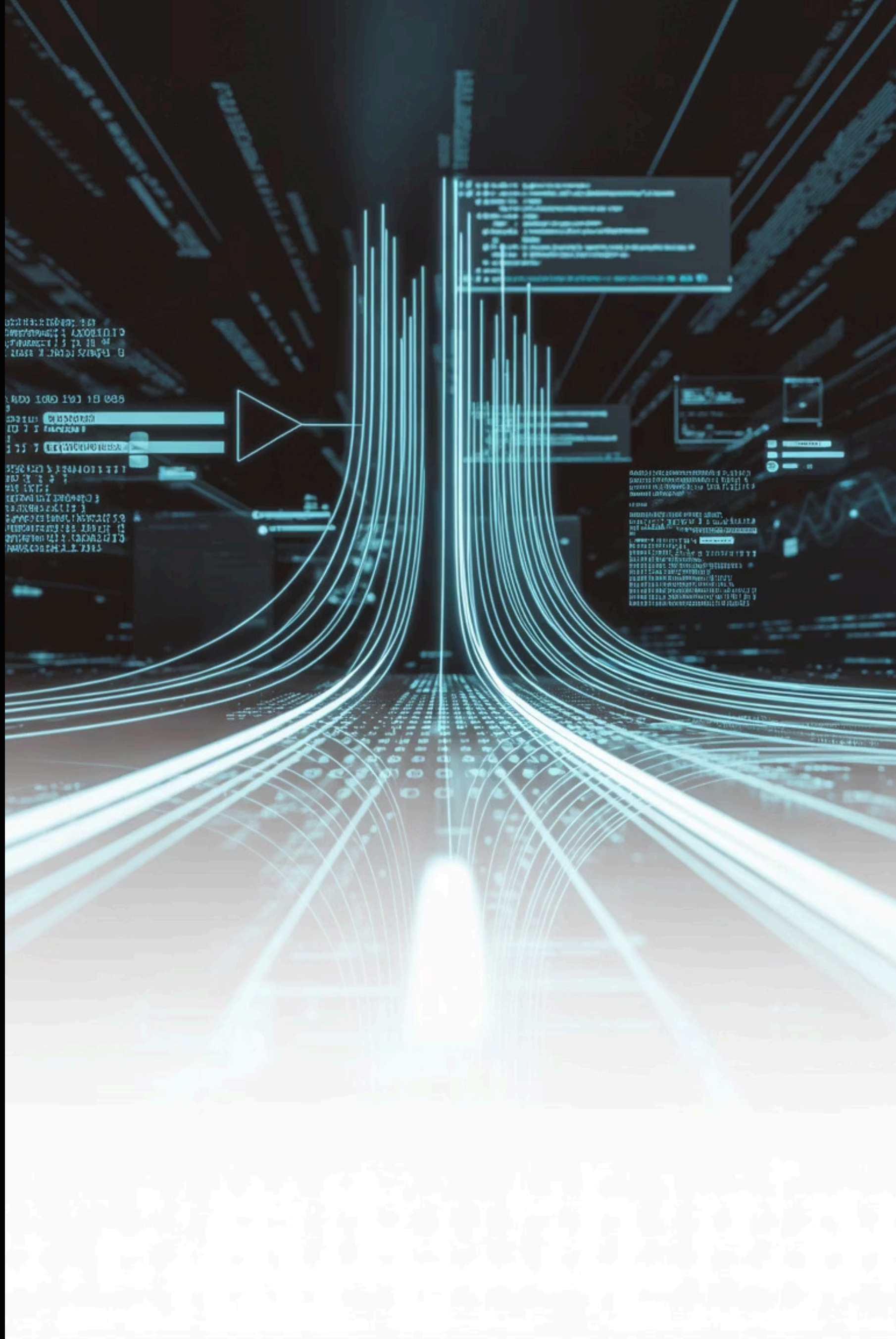
Linux Observability and Tuning using bpftrace

Module 4: Networking & Security Observability

Instructor: Chandrashekar Babu <training@chandrashekar.info>



Agenda

Network Tracing Fundamentals

Understanding socket operations, TCP/UDP tracing basics, and network stack instrumentation points

Advanced Network Observability

Deep packet inspection, NIC queue monitoring, and network performance analytics

Security Monitoring with bpftrace

Detecting suspicious behavior, unauthorized connection attempts, and monitoring process activities

Hands-On Labs

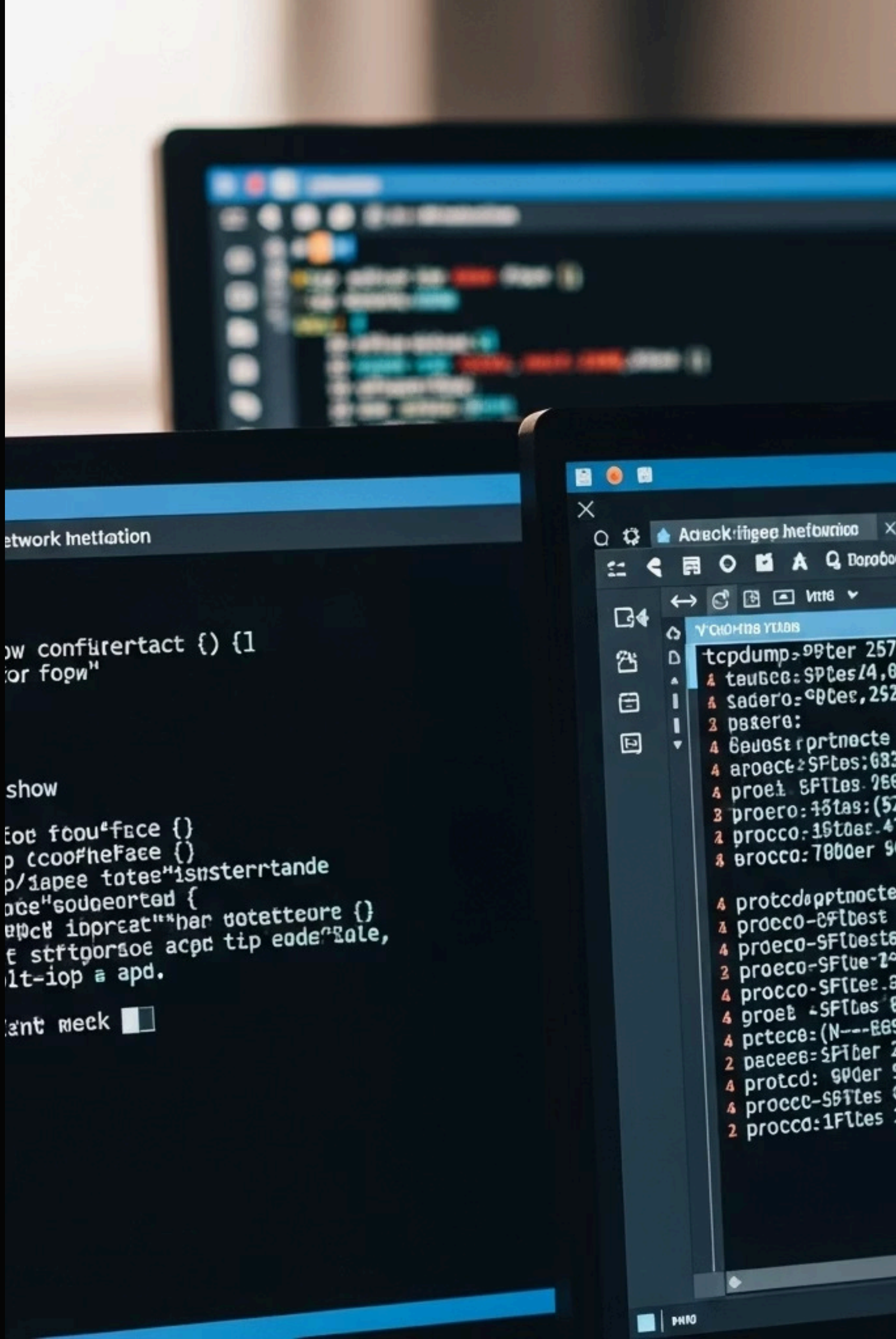
Practical exercises implementing network and security monitoring solutions with bpftrace

Network Tracing: Linux Networking Basics

Before diving into bpftrace, let's review where we can hook into the Linux networking stack:

- **Socket Layer**: High-level interface used by applications
- **Transport Layer**: TCP/UDP protocol implementations
- **Network Layer**: IP routing and forwarding
- **Link Layer**: Device drivers and physical transmission

bpftrace gives us visibility at all these layers, from socket operations down to device driver interactions.



Key bpftrace Probe Points for Networking

Socket Operations

Tracepoints:

- `sock:*`
- `syscall:socket*`
- `syscall:connect`
- `syscall:accept*`

TCP Operations

Tracepoints:

- `tcp:*`
- `net:tcp_*`
- `kprobe:tcp_*`

Packet Processing

Tracepoints:

- `net:netif_receive_skb`
- `net:net_dev_xmit`
- `kprobe:dev_queue_xmit`

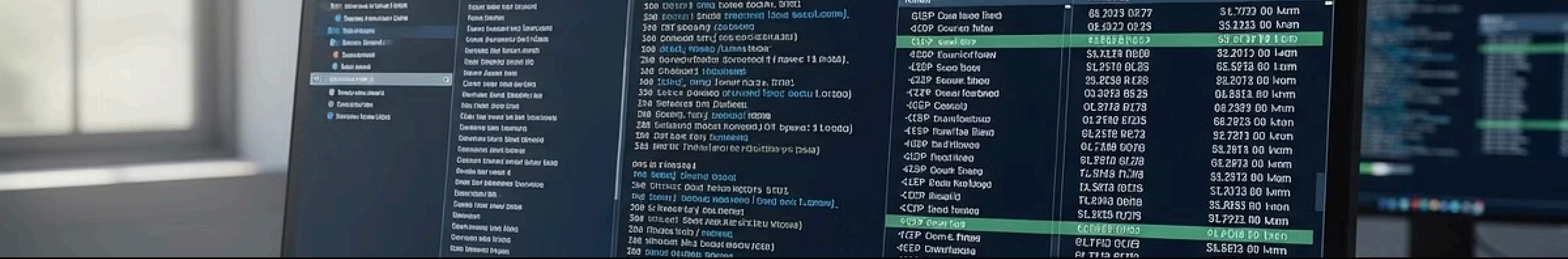
These provide the foundation for our networking observability scripts.

Basic Network Connection Monitoring

Let's start with a simple bpftrace one-liner to monitor new TCP connections:

```
#!/usr/bin/env bpftrace

tracepoint:syscalls:sys_enter_connect
{
    $user_addr = arg2;  // struct sockaddr * uservaddr
    $family = *(uint16 *)($user_addr);

    if ($family == 2) {  // AF_INET
        $port_be = *(uint16 *)($user_addr + 2);
        $ip = *(uint32 *)($user_addr + 4);

        // Convert network byte order to host byte order for port
        $port = (($port_be >> 8) & 0xFF) | (($port_be & 0xFF) << 8);

        printf("CONNECT: PID %d (%s) -> %s:%d\n",
            pid, comm,
            ntop($ip),
            $port);
    }
}
```

This one-liner tracks all connection attempts, showing the process name, PID, and destination IP/port.



Tracking TCP Connection Lifetimes

The `tcplife.bt` script tracks the entire lifecycle of TCP connections:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("%-5s %-10s %-15s %-5s %-15s %-5s %s\n",
 "PID", "COMM", "LADDR", "LPORT", "RADDR", "RPORT", "MS");
}

kprobe:tcp_set_state
{
 $sk = (struct sock *)arg0;
 $newstate = arg1;

 if ($newstate == 1) {
 @start[$sk] = nsecs;
 }
}

kprobe:tcp_close
{
 $sk = (struct sock *)arg0;
 $delta = nsecs - @start[$sk];
 $durationms = $delta / 1000000;

 $family = $sk->__sk_common.skc_family;

 if ($family == 2) {
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
 $lport = $sk->__sk_common.skc_num;
 $dport = $sk->__sk_common.skc_dport;

 printf("%-5d %-10s %-15s %-5d %-15s %-5d %d\n",
 pid, comm, $saddr, $lport, $daddr, $dport, $durationms);
 }
 delete(@start[$sk]);
}
```

Socket Connection Statistics

This one-liner counts active socket connections by process name:

```
bpftrace -e 'tracepoint:syscalls:sys_enter_connect
  { @connects[comm] = count(); }
  interval:s:5
  { print(@connects); clear(@connects); }'
```

For more detailed statistics including duration, use:

```
bpftrace -e 'kprobe:tcp_connect { @start[arg0] = nsecs; }
  kretprobe:tcp_connect /@start[arg0]/ {
    @duration_ns = hist(nsecs - @start[arg0]);
    delete(@start[arg0]);
  }'
```

This shows a histogram of TCP connection durations, helping identify performance issues.

Monitoring TCP Retransmits

TCP retransmissions can indicate network problems. Track them with:

```
bpftrace -e 'kprobe:tcp_retransmit_skb {
  @[pid, comm] = count();
}'
```

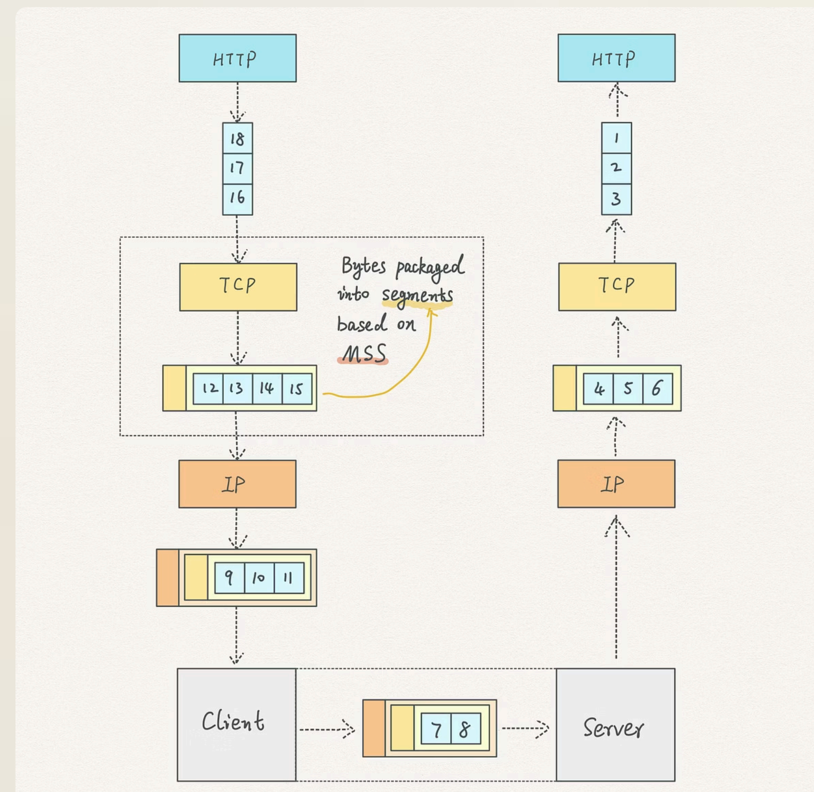
For more context about what's being retransmitted:

```
bpftrace -e 'kprobe:tcp_retransmit_skb {
 $sk = (struct sock *)arg0;
 $inet_family = $sk->__sk_common.skc_family;
 if ($inet_family == AF_INET || $inet_family == AF_INET6) {
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
 printf("Retransmit: %s -> %s (PID: %d, %s)\n",
 $saddr, $daddr, pid, comm);
 }
}'
```

Detailed TCP Retransmit Analysis

A more comprehensive script for detailed retransmit analysis:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Tracing TCP retransmits... Hit Ctrl-C to end.\n");
 printf("%-8s %-8s %-16s %-16s %-5s %-5s %s\n",
 "TIME", "PID", "SOURCE", "DESTINATION", "SPORT", "DPORT", "STATE");
}

kprobe:tcp_retransmit_skb {
 $sk = (struct sock *)arg0;
 $inet_family = $sk->__sk_common.skc_family;

 if ($inet_family == AF_INET || $inet_family == AF_INET6) {
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
 $lport = $sk->__sk_common.skc_num;
 $dport = ntohs($sk->__sk_common.skc_dport);
 $state = $sk->__sk_common.skc_state;

 time("%H:%M:%S ");
 printf("%-8d %-16s %-16s %-5d %-5d %d\n",
 pid, $saddr, $daddr, $lport, $dport, $state);

 @retransmits[$saddr, $daddr] = count();
 }
}

END {
 printf("\nRetransmit counts by connection:\n");
 print(@retransmits);
}
```



Measuring TCP RTT (Round Trip Time)

This script measures TCP round-trip time by tracking ACK packets:

```
#!/usr/bin/env bpftrace

#include <linux/socket.h>
#include <net/inet_sock.h>

BEGIN {
 printf("Measuring TCP RTT...\n");
}

kprobe:tcp_transmit_skb {
 $sk = (struct sock *)arg0;
 $seq = $sk->tcp_sk.snd_nxt;
 @start[$sk, $seq] = nsecs;
}

kprobe:tcp_ack {
 $sk = (struct sock *)arg0;
 $seq = arg1;
 $start = @start[$sk, $seq];

 if ($start) {
 $rtt = (nsecs - $start) / 1000000; // ms

 // Only record if valid RTT
 if ($rtt > 0 && $rtt < 10000) {
 // Get connection details
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
 $lport = $sk->__sk_common.skc_num;
 $dport = ntohs($sk->__sk_common.skc_dport);

 printf("RTT: %6d ms (%s:%d -> %s:%d)\n",
 $rtt, $saddr, $lport, $daddr, $dport);

 // Record histogram by destination
 @rtt_hist[$daddr] = hist($rtt);
 }
 delete(@start[$sk, $seq]);
 }
}

interval:s:10 {
 print(@rtt_hist);
 clear(@rtt_hist);
}
```

TCP SYN Flood Detection

Detect potential SYN flood attacks with this script:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring for SYN flood attacks...\n");
 printf("Threshold: >100 SYN packets per second from a single source\n\n");
}

tracepoint:net:netif_receive_skb {
 $skb = (struct sk_buff *)args->skb;
 $ip = (struct iphdr *)($skb->head + $skb->network_header);

 // Check if it's an IPv4 packet
 if ($ip->version == 4) {
   // Extract the TCP header
   $tcp = (struct tcphdr *)($skb->head + $skb->transport_header);

   // Check if it's a SYN packet (SYN flag set, ACK flag not set)
   if (($tcp->syn == 1) && ($tcp->ack == 0)) {
      $saddr = ntop($ip->saddr);
      @syn_count[$saddr] = count();
    }
 }
}

interval:s:1 {
 // Check for potential SYN flood (high number of SYN packets from same source)
 printf("--- %s ---\n", strftime("%H:%M:%S", nsecs));

 foreach ($saddr in @syn_count) {
    $count = @syn_count[$saddr];
    if ($count > 100) {
       printf("ALERT: Possible SYN flood from %s (%d SYN packets/sec)\n",
     $saddr, $count);
  }
 }

 print(@syn_count);
 clear(@syn_count);
}
```

Monitoring Network Socket Buffers

Track socket buffer usage to identify potential bottlenecks:

```
bpftrace -e 'kprobe:sock_alloc_send_pskb {
  @bytes[comm] = sum(arg1);
}
interval:s:5 {
  printf("Socket send buffer allocation by process:\n");
  print(@bytes);
  clear(@bytes);
}'
```

To monitor socket receive buffers:

```
bpftrace -e 'kprobe:__netif_receive_skb_core {
 $skb = (struct sk_buff *)arg0;
 @recv_bytes[cpu] = sum($skb->len);
}
interval:s:5 {
 printf("Bytes received per CPU:\n");
 print(@recv_bytes);
 clear(@recv_bytes);
}'
```

DNS Query Monitoring

Track DNS queries on your system with this script:

```
#!/usr/bin/env bpftrace

#include <linux/socket.h>
#include <net/sock.h>
#include <linux/types.h>

BEGIN {
 printf("Tracing DNS queries... Hit Ctrl-C to end.\n");
}

// Catch UDP packets on port 53 (DNS)
kprobe:udp_sendmsg {
 $sk = (struct sock *)arg0;
 $dport = (uint16)($sk->__sk_common.skc_dport);

 // Check if it's to port 53 (DNS)
 if (ntohs($dport) == 53) {
 $daddr = ntop($sk->__sk_common.skc_daddr);
 printf("%-6d %-16s DNS query to %s (resolver)\n",
 pid, comm, $daddr);
 @dns_queries[comm] = count();
 }
}

END {
 printf("\nDNS query counts by process:\n");
 print(@dns_queries);
}
```

Inspecting HTTP Request Headers

This advanced script detects and parses HTTP headers in network traffic:

```
#!/usr/bin/env bpftrace

#include <linux/socket.h>
#include <net/sock.h>
#include <linux/types.h>

BEGIN {
 printf("Monitoring HTTP requests... Hit Ctrl-C to end.\n");
 printf("%-6s %-16s %-16s %-5s %-5s %s\n",
 "PID", "COMM", "DSTIP", "PORT", "BYTES", "PATH");
}

kprobe:tcp_sendmsg {
 $sk = (struct sock *)arg0;
 $size = arg2;

 // Only process packets with a reasonable HTTP request size
 if ($size > 10 && $size < 1000) {
    // Get source buffer
    $iovbase = (char *)((struct iovec *)arg1)->iov_base;

    // Look for HTTP request pattern (GET, POST, PUT, etc.)
    $method = str($iovbase, 4);

    if (($method == "GET " || $method == "POST" ||
      $method == "PUT " || $method == "HEAD")) {

      // Extract destination IP and port
      $daddr = ntop($sk->__sk_common.skc_daddr);
      $dport = ntohs($sk->__sk_common.skc_dport);

      // If it's to standard HTTP/HTTPS ports
      if ($dport == 80 || $dport == 443 || $dport == 8080) {
        // Try to extract the request path
        $path_start = 0;
        $space_count = 0;

        // Find the second space which comes after the path
        for ($i = 0; $i < 80 && $i < $size; $i++) {
            if ($iovbase[$i] == ' ') {
              $space_count++;
              if ($space_count == 1) {
                  $path_start = $i + 1;
              } else if ($space_count == 2) {
                  // Found the end of the path
                  $path = str($iovbase + $path_start, $i - $path_start);
                  printf("%-6d %-16s %-16s %-5d %-5d %s\n",
                         pid, comm, $daddr, $dport, $size, $path);
                  break;
              }
            }
        }

        @http_reqs[comm, $daddr] = count();
      }
    }
 }
}

END {
 printf("\nHTTP request counts by process and destination:\n");
 print(@http_reqs);
}
```

NIC Queue Monitoring

This one-liner tracks NIC transmit queue lengths:

```
bpftrace -e 'kprobe:__netdev_pick_tx {
  @qlen[arg0->name] = hist(arg0->tx_queue_len);
}'
```

A more advanced script for monitoring NIC queue health:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring NIC queue metrics... Hit Ctrl-C to end.\n");
}

// Track queue fill level on transmit
kprobe:__dev_queue_xmit {
 $skb = (struct sk_buff *)arg0;
 $dev = $skb->dev;
 if ($dev) {
   @tx_qlen[$dev->name] = hist($dev->tx_queue_len);
   @tx_bytes[$dev->name] = sum($skb->len);
 }
}

// Track packet drops
kprobe:dev_hard_start_xmit {
 $ret = retval;
 if ($ret == -1) {
   $skb = (struct sk_buff *)arg0;
   $dev = $skb->dev;
   @drops[$dev->name] = count();
 }
}

interval:s:5 {
 time("%H:%M:%S\n");
 printf("NIC transmit queue lengths:\n");
 print(@tx_qlen);
 printf("\nBytes transmitted per interface:\n");
 print(@tx_bytes);
 printf("\nDropped packets per interface:\n");
 print(@drops);
 clear(@tx_qlen);
 clear(@tx_bytes);
}
```

TCP Receive Window Analysis

This script monitors TCP receive window sizes to identify flow control issues:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Analyzing TCP receive windows... Hit Ctrl-C to end.\n");
}

kprobe:tcp_rcv_established {
 $sk = (struct sock *)arg0;
 $tp = (struct tcp_sock *)$sk;

 // Calculate current receive window in bytes
 $rcv_wnd = $tp->rcv_wnd;
 $window_scaling = 1 << $tp->rx_opt.rcv_wscale;
 $effective_window = $rcv_wnd * $window_scaling;

 // Get connection details
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $saddr = ntop($sk->__sk_common.skc_rcv_saddr);

 // Record histograms of receive window sizes
 @rcv_wnd_bytes[$daddr] = hist($effective_window);

 // Detect small windows (potential bottleneck)
 if ($effective_window < 4096) {
 printf("Small window alert: %s->%s window: %d bytes\n",
 $saddr, $daddr, $effective_window);
 @small_windows[$saddr, $daddr] = count();
 }
}

interval:s:10 {
 printf("\n=== TCP Receive Window Analysis ===\n");
 printf("Receive window size distributions by remote host:\n");
 print(@rcv_wnd_bytes);

 if (@small_windows) {
 printf("\nConnections with small receive windows:\n");
 print(@small_windows);
 }

 clear(@rcv_wnd_bytes);
 clear(@small_windows);
}
```

Network Device Driver I/O

This script profiles network device driver activity:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Tracing network device driver I/O... Hit Ctrl-C to end.\n");
}

// Trace network packet receipt
kprobe:netif_receive_skb {
 @receive_stack[kstack] = count();
 @dev_rx[arg0->dev->name] = count();
}

// Trace network packet transmission
kprobe:dev_hard_start_xmit {
 @transmit_stack[kstack] = count();
 @dev_tx[arg0->dev->name] = count();
}

interval:s:5 {
 printf("\n=== Network I/O by device ===\n");
 printf("Packets received:\n");
 print(@dev_rx);
 printf("\nPackets transmitted:\n");
 print(@dev_tx);

 clear(@dev_rx);
 clear(@dev_tx);
}

END {
 printf("\n=== Top packet receive stacks ===\n");
 print(@receive_stack, 5);
 printf("\n=== Top packet transmit stacks ===\n");
 print(@transmit_stack, 5);
}
```

Socket Open & Close Tracking

Track socket lifecycle with this one-liner:

```
bpftrace -e 'tracepoint:syscalls:sys_enter_socket { @opens[comm] = count(); }
  tracepoint:syscalls:sys_enter_close /@socket_fds[tid]/ {
    @closes[comm] = count();
    delete(@socket_fds[tid]);
  }
  tracepoint:syscalls:sys_exit_socket /retval >= 0/ {
    @socket_fds[tid] = retval;
  }'
```

For a more detailed view of socket operations by protocol:

```
bpftrace -e 'tracepoint:syscalls:sys_enter_socket {
 @socket_by_type[arg0, arg1, arg2] = count();
}
interval:s:5 {
 printf("Socket creation by (domain, type, protocol):\n");
 print(@socket_by_type);
 clear(@socket_by_type);
}'
```

Connection Tracking by Process

This comprehensive script tracks connections by process name:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Tracing connection activity by process... Hit Ctrl-C to end.\n");
}

// Track socket creation
tracepoint:syscalls:sys_enter_socket {
 @socket_ops[pid, comm, "create"] = count();
}

// Track connection attempts
tracepoint:syscalls:sys_enter_connect {
 $sa = (struct sockaddr *)args->uservaddr;
 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $daddr = ntop($in->sin_addr.s_addr);
 $dport = ntohs($in->sin_port);

 printf("%-6d %-16s connect %s:%d\n", pid, comm, $daddr, $dport);
 @connect_to[comm, $daddr, $dport] = count();
 @socket_ops[pid, comm, "connect"] = count();
 }
}

// Track bind operations
tracepoint:syscalls:sys_enter_bind {
 $sa = (struct sockaddr *)args->umyaddr;
 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $addr = ntop($in->sin_addr.s_addr);
 $port = ntohs($in->sin_port);

 printf("%-6d %-16s bind %s:%d\n", pid, comm, $addr, $port);
 @bind_to[comm, $addr, $port] = count();
 @socket_ops[pid, comm, "bind"] = count();
 }
}

// Track listening sockets
tracepoint:syscalls:sys_enter_listen {
 @socket_ops[pid, comm, "listen"] = count();
}

// Track accept operations
tracepoint:syscalls:sys_enter_accept {
 @socket_ops[pid, comm, "accept"] = count();
}

// Track socket close
tracepoint:syscalls:sys_enter_close /@socket_fds[tid]/ {
 @socket_ops[pid, comm, "close"] = count();
 delete(@socket_fds[tid]);
}

tracepoint:syscalls:sys_exit_socket /args->ret >= 0/ {
 @socket_fds[tid] = args->ret;
}

interval:s:10 {
 printf("\n=== Socket operations by process ===\n");
 print(@socket_ops);

 printf("\n=== Connection attempts by destination ===\n");
 print(@connect_to);

 printf("\n=== Bind operations by address ===\n");
 print(@bind_to);

 clear(@socket_ops);
 clear(@connect_to);
 clear(@bind_to);
}
```

Security Monitoring: Unauthorized Process Activity

This script detects suspicious network activity by unauthorized processes:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring for unauthorized network activity...\n");

 // Define allowed network binaries - customize for your environment
 @allowed_net_bins["curl"] = 1;
 @allowed_net_bins["wget"] = 1;
 @allowed_net_bins["ssh"] = 1;
 @allowed_net_bins["scp"] = 1;
 @allowed_net_bins["rsync"] = 1;
 @allowed_net_bins["nc"] = 1;
 @allowed_net_bins["netcat"] = 1;
 @allowed_net_bins["firefox"] = 1;
 @allowed_net_bins["chrome"] = 1;
 @allowed_net_bins["chromium"] = 1;
}

tracepoint:syscalls:sys_enter_connect {
 $sa = (struct sockaddr *)args->uservaddr;

 // Only monitor IPv4 and IPv6 connections
 if ($sa->sa_family == AF_INET || $sa->sa_family == AF_INET6) {
 // Check if this is an allowed binary
 if (@allowed_net_bins[comm] != 1) {
 // This is a potential unauthorized connection
 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $daddr = ntop($in->sin_addr.s_addr);
 $dport = ntohs($in->sin_port);

 printf("ALERT: Unauthorized connection attempt by %s (PID %d): %s:%d\n",
 comm, pid, $daddr, $dport);
 @unauth_conns[comm, $daddr, $dport] = count();

 // Record stack trace for investigation
 @stacks[comm, pid] = ustack;
 }
 }
 }
}

interval:s:10 {
 printf("\n=== Unauthorized connection summary ===\n");
 print(@unauth_conns);

 if (@stacks) {
 printf("\n=== Stack traces for investigation ===\n");
 print(@stacks);
 clear(@stacks);
 }
}
```

Network Data Exfiltration Detection

Detect possible data exfiltration with large outbound transfers:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring for potential data exfiltration...\n");
 printf("Threshold: >10MB outbound in 1 minute\n");

 // Whitelist known data transfer processes - customize for your environment
 @allowed_data_xfer["rsync"] = 1;
 @allowed_data_xfer["scp"] = 1;
 @allowed_data_xfer["sftp"] = 1;
 @allowed_data_xfer["backup"] = 1;
}

kprobe:tcp_sendmsg {
 $sk = (struct sock *)arg0;
 $size = arg2;

 // Only track outbound connections (not localhost)
 if ($sk->__sk_common.skc_daddr != 0x0100007F && // 127.0.0.1
 $sk->__sk_common.skc_daddr != 0) {

 $daddr = ntop($sk->__sk_common.skc_daddr);
 $dport = ntohs($sk->__sk_common.skc_dport);

 // Track bytes by process
 @bytes[comm, $daddr, $dport] += $size;

 // Alert on large single writes
 if ($size > 1000000 && @allowed_data_xfer[comm] != 1) {
 printf("Large send: %s (PID %d) sending %d bytes to %s:%d\n",
 comm, pid, $size, $daddr, $dport);
 }
 }
}

interval:s:60 {
 printf("\n=== Checking for suspicious data transfers ===\n");

 // Detect potential exfiltration by checking total bytes transferred
 foreach ([$comm, $daddr, $dport] in @bytes) {
 $total = @bytes[$comm, $daddr, $dport];

 // Alert on large total transfers by non-whitelisted processes
 if ($total > 10000000 && @allowed_data_xfer[$comm] != 1) {
 printf("ALERT: Possible data exfiltration by %s to %s:%d (%d bytes)\n",
 $comm, $daddr, $dport, $total);
 }
 }

 printf("\n=== Data transfer summary (bytes) ===\n");
 print(@bytes);
 clear(@bytes);
}
```

Port Scanning Detection

This script identifies potential port scanning activity:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring for port scanning activity...\n");
 printf("Threshold: >15 different ports in 5 seconds from same source\n");
}

tracepoint:syscalls:sys_enter_connect {
 $sa = (struct sockaddr *)args->uservaddr;

 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $daddr = ntop($in->sin_addr.s_addr);
 $dport = ntohs($in->sin_port);

 // Track unique destinations by source process
 @dest_ports[pid, comm, $daddr, $dport] = count();

 // Count unique ports per destination
 @port_count[pid, comm, $daddr] = count();
 }
}

interval:s:5 {
 printf("\n=== Checking for port scan activity ===\n");

 // Check for processes connecting to many ports on the same host
 foreach ([$pid, $comm, $daddr] in @port_count) {
 $count = @port_count[$pid, $comm, $daddr];

 if ($count > 15) {
 printf("ALERT: Possible port scan by %s (PID %d) to %s (%d ports)\n",
 $comm, $pid, $daddr, $count);

 // Detailed breakdown of ports accessed
 printf("Ports accessed:\n");
 foreach ([$pid2, $comm2, $daddr2, $dport] in @dest_ports) {
 if ($pid2 == $pid && $daddr2 == $daddr) {
 printf(" %d\n", $dport);
 }
 }
 }
 }

 // Only clear counters after alert generation
 clear(@dest_ports);
 clear(@port_count);
}
```

Detecting Suspicious DNS Queries

This script monitors for DNS tunneling and other suspicious DNS activities:

```
#!/usr/bin/env bpftrace

#include
#include
#include
#include

BEGIN {
 printf("Monitoring for suspicious DNS activity...\n");
 printf("Checking for: long queries, high volume, and encoded data\n");
}

// Examine DNS packets (UDP port 53)
kprobe:udp_sendmsg {
 $sk = (struct sock *)arg0;
 $data = arg1;
 $size = arg2;
 $dport = ntohs($sk->__sk_common.skc_dport);

 // Only process DNS packets
 if ($dport == 53) {
 $daddr = ntop($sk->__sk_common.skc_daddr);

 // Count queries by process
 @dns_count[comm] = count();

 // Track total bytes (to detect data exfiltration)
 @dns_bytes[comm] += $size;

 // Basic checks for suspicious DNS activity
 if ($size > 100) {
 // Unusually large DNS query
 printf("Large DNS query (%d bytes) from %s (PID %d) to %s\n",
 $size, comm, pid, $daddr);
 @large_queries[comm, $daddr] = count();
 }
 }
}

interval:s:10 {
 printf("\n=== DNS Activity Analysis ===\n");

 // Check for high volume DNS queries (potential tunneling)
 foreach ($comm in @dns_count) {
 $count = @dns_count[$comm];
 $bytes = @dns_bytes[$comm];

 if ($count > 100) {
 printf("ALERT: High volume DNS queries from %s: %d queries, %d bytes\n",
 $comm, $count, $bytes);
 }
 }

 printf("\nDNS queries by process:\n");
 print(@dns_count);

 printf("\nLarge DNS queries (potential data encoding):\n");
 print(@large_queries);

 clear(@dns_count);
 clear(@dns_bytes);
}
```

Process Network Isolation Monitoring

Track processes that should not be making network connections:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring network activity by isolated processes...\n");

 // Define processes that should not make network connections
 // Add your specific processes here
 @isolated_procs["mysql"] = 1;
 @isolated_procs["postgres"] = 1;
 @isolated_procs["redis-server"] = 1;
}

tracepoint:syscalls:sys_enter_socket,
tracepoint:syscalls:sys_enter_connect,
tracepoint:syscalls:sys_enter_accept,
tracepoint:syscalls:sys_enter_sendto,
tracepoint:syscalls:sys_enter_recvfrom
{
 // Check if this is an isolated process
 if (@isolated_procs[comm] == 1) {
 printf("ALERT: Isolated process %s (PID %d) attempting network operation: %s\n",
 comm, pid, probe);
 @violations[comm, probe] = count();
 @stacks[comm, pid, probe] = ustack;
 }
}

interval:s:30 {
 if (@violations) {
 printf("\n=== Network isolation violations ===\n");
 print(@violations);

 printf("\n=== Stack traces for investigation ===\n");
 print(@stacks);
 clear(@stacks);
 }
}
```

Socket Permission Violation Detection

Detect socket operations with insufficient permissions:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring for socket permission violations...\n");
}

// Define a helper function to check uid/capability
// Note: this is simplified and should be enhanced in real scenarios
function check_socket_perm(prot, op) {
 // For demonstration - this should be customized for your environment

 // Check for privileged ports (<1024) being used by non-root
 if (((uint16)arg1 < 1024) && (uid != 0)) {
 printf("ALERT: Non-root user (UID %d) attempting to %s on privileged port %d\n",
 uid, op, (uint16)arg1);
 @priv_port_violations[comm, uid, (uint16)arg1, op] = count();
 return 1;
 }

 // Check for raw sockets (requires CAP_NET_RAW)
 if (prot == 3 && uid != 0) { // IPPROTO_RAW = 3
 printf("ALERT: Non-root user (UID %d) attempting to create raw socket\n", uid);
 @raw_socket_violations[comm, uid] = count();
 return 1;
 }

 return 0;
}

tracepoint:syscalls:sys_enter_socket {
 $domain = args->family;
 $type = args->type;
 $protocol = args->protocol;

 // Check permissions for socket creation
 check_socket_perm($protocol, "create socket");
}

tracepoint:syscalls:sys_enter_bind {
 $sa = (struct sockaddr *)args->umyaddr;

 // For IPv4 sockets
 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $port = ntohs($in->sin_port);

 // Check permissions for binding
 if ($port < 1024 && uid != 0) {
 printf("ALERT: Non-root user (UID %d) attempting to bind to privileged port %d\n",
 uid, $port);
 @bind_violations[comm, uid, $port] = count();
 }
 }
}

END {
 printf("\n=== Socket permission violation summary ===\n");
 printf("Privileged port violations:\n");
 print(@priv_port_violations);

 printf("\nRaw socket violations:\n");
 print(@raw_socket_violations);

 printf("\nBind violations:\n");
 print(@bind_violations);
}
```

Monitoring File Descriptor Sharing

This script tracks file descriptor sharing between processes, which can include socket sharing:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring for socket descriptor sharing between processes...\n");
}

// Track process relationships
tracepoint:syscalls:sys_enter_fork,
tracepoint:syscalls:sys_enter_vfork,
tracepoint:syscalls:sys_enter_clone {
 @parent[tid] = pid;
}

// Track socket creations
tracepoint:syscalls:sys_exit_socket /args->ret >= 0/ {
 $fd = args->ret;
 @socket_owner[$fd, pid] = comm;
}

// Track socket sharing via sendmsg with SCM_RIGHTS
kprobe:__sys_sendmsg {
 $msghdr = (struct msghdr *)arg1;
 $control = $msghdr->msg_control;

 // If control data exists, it might contain SCM_RIGHTS
 if ($control != 0) {
 // This is a simplification - actual SCM_RIGHTS detection would require
 // more detailed parsing of the cmsghdr structure
 printf("Potential FD sharing: %s (PID %d) sending control message\n",
 comm, pid);
 @potential_fd_sharing[comm, pid] = count();
 }
}

// Track when a process uses a socket it didn't create
kprobe:sock_sendmsg,
kprobe:sock_recvmsg {
 $sock = (struct socket *)arg0;
 $owner_pid = 0;
 $owner_found = 0;

 // Search for the real owner
 // This is simplified - in reality, we'd need to track FDs throughout their lifecycle
 foreach ([$fd, $opid] in @socket_owner) {
 if ($opid != pid) {
 $owner_found = 1;
 $owner_pid = $opid;
 $owner_comm = @socket_owner[$fd, $opid];

 printf("Socket sharing detected: %s (PID %d) using socket owned by %s (PID %d)\n",
 comm, pid, $owner_comm, $owner_pid);
 @shared_sockets[comm, pid, $owner_comm, $owner_pid] = count();
 break;
 }
 }
}

interval:s:30 {
 printf("\n=== Socket sharing summary ===\n");

 if (@shared_sockets) {
 printf("Processes using sockets they didn't create:\n");
 print(@shared_sockets);
 }

 if (@potential_fd_sharing) {
 printf("\nPotential file descriptor sharing via SCM_RIGHTS:\n");
 print(@potential_fd_sharing);
 }
}
```

Network Connection Profiling

This script builds network connection profiles to detect anomalies:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Building network connection profiles...\n");
 printf("Will alert on deviations from established patterns\n");
}

// Track connection patterns by process
tracepoint:syscalls:sys_enter_connect {
 $sa = (struct sockaddr *)args->uservaddr;

 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $daddr = ntop($in->sin_addr.s_addr);
 $dport = ntohs($in->sin_port);

 // Build profile by process
 @conn_count[comm] = count();
 @dest_by_proc[comm, $daddr] = count();
 @port_by_proc[comm, $dport] = count();

 // Record first seen timestamp if this is a new destination
 if (@first_seen[comm, $daddr] == 0) {
 @first_seen[comm, $daddr] = nsecs;
 }
 }
}

// Analyze data transfer patterns
kprobe:tcp_sendmsg {
 $sk = (struct sock *)arg0;
 $size = arg2;

 if ($sk->__sk_common.skc_family == AF_INET) {
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $dport = ntohs($sk->__sk_common.skc_dport);

 // Track data volumes
 @bytes_sent[comm, $daddr, $dport] += $size;

 // Detect sudden large transfers
 if (!@max_size[comm, $daddr, $dport]) {
 @max_size[comm, $daddr, $dport] = $size;
 } else if ($size > @max_size[comm, $daddr, $dport] * 5) {
 printf("Unusual large transfer: %s sending %d bytes to %s:%d (5x previous max)\n",
 comm, $size, $daddr, $dport);
 @anomalies[comm, "large_transfer", $daddr, $dport] = count();
 } else if ($size > @max_size[comm, $daddr, $dport]) {
 @max_size[comm, $daddr, $dport] = $size;
 }
 }
}

interval:s:60 {
 printf("\n=== Network Connection Profile Analysis ===\n");

 // Calculate hourly rate for new connections
 $now = nsecs;

 foreach ([$comm, $daddr] in @first_seen) {
 $elapsed = ($now - @first_seen[$comm, $daddr]) / 1000000000; // seconds

 if ($elapsed < 3600 && @dest_by_proc[$comm, $daddr] > 10) {
 printf("New destination with high activity: %s connecting to %s (%d times in %.1f min)\n",
 $comm, $daddr, @dest_by_proc[$comm, $daddr], $elapsed/60);
 @anomalies[$comm, "new_high_activity", $daddr] = count();
 }
 }

 // Report anomalies
 if (@anomalies) {
 printf("\nDetected network anomalies:\n");
 print(@anomalies);
 clear(@anomalies);
 }

 // Clean up old records - in real use this would be more sophisticated
 if (@conn_count) {
 clear(@first_seen);
 }
}
```

Socket Bufferbloat Detection

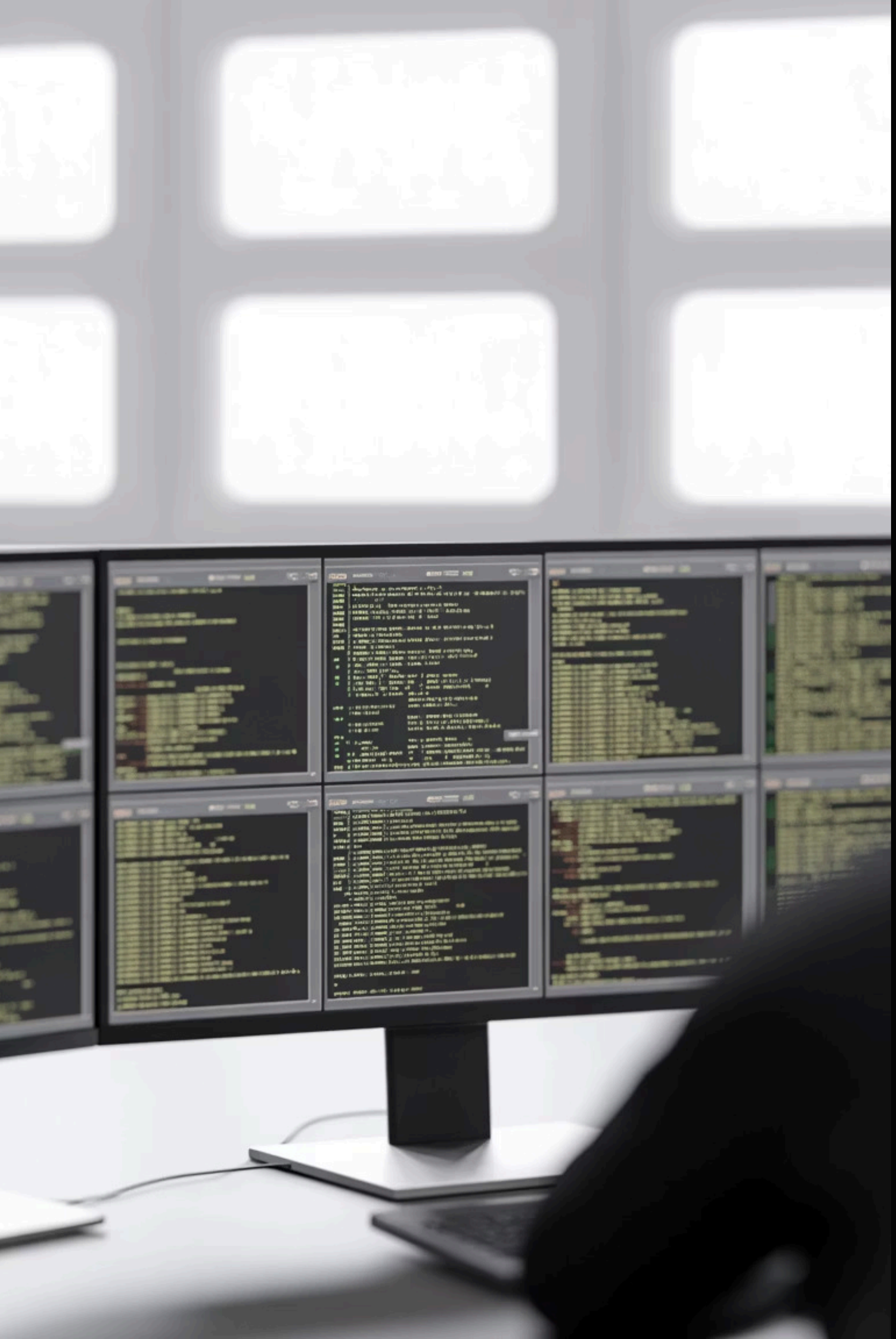
This script identifies socket buffer issues that can lead to latency:

```
#!/usr/bin/env bpftrace

BEGIN {
 printf("Monitoring for socket bufferbloat issues...\n");
}

// Track queued data in socket write buffers
kprobe:sock_alloc_send_pskb {
 $sk = (struct sock *)arg0;
 $size = arg1;

 if ($sk->__sk_common.skc_family == AF_INET) {
 $daddr = ntop($sk->__sk_common.skc_daddr);
 $dport = ntohs($sk->__sk_common.skc_dport);

 // Get current queue size
 $wmem_queued = $sk->sk_wmem_queued;

 // Record metrics
 @wmem_queued_hist[$daddr, $dport] = hist($wmem_queued);

 // Alert on large queues
 if ($wmem_queued > 1000000) { // 1MB threshold
 printf("Large socket write queue: %s -> %s:%d (%d bytes queued)\n",
 comm, $daddr, $dport, $wmem_queued);
 @large_queues[comm, $daddr, $dport] = $wmem_queued;
 }
 }
}

// Track receive buffer pressure
kprobe:tcp_rcv_established {
 $sk = (struct sock *)arg0;

 if ($sk->__sk_common.skc_family == AF_INET) {
 $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
 $daddr = ntop($sk->__sk_common.skc_daddr);

 // Get receive buffer occupancy
 $rmem_alloc = $sk->sk_rmem_alloc;
 $rcvbuf = $sk->sk_rcvbuf;

 // Calculate percentage full
 $pct_full = ($rmem_alloc * 100) / $rcvbuf;

 // Record metrics
 @rmem_pct_hist[$saddr, $daddr] = hist($pct_full);

 // Alert on nearly full buffers
 if ($pct_full > 90) {
 printf("Receive buffer pressure: %s <- %s (%d%% full)\n",
 $saddr, $daddr, $pct_full);
 @rcv_pressure[$saddr, $daddr] = $pct_full;
 }
 }
}

interval:s:10 {
 printf("\n=== Socket Buffer Analysis ===\n");

 printf("Socket write buffer histograms (bytes queued):\n");
 print(@wmem_queued_hist);

 printf("\nSocket receive buffer fill percentage:\n");
 print(@rmem_pct_hist);

 if (@large_queues) {
 printf("\nLarge socket write queues (potential bufferbloat):\n");
 print(@large_queues);
 clear(@large_queues);
 }

 if (@rcv_pressure) {
 printf("\nReceive buffer pressure points:\n");
 print(@rcv_pressure);
 clear(@rcv_pressure);
 }

 clear(@wmem_queued_hist);
 clear(@rmem_pct_hist);
}
```

Detecting Network Covert Channels

This advanced script detects potential covert channels in network traffic:

```
#!/usr/bin/env bpftrace

#include
#include
#include
#include

BEGIN {
 printf("Monitoring for network covert channels...\n");
 printf("Looking for: timing-based channels, uncommon protocol usage, header manipulation\n");
}

// Track TCP header field patterns
kprobe:tcp_sendmsg {
 $sk = (struct sock *)arg0;
 $tcp = (struct tcp_sock *)$sk;

 // Check for unusual TCP header options or values
 if ($tcp->tcp_header_len > 20) {
 // TCP header larger than standard 20 bytes indicates options
 @tcp_options_usage[comm, $tcp->tcp_header_len] = count();
 }

 // Check for abnormal sequence numbers or other patterns
 // This is a simplification - real detection would be more sophisticated
 if ($tcp->rcv_nxt == $tcp->copied_seq && $tcp->rcv_wup == $tcp->copied_seq) {
 @suspicious_seq[comm] = count();
 }
}

// Track uncommon protocol usage
tracepoint:syscalls:sys_enter_socket {
 $domain = args->family;
 $type = args->type;
 $protocol = args->protocol;

 // Look for rare protocol combinations
 if ($protocol > 10 && $protocol != 17 && $protocol != 6) {
 // Uncommon protocol number (not TCP or UDP)
 printf("Uncommon protocol: %s using protocol %d\n",
 comm, $protocol);
 @rare_protocols[comm, $protocol] = count();
 }
}

// Track packet timing patterns (potential timing channel)
kprobe:dev_hard_start_xmit {
 // Record timestamp of packet transmission
 @last_xmit[pid] = nsecs;
}

kprobe:dev_hard_start_xmit /@last_xmit[pid]/ {
 $interval = nsecs - @last_xmit[pid];

 // Look for suspicious timing patterns (e.g., very regular intervals)
 if ($interval > 0) {
 @xmit_intervals[comm] = hist($interval / 1000000); // ms

 // Check if previous 5 intervals were nearly identical
 // This is simplified - real detection would use more robust statistics
 if (@last_intervals[pid, 0] > 0 &&
 @last_intervals[pid, 1] > 0 &&
 @last_intervals[pid, 2] > 0 &&
 @last_intervals[pid, 3] > 0 &&
 @last_intervals[pid, 4] > 0) {

 $avg = (@last_intervals[pid, 0] +
 @last_intervals[pid, 1] +
 @last_intervals[pid, 2] +
 @last_intervals[pid, 3] +
 @last_intervals[pid, 4]) / 5;

 $dev0 = ($avg > @last_intervals[pid, 0]) ?
 ($avg - @last_intervals[pid, 0]) :
 (@last_intervals[pid, 0] - $avg);
 $dev1 = ($avg > @last_intervals[pid, 1]) ?
 ($avg - @last_intervals[pid, 1]) :
 (@last_intervals[pid, 1] - $avg);
 // ... (similar for dev2-4)

 // If all deviations are very small (< 1% of average)
 if ($dev0 < ($avg / 100) &&
 $dev1 < ($avg / 100)) {

 printf("Suspicious timing pattern detected: %s (PID %d) - regular intervals of ~%d ms\n",
 comm, pid, $avg / 1000000);
 @timing_channels[comm, pid] = count();
 }
 }

 // Shift and store last 5 intervals
 @last_intervals[pid, 4] = @last_intervals[pid, 3];
 @last_intervals[pid, 3] = @last_intervals[pid, 2];
 @last_intervals[pid, 2] = @last_intervals[pid, 1];
 @last_intervals[pid, 1] = @last_intervals[pid, 0];
 @last_intervals[pid, 0] = $interval;
 }

 @last_xmit[pid] = nsecs;
}

interval:s:30 {
 printf("\n=== Covert Channel Detection Results ===\n");

 if (@tcp_options_usage) {
 printf("Processes using unusual TCP header options:\n");
 print(@tcp_options_usage);
 }

 if (@suspicious_seq) {
 printf("\nProcesses with suspicious TCP sequence patterns:\n");
 print(@suspicious_seq);
 }

 if (@rare_protocols) {
 printf("\nUncommon protocol usage:\n");
 print(@rare_protocols);
 }

 if (@timing_channels) {
 printf("\nPotential timing-based covert channels:\n");
 print(@timing_channels);
 clear(@timing_channels);
 }

 printf("\nPacket timing interval distributions:\n");
 print(@xmit_intervals);

 clear(@tcp_options_usage);
 clear(@suspicious_seq);
 clear(@xmit_intervals);
}
```



Hands-On Lab: Unauthorized Connection Monitoring

In this lab, we'll develop and deploy a comprehensive tool to monitor for unauthorized connection attempts.

Lab Overview:

1. Create a bpftrace script that identifies unauthorized connections
2. Enhance it to create detailed logs for forensic analysis
3. Integrate with a notification system for real-time alerts

Lab Setup: Define Allowed Connections

First, let's create a baseline of allowed connections:

```
#!/usr/bin/env bpftrace

BEGIN {
    printf("Unauthorized connection monitoring started...\n");

    // Define allowed connections by process and destination
    // Format: process_name:destination:port
    @allowed["nginx:10.0.0.5:3306"] = 1;  // nginx to MySQL
    @allowed["java:10.0.0.6:27017"] = 1;  // java app to MongoDB
    @allowed["python:10.0.0.7:6379"] = 1; // python app to Redis

    // Add your environment-specific rules here
}
```

Customize the allowed connections list to match your environment's legitimate network paths. This is critical for reducing false positives.

Lab Part 1: Basic Connection Monitoring

Now let's add the core monitoring functionality:

```
// Add this to the previous script

// Track all connection attempts
tracepoint:syscalls:sys_enter_connect {
 $sa = (struct sockaddr *)args->uservaddr;

 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $daddr = ntop($in->sin_addr.s_addr);
 $dport = ntohs($in->sin_port);

 // Create connection identifier string
 $conn_id = strcat(comm, ":");
 $conn_id = strcat($conn_id, $daddr);
 $conn_id = strcat($conn_id, ":");
 $conn_id = strcat($conn_id, str($dport));

 // Check if this connection is allowed
 if (@allowed[$conn_id] != 1) {
 printf("UNAUTHORIZED CONNECTION: %s (PID %d) -> %s:%d\n",
 comm, pid, $daddr, $dport);
 @unauth_conn[comm, $daddr, $dport] = count();
 }
 }
}
```

Lab Part 2: Enhanced Forensic Logging

Let's extend our script with detailed forensic capabilities:

```
// Add this to the previous script

// Add user and process path information
tracepoint:syscalls:sys_enter_connect /@unauth_conn[comm, ntop(((struct sockaddr_in *)args->uservaddr)->sin_addr.s_addr), ntohs(((struct sockaddr_in *)args->uservaddr)->sin_port)]/ {
 // Get user information
 $uid = uid;
 $gid = gid;

 // Log detailed information about this connection attempt
 printf("FORENSIC DETAIL - Time: %s\n", strftime("%H:%M:%S", nsecs));
 printf(" Process: %s (PID: %d, PPID: %d)\n", comm, pid, ppid);
 printf(" User: UID %d, GID %d\n", $uid, $gid);

 // Record stack trace for investigation
 printf(" User-space stack trace:\n");
 print(ustack);

 // Record kernel-space stack trace
 printf(" Kernel-space stack trace:\n");
 print(kstack);

 @detailed_events[comm, pid, $uid] = count();
}
```

Lab Part 3: Implementing Real-Time Alerts

Now let's add real-time alerting functionality:

```
// Add this to the previous script

// Function to generate an alert message
function generate_alert(process, pid, uid, dst_ip, dst_port) {
 // In a real system, this would send to an external alerting system
 // For this lab, we'll simulate with a printf

 printf("\n!!! SECURITY ALERT !!!\n");
 printf("Unauthorized connection attempt detected:\n");
 printf(" Process: %s (PID: %d)\n", process, pid);
 printf(" User ID: %d\n", uid);
 printf(" Destination: %s:%d\n", dst_ip, dst_port);
 printf(" Timestamp: %s\n", strftime("%Y-%m-%d %H:%M:%S", nsecs));
 printf("!!! SECURITY ALERT !!!\n\n");
}

// Trigger alerts for unauthorized connections
tracepoint:syscalls:sys_enter_connect /@unauth_conn[comm, ntop(((struct sockaddr_in *)args->uservaddr)->sin_addr.s_addr), ntohs(((struct sockaddr_in *)args->uservaddr)->sin_port)]/ {
 $sa = (struct sockaddr *)args->uservaddr;

 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $daddr = ntop($in->sin_addr.s_addr);
 $dport = ntohs($in->sin_port);

 // Generate real-time alert
 generate_alert(comm, pid, uid, $daddr, $dport);
 }
}

// Summary report
interval:s:60 {
 printf("\n=== Unauthorized Connection Summary ===\n");
 time("%Y-%m-%d %H:%M:%S\n");

 if (@unauth_conn) {
 printf("Unauthorized connection attempts:\n");
 print(@unauth_conn);
 } else {
 printf("No unauthorized connections detected in the last minute\n");
 }

 clear(@unauth_conn);
}
```

Lab Part 4: Running and Testing the Script

To run your unauthorized connection monitor:

```
# Save the complete script as unauthorized_conn_monitor.bt
# Make it executable
chmod +x unauthorized_conn_monitor.bt

# Run it with root privileges
sudo ./unauthorized_conn_monitor.bt

# Or use bpftrace directly
sudo bpftrace unauthorized_conn_monitor.bt
```

To test the script, try making connections from unauthorized processes:

```
# In another terminal, try an unauthorized connection
nc 10.0.0.5 3306

# You should see alerts in your bpftrace output
```

Lab Part 5: Advanced Connection Fingerprinting

Let's enhance our monitor with connection fingerprinting:

```
// Add this feature to our monitoring script

// Track connection patterns
tracepoint:syscalls:sys_enter_connect {
 $sa = (struct sockaddr *)args->uservaddr;

 if ($sa->sa_family == AF_INET) {
 $in = (struct sockaddr_in *)$sa;
 $daddr = ntop($in->sin_addr.s_addr);
 $dport = ntohs($in->sin_port);

 // Record connection timestamp
 @conn_time[pid, $daddr, $dport] = nsecs;

 // Count connections by process
 @conn_by_proc[comm] = count();
 @dest_by_proc[comm, $daddr] = count();
 }
}

// Track connection frequency patterns
interval:s:10 {
 printf("\n=== Connection Pattern Analysis ===\n");

 // Detect processes with unusual connection patterns
 foreach ($comm in @conn_by_proc) {
 $count = @conn_by_proc[$comm];

 if ($count > 20) {
 printf("High connection rate: %s made %d connections in 10 seconds\n",
 $comm, $count);
 }
 }

 // Detect processes connecting to many unique destinations
 $dest_count = 0;
 $prev_comm = "";

 foreach ([$comm, $daddr] in @dest_by_proc) {
 if ($prev_comm == $comm) {
 $dest_count++;
 } else {
 if ($dest_count > 10) {
 printf("Process connecting to many destinations: %s -> %d unique destinations\n",
 $prev_comm, $dest_count);
 }
 $dest_count = 1;
 $prev_comm = $comm;
 }
 }

 // Clear counters for next interval
 clear(@conn_by_proc);
 clear(@dest_by_proc);
}
```

Extending Our Security Observability

Beyond our lab, here are additional areas to explore for comprehensive security observability:

Behavioral Baselining

Create process-specific network behavior profiles and alert on deviations

Traffic Correlation

Correlate network activity with system calls, file access, and memory patterns

Lateral Movement Detection

Identify suspicious internal network traffic patterns that might indicate compromise

Integration

Connect bpftrace monitors with SIEM systems, log analyzers, and alerting platforms

Key Takeaways: Network & Security Observability

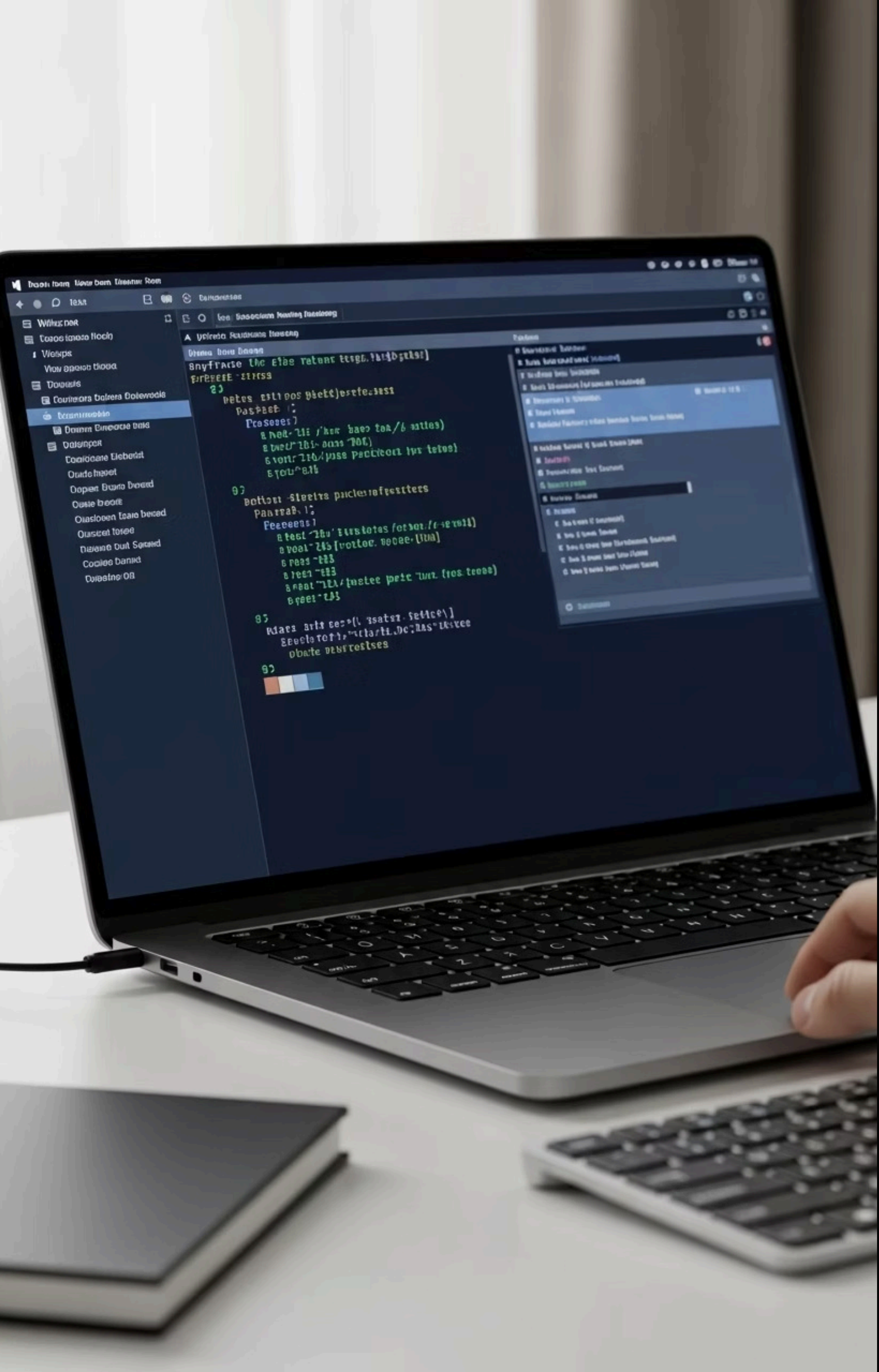
Network Observability

- bpftrace provides unprecedented visibility into network stack operations
- Trace from socket layer down to device driver for complete understanding
- Performance analysis can identify bottlenecks in packet processing
- Socket buffer monitoring helps detect queuing and latency issues

Security Monitoring

- Real-time detection of unauthorized connections
- Identify suspicious network behavior patterns
- Monitor for covert channels and data exfiltration
- Track process network isolation violations

bpftrace empowers Linux systems engineers and security practitioners with deep network and security observability capabilities without requiring kernel modifications.



Next Steps and Resources

Additional Learning

- Brendan Gregg's "BPF Performance Tools" book
- Linux Tracing Workshops: **https://github.com/iovisor/bcc/blob/master/docs/tutorial.md**
- Networking Stack Internals: **Linux Networking Stack Guide**

Build Your Own

- Extend the lab scripts for your specific environment
- Create a comprehensive network observability dashboard
- Implement custom bpftrace collectors for your monitoring system

Coming in Module 5

- Filesystem & Storage Observability
- Block I/O tracing
- Page cache analysis
- Storage performance monitoring