CH 1-2: Networking Basics/Capturing App Traffic

Friday, December 21, 2018 7:41 PM

Network protocol functions:

Session state	Create new/terminate existing connections
Addressing	ID specific nodes/group of
Flow	Data xfer: Implement ways of managing data to increase throughput/reduce latency
Arrival	Guaranteeing order data sent: Protocols can reorder to ensure delivery correct
Find/correct errors	Detect corruption
Fmt/encode data	Protocols can specify ways of encoding

TCP/IP:

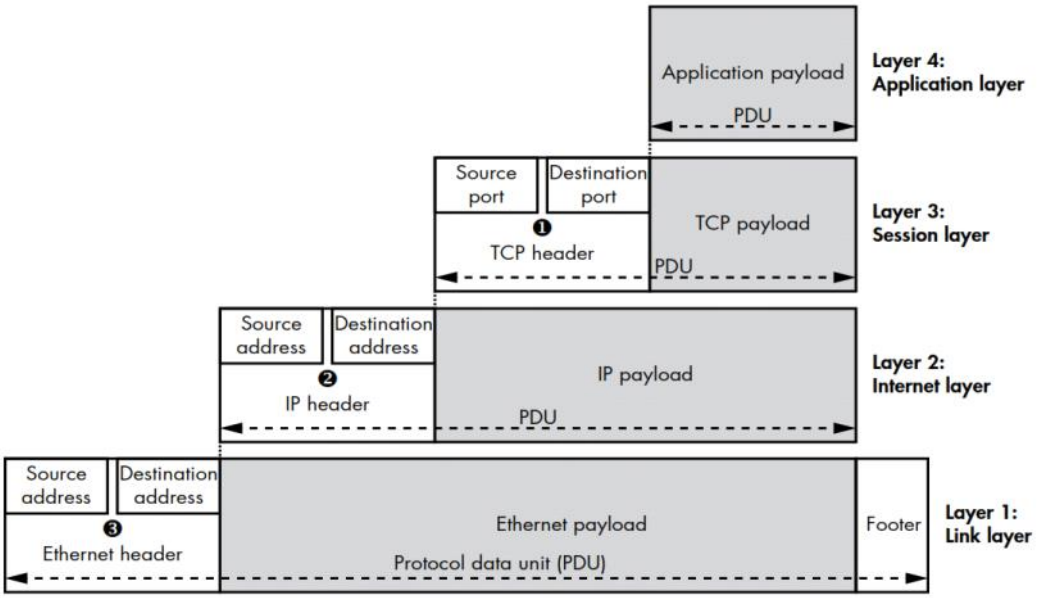
Link	Phys: Xfer info bet nodes: Ethernet/PPP
Internet	Addr network nodes: IPv4/6
Transport	Connections bet client/server: Correct order of packets/multiplexing Service multiplexing: TCP/UDP: Single node support multiple diff services ▪ Assigns diff #'s for each port
App	Network protocols: HTTP/SMTP/DNS etc..

Apps typically have following components:

Network comm	Process inc/out data: SMTP/POP3
Content parsers	Data xferred: Content must be extracted/processed: Txtual data/email/pics/vid
UI	View things: Browser

Data Encapsulation: PDU: Each layer contains payload data transmitted: Common to prefix header

- **Header**: Info req for payload data to be transmitted (addr: source/dest)
- **Footer:** Sometimes: Values needed to ensure correct transmission: Error-checking



TCP header: Source/Dest port: Multiple unique connections 0-65535: **/etc/services**

Segment: TCP payload/header | Datagram: UDP payload/header

3 layers of protocol analysis:

1. **Content:** What's being comm
2. **Encoding:** Rules: Govern how you represent content
3. **Transport:** Rules: Govern how data xferred: HTTP GET

Capturing App Traffic: 2 diff ways:

Passive	Not directly interacting w/traffic: Extracts data as it travels on wire
Active	Interferes w/traffic bet client app/server: Problems like proxies/MiTm's

Extracting traffic from local app w/out using packet-sniffer like WS

System Call Tracing: Many OS provide 2 modes of execution

1. Kernel: High priv: Code implementing core functionality
   - Services to to usr mode by exporting collection of special syscalls
2. User: Everyday processes

When app wants to connect to remote server: Syscall to kernel to open connection

- App reads/writes network data: Can monitor calls directly to passively extract data

Common Unix Sys Calls for Networking

Name	Description
socket	New socket file descriptor
connect	Connects socket to known IP/port
bind	Binds socket to local known IP/port
recv, read, recvfrom	Receive data from network via socket: ▪ Function read from file descriptor ▪ recv/recvfrom specific to socket's API
send, write, sendfrom	Sends data over network via socket

strace Utility on Linux: Can monitor sys calls from usr program w/out special perms unless app runs as priv

strace -e trace=network,read,write /path/to/app args

Monitor networking app that reads/writes a few strings/look at output from strace:

strace -e trace=network,read,write customapp

```
1. socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
2. connect(3, {sa_family=AF_INET, sin_port=htons(5555),
    sin_addr=inet_addr("192.168.10.1")}, 16) = 0
3. write(3, "Hello World!\n",13)     = 13
4. read(3, "Boo!\n", 2048)           = 5
```

1. Creates new TCP socket: Assigned handle **3**
2. **connect** syscall used to make connection to IP: Port **5555**
3. App writes **Hello World!** before reading **Boo!**

Monitoring Connections w/DTrace

- Set sys-wide probes on special trace providers: Inc syscalls
- Config by writing scripts in lang w/C-like syntax

Example:

```
traceconnect.d /* Monitor connect sys call */
1.
struct sockaddr_in {
    short          sin_family;
    unsigned short sin_port;
    in_addr_t      sin_addr;
    char           sin_zero[8];
};
2. syscall::connect:entry
3. /arg2 == sizeof(struct sockaddr_in)/ {
    4.   addr = (struct sockaddr_in*)copyin(arg1, arg2);
    5. printf("process:'%s' %s:%d", execname, inet_ntop(2, &addr->sin_addr),
        ntohs(addr->sin_port));
}
```

syscall: 3 params arg0, arg1, arg2 initialized in kernel

- **arg0:** Socket file descriptor
  - Handle: Not needed
- **arg1:** Addr of socket connecting to
  - Usr process mem addr of socket addr struct: addr to connect to
  - Can be diff sizes depending on socket type
- **arg2:** length of addr
  - Length of socket addr struct in bytes

IPv4 connections: Script defines **sockaddr_in** struct

1. In many cases: These structs can be directly copied from sys C headers

2. Syscall to monitor
3. DTrace filter to ensure only connect calls where sock addr same size as **sockaddr_in**
4. **sockaddr_in** copied from your process into local struct for DTrace to inspect
5. Process name/dest IP/port printed to console

dtrace -s traceconnect.d as root to run

- Individual connections to IP's/process name

Procmon Win: Usr-mode network functions w/out direct sys calls

- Networking stack exposed through driver
- Establishes connection: Uses **open,read,write** syscalls to config network socket for use

Vista/later: Supported event generation framework: Allows apps to monitor network activity

- Can capture state of current calling stack: Determines where in app connections made
- Can't capture data, but can add info to analysis through active captures

Advantages/Disadvantages of Passive Capture:

Advantage	Doesn't disrupt client/server app comm ▪ Won't change dest/source addr of traffic ▪ Doesn't req mods/reconfigs of apps ▪ May be only technique useful when no direct control over client/server
Disadvantage	Sniffing at very low lvl: Difficult to interpret what app received ▪ May not be possible to easily take apart protocol w/out interacting directly ▪ Doesn't always make it easy to mod traffic produced ▪ Useful: ▫ Encrypted protocols ▫ Disabling compression ▫ Changing traffic for exploitation

Active Network Traffic Capture: Differs from passive: Try to influence flow of traffic

- Device capturing usually sits bet client/server apps: Acts as bridge

Advantages: Disable encryption/compression: Easier to analyze/exploit protocol

Disadvantages: Need to reroute apps traffic through active capture sys

- Example: Change network addr of server/client to proxy? Confusion: Traffic sent to wrong place

Network Proxies: Most common way to MitM: Force app to comm through proxy

Simple Implementation: To create proxy - using built-in TCP port fwdr included w/Canape Core libs

```
// PortFmtProxy.csx
// Exposure methods like WriteLIne/WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

//Create proxy template
var template = new FixedProxyTemplate();
template.LocalPort = LOCALPORT;
template.Host = "REMOTEHOST";
template.Port = "REMOTEPORT";

// Create proxy instance/start
var service = template.Create();
service.Start();

WirteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Dump packets
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

LOCALPORT, REMOTEHOST, REMOTEPORT: Replace w/values as needed

1. Creates instance of **FixedProxyTemplate**
    - Canape Core: Template model: If req can work w/low-lvl config
    - Configs template w/desired local/remote network info
2. Creates service instance at **var service = template.Create(); service.Start();**
3. All captured packets written to console using **WritePackets(**) method

Should bind instance of fwding proxy to LOCALPORT # for localhost int only

- When new connection made to port: Proxy should establish it to REMOTEHOST w/REMOTEPORT
- Links the 2 connections together

NOTE **Binding proxy to all addr bad sec**
- Proxies written for testing protocols rarely implement sec mechs
- Unless you have control: Only bind to local loopback
- Default: LOCALHOST: Bind all ints: Set **AnyBind** property to true

Redirecting Traffic to Proxy:
- **Browser** http://localhost.localport/resource : Pushes req through port-fwding proxy
- Other apps: May have to dig into app config settings
  - Sometimes: Only dest IP allowed setting change
  - Can lead to scenario: Unsure TCP/UDP ports app may be using w/addr

Example: RPC protocols: CORBA (Common Object Request Broker Arch)
- Usually makes initial network connection to broker
- Broker acts as dir of avail services
- 2nd connection made to req service over instance specific port

In case above: Use as many network-connected features of app as possible: Monitor w/passive capture
If app doesn't allow dest IP change: Custom DNS server: Respond to reqs w/IP of proxy
- Use hosts file: During hostname resolving: OS 1st refers to hosts file to see if local entries exist
- Makes DNS req if one not found
- **/etc/hosts | C:\Windows\System32\Drivers\etc\hosts**

Port-Fwding Proxy

Advantages **Simplicity**
- Wait for open connection to original dest: Pass traffic bet 2
- No protocol associated w/proxy to deal w/
- No special support req by app you're capturing traffic from
- Primary way of proxying UDP BC/connectionless

Disadvantages Only fwd traffic from listening connection to single dest
- Multiple instances of proxy: App uses multiple protocols/diff ports
  - Can mitigate if app supports specifying dest addr/port using DNAT

DNAT: Destination Network Addr Translation
- Used to redirect specific connections to unique fwding proxies
- Protocol might use dest addr for own purposes
  - Example: Host header in HTTP can be used for Virtual Host decisions
  - Might make port-fwding protocol work diff/not at all
  - Reverse HTTP Proxy workaround

SOCKS Proxy: Fwds TCP connections to desired network loc
- All new connections start w/simple handshake protocol: Informs proxy of ult. dest
- Can support listening connections (FTP: Needs to open new local ports for server to send data)

3 common variants in use: SOCKS 4, 4a and 5

4	**Most commonly supported version of protocol**: Only IPv4 ▪ Dest addr must be specified as 32-bit IP
4a	Allows connections by hostname (useful if no DNS server)
5	Hostname support: IPv6: UDP fwding: Improved auth: **RFC 1928**

```
// SocksProxy.csx - SOCKS proxy
// Expose methods like WriteLine/WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Create SOCKS proxy template
var template = new SocksProxyTemplate();
template.LocalPort = LOCALPORT;

// Create proxy instance/start
var service = template.Create();
service.Start();

WriteLine("Created{0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Dump packets
var packets = service.Packets;
WriteLine("Capture{0} packets:",
    packets.Count);
WritePackets(packets);
```

Redirecting Traffic to Proxy: Check through app for SOCKS proxy info first (Mozilla/Burp)

- Sometimes SOCKS support not obvious: Java app
- Runtime accepts CLI params that enable SOCKS support for any outbound connection

Example:

```
// SocketClient.java - Simple Java TCP socket client
import java.io.PrintWriter;
import java.netSocket;

public class SocketClient {
    public static void main(Stringp[]args) {
        try {
            Socket s = new Socket("192.168.10.1", 5555);
            PrintWriter out = new PrintWriter(s.getOutputStream(), true);
            out.println("Hello World!");
            s.close();
            } catch(Exception e) {
        }
    }
}
```

- If CLI: Pass 2 special sys properties **socksProxyHost/socksProxyPort:** Can specify SOCKS for any TCP connection

java -DsocksProxyHost=localhost -DsocksProxyPort=1080 SocketClient

Another place to look: OS default proxy: **macOS: System Preferences > Network > Advanced > Proxies**

If app won't support SOCKS: Certain tools will add function:

- Linux: **Dante** https://www.inet.no/dante/
- Win/macOS: **Proxifier:** https://www.proxifier.com
  - Inject into app to add SOCKS support/mod op of socket functions

Advantages	Should capture all TCP connections app makes (maybe UDP if SOCKS 5) ▪ Preserves dest of connection from POV of client app ▫ If client app sends in-band data that refers to endpoint: What server expects
Disadvantages	Inconsistency bet apps/platforms Win proxy supports ONLY ver 4: Will resolve only local hostnames: No IPv6: No robust auth mech

HTTP Proxies: Can be used as transport mech for non-web protocols

- **RMI: Remote Method Invocation (Java)**
- **RTMP: Real Time Messaging Protocol**

It can tunnel through most restrictive FW's: **2 main types of HTTP proxy:**

1. **Forwarding**
2. **Reverse**

Fwding HTTP Proxy: RFC 1945 ver 1.0 || RFC 2616 ver 1.1: Simple mech for proxying HTTP requests

Example	**HTTP 1.1:** 1st full line of a req (request line) has fmt: **GET /image.jpg HTTP/1.1** ▪ Specifies what to do in req using **GET,POST,HEAD**

- Doesn't change from normal HTTP connection
- Absolute path indicates resource method will act on

Path can also be absolute URI: Uniform Request Identifier

- Specifying absolute URI: Proxy server can establish new connection to dest/fwd traffic on/return back to client
- Can manip traffic to add auth/hide ver 1.0 servers from 1.1 clients: Can add xfer compression
- **Cost:** Proxy server must be able to process HTTP traffic: Complex

Example **GET http://www.domain.com/image.jpg HTTP/1.1**

- HTTPS transports HTTP over encrypted TLS: Could break out encrypted traffic: Normal env: Unlikely client would trust cert
- TLS intentionally designed to make impossible to MitM other ways

RFC: 2817: Solutions

1. Ability to upgrade HTTP connection to encryption: Specifies **CONNECT HTTP method**
   - Transparent tunneled connections over HTTP proxies

Example **Browser establish proxy connection to HTTPS: CONNECT www.domain.com:443 HTTP/1.1**
Successful response: HTTP/1.1 200 Connection Established

- TCP connection transparent: Browser able to establish negotiated TLS w/out proxy

```
// HTTPProxy.csx - Simple HTTP Proxy
// Expose methods like WriteLine/WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Create proxy template
var template = new HttpProxyTemplate();
template.LocalPort = LOCALPORT;

// Create proxy instance / start
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Dump Packets
var packets = service.Packets;
WriteLine("Captured{0}, packets:", packets.Count);
WritePackets(packets);
```

Redirecting Traffic to Proxy: 1st port app: Rare app uses HTTP to not have a proxy config

- **If app has no settings for HTTP proxy support: OS config:** Same place as SOCKS proxy config

Windows: Control Panel > Internet Options > Connections > LAN Settings
Linux: curl, wget, apt: Support setting HTTP proxy config through env vars

- Can set env var **http_proxy** to URL for HTTP proxy for use: http://localhost:3128 - app will use
- Sec traffic: **https_proxy** || **socks4://**

Advantages

App uses HTTP exclusively:

- All it needs to add proxy support? Change absolute path in Request Line to an absolute URI
- Sends data to listening proxy server: Few apps that use HTTP for transport don't support proxying

Disadvantages

Idiosyncrasies: Processing/sec issues:

- More diff to retrofit HTTP proxy support to existing app through external techniques
- Have to convert connections w/**CONNECT** method

HTTP 1.1 Common for proxies to disconnect clients after 1 req/downgrade to 1.0

Reverse HTTP Proxy: Envs where internal client connecting to outside network: May want to proxy inbound connections
Instead of req dest host to be specified in req line:

- **Can abuse HTTP 1.1: Specifies original hostname used in URI of req**
- **Host header info:** Can infer original destination of req making proxy connection to server

Example:
GET /image.jpg HTTP/1.1
User-Agent: Super Funky HTTP Client v1.0

Host: www.domain.com
Accept: */*
Typical Host header where HTTP req was to URL: Reverse proxy can take this info/reuse it to construct original dest

- No req for parsing HTTP headers: More diff to use for HTTPS traffic protected by TLS
- Most TLS implementations take wildcard certs where subject is in form of *.domain.com/similar

```
// ReverseHttpProxy.csx - Simple reverse HTTP proxy
// Expose methods like WriteLine/WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Create proxy template
var template = new HttpReverseProxyTemplate();
template.LocalPort = LOCALPORT;

// Create proxy instance and start
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Dump packets
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WriteLine(packets);
```

Similar to TCP port-fwding: Can't change destination hostname: Would change host header: Causes proxy loop
Proxy loop: When a proxy repeatedly connects to itself: Recursive: Runs out of avail resources

- App testing running on device: Doesn't allow changes to host file? Custom DNS server: Tools: dnsspoof: Can use w/Canape

```
//DnsServer.csx - Simple DNS Server
// Expose console methods like WriteLine at global level
using static System.Console;

// Create the DNS server template
var template = new DnsServerTemplate();

// Setup response address
template.ResponseAddress = "IPV4ADDRESS";
template.ResponseAddress6 = "IPV6ADDRESS";
template.ReverseDns = "REVERSEDNS";

// Create DNS server instance/start
var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

If you config DNS server for app to point to spoofing DNS server: App should send traffic through

Advantages	Doesn't req client app to support typical fwding proxy config: Useful if isn't under direct control/fixed config
Disadvantages	Same as fwding proxy: Must be able to parse HTTP req/handle idiosyncrasies

CH 3: Network Protocol Structures

Wednesday, January 30, 2019 8:44 PM

Binary Protocol Structures: Smallest unit of data single binary digit: **octet:** 8-bit units/bytes: unit of network protocols

Bit fmt:

0 (Bit 7/MSB)	1	0	0	0	0	0	**1 (Bit 0/LSB)**

= 0x41/65: Octet: 0x41

MSB: Most Significant Bit || LSB: Least Significant Bit

Numeric Data: Data values represented: Core of binary protocol: Ints/dec values: Length of data/ID tags

Unsigned ints: Based on position: Values added together to represent the int

Bit	Dec	Hex
0	1	0x01
1	2	0x02
2	4	0x04
3	8	0x08
4	16	0x10
5	32	0x20
6	64	0x40
7	128	0x80

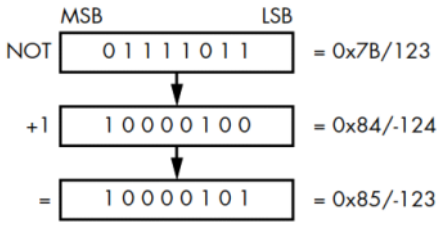
Signed ints: Not all ints positive: Neg ints req: Only signed ints can hold neg values

- CPU can only work w/same set of bits
- Reqs way of interpreting unsigned int value as signed: Two's complement

Two's complement: Way in which signed int represented w/in native int value in CPU

- Conversion bet unsigned/signed values in 2's complement done by taking bitwise NOT
  - 0 bit converted to 1 vice versa: Then adding 1

Example:



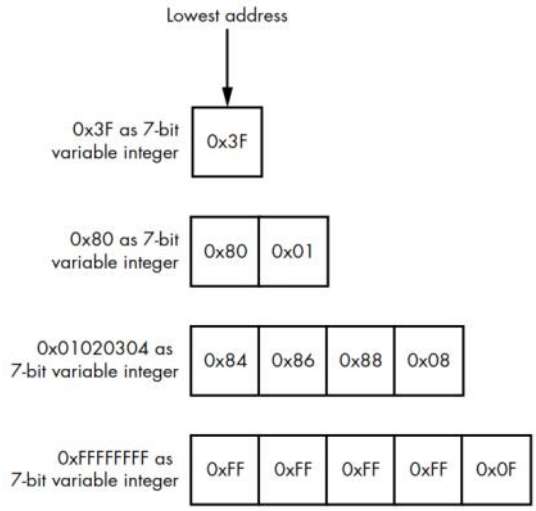
2's complement representation sec consequence:

- 8-bit signed int range: -128 to 127: Magnitude of min larger than max
- **If min value negated result is itself -(-128) is -128**
- Calcs incorrect parsed fmt leading to vuln

Note: Above img is confusing: NOT is auto flipped then +1 added for clarity

Var-Length Ints:

Length fields: When sending blocks of data bet 0-127 bytes in size: **Could use a 7-bit var int representation**



Parse more than 5 octets? Resulting int from parsing op will depend on parsing program

- Some programs will drop any bits beyond given range
- Other envs will generate int overflow: Possible BoF

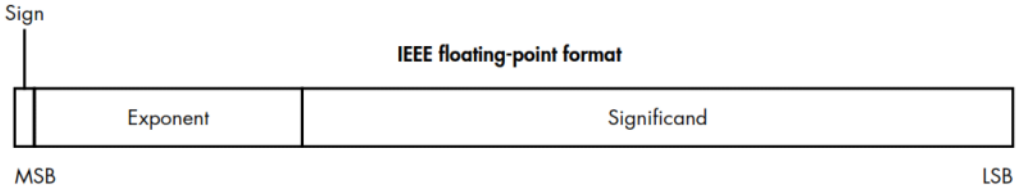
Floating-Point Data: Sometimes ints not enough to represent range of dec values needed for a protocol

- Could run against limited range of 32-/64-bit fixed-point value

IEEE Standard for Floating-Point Arithmetic [IEEE 754]

Standard specifies num of diff bin/dec fmts for floating-point values: Likely to encounter 2

1. **32-bit**
2. **64-bit: Double-precision**
    - Each specifies position/bit size of significand/exponent
    - **Sign bit also specified**: Indicates whether value positive/negative



Bit size	Exponent bits	Significant bits	Value range
32	8	23	+/-3.402823 x 10^38
64	11	52	+/-1.79769313486232 x 10^308

Booleans: Protocols: How to represent **true[1]/false[0]**

Bit Flags: 1 way to represent specific Booleans in protocol

Example: TCP: Bit flags to determine current state of connection

- Client: Sends packet SYN: Indicates connection should sync timers
- Server: Respond ACK to indicate client req as SYN: Establishes sync
- Handshake: Single enumerated values: Dual state impossible w/out SYN/ACK state

Binary Endian: How computers store data in mem:

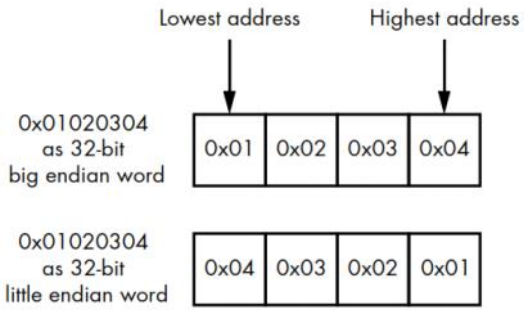
- Octets transmitted sequentially: Possible to send most significant octet of value as 1st part of transmission
- Least significant octet: Also possible to send as value of 1st part of transmission

Order in which octets sent determines endianness of data:

- Failure to handle endian fmt: Bugs in parsing protocols

Main Endian fmts:

1. **Big endian:** Stores **most** significant byte at lowest addr
2. **Little endian:** Stores **least** significant byte in lowest addr



Network/Host order: Endianness of value: Internet RFC's typically use big endian as preferred type for network protocols

- Big endian referred to as network order
- Computer could be big/little
- Proc arch: x86: Little endian: SPARC: Big endian

Text/Human-Readable Data: English chars: Encoded using ASCII

- Original ASCII standard defined 7-bit char set from 0 to 0x7F: Most to represent English

Control character | Printable character

Upper 4 bits \ Lower 4 bits	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	l	}	~	DEL

Originally for txt terms: Control chars used to send msgs to terminal to move printing head to sync serial comms bet computer/term

ASCII char set 2 types of chars:

Control	
Printable	**Ones seen:** Familiar symbols/alphanumeric chars: Not useful to represent intl chars ▪ Can't represent fraction of possible chars in all world languages w/7-bit num **Strategies to counter limit:** **1. Code pages** **2. Multibyte character sets** **3. Unicode** Protocols: Req 1-of-3 ways to represent txt: Offers option app can select

More on counter limitations:

Code pages	**Code pages/char encodings:** Which chars mapped to which values codified in specifications **Simplest:** Extend ASCII char set: Recognized if all data stored in octets: ▪ **128 unused values 128-255:** Can be repurposed for storing extra chars ▪ **256 values:** Not enough to store all chars in all lang: Diff ways to use unused range
Multibyte char sets	**Multibyte char sets:** Allow use 2/more octets in seq to encode desired char Languages: **CJK:** AKA: Chinese/Japanese/Korean: ▪ Uses multibyte char sets combined w/ASCII to encode languages **Common encodings:** ▪ **Shift-JIS:** Japanese ▪ **GB2312:** Chinese
Unicode	**Standard: 1991:** Aim: Represent all languages w/in Unified Char Set: Multibyte char set ▪ Tries to encode all written languages: Archaic/constructed **Unicode defines 2 related concepts**

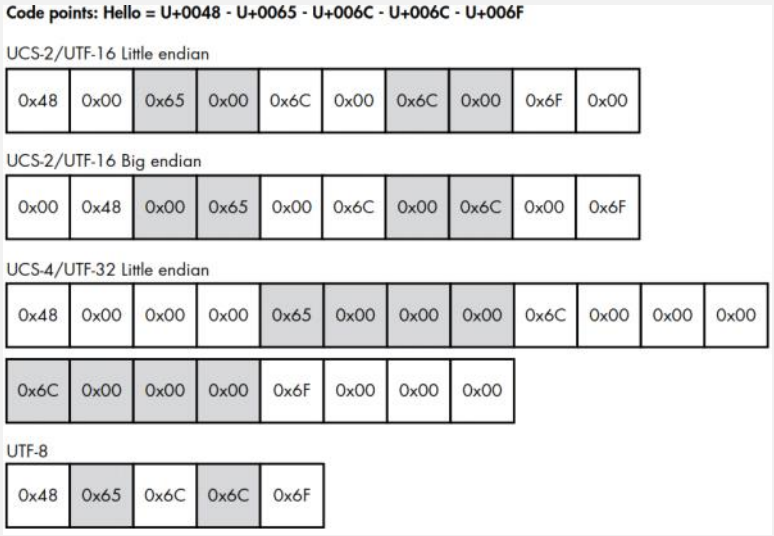
1. **Character mapping**
    - Mappings bet num value/char: Rules/reg on how chars used/combined
2. **Character encoding**
    - Define way num values encoded in underlying file/network protocol

Code point: Each char in Unicode assigned code point: Represents unique char

- Code points commonly written in fmt **U+ABCD**
- **ABCD: Code point's hex value**
- **1st 128 code points:** What's specified in ASCII
- **2nd 128 code points:** From **ISO/IEC 8859-1**

UCS: Universal Char Set || UTF: Unicode Transformation Fmt

Encodings: Resulting value encoded in one of said schemes



3 Common Unicode encodings:

UCS-2/UTF-16 Native of MS Win/Java/.NET VM's when running code
- Code points: Seq of 16-bit ints
- Little/big endian variants

UCS-4/UTF-32 UNIX apps: Default wide-char fmt many C/C++ compilers
- Code points: Seq of 32-bit ints
- Diff endian variants

UTF-8 Most common UNIX: Default input/output fmt for platforms like XML
- Int size using simple var-length value: No fixed int size

Bits of code point	First code point (U+)	Last code point (U+)	Byte 1	Byte 2	Byte 3	Byte 4
0–7	0000	007F	0xxxxxxx			
8–11	0080	07FF	110xxxxx	10xxxxxx		
12–16	0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx	
17–21	10000	1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx
22–26	200000	3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx
26–31	4000000	7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx

Incorrect/naïve char encoding: Source of subtle sec issues:
- Range: Bypassing filtering (req resource path)/BoF

Variable Binary Length Data:
- Dev knows exactly what data to be transmitted? Can ensure all values w/in protocol fixed length

Protocols use a few strategies to produce variable-length data values
1. **Terminated data**
2. **Length-prefixed data**
3. **Implicit-length data**
4. **Padded data**

Terminated

Variable-length int value terminated when octet's MSB 0

- Can extend concept of terminating values further to elements like strings/data arrays

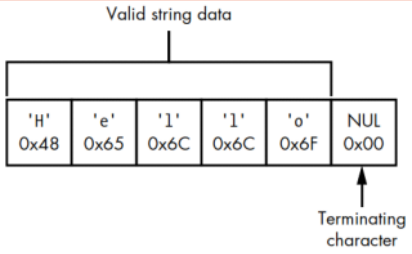
Terminated data value: Symbol defined: Tells parser end of data reached

- Unlikely present in typical data: Ensures value isn't terminated prematurely
- String data: Terminating value can be NUL value/1 of other control chars in ASCII set
- If term symbol occurs during normal data xfer: Need escape symbols

With strings:

- Common to see terminating char prefixed with \ or repeated 2x to prevent ID as term symbol
- Useful when protocol doesn't know ahead of time how long value is

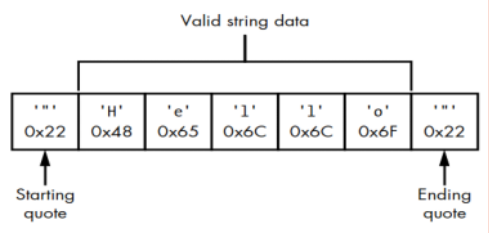
Example: Generated dynamically



Bounded data often terminated by symbol that matches 1st char in var-length sequence

Example: String data w/quoted string in bet **" "**

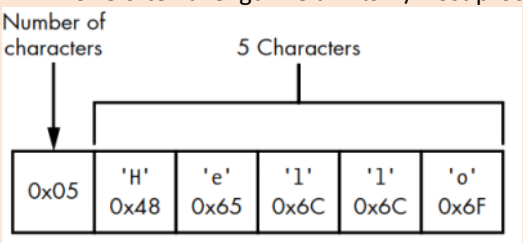
- Initial " " tells parser to look for matching char to end data



Length-Prefixed

If data value known: Possible to insert length into protocol directly

- Parser can read value/appropriate # of units (chars/octets) to extract original value: Common
- Actual length prefix/size not impt: Representative of data type transmitted
- Most protocols don't need to specify full range of 32-bit int
  - Size often a length field: Fits w/most processor arch/platforms



Implicit-Length

Sometime length implicit in values around it

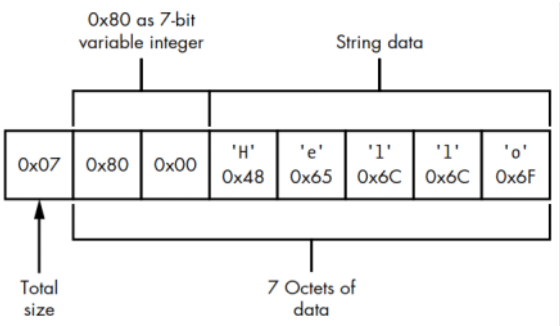
Example: TCP

- Protocol sending data back to client using connection-oriented protocol
- Instead of specifying data size: Server could close TCP connection
  - Implicitly signifies end of data: How data returned in HTTP version 1.0 response

Example II: Higher-lvl protocol/struct that already specified length of set of values

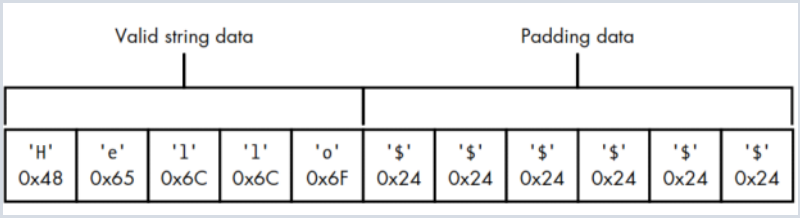
- Parser might extract higher-lvl struct 1st then read the values contained w/in
- Protocol could use struct w/finite length to implicitly calc length similar to closing connection

7-bit var int/str contained w/in single block:



Padded Data **Used when max upper bound on length of value like 32-octet limit**

- Instead of prefixing value w/length/explicit terminating value
- Protocol could send entire fixed-length str but terminate value by padding unused data w/known value



Dates/Times: Impt for protocols: Metadata: File mod timestamps: Determine expiration of auth credentials

- Failure: Serious sec issues: Depends on usage reqs platform/protocol space reqs

POSIX/UNIX Time **Stored as 32-bit signed int:** Represents num of sec elapsed since UNIX epoch **00:00:00 (UTC), 1 January 1980**

- **Value limited to 03:14:07 (UTC), 19 January 2038**
  - Representation will overflow
  - Some OS's use 64-bit representation to address issue

Windows FILETIME **MS filesystem timestamps:** Only fmt on Win w/simple bin representation

- In a few protocols: **Stored as 64-bit unsigned int**
- One unit of int: 100 ns interval

Epoch: **00:00:00 (UTC), 1 January 1601**

- Larger range than POSIX/UNIX

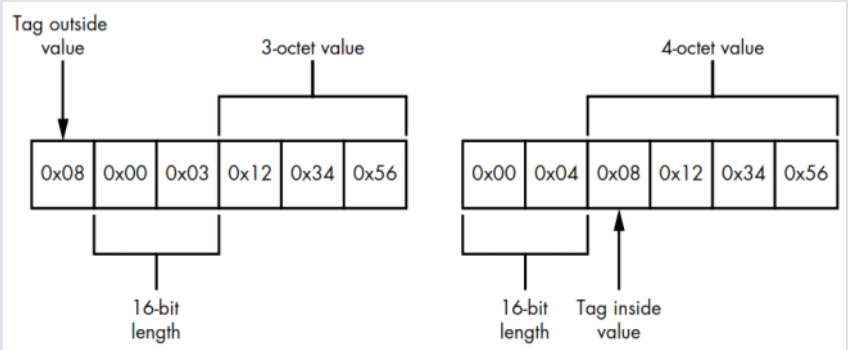
TLV: Tag, Length Value Pattern: Protocol can send diff types of structures must have way to represent bounds of struct/type

Tag value **Type of data being sent by protocol: Commonly num:** Can be anything that provides data structs w/unique pattern

- Can be used to determine how to further process data

Example: 2 types of Tags: 1 auth credentials to app: Other msg transmitted to parser

- Allows us to extend protocol w/out breaking apps not updated to support it
- Protocol parser can ignore structs that it doesn't understand



Length Variable-length value

Value Pattern Variable-length value

Multiplexing/Fragmentation: Multiple tasks happening at once

Multiplexing	**Allows multiple connections to share same underlying network connection**: ▪ Multiple types of traffic by fragmenting large transmissions to smaller chunks ▪ Combines chunks into single connection **Protocol analysis?** Demultiplex chan to get original data out Some protocols restrict type of data transmitted: How large each packet can be ▪ IP: Max traffic frames: 1500 octets: Packets 65,535
Fragmentation	Mech that allows network stack to convert large packets into smaller fragments ▪ OS knows entire packet can't be handled by next layer

Structured Bin Fmts:

ASN.1	**Abstract Syntax Notation 1:** ▪ Basis for protocols like SNMP: Simple Network Management Protocol ▪ Encoding mechanism for cryptographic values: X.509 certificates ▪ Standard: ISO/IEC/ITU: X.680 series **Defines abstract syntax to represent structured data** ▪ Data represented depending on encoding rules ▪ **DER: Distinguished Encoding Rules** ▪ Designed to represent ASN.1 structures that can't be misinterpreted ▪ Property for cryptographic protocols ▪ Representation of TLV protocol

Text Protocol Structures: Good choice when purpose to xfer txt: Mail/Msg/News

- Must have structures similar to bin protocols

Common text protocol structures:

Numeric Data

Integers	Simple representation: Size limitations no concern: Num larger than bin word/can add digits ▪ Hope protocol parser can handle the extra digits or sec issues: Make signed num: ▪ Add - char to front of num ▪ Add + char for positive num
Dec Num	Defined using human-readable forms: Bin representations [floating points] can't represent all dec ▪ Can make some values diff to represent in txt fmt: Can cause sec issues
Txt Booleans	**True/False**: Some may require words be capitalized exactly to be valid
Dates/Times	Not everyone can agree on standard fmt: Many competing representations: Issue w/mail clients
Var-Length	When txt field separated out of original protocol: Token ▪ Some protocols specify fixed length for tokens: More common to req type of var-length data
Delimited txt	Separating tokens/field w/delimiting chars very common: Any char can be used as delimiter ▪ Whitespace usually encountered: Doesn't have to be ▪ **FIX: Financial Info Exchange** protocol delimits tokens using **ASCII SOH:** ▪ **Start of Header** char w/value 1
Terminated txt	If separate individual tokens: Must also have way to define End of Command condition ▪ If protocol broken into separate lines: Must be terminated in some way ▪ HTTP/IRC: Line terminated protocols ▪ Typically delimit entire structs such as end of a cmd **OS dev: Usually define EOL char as:** ▪ **LF: Line Feed: ASCII: Value 10** ▪ **CR: Carriage Return: Value 13** ▪ **Combo CR LF: EOL: End of Line combo**

Structured Txt Fmts:

MIME	**Multipurpose Internet Mail Extensions:** Dev for multipart email msgs: HTTP: RFC's 2045/46/47 ▪ Separates body parts by defining common separator line prefixed w/2 -- ▪ Msg terminated by following separator w/same 2 -- ▪ Common uses: Content-Type values: MIME types

MIME type: Widely used w/HTTP content in OS to map app to particular content type
Each type consists of form of data: Txt/app

JSON

JavaScript Object Notation: Simple representation for struct based on object fmt
- Originally used to xfer data bet web page/backend service such as AJAX
- **AJAX: Asynchronous JavaScript/XML**
- Commonly used for web service data xfer/all manner of other protocols

JSON fmt: JSON object enclosed using {}
- W/in braces 0/more member entries
- Each consists of key/value

Example:

```
{
    "index"  : 0,
    "str"  : "Hello World!",
    "arr" : [ "A", "B" ]
}
```

Also designed for JS processing: Can be parsed using "eval" function
- Sec risk: Possible to insert arbitrary script code during object creation
- Lead to XSS

XML

Extensible Markup Language: Describing struct doc fmt
- Dev by W3C: Derived from **SGML: Standard Generalized Markup Lang**
- Similarities to HTML: Aims to be stricter in def to simplify parsers/create fewer sec issues

Consists of elements/attributes/txt

Elements: Main structural values
- Have name/can contain child elements/txt content
- Only 1 root element allowed in single doc

Attributes: Addl name-value pairs: Can be assigned to element
- Take form of **name="Value"**
- Txt is child of an element/value component of an attribute

Example:

```
<value index="0">   <str>Hello World!</str>
  <arr><value>A</value><value>B</value></arr>
</value>
```

All XML data txt: No type info provided: Parser must know what values represent
- Used in many ways: RSS: Rich Site Summary
- XMPP

Encoding Bin Data: Early comms: 8-bit bytes not norm: Most comm txt based: 7 bits per byte req by ASCII
- Allowed other bits to provide control for serial link protocols for perf
- SMTP/NNTP: Network News Transfer Protocol: Assume 7-but comm chans

7-but limitations: Problems w/pics/non-English char set
- Dev devised ways to encode bin data as txt
- Still has advantages: Ex. Sending bin data in structured txt fmt: JSON/XML: Delimiters properly escaped
- Can choose encoding fmt like Base64 to send bin data

Hex Encoding

Each octet split into 2 4-bit values converted to 2 txt chars denoting hex representation
- Not space efficient: Bin data auto becomes 100% larger
- Advantage: Encoding/decoding ops fast/simple

HTTP: Similar encoding for URL's/txt protocols

Percent encoding: Only nonprintable data converted to hex
- Values signified by prefixing w/% char

Base64

Counters inefficiencies w/Hex encoding: Dev as part of MIME spec:
- 64: Num of chars used to encode data
- Input bin separated into individual 6-bit values: 0-63
- Used to look up corresponding char in encoding table

Problems: 8-bits divided by 6: 2 bits remain

Counter?
- Input taken in units of 3 octets: Dividing 24 bits by 7 bits = 4 value
- Encodes 3 bytes into 4: Increase of 33%
- Better than hex

What if 1/2 octets to encode? Placeholder char =

- If no valid bits avail: Encoder will encode value as placeholder

CH 4: Advanced App Traffic Capture

Wednesday, January 30, 2019 8:45 PM

Traceroute

Windows	**tracert**
***nix**	**traceroute**

Max

Windows	**-h NUM**
***nix**	**-m NUM**

Routing tables:

Windows	**route print**
***nix**	**netstat -r**

Enabling Routing: 0 to disable

Windows	**reg add HKLM\System\CurrentControlSet\Services\Tcpip\Parameters ^ /v IPEnableRouter /t REG_DWORD /d 1**
***nix**	**sysctl net.ipv4.conf.all.forwarding=1 | sysctl net.ipv6.conf.all.forwarding=1**
MacOS	**sysctl -w net.inet.ip.forwarding=1**

NAT: 2 types common today:

1. **SNAT: Source Network Address Translation**
2. **DNAT: Destination Network Address Translation**
   - Diff bet 2: Which address is modified during NAT processing of traffic

Enabling SNAT:

- When you want rtr to hide multiple machines behind single IP
- Source IP addr in packets rewritten to match addr made by SNAT

Config SNAT on Linux: Make sure to: Enable IP routing: Find name of outbound net int w/ifconfig [eth0]

Flush existing NAT rules	**iptables -t nat -F**
Outbound int has fixed addr	**iptables -t nat -A POSTROUTING -o INTNAME -j SNAT --to INTIP**
IP addr config dynamically	**iptables -t nat -A POSTROUTING -o INTNAME -j MASQUERADE**

Enabling DNAT: Useful if redirecting traffic to proxy/service to terminate before fwding traffic

- Rewrites dest IP/port

Flush existing NAT rules

run as root	**iptables -t nat -A PREROUTING -d ORIGIP -j DNAT --to-destination NEWIP**

Apply rule only to specific TCP/UDP change:

iptables -t nat - A PREROUTING -p PROTO -d ORIGIP --dport ORIGPORT -j DNAT \ --to-destination NEWIP:NEWPORT

DHCP spoofing: Ettercap: GUI mode: **ettercap -G**

Sniff > Unified Sniffing | Mitm > DHCP spoofing | Start > Start sniffing

ARP Poisoning: Ettercap > Unified Sniffing > Hosts > Scan for Hosts > Hosts Host List

- **Add to Target 1: Select host to poison > Add to Target 2**
- **Mitm > ARP poisoning > OK**

Utilizing WS: Statistics > Conversations after capture > Follow Stream [TCP]

ID Packet Structure w/Hex Dump:

- WS has a Hex Dump option when Following TCP Streams from drop down menu

- 1: Byte offset into the stream for a direction:
  - Byte at 0: 1st byte sent in that direction
  - Byte 4: is the 5th
- 2: Shows bytes as hex dump
- 3. ASCII representation

Viewing Individual Packets:

- Each block is a single TCP packet/segment: Only about 4 bytes of data

TCP: Stream-based protocol:

- No real boundaries bet consecutive blocks of data when reading/writing to sockets
- Sends individual packets consisting of TCP header containing info
- **Edit > Find Packet**

Determining Protocol Structure: Look only at 1 direction of network comm

Binary Conversion w/Python Script:

- Can use Python built-in struct lib to do binary conversions
- Should fail if something isn't right: Ex. Not being able to read all data expected from file

```
from struct import unpack
import sys
import os

# Read fixed number of bytes
def read_bytes(f,1):
    bytes = f.read(1)
    if len(bytes) != 1:
        raise Exception("Not enough bytes in stream")
        return bytes

# Unpack 4-byte network bye order int
def read_int(f):
    return unpack("!i", read_bytes(f,4))[0]

# Read single byte
def read_bte(f):
    return ord(read_bytes(f,1))

    filename = sys.argv[1]
    file_size = os.path.getsize(filename)

    f = open(filename, "rb")
    print("Magic: %s" % read_bytes(f,4))

# Keep reading until EOF
while f.tell() < file_size:
    length = read_int(f)
    unk1 = read_int(f)
    unk2 = read_byte(f)
    data = read_bytes(f, length -1)
    print("Len: %d, Unk1: %d, Unk2: %d, Data: %s"
        % (length, unk1, unk2, data))
```

1. **read_bytes()**: Reads fixed # of bytes from file specified as param
   - If not enough in file: Exception thrown
2. **read_int()**: Reads 4-byte int from file in network byte order
   - Most significant byte of int is 1st in file/defines a func to read single byte
3. Opens file passed on cli/1st 4-byte value
4. **Loop:** Data to read: Length, 2 unknown values, data, prints values to console

python3 read_protocol.py bytes_inbound.bin

Calculating the Checksum

- If we assume that the unknown value is a simple checksum
- Can sum all bytes in the ex. outbound/inbound packets

2 easy ways to determine whether guessed correctly:

1. Send simple incrementing msgs from a client (A/B/C/etc): Capture/analyze
   - **If checksum simple addition:** Value should increment by 1 for each msg
2. Add function to calc checksum to see whether it matches bet capture on network/value

```
35  # Checksum function
36  def calc_chksum(unk2, data):
37      chksum = unk2
38      for i in range(len(data)):
39          chksum += ord(data[i:i_1])
40      return chksum
```

Dev WS Dissectors in Lua:

- Easy to analyze a protocol like HTTP w/WS bc SW can extract all necessary info
- Custom protocols more challenging:
  - Manually extract all relevant info from byte representation of network traffic
  - Can use WS plug-in Protocol Dissectors to add addl analysis
  - Modern versions support Lua scripting
  - Will also work w/tshark cli tool

Load Lua files: Put scripts in **%APPDATA%\Wireshark\plugins** dir

Linux/macOS: **~/.config/wireshark/plugins** dir

- Can also load Lua script by specifying on cli: **wireshark -X lua_script:</path.lua>**

Creating the Dissector:

```
1   -- Declare chat protocol for dissection
2   chat_proto = Proto("chat", "SuperFunkyChat Protocol")
3
4   -- Specify protocol fields
5   chat_proto.fields.chksum = ProtoField.uint32("chat.chksum", "Checksum",
6       base.HEX)
7   chat_proto.fields.command = ProtoField.unint8("chat.command", "Command")
8   chat_proto.fields.data = ProtoField.bytes("chat.data", "Data")
9
0   -- Dissector function
1   -- buffer: UDP packet data as a "Testy Virtual Buffer"
2   -- pinfo: Packet info
3   -- tree: Root of the UI tree
4   function chat_proto.dissector(buffer, pinfo, tree)
5       -- Set name in the protocol column in the UI
6       pinfo.cols.protocol = "CHAT"
7
8   -- Create sub tree which represents entire buffer
9   local subtree = tree:add(chat_proto, buffer(),
0       "SuperFunkyChat Protocol Data")
1   subtree:add(chat_proto.fields.chksum, buffer(0,4))
2   subtree:add(chat_proto.fields.command, buffer(4,1))
3   subtree:add(chat_proto.fields.data, buffer(5))
4   end
5
6   -- Get UDP dissector table/add for port 12345
7   udp_table = DissectorTable.get("udp.port")
8   udp_table:add(12345, chat_proto)
```

1. Creates new instance of Proto class: Represents instance of a WS protocol/assigns name **chat_proto**
   - Although you can build dissected tree manually: Chosen to define specific fields for protocol
2. Fields will be added to display filter engine: You'll be able to set display filter of **chat.command == 0**
   - WS won't only show packets w/cmd 0
   - Useful for analysis: You can filter down to specific packets easily/analyze separately

3. Script creates dissector() function on instance of Proto class: Will be called to dissect packet
    - **Function has 3 params:**
        - Buffer containing packet data is an instance of something WS calls TVB: Testy Virtual Buffer
        - Packet info instance represents display info for dissection
        - Root tree object for UI: Can attach subnodes to tree to generate display of packet data
4. **Set the name of protocol in UI column**
5. **Build a tree of protocol elements dissecting**
    - UDP doesn't have explicit field length: Don't need to bother
    - Only need to extract checksum field
    - Add to subtree using protocol fields
    - Use the buffer param to create a range: Takes a start index into buffer/optional length
    - No length? Rest of buffer used
    - Register protocol dissector w/WS's UDP dissector table
6. Get UDP table/add **chat_proto** object to table w/port 12345

CH 6 App RE

Wednesday, January 30, 2019 8:45 PM

2 main kinds of reverse engineering:

1. **Static**
2. **Dynamic**

Static Process of disassembling a compiled executable into native machine code
- Using that code to understand how the executable works

Dynamic Executing an application/using tools like debuggers/function monitors
- Inspect the application's runtime operation

Compilers, Interpreters and Assemblers:

- The way a program executes determines how it's reverse engineered

Interpreted Languages: Ex. Python/Ruby/Scripting langs

- Commonly run from short scripts written as text files
- Dynamic/speed up dev time
- Interpreters execute programs more slowly than code that has been converted to machine code

Compiled Languages: Use a compiler to parse source code/generate machine code

- Typically generating intermediate lang first
- For native code generation: Assembly language specific to CPU

Static vs. Dynamic Linking

Linking: Process that uses a linker program after compilation

- Takes app-specific machine code generated by compiler along w/external libs/embeds everything into exe

Static Process produces single/self-contained exe that doesn't depend on original libs
- OS-specific implementations could change

Dynamic Instead of embedding machine code in final exe:
- Compiler stores only a ref to dynamic lib/required function
- OS must resolve linked references when app runs

x86 Architecture: Originally released by Intel 1978 w/8086 CPU

- Support over the years to 16/32/64-bit operations

ISA: Instruction Set Architecture

- Defines how machine code works/interacts w/CPU-rest of computer
- Defines set of instructions avail to a program:
  - Each individual machine lang instruction represented by mnemonic instruction
- **Mnemonics**: Name each instr/determine how params/operands represented

Common x86 Instruction Mnemonics

Instruction	Description
MOV dest, src	Moves value from source to dest
ADD dest, value	Adds int value to dest
SUB dest, value	Subs int value from dest
CALL address	Calls subroutine at specified addr
JMP address	Jumps unconditionally to specified addr
RET	Returns from previous subroutine
RETN size	Returns from previous subroutine/increments stack by size
Jcc addr	Jumps to specified addr if condition indicated by cc true
PUSH value	Pushes value onto stack/decrements stack pointer
POP dest	Pops top of stack into dest/increments stack pointer
CMP valuea, valueb	Compares valuea-b/sets appropriate flags
TEST valuea, valueb	Bitwise AND on valuea-b/sets appropriate flags

AND dest, value	Bitwise AND on dest w/value
OR dest, value	Bitwise OR on dest w/value
XOR dest, value	Bitwise Exclusive OR on dest w/value
SHL dest, N	Shifts dest to left by N bits [left being higher bits]
SHR dest, N	Shifts dest to right by N bits [right being lower bits]
INC dest	Increments dest by 1
DEC dest	Decrements dest by 1

Mnemonic instructions take 1 of 3 forms depending on how many ops instruction takes

Intel Mnemonic Forms

Num of operands	Form	Example
0	NAME	POP, RET
1	NAME input	PUSH 1; CALL func
2	NAME output, input	MOV EAX, EBX; ADD EDI, 1

2 common ways to represent x86 instructions:

1. **Intel**
2. **AT&T syntax**

Example: Add 1 to value in EAX register: **Intel: ADD EAX, 1** || **AT&T: add1 $1, %eax**

CPU Registers:

- The CPU has a number of registers for fast temp storage of current state of execution
- x86: Each registered referred to by 2/3-char label

Split into 4 main categories:

1. **General purpose**
2. **Memory index**
3. **Control**
4. **Selector**

Register Categories

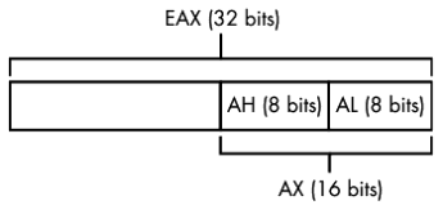
General Purpose

EAX, EBX, ECX, EDX: 32 bit: Can access in 16/8 bit versions

- Temp stores for nonspecific values of computation
- Results of addition/subtraction

Examples:

- EAX accessed as AX for 16 bit
- EAX accessed as AH/AL as 8 bit



Memory Index

ESI, EDI, ESP, EBP, EIP: Mostly general purpose except ESP/EIP

ESP register: Used by PUSH/POP

- Subroutine calls indicate current mem loc of base of stack
- Can be used for purposes other than indexing into stack
  - Not good: Mem corruption/unexpected behavior
  - Some instructions implicitly rely on value of register

EIP: Can't be directly accessed as general purpose register

- Indicates next addr in mem where instruction will be read from
- Only way to change value of EIP is by using a control instruction
  CALL, JMP, RET

EFLAGS: Boolean flags that indicate results of instruction execution

- Whether last op resulted in value 0
- Implement conditional branches on x86 processor
- Also impt sys flags: Whether interrupts enabled

Selector	**CS, DS, ES, FS, GS, SS:** Addr mem locs: Specific block you can read/write ▪ Real mem addr used in read/write value looked in internal CPU table ▪ Usually OS specific ops

Mem: Accessed using little endian byte order: LSB stored at lowest mem addr

- x86 arch doesn't req its mem ops to be aligned
- All reads/writes to main mem on aligned processor arch must be aligned to size of op
  - Example: To read 32-bit value: Would have to read from mem addr multiple of 4
    - Archs like SPARC: Reading unaligned addr would generate error
- x86 permits you to read from or write to any mem addr regardless

MOV EAX, [ESI + EDI * 8 + 0x50] ; Read 32-bit value from memory address

Important EFLAGS Status Flags

Bit	Name	Description
0	Carry flag	Whether carry bit generated from last op
2	Parity flag	Parity of LSB of last op
6	Zero flag	Whether last op had 0 as result: Comparison ops
7	Sign flag	Sign of last op: MSB of result
11	Overflow flag	Whether last op overflowed

Program/Control Flow: How a program determines which instructions to execute

x86: 3 main types of program flow instructions

1. **Subroutine calling**
2. **Conditional branches**
3. **Unconditional branches**

Subroutine calling	**Redirects flow of program to subroutine** **Subroutine**: Specified sequence of instructions: Achieved w/**CALL** instruction ▪ Changes EIP to loc of subroutine CALL ▪ CALL places mem addr of next instr onto current stack ▪ Tells program flow where to ret after performed subroutine ▪ Return performed using RET ▫ Changes EIP to top address in stack
Conditional branches	**Allow code to make decisions on prior ops** Example: CMP compares 2 operands/calcs values for **EFLAGS** register Does this by: ▫ Subtracting 1 value from other ▫ Setting **EFLAGS** as appropriate ▫ Discards result **TEST instruction:** Does the same: Performs **AND** op instead of sub ▪ After EFLAGS value calc: Conditional branch can be exe ▪ Addr it jumps to depends on state of **EFLAGS** Example: **JZ** will conditionally jump if 0 flag set ▫ Happens if 2 values equal: Otherwise instruction is no-op **EFLAGS:** Can also be set by arithmetic/other instructions ▪ **SHL instruction**: Shifts value of dest by certain num of bits from low to high ▪ Implemented through **JMP**: Just jumps unconditionally to a dest addr

Exe File Formats: Modern exe fmts include:

- Mem allocation for exe instructions/data
- Support for dynamic linking of external libs
- Support for crypto sigs to validate source of exe
- Maintenance of debug info to link exe code to original src code for debugging
- Reference to addr in exe file where code begins executing (start addr)
  - Necessary: Program's start addr might not be 1st instruction in exe file

Windows: PE: Portable Executable fmt: Typically .exe extension

- **.dll Dynamic libraries**
- Doesn't actually need these extensions for a new process to work correctly: Convenience

Unix-like sys: ELF: Executable Linking Format: Primary exe fmt

MacOS: Mach-O fmt
Sections: Mem sections probably most impt info stored in an exe
All nontrivial exe's have at least 3 sections

1. **Code:** Contains native machine code for exe
2. **Data:** Contains initialized data that can be read/written during exe
3. **BSS:** Special section to contain uninitialized data:

Every section contains 4 basic pieces of info:

- Txt name
- Size/loc of data for section contained in exe file
- Size/addr in mem where data should be loaded
- Mem protection flags: Indicate whether section can be written/exe when loaded into mem

Processes/Threads:

Process	Acts as a container for an instance of a running exe ▪ Stores all private mem the instance needs to operate ▫ Isolates it from other instances of the same exe ▫ Also sec boundary: Runs under a particular usr of OS
Thread	Allows OS to rapidly switch bet multiple processes: ▪ Makes it seem like they're all running at the same time ▪ Defines current state of execution ▪ Own block of mem for a stack/somewhere to store its state when OS stops it
Multitasking	To switch bet processes: OS must interrupt CPU/store current processes state ▪ Restore an alternate process's state ▪ When CPU resumes: Running another process

OS Networking Int: Needs to provide a way for apps to int w/network
Berkeley sockets model: Most common network API: Dev at Uni. of CA, in 70's for BSD: All UNIX-like sys:

- Built-in support for Berkeley sockets

Winsock: Windows library provides similar programming int
Creating a Simple TCP Client Connection to a Server:

```
int port = 12345;
const char* ip ="1.2.3.4";
sockaddr_in addr = {0};

int s = socket(AF_INET, SOCK_STREAM, 0);

addr.sin_family = PF_INET;
addr.sin_port = htons(port);
inet_pton(AF_INET, ip, &addr.sin_addr);

if(connect(s, (sockaddr*) &addr, sizeof(addr)) == 0)
{
    char buf[1024];
    int len = recv(s, buf, sizeof(buf), 0);

    send(s, buf, len, 0);
}
close(s);
```

1. **Creates new socket: AF_INET** indicates we want to use IPv4/6
   - 2nd param **SOCK_STREAM** indicates to use streaming connection: **TCP**
   - To create UDP socket: **SOCK_DGRAM**
2. Call to **inet_pton** converts str representation of IP to 32-bit int
   - Convert value host-byte-order (x86 Little Endian) to network-byte-order (BE: Big Endian)
   - Applies to IP also: 1.2.3.4 will become int 0x01020304 when sorted in BE fmt
3. Issue call to connect to destination addr
   - Main point of failure: OS has to make outbound call to dest addr to see if anything listening
   - **New socket connection established?** Program can read/write data to socket as if it were a file via recv/send sys calls

Creating a Client Connection to a TCP Server

```
sockaddr_in bind_addr = {0};

int s = socket(AD_INET, SOCK_STREAM, 0);

bind_addr.sin_family = AD_INET;
bind_addr.sin_port = htons(12345);
inet_pton("0.0.0.0", &bind_addr.sin.addr);

bind(s, (sockaddr*)&bind_addr, sizeof(bind_addr));
listen(s, 10);

sockaddr_in client_addr;
int socksize = sizeof(client_addr);
int newsock = accept(s, (sockaddr*)&client_addr, &socksize);
```

1. Bind socket to an addr on local network int
2. Ensure server socket will be accessible from outside current sys assuming no fw
3. Listing asks network int to listen for new incoming connections/calls accept
4. Returns next new connection: New socket can be read/written w/recv/send calls

ABI: Application Binary Interface:

- Int defined by OS to describe conventions of how app calls API function
- Most languages/OS's pass params left to right:
  - Leftmost param in original source code placed at lowest stack addr
  - If params built by pushing them to a stack: Last param pushed first
- How return value provided to function's caller when API call is complete
  - **x86:** As long as value less than/equal to 32 bits: Passed back in EAX register
  - If value bet 32-64 bits: Passed back in combo of EAX/EDX

EAX/EDX: Scratch registers in ABI

- Register values aren't preserved across function calls
- When calling a function: Caller can't rely on any value stored in these registers to still exist
- Model of designating registers as scratch done for pragmatic reasons
  - Allows functions to spend less time/mem saving registers
  - ABI specifies an exact list of which registers must be saved into a loc on stack by called function

Saved Register List

Register	ABI usage	Saved?
EAX	Pass return value of function	No
EBX	General purpose	Yes
ECX	Local loops/counters: Sometimes pass object ptrs in C++	No
EDX	Extended return values	No
EDI	General purpose	Yes
ESI	General purpose	Yes
EBP	Ptr to base of current valid stack frame	Yes
ESP	Ptr to base of stack	Yes

Static RE: Process of dissecting exe to determine what it does:

- Ideally: Reverse compilation process to original source code: Usually difficult: More common to disassemble

objdump: Prints disassembled output to console/file

IDA Pro: Hex Rays: Go to tool for static RE

debug symbols	Info about original source code line associated w/an instruction in mem ▪ Type info for functions/vars: Devs rarely leave debug symbols intentionally
PDB	**Program Database File:** All debug info stored in file ▪ Separation of debug symbols from exe: Easy to distribute w/out debug info ▪ Rarely distributed w/exe's in closed-source software MS Windows exception to this: Releases public symbols for most exe's: Including kernel to aid debugging
dSYM	**Debugging Symbols Package**: Created alongside exe rather than single PDB file ▪ Separate macOS package dir/rarely distributed w/commercial apps
magic	**Numbers defined by an algorithm that are chosen for particular mathematical properties**

constants

- Tells if encryption alg compiled into exe

Example: MD5 hashing alg:

```
void md5_init( md5_context *ctx )
{
ctx -> state[0] = 0x67452301;
ctx -> state[1] = 0xEFCDAB89;
ctx -> state[2] = 0x98BADCFE;
ctx -> state[3] = 0x10325476;
}
```

IDA: Search > Immediate value > OK

Better tools can do searches for you

PEiD: Determines whether a Windows PE file is packed w/known packing tool like UPX

- Plugins which can detect potential encryption algs/indicate where in exe referenced
- **Plugins > Krypto Analyzer**

Dynamic RE: Inspecting the op of a running exe: Useful when analyzing complex functionality

- Example: Custom crypto/compression routines
- Can step through one instruction at a time: Let's you test understanding of cold (inject test inputs)

Common way: Debugger halts running ap at specific points/inspects data values

- **IDA: Debugger > Process > options > Parameters** txt || To stop debugging running process: **CTRL F2**

Setting Breakpoints: Places of interest in disassembly: **F2**

- When program tries to exe instruction at breakpoint: Debugger should stop/give access to current program state

Debugger Windows: Default: IDA Pro debugger 3 impt win when hits breakpoint

1. **EIP Window**
2. **ESP Window**
3. **State of General Purpose Registers**

EIP Window Displays disassembly view based on instruction in EIP register currently being executed

- Much like disassembly window
- Hovering mouse over register: Quick preview of value

ESP Window Reflects current location of ESP register: Points to base of current thread's stack

- Can ID params being passed to function calls/value of local vars

General Purpose Stores current values of various program states (loop counters/mem addr)

- Mem addr: Window provides way to navigate to mem view window
  - Click arrow next to each addr to navigate from last active mem to value
  - Create new mem window: Right click array > Jump in new window
  - Lists condition flags from EFLAGS register on right side

Where to set breakpoints? send/recv functions/crypto functions

RE Managed Languages: Not apps distributed as native executables

- Apps written in managed languages: .NET/Java: Compile to an intermediate machine lang
- Commonly designed to be CPU/OS agnostic
- When app exe: VM/runtime executes code

.NET: Intermediate language called CIL: Common Intermediate Language

Java: Byte code: Substantial amts of metadata: Names of classes/internal-external-facing method names

- Output of managed languages fairly predictable: Ideal for decompiling

.NET app relies on:

1. **CLR: Common Runtime Language Runtime**
2. **BCL: Base Class Library**

Assemblies .NET uses exe/dll fmts as convenient containers for CIL code

- Contain 1/more classes enums/structs
- Each type referred to by a name: Namespace/short name
- Namespace reduces likelihood of conflicting names but useful for categorization

ILSpy: Tools like Reflector/ILSpy can decompile CIL data into C#/CB source and display original Java Apps: Differ from .NET apps bc **Java compiles don't merge all types into single file**

- Compiles each source code file into single class file w/.class extension
- Java apps packaged into JAR: Java Archive format (just a zip w/addl files to support Java)

JD-GUI: Decompilation: Same as ILSpy for .NET

Obfuscation: Tackling Tips

- **External lib types/methods [core class] can't be obfuscated:** Calls to socket API's must exist in app if doing any networking
- **.NET/Java easy to load/exe dynamically:** Write simple test harness to load obfuscated app: Run str/code decryption routines
- **Use dynamic RE as much as possible to inspect types at runtime**: Determine what used for

Resources: OpenRCE http://www.openrce.org || **ELF:** http://refspecs.linuxbase.org/elf/elf.pdf

CH 7

Wednesday, January 30, 2019 8:46 PM

All secure protocols should do the following:

- Confidentially: Protect data from being read
- Integrity: Protect data from being modified
- Prevent attacker from impersonating server/client via server/client auth

Encryption: Data confidentiality
Signing: Data integrity/auth

Substitution ciphers	**Simplest form of encryption** ▪ Alg to encrypt a value based on a sub table that contains 1-to-1 mapping bet plaintxt/cipher txt value ▪ Cipher value is looked up in a table/original txt replaced ▪ Fails to withstand cryptanalysis **Frequency analysis:** Commonly used to crack substitution ciphers ▪ Correlates frequency of symbols found in cipher txt w/plaintxt data sets
XOR Encryption	**Simple:** Applies bitwise XOR op bet byte of plaintxt/byte of key: Results in cipher txt Example: Byte 0x48 & key byte 0x82 and result of XORing would be 0xCA ▫ **Symmetric**: Applying same key byte to cipher returns original plaintxt **Only way to securely use XOR encryption?** ▪ Message/values in key chosen completely at random ▪ **OTP: One-Time Pad encryption:** Hard to break ▪ Alg also has problems/rarely used in practice ▪ OTP: Size of key material you send must be same size as any msg to sender/recipient ▪ Only secure if every byte in msg encrypted w/completely random value ▪ Can never re-use a OTP: If attacker decrypts msg: Can recover key: Compromised
RND	**Random Number Generators:** ▪ Computers are deterministic: Getting truly random data difficult ▪ Sampling physical processes can generate relatively unpredictable data ▫ Don't provide much data: Few hundred bytes every second at best ▫ 4096bit-RSA key requires at least 2 random 256-byte numbers: 7 sec to generate **PRNG: Pseudorandom Number Generators:** ▪ Use initial seed value/generate seq of num that shouldn't be predictable ▫ w/out knowledge of internal state of generator ▫ C lib rand(): Completely useless for crypto secure protocols

Symmetric Key: Send completely random key that's same size as msg before encryption can take place as OTP

- Can construct symmetric key alg that uses math constructs to make cipher
- Easier to distribute if alg has no obv weakness: Limiting factor for sec key size

2 types of symmetric ciphers

1. **Block**
2. **Stream**

Block Ciphers	**AES: Advanced Encryption Standard, DES: Data Encryption Standard** ▪ Encrypt/decrypt fixed number of bits (block) every time encryption alg applied ▪ To encrypt/decrypt msg: Alg reqs key ▪ If msg longer than size of block: Must be split into smaller blocks/alg applied to each ▪ Each app of alg uses same key
DES	**Data Encryption Standard:** ▪ Oldest block cipher still used in modern apps: Dev by IBM: Published as FIPS: 1979 ▪ FIPS: Federal Info Processing Standard **Fiestel network**: Repeatedly applies function to input for number of rounds ▪ Takes input value from previous round [original plaintxt] ▪ Specific subkey derived from original key using key-scheduling alg

DES: 64-bit block size/64-bit key
- 8 bits of key used for error checking [so 56 bits]
- Unsuitable for modern apps: 1998: EFF's DES cracker
  - HW-key brute-force attacker that discovered unknown DES key in 56 hrs

3DES **Modified form that applies alg 3 times:**
- Uses 3 separate DES keys: 168 bits
- Encrypt function applied 1st time
- Output decrypted using 2nd key
- Output encrypted again using 3rd key
  - Ops reversed perform decryption

AES **Advanced Encryption Standard:**
- Rijndael alg: Fixed 128 bit block size
- Can use 3 diff key lengths: 128/192/256 bits

Substitution-permutation network: 2 main components
1. **S-Box: Substitution Boxes**
2. **P-Box: Permutation Boxes**

2 components chained together to form round of alg:
- As with Fiestel network: Can be applied many times w/diff values of S/P-Box

S-box: Basic mapping table unlike sub cipher
- Takes input/looks up in table/produces output

Other Block Ciphers

Name	Block size (bits)	Key size (bits)	Year introduced
DES	64	56	1979
Blowfish	64	32-448	1993
3DES	64	56, 112, 168	1998
Serpent	128	128, 192, 256	1998
Twofish	128	128, 192, 256	1998
Camellia	128	128, 192, 256	2000
AES	128	128, 192, 256	2001

Block Cipher Modes: Defines how cipher ops on blocks of data
Mode of operation: Cipher combined w/this alg: Provides addl sec properties: Less predictability

ECB **Electronic Code Book:**
- Simplest/default mod of op: Alg applied to each fixed-size block from plaintxt to generate cipher blocks
- Size of block defined by alg in use

CBC **Cipher Block Chaining:**
- Encryption of single plaintxt block depends on encrypted value of previous block
- Previous encrypted block is XORed w/current plaintxt block: Alg applied to combined result
- 1st block of plaintxt no previous cipher block: Combined w/IV: Initialization Vector

Common Block Cipher Modes of Operation:

Name	Abbreviation	Mode Type
Electronic Code Block	ECB	Block
Cipher Block Chaining	CBC	Block
Output Feedback	OFB	Stream
Cipher Feedback	CFB	Stream
Counter	CTR	Stream
Galois Counter Mode	GCM	Stream w/data integrity

Block Cipher Padding: Op on fixed-size msg unit: Block
- Padding schemes determine how to handle unused remainder of a block during encryption/decryption

PKCS #7: Public Key Crypto Standard #7: All padded bytes set to value that represents how many padded bytes

present

- Each byte set to value 3

Padding Oracle Attack	Occurs when CBC mode of op combined w/PKCS #7 padding scheme ▪ Allows attacker to decrypt data: Some cases encrypt own data (session token) via protocol ▪ If attacker can decrypt a session token: Might recover sensitive info If they can encrypt the token: ▪ Might be able to circumvent access controls on a website for ex

Common Stream Ciphers

Cipher Name	Key size (bits)	Year Introduced
A5/1 and A5/2 (GSM voice)	54/64	1989
RC4	Up to 2048	1993
CTF: Counter Mode	Depends on block cipher	NA
OFB: Output Feedback Mode	Depends on block cipher	NA
CFB: Cipher Feedback Mode	Depends on block cipher	NA

Signature algs: Generate a unique signature for a msg: Msg recipient can use same alg to generate sig to prove auth

- Protects against tampering over untrusted network
- Built on crypto hashing algs

Cryptographic Hashing Algs: AKA Message Digest Algs

- Funcs applied to a msg to generated fixed-length summary of msg: Usually shorter than original

For hashing alg to be suitable needs 3 reqs:

Pre-Image resistance	Given a hash value: Diff to recover msg
Collision resistance	Diff to find 2 diff msgs that has to same value
Nonlinearity	Diff to create a msg that hashes to any given value

Most common:

- **MD: Message Digest:** MD4/5: Ron Rivest
- **SHA: Secure Hashing Alg:** SHA-1/2: NIST
- **CRC:** Useful for detecting changes in data: Not useful for secure protocols: Can change checksum

Asymmetric Signature Algorithms: Properties of asymmetric crypto to generate msg signature

- **DSA: Digital Signature Alg:** Designed for sigs only
- **RSA:** Possible to encrypt msg using private key/decrypt w/public one (no longer secure)

Message Authentication Codes: MACs: Symmetric sig algs: Rely on sharing key bet sender/recipient

HMAC: Hashed Message Auth Code: Counters attacks

- Instead of directly appending key to msg/using hashed output to produce sig: Splits process into 2 parts
  1. **Key is XORed w/padding block equal to block size of hashing alg**
     - 1st padding block filled w/repeating value: Typically byte 0x36
     - Prefixed to msg/hashing alg applied
  2. **Takes hash value from 1st step: Prefixes hash w/new key (outer padding block: Uses constant 0x5C)**
     - Applies hash alg again

PKI: Public Key Infrastructure:

- Combined set of protocols/encryption key fmts/usr roles/policies to manage asymmetric public key info across network

WOT	**Web of Trust:** Used by PGP: ID of public key attested to by someone you trust

X.509 Certificates: Generate strict hierarchy of trust rather than relying on directly trusting peers

Used to: Verify web servers | Sign exe programs | Auth to a network service

- Trust provided through hierarchy of certs using asymmetric sig algs like RSA/DSA

Chain of trust: Certs must contain at least 4 pieces of info:

1. **Subject: Specifies ID for cert**
2. **Subject's public key**
3. **Issuer: ID's signing cert**
4. **Valid sig applied over cert/auth by issuer's priv key**

TLS: Transport Layer Security: Formerly SSL: Secure Sockets Layer:

- Most common sec protocol in use on the internet
- Originally dev as SSL by Netscape in mid-90's for securing HTTP connections

Protocol went through multiple revisions: SSL ver 1-3.0 || TLS 1.0-1.2

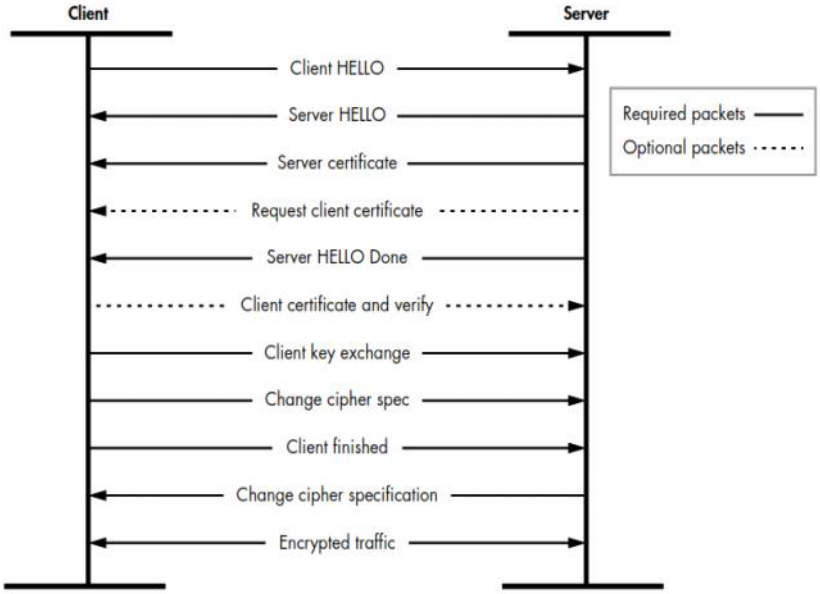
- Originally designed for HTTP: Can use TLS for any TCP protocol

DTLS: Datagram Transport Layer Security: For use w/UDP/unreliable protocols

TLS: Symmetric/asymmetric encryption/MACs/Secure key exchange/PKI

TLS Handshake: Client/server negotiate type of encryption they'll use: Exchange unique key for connection/verify ID's

- TLS Record protocol: All comm uses a predefined TLV structure
  - Allows protocol parser to extract individual records from stream of bytes
  - Handshake packets assigned a tag value of 22 to distinguish from other packets
- Handshake process can be time-intensive
- Sometimes truncated/bypassed entirely by caching previously negotiated session key
  - Or by client's asking server to resume previous session by unique session identifier
  - Client still won't know private negotiated session key sec wise



Initial Negotiation:

- **1st step:** Client/server negotiate sec params they want to use for TLS connection using **HELLO** msg
  - Piece of info in **HELLO** client random
  - **Client random:** Random value ensures connection can't be easily replayed
  - **HELLO** msg indicates types of ciphers supported
- **TLS designed to be flexible w/regard to what encryption algs it uses:**
  - Only supports symmetric ciphers like RC4/AES
  - Using public key encryption would be too expensive from computational perspective
- **Server responds w/its own HELLO msg:** Indicates what cipher chosen from avail list
- **Server HELLO also contains server random**
  - **Server random:** Value that adds addl replay protection to connection
- **Server sends its X.509 cert/any necessary intermediate CA certs so client can make informed decision about ID**
- **Server sends HELLO done packet to inform client it can proceed to auth connection**

Endpoint Authentication: Client must verify server certs legitimate/meet client's sec reqs

- **Client must verify ID in cert by matching Subject field to server's domain name**
  - Cert's Subject/Issuer fields not simple strings but X.500 names
  - Contains other fields like Organization/Email
  - Only CN ever checked during handshake to verify ID
  - Possible to have wildcards in CN field: Sharing certs w/multiple servers running on a subdomain
    - *.domain for www.domain.com or blog.domain.com
- **After client checks ID of endpoint**: Ensures cert trusted
  - Builds chain of trust for cert/any intermediate CA certs
  - Checks to make sure none of them appear on revocation lists

- **Optional: Client certificate:** Allows server to auth client
  - If server reqs client cert: Sends list of acceptable root certs to client during HELLO
  - Client can search avail certs/choose most useful to send back to server
  - Cert + verification msg containing hash of all handshake msgs sent/received/signed w/cert's priv key
  - Sig proves to server client possesses priv key associated w/cert

Establishing Encryption: When endpoint auth: Client/server finally establish encrypted connection

- Client sends randomly generated pre-master secret to server encrypted w/server's cert public key
- Both client/server combine pre-master secret w/client/server randoms
- They use this combined value to seed a RNG that generates a 48-byte master secrete
  - Will be session key for encrypted connection
- **When both endpoints have master secret:**
  - Client issues change cipher spec packet: Tells server it will only send encrypted msgs from there on
  - Client needs 1 msg to server before normal traffic: Finished packet
    - Packet encrypted w/session key/contains hash of all handshake msgs sent/received during process
    - Crucial against downgrade attack

Downgrade attack: Attacker mods handshake process to reduce sec of connection by selecting weak encryption algs

- Once server receives finished msg: Can validate negotiated session key correct: If not: Closes connection

How TLS Meets Sec Reqs

Sec Req	How met
Confidentiality	Selectable strong cipher suites: Sec key exchange
Integrity	Encrypted data protected by an HMAC: Handshake packets verified by final hash verification
Server Auth	Client can choose to verify server endpoint using PKI + issued cert
Client Auth	Optional cert-based client auth

Problems:

- Reliance on cert-based PKI
- Depends entirely on trust that certs issued to correct people/orgs
- Subversion of CA process to generate certs an issue
- CA's don't always perform due diligence/issued bad certs

Certificate pinning: App restricts acceptable certs/CA issuers for certain domains

- Issue: Management of pinning list: Dev can't easily migrate/change certs to another CA
- TLS connections can be captured from network/stored by attacker until needed
- If attacker obtains server's priv key: All historical traffic could be decrypted

Why using DH alg in addition to certs for ID verification impt

Perfect Forward Secrecy: Even if private key compromised: Not easy to also calc DH-generated key

CH 9

Wednesday, January 30, 2019 8:46 PM

Vulnerability Classes:

RCE	**Remote Code Execution:** Running arbitrary code w/app that implements protocol ▪ Hijacking logic of app/influencing cli subprocesses created in normal op **Allows attacker to compromise sys of app executing** ▪ Access to anything app can access: Maybe hosting network compromised too
DoS	**Denial of Service:** Causes crash/unresponsiveness • Denies usr access to app/service **Categorized as:** **1. Persistent:** Perm prevents usr from accessing service **2. Non-persistent:** As long as attacker attacks
Info Disclosure	Exists if there's a way to get an app to provide info it wasn't designed to ▪ Contents of mem/fs paths/auth creds
Auth Bypass	**Authentication Bypass:** Way to auth to app w/out providing all creds ▪ Incorrectly checking for a password/brute force/SQLi ▪ Allows to auth as a specific usr
Autho Bypass	**Authorization Bypass:** Can gain rights/access to resources not priv to access ▪ Allows attacker to access resource from incorrect auth

Memory Corruption Vulns: Mem-Safe vs. Mem-Unsafe Languages:

Memory safe languages:

- Java/C#/Python/Ruby don't normally req dev to deal w/low-level mem mgmt
- Can provide libs/constructs to perform unsafe ops: C# unsafe keyword
- Bounds checking for in-mem buffer access to prevent out-of-bounds reads/writes
- Not completely immune to mem corruption
- More likely to be a bug in the runtime

Memory-Unsafe languages:

- C/C++ perform little mem access verification/lack robust mechs for auto managing mem
- Many types of mem corruption can occur
- How exploitable? Depends on OS/compiler used/how app structured

Buffer Overflows: When app tries to put more data into region of mem than designed to hold

Can occur for 2 reasons:

1. **Fixed-length:** Incorrect input buffer fitting into allocated buffer
2. **Variable-length**: Size of allocated buffer incorrectly calc

Fixed-Length **App incorrectly checks length of external data value**

- Relative to fixed-length buffer in mem
- Might be in stack/on a heap/exist as global buffer defined at compile time
- Mem length determined prior to knowledge of actual data length

```
1  def read_string() {
2      byte str[32];
3      int i = 0;
4
5      do {
6      str[i] = read_byte();
7      i = i + 1;
8      }
9  while(str[i-1] != 0);
10 printf("Read String: %s\n", str);
11 }
```

1. Allocates buffer where it will store string on stack: 32 bytes
   - Loop reads byte from network/stores into incrementing index in buffer

2. Loop exits when last byte read from network eq to 0: Indicates value sent

3. Mistake: Loop doesn't verify length/reads as much data as avail from network

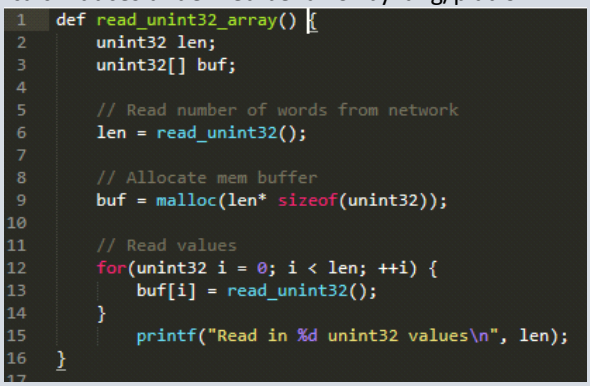
Unsafe String Functions: C doesn't define str type: Uses ptrs to list of char types

- End of str indicated by 0-value char

strcpy: Function copies strings: Takes only 2 args

1. **Ptr to source string**
2. **Ptr to destination mem buffer to store copy**
    - Nothing indicates length destination mem buffer
    - Recent C compilers added more sec vers of these
        - **strcpy_s**: adds a destination length arg

off-by-one error: Shift in index positions (screwing up arrays)

Var-Length	Possible for app to allocate buffer of correct size for data being stored ▪ If incorrectly calcs buffer size var-length b0f could happen ▪ Issue if calc induces undefined behavior by lang/platform  1. Buffer dynamically allocated at runtime to contain total size of input ▫ 32-bit int: Uses to determine num of next 32-bit value ▫ Determines total allocation size/allocates buffer corresponding size 2. Loop reads each value from protocol into allocated buffer
Int Overflows	**modulo arithmetic:** At processor instruction lvl: int math ops ▪ Allows values to wrap if they go above certain value: **modulus** ▪ Processor uses modulo if supports certain native int such as 32/64 bits ▪ Result of any op must be w/in ranged allowed for fixed-size int values Example: 8 bit int: Can only take values bet 0-255 ▪ Multiplying a value by 4 on 32-bit ints like 65 x 4 = 0x104 or 260 ▪ Processor drops the overflowed bit
Out-of-bounds Buffer Indexing	Sometimes vuln occurs bc size of buffer incorrect ▪ Instead of incorrectly specifying size of value ▪ Some control over position in buffer ▪ If incorrect bounds checking on access position: Vuln exists **Selective mem corruption:** Can be exploited to write data outside buffer: ▪ Exploited reading value outside buffer: Info disclosure/RCE Doesn't just have to involve writing: ▪ Works when values read from buffer w/incorrect index ▪ If index used to read value/ret to client: Simple info disclosure ▪ Vuln could occur if index used to ID functions w/in app to run
Data Expansion Attack	Modern high-speed networks compress data to reduce num of raw octets ▪ At some point data must be decompressed ▪ If compression done by app: Data expansion possible

Dynamic Memory Allocation Failures: System memory finite: When mem pool runs dry:

- Dynamic mem allocation pool handles situations where app needs more
- Results in error value being ret from allocation functions (NULL ptr)

Possible vulns may arise from not correctly handling dynamic mem allocation failure: DoS/app crash

Default/Hardcoded Creds	Default creds commonly added as part of installation process ▪ Usually default usrname/passwd associated w ▪ Problem if they aren't changed
User Enumeration	Most usr-facing auth use usernames to control access to resources ▪ Typically name combined with token

- User ID doesn't have to be a secret: Often publicly avail emails
- More likely you could brute force passwds by valid accts

Incorrect Resource Access: Protocols provide access to resources (HTTP)/file-sharing/ID for resource

- Identifier could be file path/unique: App must resolve identifier in order to access target resource
- Many vulns can affect such protocols when processing resource identifiers

Canonicalization

If resource identifier hierarchical list of resources/dirs: Referred to as path
- OS defines way to specify relative path info using .. (parent dir)
- Before a file can be accessed: OS must find it using this path info

Naïve remote file protocol: Pass directly to OS
- Could take path supplied by remote user: Concatenate it w/base dir

Verbose Errors

When app tries to retrieve resource/isn't found: Returns error info
- Simple as error code w/full description of what doesn't exist
- Shouldn't disclose any more info than required

Mem Exhaustion

Resources of sys on which app runs finite: Exhausting them
- Allocating mem dynamically based on absolute value transmitted in protocol

CPU Exhaustion

CPU's can only do certain # of tasks a time

2 main ways:

1. **Algorithmic complexity**
2. **Identifying external controllable params to cryptographic systems**

Algorithmic Complexity:
- All algs have associated computational cost
- How much work performed for particular input to get desired output
- More work alg needs? More time from processor
- Some algs become expansive as num of input params increase

Example: Bubble Sort: Inspects each value pair in a buffer/swaps them
- If left value of pair greater than right
- Bubbling higher values at end of buffer until buffer is sorted
- Amt of work alg req proportional to num of elements in buffer to sort

Best case: Single pass through buffer req N iterations: All elements already sorted

Worse case: Buffer sorted In reverse: Alg needs to repeat sort process N^2 times
- If attacker could specify a large num of reverse-sorted values
  - Computational cost becomes significant
  - Could consume 100% of CPU's processing time: DoS

Configurable Crypto:
- Primitives processing: Hashing create significant amt of workload
  - Authentication creds
- Passwds should always be hashed using digest alg before stored
- Converts pass into hash value: Impossible to reverse
- Someone could still guess pass/generate hash
- If guessed passwd matches when hashed: Original pass discovered

To mitigate: Typical to run hashing op multiple times
- Increase computational cost for app
  - DoS: Long time bc of size/alg # of iterations specified externally

Format String Vulnerabilities: Most lang have mech to convert arbitrary data into str

- Common to define some fmting mech to specify output
- Attacker can supply str value to app used directly as fmt str
- **printf**/variants such as **sprintf** which print to str
  - Takes fmt str as first arg/list of values to fmt
- Specifies position/type of data using a %? syntax (? replaced by alphanumeric char)
  - Fmt specifier can also include fmt info: num of dec places in num
  - Attacker who can directly control fmt str could corrupt mem/disclose info

List of Commonly Exploitable printf Fmt Specifiers

Fmt Specifier	Description	Potential Vulns
%d, %p, %u, %x	Prints ints	Info disclosure from stack if ret to an attacker
%s	Prints 0 terminated str	Info disclosure from stack if ret to an attacker

		Cause invalid mem accesses to occur: DoS
%n	Writes current # of printed chars to ptr specified in args	Selective mem corruption/app crash

Command Injection: Most OS: Set of utilities for various tasks

- Some decide easiest way to exe task is to exe an external app/os util
- Some data from network client inserted into cli to perform desired op

SQLi: Simplest app may need to persistently store/retrieve data: Relational DB

- SQL: Structured Query Language: Defines what data tables to read/how to filter them
- Can easily result in vuln like cmd injection:
- Instead of inserting untrusted data into CLI w/out appropriately escaping
- Attacker inserts data into SQL query: Executed on DB: Can mod op of query

Txt-Encoding Char Replacement: Some conversions bet txt encodings can't be round-tripped:

- Converting from 1 encoding to another loses impt info
  - If reverse applied original txt can't be restored
- Converting from wide char set (Unicode) to narrow (ASCII)
  - Impossible to encode entire Unicode char set in 7 bits

Conversions handle this 2 ways:

1. **Replaces char that can't be represented w/placeholder (?)**
   - Problem if data value refers to something where ? is delimiter/special char
2. **Best-fit mapping**: Used for chars for similar char in new encoding
   - Problem when converted txt processed by app

Implementation issue: App 1st verifies sec condition using 1 encoded form of a str

- Then uses other encoded form of str for specific action:
  - Reading resource/executing cmd

CH 10

Wednesday, January 30, 2019 8:47 PM

Fuzzing: Feeds random/not-so-random data into protocol to force processing app to crash in order to ID vulns

- Yields results no matter complexity
- Produces simple multiple test cases: Sent to app for processing
- Can be generated auto using random mods/under direction from analyst

Simplest: Sends random garbage to see what happens: **cat /dev/urandom | nc hostname port**

- Reads data from system's RNG device using cat: Piped into netcat: Opens connection as instructed

Mutation Fuzzer: Using existing protocol data/mutate it in some way/send it to receiving app

Simplest: Random bit flipper

```
void SimpleFuzzer(cons char* data, size_t length) {
    size_t position = RandomInt(length);
    size_t bit = RandomInt(8);

    char* copy = CopyData(data, length);
    copy[position] ^= (1 << bit);
    SendData(copy, length);
}
```

1. SimpleFuzzer() function: Takes in data/length of data to fuzz:
    - Generates random num bet 0/length of data to mod
2. Decides which bit in byte to change by generating num between 0-7
    - Toggles bit using XOR/sends mutated data to network destination

Vulnerability Triaging: Taking a series of steps to search for root cause of a crash

Debugging Applications: Diff platforms allow diff lvls of control over triaging: Can attach debugger to process

Cmds: Running debuggers on Win/Linux/MacOS

Debugger	New Process	Attach process
CDB (Win)	cdb application.exe [args]	cdb -p PID
GDB (Linux)	gdb --args application [args]	gdb -p PID
LLDB (macOS)	lldb -- application [args]	lldb -p PID

- Debugger will suspend execution of process after you've created/attached debugger: Run process again

Simplified App Execution Cmds

Debugger	Start Execution	Resume Execution
CDB	g	g
GDB	run, r	continue, c
LLDB	process launch, run, r	thread continue, c

When new process creates child process: Might be child process that crashes instead of one debugging

- Can follow child/not parent

Debugging Child Processes

Debugger	Enabled child process debugging	Disable child process debugging
CDB	.childdbg 1	.childdbg 0
GDB	set follow-fork-mode child	set follow-fork-mode parent
LLDB	process attach --name NAME --waitfor	exit debugger

Analyzing the Crash: Look for crashes that indicate corrupted mem:

Windows: Access violation | Linux: SIGSEGV

Instruction Disassembly Commands

Debugger	Disassemble from crash location	Disassemble from specific location
CDB	u	u ADDR

GDB	**disassemble**	**disassemble ADDR**
LLDB	**disassemble -frame**	**disassemble --start-address ADDR**

Displaying/Setting Processor Register State

Debugger	Show general purpose registers	Show specific registers	Set specific register
CDB	**r**	**r @rcx**	**r @rcx = NEWVALUE**
GDB	**info registers**	**info registers rcx**	**set $rcx = NEWVALUE**
LLDB	**register read**	**register read rcx**	**register write rcx NEWVALUE**

- Can use these to set the value of register: Allows you to keep app running by fixing crash/restarting execution

Creating a Stack Trace: When app debugging crashes: Want to display how current function was called

- Can narrow down which parts of protocol needed to focus on reproducing crash
- Can get context by generating stack trace
- Displayed functions called prior to execution of vuln: Including some local vars/args passed to them

Creating a Stack Trace

Debugger	Display stack trace	Display stack trace with arguments
CDB	**K**	**Kb**
GDB	**backtrace**	**backtrace full**
LLDB	**backtrace**	

Displaying Memory Values

Debugger	Display bytes/words/dwords/qwords	Display ten 1-byte values
CDB	**db, dw, dd, dq ADDR**	**db ADDR L10**
GDB	**x/b, x/h, x/w, x/g ADDR**	**x/10b ADDR**
LLDB	**memory read --size 1,2,4,8**	**memory read --size 1 --count 10**

CMDS for Displaying Process Mem Map

Debugger	Display process memory map
CDB	**!address**
GDB	**info proc mappings**
LLDB	**No direct equivalent**

- Determines what type of mem an addr corresponds to: Heap/stack/mapped executable
- Helps narrow down type of issue
- Example: Memory value corruption occurred? Distinguish whether stack/heap mem corruption

Rebuilding apps w/Addr Sanitizer:

Asan Address Sanitizer: Extension for CLANG C compiler: Detects mem corruption bugs

-fsantize=address when running compiler:

- Specify option using CFLAGS env var
- Rebuilt app will have addl instrumentation to detect common mem errors
- Mem corruption/out-of-bound writes/use-after-free/double-free
- Stops app as soon as vuln condition has occurred

Page Heap Win Access to source code of app more restricted:

- **Page Heap:** Can enable chances of tracking down mem corruption

gflags.exe -i appname.exe +hpa

- Comes installed w/CDB debugger
- **-i**: specify img filename to enable page heap on
- **+hpa**: What actually enables page heap when app executes

Works by allocating special OS-defined mem pages: AKA: **guard pages** after every heap allocation

- If an app tries to read/write these guard pages: Error will be raised/debugger notified
- Useful for detecting heap overflows
- If overflow writes immediately at end of buffer: Guard page will be touched by app/error

Cons: Wastes a huge amt of mem b/c each allocation needs a separate guard page

- Requires a syscall which reduces allocation performance

Exploiting Common Vulns

Stack Overflows

Occurs when code underestimates length of buffer to cp into a loc on the stack
- Many archs: Return addr for function stored on stack/corruption of ret addr gives direct execution
- Corrupt ret addr on stack to point to buffer containing shell code w/instructions
- Need to craft data into overflowed buffer to ensure rt addr points to mem region you control
- If caused by C-style str copy: Won't be able to use multiple 0 bytes in overflow
- **C uses a 0 byte as terminating char for string**
  - Overflow will stop immediately
- Direct shell code to addr value with no 0 bytes

Heap Overflows

Often less predictable mem addr: No guarantee
- Exploit the structure of C++ objects: specifically Vtables

VTable: List of pointers to functions that the object implements
- Allows dev to make new classes derived from existing base classes/override some functionality
- Each allocated instance of a class must contain a ptr to the mem loc of the function table
- When virtual func called on object: Compiler generates code that looks up addr of Vtable
- Then looks up virtual function inside table/calls addr
- Can't corrupt the ptrs in the table: Likely stored in read-only part of mem
- CAN corrupt ptr to the Vtable to gain code execution

Use-After-Free

Corruption of the state of the program/not exactly mem
- When mem block is freed but ptr to block stored by some part of app
- Later in app execution: ptr to freed block re-used

Bet time mem block freed/ptr reused opportunity to replace contents of block w/arbitrary values
- Gain code execution
- When mem block freed: Will be given back to heap to be reused for another mem allocation
- As long as you can issue allocation req of same size as original allocation
  - Strong possibility freed mem block would be reused w/your crafted contents

App first allocations an object p on heap: Contains a Vtable ptr we want to control
- App calls del on ptr to free mem
- App doesn't reset value of p: Object free to be reused in the future
- Exploit allocates mem of exact size/has control over contents of mem p points to
- Heap allocator reuses as allocation for p
- If app reuses p to call a virtual function: Can control lookup/gain execution

Manipulating Heap Layout: Key to success usually is in forcing suitable allocation to occur at a reliable loc
- Heap implementation for an app may be based on virtual mem mgmt features of platform app exe on

Using OS virtual mem allocator has problems:
- Poor perf: Each allocation/free-up requires OS to switch to kernel mode/back
- Wasted mem: Virtual mem allocations done at page level: At least 4096 bytes
  - If you allocate mem smaller than page size: Rest of page wasted

Free-list

Maintains a list of freed allocations inside a larger allocation
- When allocation req made:
- Heap's implementation scans list of free blocks looking for sufficient size
- Would use free block/allocate req block at start
- Update free-list to reflect new free size

Defined Mem Pools

Defined mem pools for diff allocations sizes:
- Groups smaller allocations appropriately
- When req made: Implementation will allocate buffer based on pool most closely matched
- Reduces fragmentation caused by small allocations

Heap mem storage

How info like free-list stored in mem: 2 methods

1. **In-band:** Metadata (block size): Whether state is free/allocated stored along allocated mem
2. **Out-of-band:** Metadata stored elsewhere in mem: Easier to exploit
    - Don't have to worry about restoring impt metadata when corrupting mem blocks
    - Useful when you don't know what values to restore for metadata to be valid

Arbitrary Mem Write Vuln: File write resulting from incorrect resource handling

- May be due to cmd that allows you to specify loc of a file write/path canonicalization
- Could occur as a by-product of another vuln like heap overflow
- Many old heap mem allocators would use linked list structure to store list of free blocks
- If linked list corrupted: Any mod of free-list could result in arbitrary write of value into attacker-supplied loc

To exploit: Need to mod loc that can directly control code execution

- Could target Vtable ptr of an object in mem/overwrite to gain control over execution

Advantage: Can lead to subverting logic of an app

Mitigating Mem Corruption:

DEP/NX **Data Execution Prevention/No-Execute:**

- Attempts to mitigate by req mem w/executable instructions to be specifically allocated by OS
- Requires processor support so if process tries to execute mem at addr not marked: Raises error
- OS terminates process in error to prevent further execution

Can determine whether executable mem is being used through memory mapping cmds

- If DEP enabled: Can use ROP: Return-Oriented Programming as a workaround

ROP **Return-Oriented Programming**

- Repurposes existing already executable restructures rather than injecting arbitrary instructions
- Sequence of instructions doesn't have to execute as originally compiled into code
- Can make small snippets of code throughout program

ROP gadgets: These small sequences of instructions

- Easier when you have a stack overflow
- Heap overflow? Will need a stack pivot

Stack pivot: ROP gadget that allows you to set current stack ptr to known value

ASLR **Address Space Layout Randomization:**

- Bypassing DEP became more diff: Randomizes the layout of a processes addr space
- Makes it harder to predict
- Location of an exe in ASLR isn't always randomized bet 2 separate processes
    - Vuln that could disclose loc of mem

Partial overwrites: Lower bits of random mem ptrs can be predictable if upper bits totally random

Canaries Detect corruption/immediately cause app to terminate:

- Random number generated by app during startup: Stored in global mem loc
- Can be accessed by all code in app
- Random num pushed onto stack when entering a function
- When function exist: Random value popped off stack/compared to global value
- If global value doesn't match what was popped: App assumes stack mem corrupted/terminates

Bypassing: Typically only protect the ret addr of currently executing func on stack

- If stack overflow has controlled length: Possible to overwrite these vars w/out corrupting canary
- Buffer underflow