Advanced App Traffic Capture

Monday, December 24, 2018 2:31 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**

<u>Apply rule only to specific TCP/UDP change:</u>

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]

<u>ID Packet Structure w/Hex Dump:</u>

- 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

```python
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

```python
# Checksum function
def calc_chksum(unk2, data):
    chksum = unk2
    for i in range(len(data)):
        chksum += ord(data[i:i_1])
    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:

```lua
-- Declare chat protocol for dissection
chat_proto = Proto("chat", "SuperFunkyChat Protocol")

-- Specify protocol fields
chat_proto.fields.chksum = ProtoField.uint32("chat.chksum", "Checksum",
    base.HEX)
chat_proto.fields.command = ProtoField.unint8("chat.command", "Command")
chat_proto.fields.data = ProtoField.bytes("chat.data", "Data")

-- Dissector function
-- buffer: UDP packet data as a "Testy Virtual Buffer"
-- pinfo: Packet info
-- tree: Root of the UI tree
function chat_proto.dissector(buffer, pinfo, tree)
    -- Set name in the protocol column in the UI
    pinfo.cols.protocol = "CHAT"

-- Create sub tree which represents entire buffer
local subtree = tree:add(chat_proto, buffer(),
    "SuperFunkyChat Protocol Data")
subtree:add(chat_proto.fields.chksum, buffer(0,4))
subtree:add(chat_proto.fields.command, buffer(4,1))
subtree:add(chat_proto.fields.data, buffer(5))
end

-- Get UDP dissector table/add for port 12345
udp_table = DissectorTable.get("udp.port")
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
    - **<u>Function has 3 params:</u>**
        - 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