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:

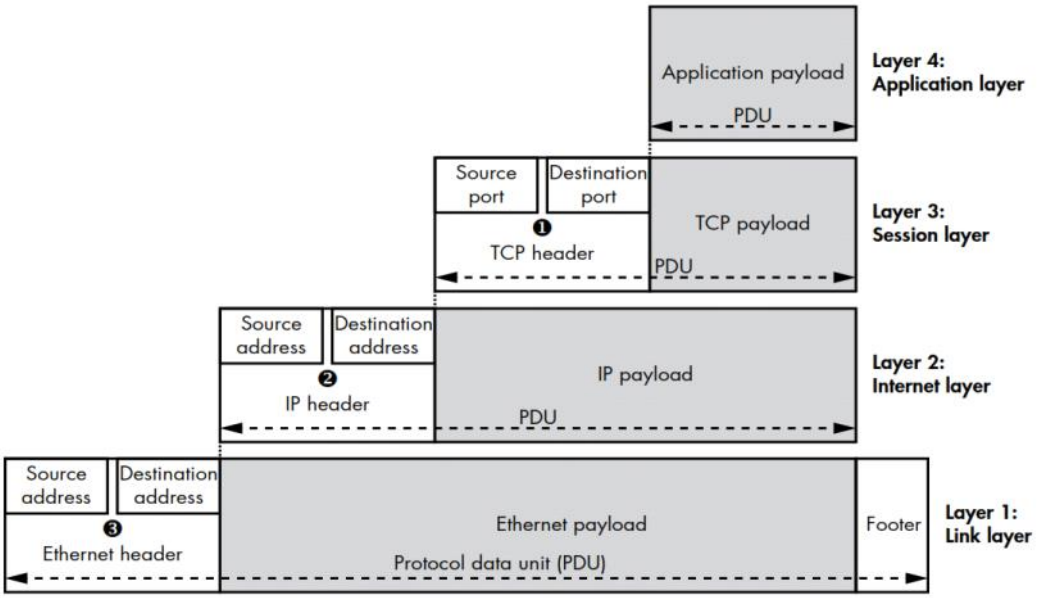
Link	Phys: Xfer info bet nodes: Ethernet/PPP
Internet	Addr network nodes: IPv4/6
Transport	Connections bet client/server: Correct order of packets/multiplexing Service multiplexing: TCP/UDP: Single node support multiple diff services ▪ Assigns diff #'s for each port
App	Network protocols: HTTP/SMTP/DNS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