv6 routable. If this is the case then it seems likely that many places around the world have the same problem. If a network is not IPv6 routable then no IPv6 packets will be transmitted to be sniffed. Maybe I'm wrong?

For each of the following problems, start with the IPv6 address as represented. For each ::, pad the address with zeroes to ensure that the address is 128 bits long. Then convert each 4-digit hexadecimal number to a decimal number and concatenate each of these decimal numbers into a vector. For example,

2001:0db8:85a3::8a2e:0370:7334

becomes

2001:0db8:85a3:0000:0000:8a2e:0370:7334

which then becomes

[8193, 3512, 34211, 0, 0, 35374, 880, 29492]

✅ Problem 11. 2001:0db8:85a3::8a2e:0370:7334

8193, 3512, 34211, 0, 0, 35374, 880, 29492

✅Problem 12. fe80::123:4567:89ab:cdef

65152, 0, 0, 0, 291, 17767, 35243, 52719

✅Problem 13. fd00::1234

64768, 0, 0, 0, 0, 0, 0, 4660

✅Problem 14. ff02::1

65282, 0, 0, 0, 0, 0, 0, 1

✅Problem 15. 2401:db00:1244::abcd

9217, 56064, 4676, 0, 0, 0, 0, 43981

Side channels using IPv6 addresses

Because I could not route IPv6 addresses on the office network, I decided to try a different kind of side channel.

In examples/ex3\_prototype\_fling\_syn, beacon.html contains script that uses fetch to open a connection to flingo.zapto.org with a specified port number. I then used tcpdump from a terminal on the same computer to capture the TCP SYN packets headed to flingo.zapto.org. I made sure that there was a valid mapping from flingo.zapto.org to an IPv4 address. I even had an instance running on AWS using this IPv4 address.

It turns out that having a node on AWS is unnecessary for this example. A TCP SYN is part of the three-way handshake whenever a connection is opened. The TCP SYN is generated even if there is no node on the other end to receive the TCP SYN. The TCP SYN contains the port number of the destination node. As such one can sniff the port numbers using tcpdump even if the TCP SYNs are being sent to some non-existent node so long as the CIDR prefix corresponds to a portion of the Internet that is not local to the node running the browser displaying beacon.html. I was able to do this successfully when tcpdump is runnig on the same node as the web browser opening the connections. When I moved the tcpdump to another node or if I ran my Mac Book Pro's network interface card in monitoring mode, I captured zero TCP packets. I captured mdns packets, but no TCP packets.

✅Problem 16 Why might this fail when tcpdump is NOT running on the same node as the browser showing beacon.html? To answer this lookup the four-way handshake when a node joins a Wifi network. Look at the the Pairwise Transient Key (PTK). Look at the TK which is derived from the PTK.

Because the encryption and a lack of access to the keys from the 4 way handshake.

✅Problem 17 What part of a frame is encrypted using the TK?

The payload or the MPDU payload.

✅Problem 18 How often does a 4-way handshake occur when a wireless node (a.k.a., a station) has stable wifi connectivity to the Access Point (AP)? How might this affect the beacon detector's ability to capture the 4-way handshake?

It occurs in the initial connection and after the transfer of the keys. This makes it to where if you did not sniff during the transfer of keys then you cannot access the encrypted information that follows.

✅Problem 19 How might we decrypt traffic between a wireless node (a.k.a., a station) and an Access Point (AP)?

By getting the key.

✅Problem 20 If the unsandboxed beacon detector does not see the 4-way handshake, is there any way for the unsandboxed initiator beacon to get access to the unencrypted IP or TCP headers or the unencrypted body of the packets?

Yes there is always a way. You could brute force for the key for years, but if you don’t wanna do that then no.

Side channels using other means

If a wireless node is on a different wireless network from the sandboxed initiator beacon, or if the sandboxed initiator beacon did not have access to the 4-way handshake, we may need to consider other types of side channels.

Problem 21 What other types of side channels might we exploit? What are the fields in an 802.11 frame? What can we modify from within a security sandbox when the IP packets are encrypted?

WE can use CSI, traffic analysis, Packet Metadata,

Frame Control: Specifies the type and subtype of the frame.

Duration/ID: Duration of the frame exchange or association ID.

Address Fields: Source and destination MAC addresses.

Sequence Control: Sequence number to manage frame order and detect losses.

Frame Body: Contains the actual payload of the frame, which could include data or management information.

Frame Check Sequence (FCS): CRC or checksum for error detection.

The source and destination of an 802.11 frame are not typically encrypted for privacy.

|  |
| --- |
| I would have mentioned the following types of side channels, since they are all doable from within a browser security sandbox:  1. Timing Attacks: |
| * Measure the time taken for specific network requests to complete. |
| * Use JavaScript to introduce deliberate delays and observe the effects on the network traffic. |
| 2. Traffic Volume and Patterns: |
| * Generate varying amounts of traffic by controlling the size and frequency of HTTP requests or WebSocket messages. |
| * Use long-polling, chunked transfer encoding, or other techniques to vary the size and timing of data sent. * Have the server-side directly control the size of packets sent. |
| 3. HTTP Headers: |
| * Use custom HTTP headers to the number of bytes in the HTTP request. |
| 4. WebSockets: |
| * Use WebSockets to send messages of varying lengths and at different intervals to create detectable patterns in the traffic. |
| 5. File Sizes: |
| * Serve files of different sizes and monitor the resulting traffic patterns. |

Problem 22 Even if an entire IP packet is encrypted, we could still modulate frame length to communicate a side channel. What is the difference between frame length and packet length? Do wifi networks pad packets to make equal-sized frames? Will wifi networks transmit packets unpadded allowing us to manipulate frame length?

✅Yes you can. Frame length is just how big the frame is but packet length is the frame’s size plus the rest of the packet. No. yeah.

Problem 24 What sized packets are we most likely to see? How big is a packet containing only a TCP ACK in an IPv4 packet? How big is an entire 802.11 frame that contains an ACK? What is the minimum size of an 802.11 frame?

The sizes vary greatly but one of the most common packets are HTTPS which are 65535 bytes. A minimum of 40 bytes. 28 to 40 bytes. 28 bytes.

|  |
| --- |
| **Question 1: What sized packets are we most likely to see?** |
| **My answer:** |
| 65535 bytes for HTTPS packets is incorrect. The Path MTU (Maximum Transmission Unit) is typically much smaller. For Wi-Fi, the MTU is usually around 1500 bytes. Common packet sizes are likely in this range for packets carrying data. We will also likely see a great deal of ACK packets without data, which for IPv4 would usually be 40 bytes including only IPv4 and TCP headers. |
|  |
| **Question 2: How big is a packet containing only a TCP ACK in an IPv4 packet?** |
| **My answer:** |
| • **IPv4 header**: 20 bytes |
| • **TCP header**: 20 bytes (minimum, without options) |
| • **Total**: 40 bytes |
|  |
| This represents the typical size of a TCP ACK packet with no payload. |
|  |
| **Question 3: How big is an entire 802.11 frame that contains an ACK?** |
| **My answer:** |
| Your answer is a bit unclear, and my question was a bit ambiguous. Am I talking about a TCP ACK or an 802.11 ACK frame? |
|  |
| • **802.11 ACK frame**: a specific control frame. It is typically 14 bytes. |
|  |
| For a data frame containing an IPv4 packet containing a TCP ACK: |
|  |
| • **802.11 data frame header**: 24 bytes (or 28 bytes with QoS) |
| • **IP header**: 20 bytes |
| • **TCP header**: 20 bytes |
| • **Total**: 64 to 68 bytes (depending on the presence of QoS) |
|  |
| **Question 4: What is the minimum size of an 802.11 frame?** |
| **My answer:** |
| • **Minimum size of an 802.11 data frame header**: 24 bytes (without QoS) |
|  |
| For 802.11 control frames, the frames can be smaller, as small as 14 bytes for an ACK frame. |
|  |
| **Interpretation of Your Answer:** |
|  |
| • **65535 bytes**: Incorrect for most likely packet sizes. |
| • **A minimum of 40 bytes**: Likely referring to the size of a TCP ACK packet in an IPv4 packet, which is correct. |
| • **28 to 40 bytes**: This might refer to the size range of 802.11 frame headers, which is somewhat correct but imprecise. |
| • **28 bytes**: This could refer to the minimum size of an 802.11 data frame header. |

Problem 25 How might we modify frame length in a request sent from a sandboxed application?

We can either use packet padding techniques or adjust data transmission.

|  |
| --- |
| **My answer:** From within a web security sandbox, we could manipulate headers or the body of the HTTP request using fetch or XmlHttpRequest. For example, with fetch:  const url = 'https://example.com/api';  const headers = new Headers();  headers.append('X-Custom-Header-1', 'SomeValue');  headers.append('X-Custom-Header-2', 'AnotherValue');  fetch(url, { method: 'GET', headers: headers }) |

Problem 26 When using a TCP SYN to communicate a side channel via the port number in the TCP header, it didn't matter if a server responds to the TCP SYN. A server node need not even exist because the TCP SYN is the first packet in any TCP connection. If there is no server, the TCP SYN never reaches the server. The Internet will try to route the packet and when a router finds that there is no node it may respond with an ICMP Destination Unreachable. Firewalls may filter out the ICMP message, so if we send a TCP SYN to a non-existent node we will either get back an ICMP message or nothing. Either way the TCP SYN is still generated and still sent. If instead of using the port number as a side channel, we want to modulate frame length, do we need a server node? Why?

X No because the TCP SYN packet is generated and sent by the client independently of whether a server exists or responds.

|  |
| --- |
| **Question 1: If instead of using the port number as a side channel, we want to modulate frame length, do we need a server node?**  **My answer:**  Yes. Data can only be transmitted AFTER a successful 3-way handshake. To complete the 3-way handshake, we need server. With each TCP segment, we also need the server to respond with an ACK or else TCP will eventually stop sending waiting for ACKs. |

Problem 27 Assuming we have a web server, what if we use the web server to modulate the length of the packets returned to the sandboxed application? In this scenario the sandboxed application is no longer performing the beacon, the server is. For such cases where the beacon is coming from the server, let's use the name sandboxed initiator to refer to the component in the sandboxed application that requests a beacon from the beacon server. Since we can implement the server however we want, we could have precise control over the size of the packets it generates.

It'll allow us control the packets much easier and guarantee that we’ll be able to adjust the packet sizes.

|  |
| --- |
| Problem 27 is super ambiguous. I apologize. |

Problem 28 If we are modulating packet length either from a beacon server (i.e., modifying downloaded packets) or from a sandboxed initiator beacon (i.e., modifying uploaded packets), how might we distinguish the packets with modulated length from those that are not?

We can make custom flags, make a unique sending pattern, and we can use reserved bits,

|  |
| --- |
| Your answer: “We can make custom flags” Yes. From the server-side we could twiddle with IP headers, but these IP headers wouldn’t be visible to the unsandboxed beacon detector. Similarly “reserved bits” wouldn’t work either because the entire payload of the IP packet will likely be encrypted.  “Unique sending pattern”. I like this answer better. Yes. But what kind of patterns? |

Problem 29 If there are a number of nodes communicating on the wireless network, how might an unsandboxed beacon detector detect the presence of a beacon that is modulating packet length?

We could use machine learning maybe :D

Or just use protocol inspection and or traffic analysis systems.

|  |
| --- |
| I agree that machine learning could help, and machine learning could be used to do the traffic analysis. I am not sure what protocol inspection means in this case. Here is one approach:   1. Identify packet lengths that are uncommon. 2. Pick a set of packet lengths that can be used to identify the presence of a beacon signal. Let’s call this the “beacon set.” 3. Have the server repeatedly send packets with the given packet lengths. The chance that all such packet lengths within the beacon set are seen within a time interval spanning maybe 5-10 seconds is super unlikely.   Along with each beacon set, the server could encode a session ID using packet pairs. For example, each packet immediately after a packet in the “beacon set” has a length between 64 bytes and 64+1024 -1 = 1087. This allows each packet pair to communicate 10 bits. Across the packet pairs in a single path through the beacon set we use an error detection code (such as a checksum) so that if a single packet pair is incorrect then we wait to receive the next beacon set. If we want to communicate a 48-bit session ID, we would need at least 5 packet pairs plus an additional packet pair for a checksum. |

Problem 30 Think outside the box. Is there anything else we migth modulate other than frame length that could be detected by an unsandboxed beacon detector even when the entire IP packet is encrypted?

We can maybe figure out how to edit the Ethernet address. Then we could easily see it and use it as a “session ID”. We should name the term “session ID” to something fancy :D.

Like autonomous ID.

|  |
| --- |
| We can’t easily modify the Ethernet address from inside a security sandbox, and the Ethernet address is only a concept that applies within a local area network. As packet moves from one network to another, the frame header is removed a new one appended. As such, the server somewhere on the public Internet doesn’t have access to the frame headers that exist inside the user’s home network. |