General Guidance

You are not expected to memorize supplemental and reading materials, however, you should be familiar enough with the content to use as a reference in a testing environment.

Socket Domains

In most cases, the domain represents a protocol family, which is virtually what set of protocols will be used for communication. The domain usually relates to the network layer protocol used to carry the traffic; but this is not always the case.

Domain	Symbol	Description
UNIX sockets domain	`AF_UNIX, AF_LOCAL`	The UNIX sockets domain is used for local communications on the same machine and does not relate to any networking protocol. The communication mechanism used in this domain is somewhat similar to pipes. The main difference with pipes is that these sockets allow two applications to send and/or receive data as if they were communicating over a network. The UNIX sockets domain is only available on UNIX/Linux platforms.
IPv4 domain	`AF_INET`	The IPv4 domain is comprised of all protocols that can directly be carried by IPv4 such as ICMP, TCP, UDP, IGMP, and so on.
IPv6 domain	`AF_INET6`	The IPv6 domain is comprised of protocols that can directly be carried by IPv6 such as ICMP6, TCP, UDP, and so on.
Low-level sockets domain	`AF_PACKET`	The low-level sockets domain does not relate to any specific network layer protocol, but it provides the capability to manually build packets from scratch before sending it on the network at the link layer. The low-level sockets domain is only supported on UNIX and UNIX-like operating systems.

Socket Types

Sockets make it possible for applications to listen to incoming connections, send and receive data, and terminate the connection once data exchange has been deemed completed. Sockets can be categorized using the concept of socket type. The socket type usually represents what communication pattern is being used within a domain.

Type	Description	In the AF_INET and AF_INET6 domains…
`SOCK_STREAM`	Provides sequenced, reliable, two-way, connection-based byte streams.	SOCK_STREAM usually corresponds to TCP protocol

SOCK_DGRAM	Supports datagrams (connectionless, unreliable messages of a fixed maximum length).	SOCK_DGRAM usually corresponds to UDP protocol
SOCK_RAW	Provides raw network protocol access.	SOCK_RAW corresponds to building the entire IP packet or just the packet payload from scratch. The packet is then passed down to the link layer that will handle the rest. The link layer uses the current machine MAC address as source address. The destination MAC address is determined by using link layer mechanisms such as ARP, default gateway, and so on. Raw sockets are supported by default on UNIX. However, to determine if a Windows system XP or above supports them, you can run the command: **netsh winsock show catalog**. In the output of that command, search for RAW/IP or RAW/IPv6 in the Description field to find those protocols that support raw sockets.

Note: With the AF_PACKET domain, SOCK_RAW corresponds to building the entire packet including transport layer, network layer, and data link layer. All checksums and signatures that are usually calculated by the networking stack will need to be handled by the application using this socket type and this domain.

Protocol Types

The protocol type is used to fully identify what IP protocol is used for communication. The protocol type is identified by a number that is defined within RFC 1700. That number also matches the protocol field in the IPv4 header or the “Next Header” field within the IPv6 header. The protocol type is sometimes called protocol number or protocol id.

Symbol	Protocol Type
IPPROTO_TCP	TCP protocol
IPPROTO_UDP	UPD protocol
IPPROTO_ICMP	ICMP protocol
IPPROTO_RAW	Some other generic protocol types can be used as well. For instance, IPPROTO_RAW does not represent any specific protocol, but it is used to indicate that the protocol is already specified in the IP header.

Bash Scripting vs Python Scripting

In this topic we will compare a Python script vs a Bash script.

Python Script

```
#!/usr/bin/python
num1= 12
str = "Hello World"
print(str)
num2 = len(str)
print ("The length of the string '%s' is %d" %(str,
num2))

# Simple for loop
for x in ("abc", 123, "cef"):
print(x)

# if…else statement
if num%2==0:
        print("Even")
else:
        print("Odd")

# Simple while loop
while num > 12:
        print(num)
        num=num-1

# Reading a text file line per line
fp = open("/etc/passwd", "r")
line = fp.readline() # reading first line
while line:
        print(line)
        line = fp.readline()
fp.close()

# Reading a file chunk by chunk
fp = open("/etc/passwd", "r")
chunk = fp.read(20) # reading first 20-byte chunk
while chunk:
        print(chunk)
        chunk = fp.read(20)
```

Bash Script

```
#!/bin/bash
num1=12
str="Hello World"
echo $str
num2=${#str}
echo "The length of the string '$str' is $num"

# Simple for loop
for x in "abc", 123, "cef"
do
        echo $x
done

# if…else statement
if [ `expr $num % 2` == 0 ]
then
        echo "Even"
else
        echo "Odd"
fi

# Simple while loop
while [ $num -gt 5 ]
do
        echo $num
        num = `expr $num - 1`
done

# Reading a text file line per line
while read line
        do
                echo $line
        done < /etc/passwd

# Reading a file chunk by chunk (ideal for non-text
files)
while read -n 20 chunk # Reading 20-byte chunks
        do
                echo $chunk
        done < /etc/passwd
```

TCP Sockets

Because TCP is a connection-based protocol, it is implied that one endpoint will be listening for incoming connections and that the other will initiate or request the connection. If the connection is authorized, a communication channel is established. The communication endpoint that listens for incoming connections is the server; whereas, the communication endpoint that initiates the connection is the client.

TCP Sockets Server Side

```
from socket import *
server=socket(AF_INET, SOCK_STREAM)
server.bind(("192.168.11.11", 12345))
server.listen(10)
client,address=server.accept()
client_data=client.recv(100)
print(client_data)
client.send=("Hello World from server")
server.close()
```

TCP Sockets Client Side

```
from socket import *
client=socket(AF_INET, SOCK_STREAM)
client.connect(("192.168.11.11", 12345))
client.send("Hello World from client")
server_data=client.recv(100)
print(server_data)
client.close()
```

UDP Sockets

UDP sockets do not require a connection to be established before data can be exchanged between two endpoints. The only requirement to receive data is for an application to bind to an address and a port. The communication endpoint that binds to receive incoming packets is the server. An endpoint that sends UDP packets to the server is a client.

UDP Sockets Server Side

```
from socket import *
server=socket(AF_INET, SOCK_DGRAM)
server.bind(("192.168.11.11", 12345))
client_data,address=server.recvfrom(100)
print(client_data)
server.close()
```

UDP Sockets Client Side

```
from socket import *
client=socket(AF_INET, SOCK_DGRAM)
client.sendto("Hello World from client",
("192.168.11.11", 12345))
client.close()
```

Raw Sockets

Raw sockets allow you to send data using existing protocols. In addition, you can also create your own data transfer protocols. This capability is possible because you can use raw sockets to build, from scratch, sections of a protocol data unit, or even a full protocol data unit.

Note that there are some limitations when using raw sockets. Specifically, if you use the SOCK_RAW socket type with the AF_NET domain, you can easily develop a new transport layer protocol that will continue to rely on IPv4, and lower level protocols to be carried over the network. In this case you would be forced to use the IP addresses and MAC addresses assigned to the machine network interface as the source of the packets sent over the network.

To alleviate the limitation, use the AF_PACKET domain to build the whole packet from scratch. Note you must calculate and fill some fields that would normally be handled by the network stack, such as the checksum fields, length fields, padding, and flags.

Raw Sockets Server Side

```
from socket import *
server=socket(AF_INET, SOCK_RAW, 102)
server.bind(("192.168.11.13",                    0))
client_data,address=server.recvfrom(100)
print(client_data)
server.close()
```

Raw Sockets Client Side

```
from socket import *
client=socket(AF_INET, SOCK_RAW, 102)
client.sendto("Hello     World     from     client",
("192.168.11.13", 0))
client.close()
```

Scapy

Scapy is an advanced packet manipulation tool for computer networks that allows you to painlessly perform all operations that can be accomplished using sockets. Scapy can be accessed via the Python interactive shell or via scripting.

Scapy Command	Description
`ls()`	View all the available protocols within Scapy
`lsc()`	List all available functions in Scapy
`pkt = IP()`	create an IP packet with default field
`pkt.show()`	View the fields of a packet
`pkt.dst="192.168.11.13"`	Set destination address of an IP packet to `192.168.11.13`

`pkt=IP(dst="192.168.11.13")`	Create a IP packet and set destination address to `192.168.11.13`
`pkt=pkt/TCP()`	Insert/Attach a TCP payload to a packet
`pkt=pkt/Raw("raw data")`	Insert/Attach a Raw data payload to a packet
`pkt["TCP"].dport=8080`	Set the destination port of the TCP portion of a packet to `8080`
`pkt.dport=8080`	
`pkt["TCP"].sport=12345`	Set the source port of the TCP portion of a packet to `12345`
`pkt.sport=12345`	
`pkt["Raw"].load = "Helloworld"`	Set the payload of the Raw portion of a packet to `"Helloworld"`
`pkt.load = "Helloworld"`	
`value = str(pkt)`	Encode packet (Retrieve the hex/string version of a packet).
`pkt = IP(value)`	Decode data to a IP packet (Create an IP packet from a string/hex dump. Any Scapy object can be created from a string in this manner by using the appropriate constructor).
`send(pkt)`	Send a packet at layer 3 (packet must be full layer 3 packet)
`psend(pkt)`	Send a packet at layer 2 (packet must be full layer 2 packet)
`sr(pkt, filter="udp port 53", timeout=10, count=10)`	Send and receive packets at layer 3. A BPF filter for replies may be specified with the `filter` argument. The number of times the packet should be sent can be specified with the `count` argument. The waiting time after the last packet is may be set with the `timeout` argument. To learn about additional arguments run `help(sr)` from Scapy command line.
`srp(pkt, filter="ether host 00:50:56:92:71:2e", timeout=10, count=10)`	Send and receive packets at layer 2. A BPF filter for replies may be specified with the `filter` argument. You can specify to accept multiple answers for the same stimulus with the `multi` argumentThe waiting time after the last packet is may be set with the `timeout` argument. To learn about additional arguments run `help(srp)` from Scapy command line.
`sr1(pkt, filter="udp port 53", timeout=10, multi=False)`	Send packets at layer 3 and return only the first reply. A BPF filter for replies may be specified with the `filter` argument. You can specify to accept multiple answers for the same stimulus with the `multi` argument. The waiting time after the last packet is may be set with the `timeout` argument. To learn about additional arguments run `help(sr1)` from Scapy command line.
`srp1(pkt, filter="udp port 53", timeout=10, multi=False)`	Send packets at layer 2 and return only the first reply. A BPF filter for replies may be specified with the `filter` argument. You can specify to accept multiple answers for the same stimulus with the `multi` argument. The waiting time after the last packet is sent may be set with the `timeout` argument. To learn about additional arguments run `help(srp1)` from Scapy command line.
`srloop(pkt, count=15)`	Send a packet in and print each reply. The packet will be sent a number of times determined by the value of the `count` argument. To learn about additional arguments run `help(srloop)` from Scapy command line.
`pkts = sniff(count=100,filter="tcp port 110" timeout=10)`	Sniff packets on the wire at layer 2. The number of packets to sniff may be specified with the `count` argument. The duration of the capture may be specified with the `timeout` argument. A BPF filter may be specified with the `filter` argument. To learn about additional arguments run `help(sniff)` from Scapy command line.

Recommended Internet Sites

- Sockets: https://docs.python.org/2/library/socket.html
- Scapy: http://www.workrobot.com/sansfire2009/SCAPY-packet-crafting-reference.html
- Scapy: http://www.secdev.org/projects/scapy/doc/usage.html