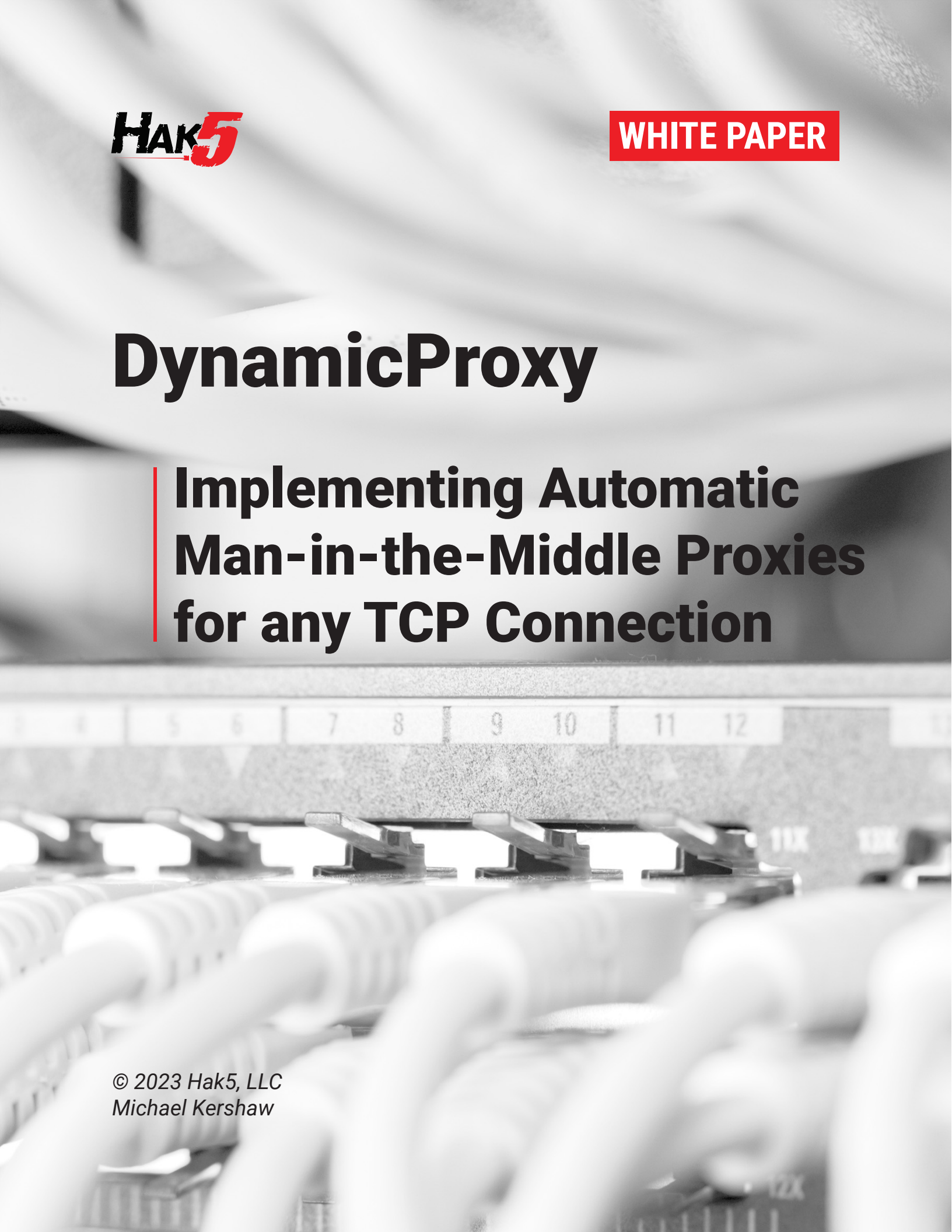


WHITE PAPER

DynamicProxy

Implementing Automatic Man-in-the-Middle Proxies for any TCP Connection

Michael Kershaw

Introduction

Transparent Proxies: Existing Tools and Limitations

Transparent Proxies

A transparent proxy is inserted into a communication stream to log or manipulate traffic between two endpoints.

Often, a transparent proxy is highly protocol specific: For instance, tools such as Burp Suite and MITMProxy offer advanced HTTP proxies, but can not be used to proxy other more generic protocols.

Protocol-specific proxies are able to act as transparent proxies to multiple destinations when the destination address is encoded in the protocol: For HTTP, the original destination and request is part of the HTTP protocol.

For protocols where there is no internal addressing, a proxy cannot derive the original destination once the packets have been redirected to the proxy service.

Proxy protocols

Proxy protocols such as SOCKS attempt to address this problem by wrapping all communications in an additional layer which informs the proxy service of the true destination.

Unfortunately, these are far from transparent: If a system is not configured to use SOCKS (or related), it will not be able to communicate. Often configuring a SOCKS proxy is at a per-application level.

Due to the invasive nature of a proxy layer like SOCKS, typically it is only applied to traffic leaving the local network; this further complicates attempts to capture and proxy intra-LAN traffic.

Introduction

Transparent Proxies: Existing Tools and Limitations

IPTables / NFTables

The Linux Kernel packet rewriting system (iptables or nftables) can trivially manipulate traffic; for instance redirecting all traffic destined to port 9100 to a local service is as simple as:

```
nft add rule ip nat prerouting 'tcp dport 9100 dnat to 127.0.0.1:9100'
```

However, this encounters the same difficulty as a traditional transparent proxy service: Once the packet has passed through the destination rewriting stage of nftables, the original destination is no longer known to the proxy.

To act as a transparent proxy for arbitrary TCP data, the destinations must already be known, and multiple proxy and multiple rewrite rules must be configured:

```
nft add rule ip nat prerouting \
    'ip daddr 192.168.1.10 tcp dport 9100 dnat to 127.0.0.1:9100' # Proxy 1
nft add rule ip nat prerouting \
    'ip daddr 192.168.1.11 tcp dport 9100 dnat to 127.0.0.1:9101' # Proxy 2
nft add rule ip nat prerouting \
    'ip daddr 192.168.1.12 tcp dport 9100 dnat to 127.0.0.1:9102' # Proxy 3
```

Introduction

Transparent Proxies: Existing Tools and Limitations

In this model, when the destinations are known, an individual proxy service can be created for each destination: The proxy no longer needs to know the original destination, as it is only receiving traffic for that single host.

The Limitations

All of these techniques require either insight into the protocol to extract the original destination (as in a HTTP proxy), disruptive and obvious configuration changes to the target system (SOCKS or other proxy protocols), capture without forwarding to the original host (as in a rewrite of all traffic to a target port), or a priori knowledge of all destinations prior to deployment and intercept (as in a proxy per target host).

By the time a packet has been rewritten and reached the userspace proxy daemon, the original destination has been overwritten by the DNAT rule, necessitating a proxy per previously identified endpoint.

Transparent Proxies
A Solution

A preferred solution should address as many of these limitations as possible, while also minimizing external dependencies and remaining performant on embedded-scale hardware.

netfilter-queue

Fortunately, the Linux kernel has a mechanism already available: NFQUEUE. Coupled with the userspace libnetfilter-queue, this allows iptables and nftables rules a target for userspace decisions.

Packets sent to NFQUEUE are held in a numbered queue until a userspace application retrieves the packet via a netlink socket, and informs the kernel of the final packet disposition.

NFQUEUE allows userspace applications to perform more complex decision making and logging processes when integrated as part of a network firewall, but also allows us an opportunity to interact with a packet before the kernel modifies it for local NAT.

Capturing a Packet

Packets are sent to NFQUEUE using a standard nftables rule:

```
chain dstnat_lan {
   mark != 1337 ip daddr != 172.16.32.1 tcp dport {1-65535}
counter queue num 30 bypass
}
```

This nftables configuration instructs the kernel to place all TCP packets into queue #30 (chosen arbitrarily).

Of note:

1. Packets are excluded by mark. Another standard component of the Linux iptables/nftables system, marking flags packets with additional attributes.

2. Packets are excluded by destination address. Packets destined to the device running the dynamic proxy are not targeted, preventing an infinite loop.

3. The `bypass` option is specified. If the userspace component is not available, the packets are allowed to continue normally based on kernel rules.

Transparent Proxies
A Solution

With this nftables configuration in place, packets are queued for a userspace service on queue #30 whenever a userspace handler is present.
This allows us to make advanced determination and manipulation of packets in userspace, via the NFQUEUE netlink interface.

Netlink

The netlink API is a datagram-based protocol between the kernel and userspace. It can be used for high-speed transfer of data between the kernel and userspace applications, and it is often used to create complex APIs that replace the older, more limited IOCTL method of control.

Netlink commands consist of one or more common netlink message headers and payloads, and a macro system for manipulating the payloads.

Fortunately, the libnetfilter-queue library abstracts most of this away for us automatically, allowing for a relatively simple interface with the netlink API:

```
h = nfq_open();
nfq_unbind_pf(h, AF_INET);
nfq_bind_pf(h, AF_INET);

qh = nfq_create_queue(h, 30, &nfq_cb, NULL);
nfq_set_mode(qh, NFQNL_COPY_PACKET, 0xFFFF);
  return go(f, seed, [])
}
```

Transparent Proxies

A Solution

Controlling Packets with NFQUEUE

A NFQUEUE tool can instruct the kernel to take multiple actions on a packet:

- NF_DROP to discard the packet (the same as the DROP target in iptables/nftables)
- NF_ACCEPT to pass the packet (the same as the ACCEPT target in iptables/nftables)
- NF_QUEUE to inject the packet into a different queue
- NF_REPEAT to return the packet to the kernel for re-processing
- NF_STOP to accept the packet, but block further re-processing in the kernel

Additionally, a NFQUEUE tool can set the mark on a packet, a feature we will leverage directly:

```
nfq_set_verdict2(qh, id, NF_REPEAT, 1337, 0, NULL);
```

Utilizing both the NF_REPEAT and mark features allows the userspace tool to return the packet to the kernel for additional processing, while marking it to be excluded from the userspace processor.

Transparent Proxies
A Solution

Connection Attributes

Now that we are able to inspect packets and return verdicts to the kernel, we need a mechanism for identifying known connections.

Fortunately, every TCP connection contains four pieces of information:

1. Source IP - the IP originating the connection
2. Source port - a random source IP on the system originating the connection
3. Destination IP - the original destination IP of the packet, prior to kernel manipulation
4. Destination port - the original destination port of the packet, prior to kernel manipulation

These four attributes are used by nearly all stateful routing systems to track a connection, including the Linux NAT connection tables. We'll use them the same way to track connections through the proxy system.

Inspecting and Remembering Connections

Connections in the dynamic proxy are tracked as a list of srcIP:srcPort entities, mapped to a record containing the original destination IP and port; since the full packet is available thanks to the NFQUEUE interface, extracting this is simple:

```
struct nfqnl_msg_packet_hdr *ph;
ph = nfq_get_msg_packet_hdr(nfa);
uint32_t id = ntohl(ph->packet_id);

ret = nfq_get_payload(nfa, (unsigned char **) &data);
IP ip((const uint8_t *) data, ret);
const auto& tcp = ip.find_pdu<TCP>();

auto stream =
    std::make_shared<tracked_stream>(ip.src_addr(), tcp-
>sport(), ip.dst_addr(), tcp->dport()
```

Transparent Proxies

A Solution

Rewriting Connections

Having inspected the original connection and recorded the destination IP and port, the packet is returned to the kernel. If the target port is one the proxy is configured for, the packet is marked with the packetmark `1337` and returned via `NF_REPEAT`, if it is not a port to be proxied it is returned with a basic `NF_ACCEPT`:

```
// Mark a packet and return it to the kernel
nfq_set_verdict2(qh, id, NF_REPEAT, 1337, 0, NULL);

// Or, accept a packet and bypass further processing
nfq_set_verdict(qh, id, NF_ACCEPT, 0, NULL);
```

An additional nftables rule is used to rewrite all connections marked with `1337` to localhost:

```
tcp ip saddr != 172.16.32.1 meta mark 1337 dnat ip to
172.16.32.1
```

This is a traditional DNAT destination rewrite: It replaces the original destination IP with the address of the device running the proxy.

Transparent Proxies
A Solution

Closing the Loop

To complete the proxy, the userspace needs the final step of creating a listening service. The target packets crossing the proxy device have been identified, tagged, and rewritten by the DNAT layer to re-target the local service.

A standard TCP socket is created via `listen()`, and each incoming connection is inspected.

The DNAT process alters the destination host, but does not change the original source host and source port: The original values are passed as part of the `accept()` system call.

```
conn_fd = accept4(fd, (struct sockaddr *) &addr, &len,
SOCK_NONBLOCK);
```

Inside the addr struct is the source address and port (addr.sin_addr and addr.sin_port), which are the same values used to make the connection key in the previous phase.

Using the source address and port, the dynamic proxy resolves the original connection destination, and creates a standard TCP connection:

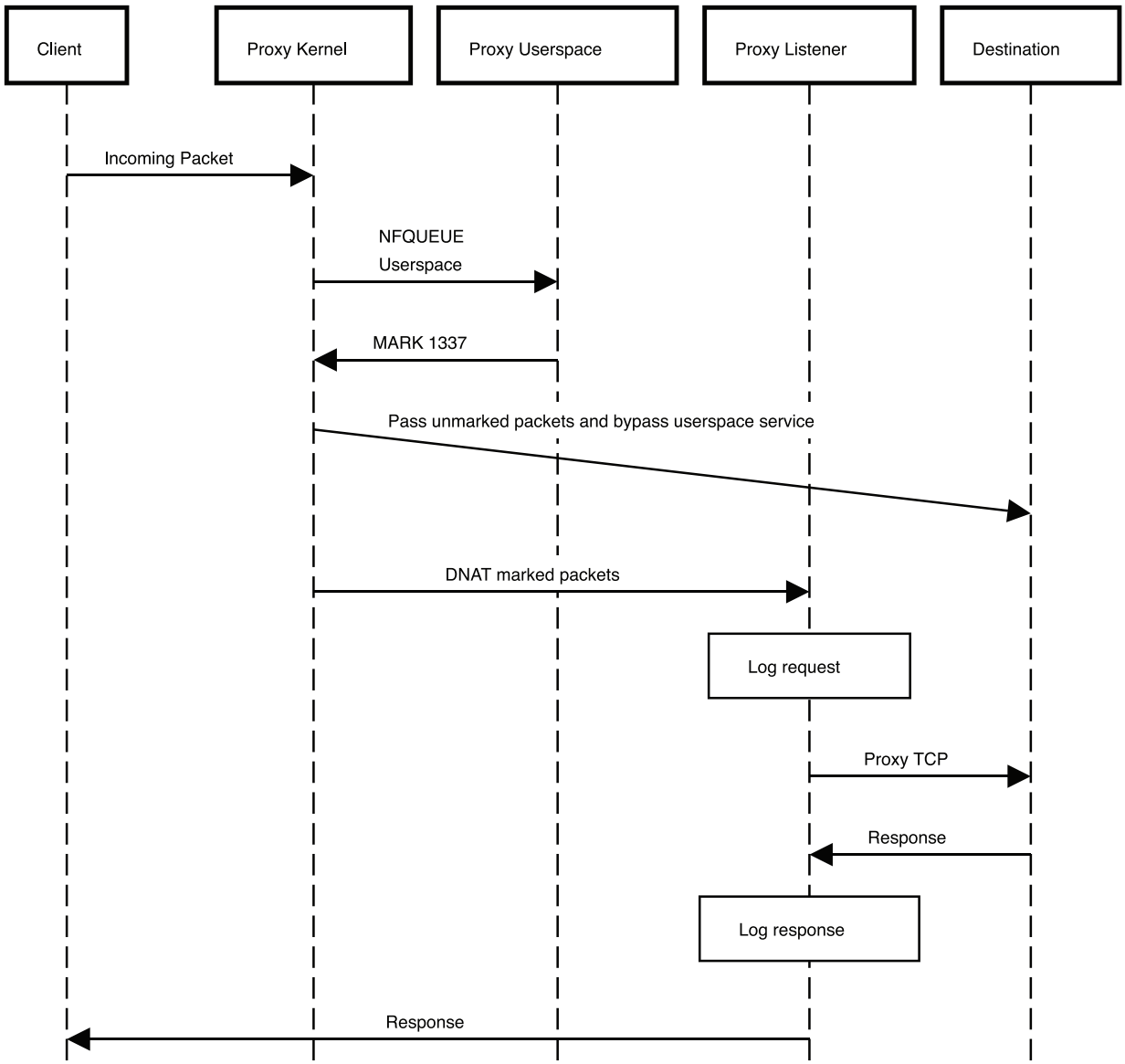
```
struct sockaddr_in cli;

stream->targetside_fd = socket(AF_INET, SOCK_STREAM, 0);

bzero(&cli, sizeof(cli));

cli.sin_family = AF_INET;
cli.sin_addr.s_addr = inet_addr(stream->orig_dst.to_string().c_str());
cli.sin_port = htons(stream->orig_port);

r = connect(stream->targetside_fd, (struct sockaddr *) &cli,
sizeof(cli));
```

Transparent Proxies

A Solution

The dynamic proxy now maintains the original connection to the target device, and the proxied connection to the original destination, and is now able to log or rewrite arbitrary TCP traffic within the connection.

The Complete Proxy Cycle



Transparent Proxies

Implementation

The TCP dynamic proxy is available now as the `DYNAMICPROXY` command on the Packet Squirrel Mark II:

```
                ____            _        _   ____              _               _ 
  __   (\_     |  _ \ __ _  ___| | _____| |_/ ___|  __ _ _   _(_)_ __ _ __ ___| |
 (_ \  ( '>    | |_) / _` |/ __| |/ / _ \ __\___ \ / _` | | | | | '__| '__/ _ \ |
   ) \/_)=     |  __/ (_| | (__|   <  __/ |_ ___) | (_| | |_| | | |  | | |  __/ |
   (___ )_     |_|   \__,_|\___|_|\_\___|\__|____/ \__, |\__,_|_|_|  |_|  \___|_|
    Hak5 Packet Squirrel Mark II                       |_|

root@squirrel:~# DYNAMICPROXY
Expected DYNAMICPROXY [CLIENT|SERVER|ANY] [prefix] [port1] ... [portN]
DYNAMICPROXY (Hak5 Network Payload Commands)
    Capture TCP streams to file by acting as a dynamic proxy,
    capable of logging client traffic, server traffic, or
    btoh types.Usage:
    DYNAMICPROXY [CLIENT|SERVER|ANY] [prefix] [port] ... [portN]
```

The `DYNAMICPROXY` system is deployed as a default payload for capturing traffic from PCL printers, but can be used for any other TCP stream as well.

```
LED SETUP
NETMODE NAT
# We have to have attached USB
USB_WAIT
# Make sure the directory exists
mkdir /usb/printer/
LED ATTACK
# Use a dynamic proxy to MITM standard PCL IP printers
DYNAMICPROXY CLIENT /usb/printer/print_ 9100
```

About Hak5

Founded in 2005, Hak5's mission is to advance the InfoSec industry. This is done through award winning podcasts, leading pentest gear, and inclusive community — where all hackers belong.

Hak5 gear have found their way into the hearts and tool-kits of enthusiasts and red-teams alike. They're notable for being effective and accessible. The design philosophy is simple — make it do the thing.