



Network Security

ithilgore
sock-raw.homeunix.org

PLUG - 0x00080c03



Session Overview

Part 0x01. Firewalls

Part 0x02. Raw Sockets

Part 0x03. Questions

Part 0x01. Firewalls

- **Firewalls overview**
- **Stateless vs Stateful**
- **Reversing Firewall Rulesets**
- **Bypassing Firewalls Case 1:
TCP piggybacking**
- **Bypassing Firewalls Case 2:
Timing games**
- **Things to watch out for**

Firewalls Overview

- Main entry points of all networks
- Some of the most complex code → bugs
- Kernel-space implementation – OS specific
- Packet filtering fws → | | ← Proxy firewalls
- Linux: < 2.4 ipchains, >= 2.4, 2.6 iptables
- FreeBSD: ipf, ipfw, pf (as kernel module)
- OpenBSD: pf (integrated), ipf

Firewalls Overview

- Major role: position of firewall
Functions rendered useless if traffic redirected
- 'Default Permit' vs 'Default Deny' policies
- Firewalls don't protect you against vulnerable services. Some ports will eventually have to stay open.
- Usually, configuration requires strong TCP/IP knowledge → more than usually, results are: misconfigurations → security holes

Stateless vs Stateful

Stateless:

- Doesn't keep state about connections.
- More simple to set up.
- Inherently **less** secure.
- Old firewalls were stateless.
- Examples: Win XP SP2/2003, ipchains, home routers/embedded devices

Stateless vs Stateful

Stateful:

- Has a state table, that keeps track of every packet/connection.
- Rules can get really complicated.
- State-of-the-art firewalls are stateful.
- Examples: pf, iptables

Stateless vs Stateful

Stateless firewalls are vulnerable to ACK and other port-scanning techniques.

See: sock-raw.homeunix.org/papers/firewalls

Other corner cases: using more than one scanflags e.g
nmap --scanflags SYNFIN

```
/usr/src/linux-2.6.*/*net/ipv4/tcp_input.c:  
static int tcp_rcv_synsent_state_process(struct sock *sk,  
struct sk_buff *skb, struct tcphdr *th, unsigned len) {  
...  
    if (th->ack) { ... } ...  
    if (th->rst) { ... goto discard ... } ...  
    if (th->syn) { ... } ...  
}
```

Stateless vs Stateful

Both Linux and *BSD kernels allow SYN-multiple flags (except RST, ACK) to pass through. Not a security hole by itself (except for an old version of ipchains).

Stateless firewalls are a good indication of a weak target (and maybe a weak OS ;).

Stateful firewalls are a good indication of more valuable targets.

Misconfigured -any type- firewalls are a good indication of a potential target to leverage more attacks.

Reversing Firewall Rulesets

TEST CASE: SCO/caldera

REASON: lawsuits anyone?

REAL REASON: odd firewall setup

TOOLS: nmap

ADDITIONAL TOOLS: none

OBJECTIVE: map firewall rulesets by reverse engineering portscan results

Reversing Firewall Rulesets

Start with SYN scanning:

```
# nmap docsrv.caldera.com -T4 -PN -F --max-retries 0 --reason -sA -vv
```

Interesting ports on docsrv.caldera.com (132.147.127.17):

Not shown: 98 filtered ports

Reason: 98 no-responses

PORT	STATE	SERVICE	REASON
------	-------	---------	--------

80/tcp	open	http	syn-ack
--------	------	------	---------

113/tcp	closed	auth	reset
---------	--------	------	-------

Reversing Firewall Rulesets

ACK Scanning:

```
# nmap docsrv.caldera.com -T4 -PN -F --max-retries 0 --reason -sA -vvvv
```

Interesting ports on docsrv.caldera.com (132.147.127.17):

PORT	STATE	SERVICE	REASON
...			
21/tcp	unfiltered	ftp	reset
22/tcp	unfiltered	ssh	reset
80/tcp	unfiltered	http	reset
113/tcp	unfiltered	auth	reset
135/tcp	filtered	msrpc	no-response

...

Reversing Firewall Rulesets

Continue with FIN Scanning:

```
# nmap docsrv.caldera.com -T4 -PN -F --max-retries 0 --reason -sF -vvvv
```

Interesting ports on docsrv.caldera.com (132.147.127.17):

PORT	STATE	SERVICE	REASON
...			
21/tcp	open filtered	ftp	no-response
22/tcp	open filtered	ssh	no-response
80/tcp	open filtered	http	no-response
113/tcp	closed	auth	reset
135/tcp	open filtered	msrpc	no-response

...

Reversing Firewall Rulesets

Let's make a table of our results:

Port	SYN	ACK	FIN
21	no-resp	reset	no-resp
22	no-resp	reset	no-resp
80	syn-ack	reset	no-resp
113	reset	reset	reset
135	no-resp	no-resp	no-resp

Noticed anything strange? Let's convert the table to a more understandable form.

Reversing Firewall Rulesets

Port	SYN	ACK	FIN
21	filtered	unfiltered	open filtered
22	filtered	unfiltered	open filtered
80	open	unfiltered	open filtered
113	closed	unfiltered	closed
135	filtered	filtered	open filtered

Still didn't notice it? The problem lies in ports 21 and 22. They replied to ACKs, but neither to SYN nor FIN. This is **not** normal.

Reversing Firewall Rulesets

Possible (you can never be sure unless you own the machine) scenario:

- src ip filtering on ports 21, 22 for SYN initiating connections
- their firewall is stateless
- Thus:
 1. ACK scans bypass it and we get back an RST
 2. FIN scans bypass it and we get back nothing since the filtering takes place only for SYN packets, and thus the port is considered open.
- More possible scenarios. Feel free to think about them before you go to sleep(3).

Bypassing Firewalls Case 1: TCP piggybacking

One of the first information gathering phases is network cartography.

Usual tool: traceroute (or nmap --traceroute)

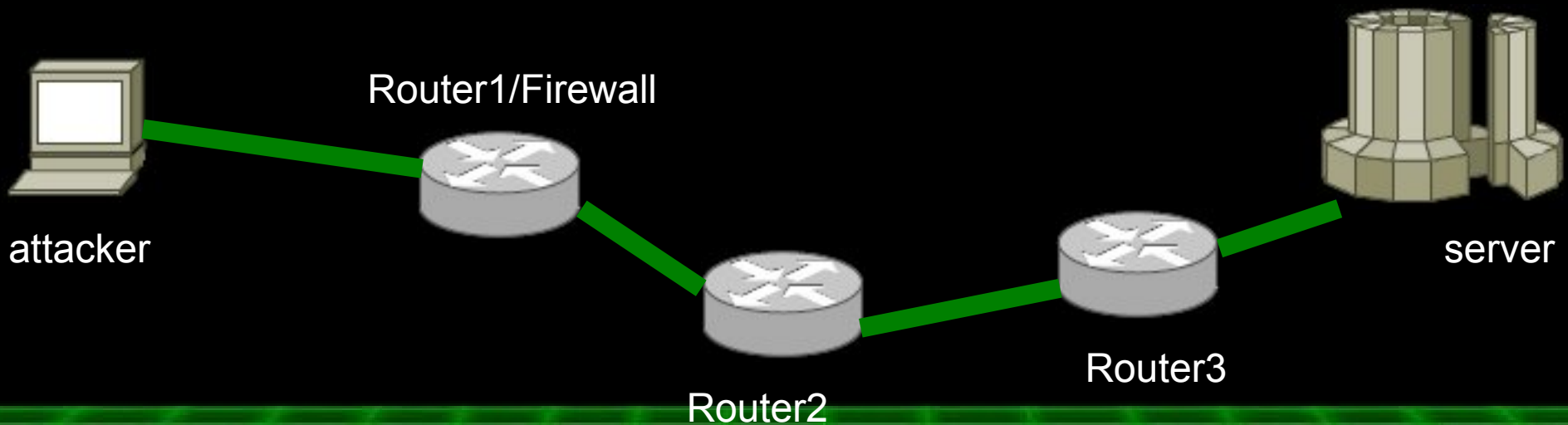
Idea: send increasing TTL packets and map the routing path

Problems:

- uses UDP as transport protocol
- most times **inbound** UDP is blocked
- also **inbound** ICMP echo requests (windows tracert uses them) are blocked

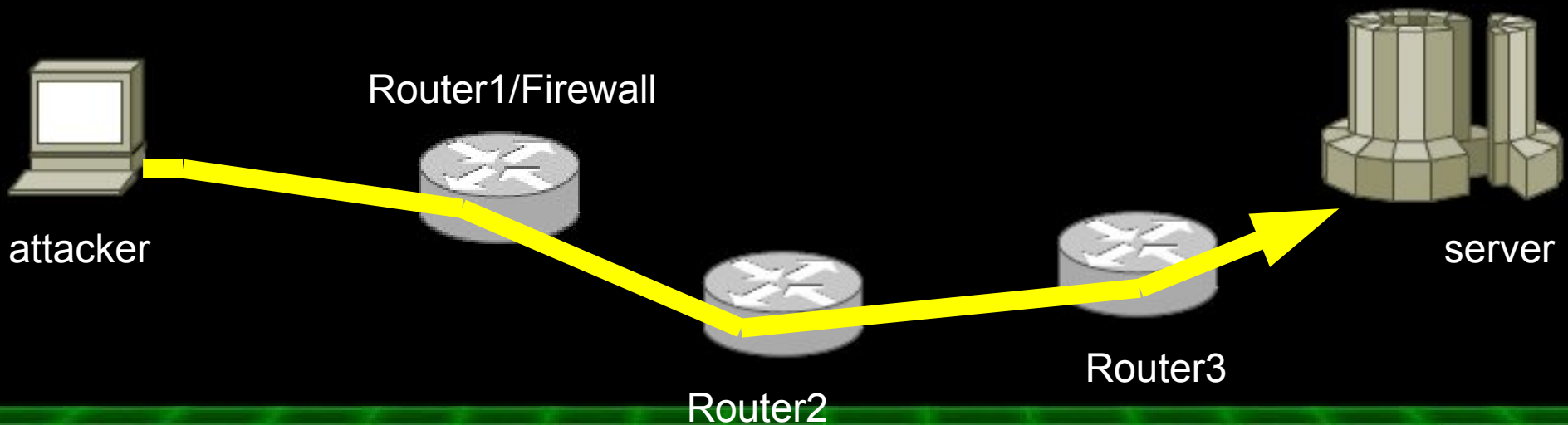
TCP piggybacking

Possible scenario: **outbound** UDP/ICMP traffic is allowed. We'll exploit this to get the replies and use already existing TCP connections to slip by the requests.



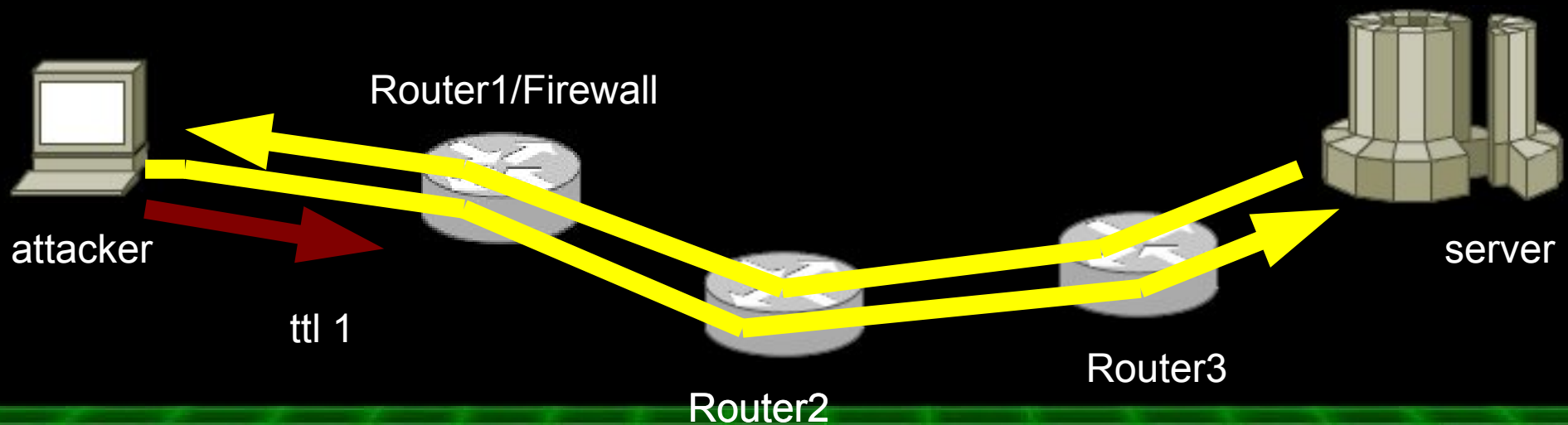
TCP piggybacking

Tool: **paratrace** from Dan Kaminsky's paketto keiretsu
First make a legitimate TCP connection to the server. (say port 80)



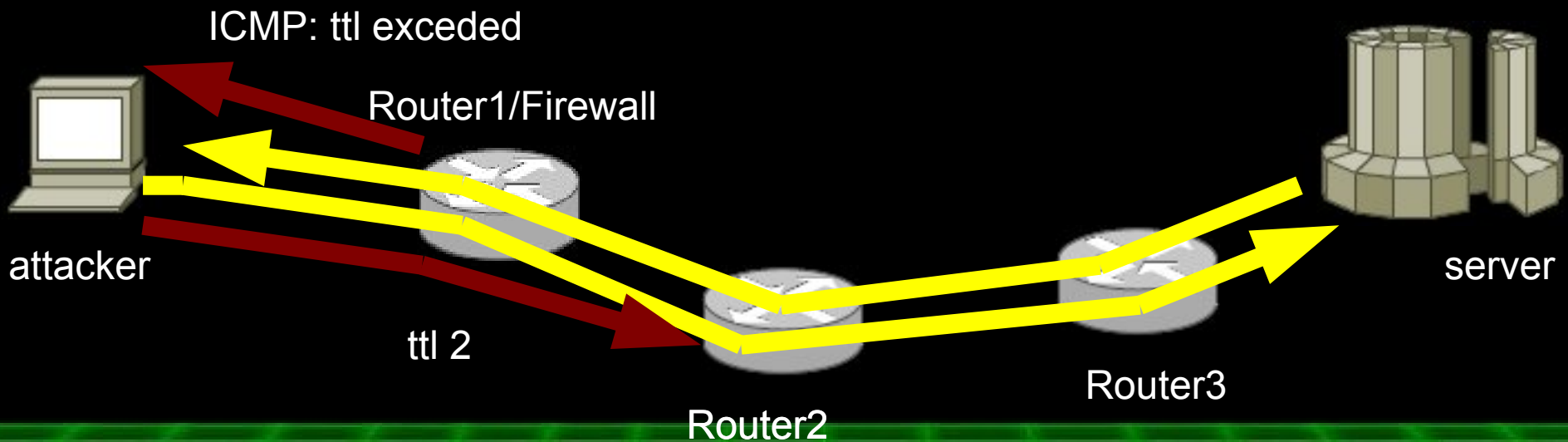
TCP piggybacking

As soon as the connection is initiated, paratrace will start sending TCP keepalive probes with increasing TTLs.



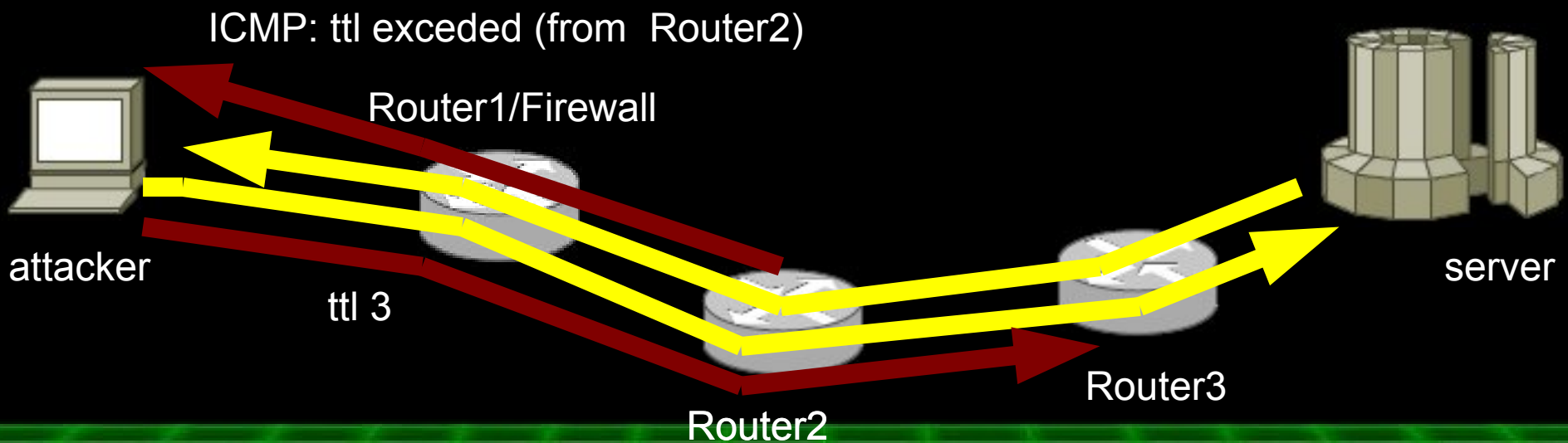
TCP piggybacking

Each injected TCP packet will pass through the firewalls since it is part of a legitimate connection, and will also trigger an ICMP TTL exceeded response.



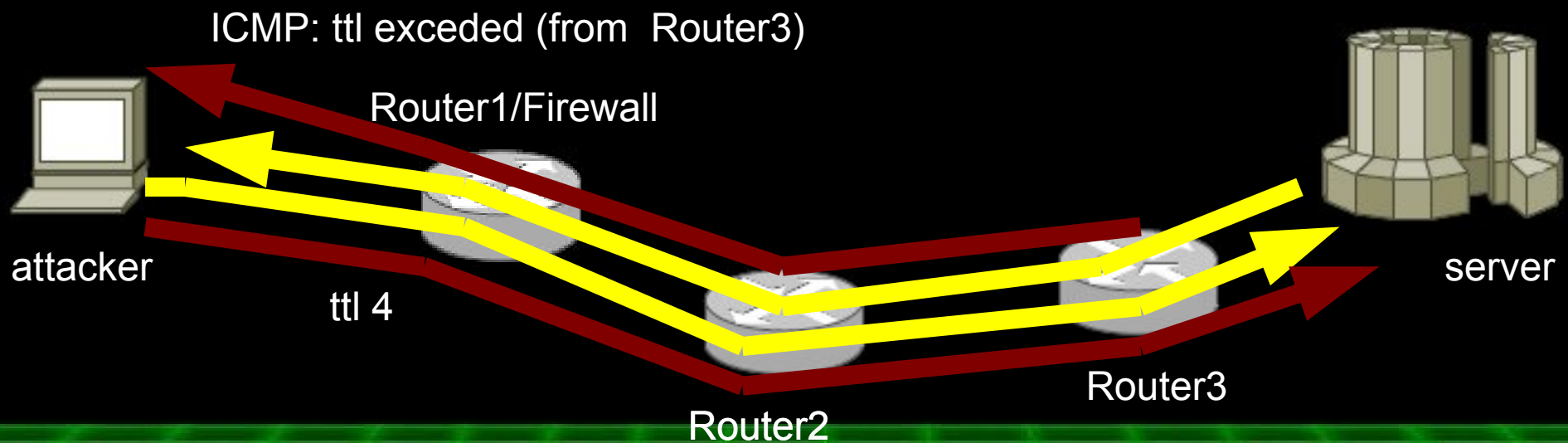
TCP piggybacking

Since ICMP traffic is allowed if direction is **outbound** (src = inside network, dst = outer network), we will get the ICMP type 11, code 0 messages.



TCP piggybacking

By the end of the normal connection, we will have eventually mapped all the path nodes.



Bypassing Firewalls Case 2: Timing games

Time is a crucial factor in everything associated with networks. More importantly, many defensive measures are based on time.

The idea is simple: if a host sends more than “threshold” packets in a certain “time” period, then “take action” (log the attempt, block IP, inform 3-letter agencies, etc)

Let's head straight into our testcase.

Timing games

Typical tactic:

- LOG/DROP further packets from src IP if: that specific src IP has sent more than <num> packets in a <time> period
- # iptables -P INPUT DROP
iptables -A INPUT -p tcp -m state --state NEW -m recent --set
iptables -A INPUT -p tcp -m state --state NEW -m recent --update --seconds <time> --hitcount <num> -j DROP

Timing games

Suppose:

<num> = 5

<seconds> = 60

So, try sending ≥ 5 packets in ≤ 60 seconds and every packet above the 4th will be dropped.

So, can the above ruleset be bypassed? Sure...
And it is far easier than you'd expect.

Timing games

Introducing (?) some Nmap timing options:

- `--max-rate <number>`: Send packets no faster than `<number>` per second
- `--scan-delay/--max-scan-delay <time>`: Adjust delay between probes

Simple math: We want no more than 4 packets in 60 seconds. Thus a `--scan-delay` of $60/4 = 15\text{s}$ will be good enough.

Timing games

Let's make some optimizations:

`--max-retries <tries>`: Caps number of port scan probe retransmissions.

Nmap by default sends 2 probes per port for reliability. Setting the option to 0 will result in a faster and slightly sneakier scanning (it almost cries out it's Nmap behind the portscan if the logs show 2 probes per port)

Timing games

And finally let's also take some more precautions by specifying decoy targets. These will mask the real attacker identity from possible logging.

```
# nmap -T4 --max-retries 0 -iR 1000 -p80 -PN -v -n |  
grep "open port" | cut -d" " -f6
```

```
69.64.16.1  
75.106.206.185  
83.80.16.235  
132.8.129.244  
80.24.231.112
```

```
...
```

Timing games

Actually, you might want to have in mind the existence of a “glitch”, before you combine decoys and complex timing options to bypass firewalls. Source code snippet: [nmap-4.76/scan_engine.cc](https://nmap.org/4.76/scan_engine.cc):

```
/* This function also handles the sending of decoys. There is
no fine-grained control of this; all decoys are sent at once on
one call of this function. This means that decoys do not
honor any scan delay and may violate congestion control
limits. */
static UltraProbe *sendIPScanProbe(UltraScanInfo *USI,
HostScanStats *hss, ...) {
```

Timing games

Summing up:

```
# nmap -p<range> --scan-delay 15 -max-  
retries 0 -PN <target_IP> -D69.64.16.1,  
75.106.206.185, 83.80.16.235, 132.8.129.244, <my_ip>,  
80.24.231.112
```

Using decoys will result in a massive burst of packets from the spoofed ips (no scan-delay from them) and the opposite from the real ip – yours. This is clearly a betraying sign that even your average sysadmin might notice. It's better than nothing though.

Timing games

In some cases applying the default nmap timing template (-T3) or less (-T2/-T1) will not trigger many alarms.

Using slightly variations of the above techniques, you can bypass the Snort IDS default nmap rulesets (which involve a sliding time window).

Nothing is impenetrable.

Things to watch out for

- Firewalls are no panacea!
- Never rely **only** on firewalls for protection.
- History of remote “denial of service” vulnerabilities. (many of them have remote root poc in the underground...)
- Watch out for traffic **direction**. What goes **out** is also as important as what goes **in** (see paratrace example).

Things to watch out for

- Avoid stateless rules.
- Embrace a '**Default Deny**' policy. Avoid the lazy approach of 'Default Permit'. On the long-term, the first will prove less strenuous.
- Avoid **not** getting a thorough TCP/IP background before you start setting up critical-point firewalls. Fragmentation-attacks anyone?

Things to watch out for

- Always test the firewall/network setup with actual tools:
 - Nmap (nmap.org -- Fyodor)
 - Hping (hping.org -- antirez)
 - Paketto keiretsu (doxpara.com -- Effugas)
 - Tcpdump (tcpdump.org)
 - IRPAS (phenoelit-us.org/irpas -- FX / for advanced routing techniques)

And of course, depending on which side you are, always watch out for PVN etc.

Part 0x02. Raw Sockets

- **Raw Sockets Overview**
- **TCP/IP packet headers**
- **Test Case: ICMP Redirection Tool**
- **Raw socket kernel handlers**
- **Things to watch out for**

Raw Sockets Overview

Normal sockets:

- old-time classic BSD socket API (used everywhere and ripped off by everyone)
- controlled network functions e.g connect, listen
- too much abstraction
- anyone can use them
- cannot really do much with them as far as low-level network tools are concerned
- (mostly) platform independent

Raw Sockets Overview

Raw sockets:

- powerful API extension that lets you build your own headers and packets
- no abstraction from kernel - (almost) total control over the network facilities
- (almost) platform independent - needs a bit caution to be portable
- need root to use them (or CAP_NET_RAW)

Raw Sockets Overview

What are the possibilities of SOCK_RAW?

- Build your own layer3/4 protocol (e.g zebra routing package)
- Make any netsec tool imaginable - most network security tools use raw sockets (or libraries which use raw sockets behind the scene) to craft their packets (e.g nmap, hping, unicornscan, scanrand, paratrace)
- Exploit vulnerable service code or bugs that are only triggered with special packets. (e.g nmap oob mem access with fake ip->ip_len and oversized TCP header: seclists.org/nmap-dev/2008/q4/0543.html)

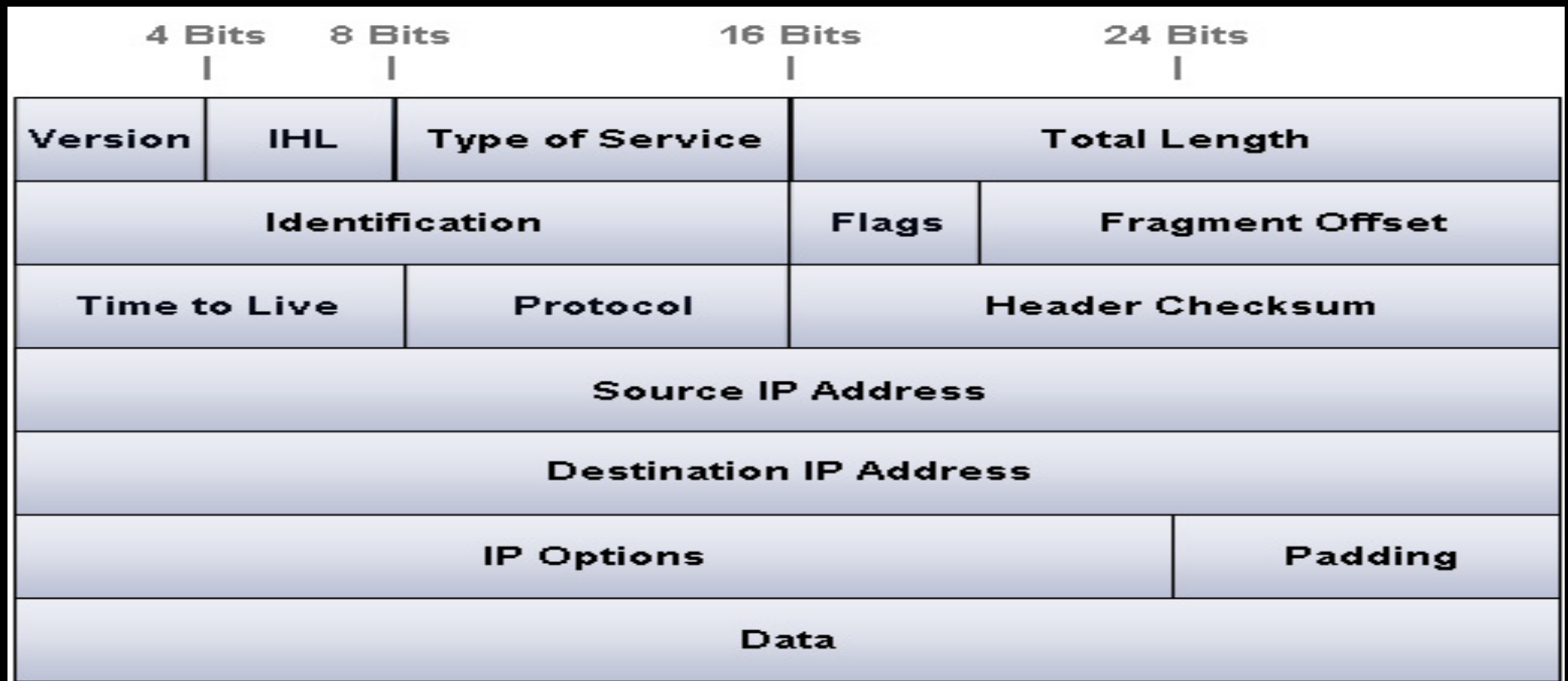
TCP/IP packet headers

One of the most frequent uses of raw sockets is for crafting our own TCP and/or IP headers.

However that requires an intimate knowledge of at least the basics of TCP/IP functionality → Stevens TCP/IP Illustrated vol1, RFC 791-793 (see ICMP too – much potential there)

Let's inspect both layer headers:

TCP/IP packet headers



TCP/IP packet headers

Interesting remarks about IP header:

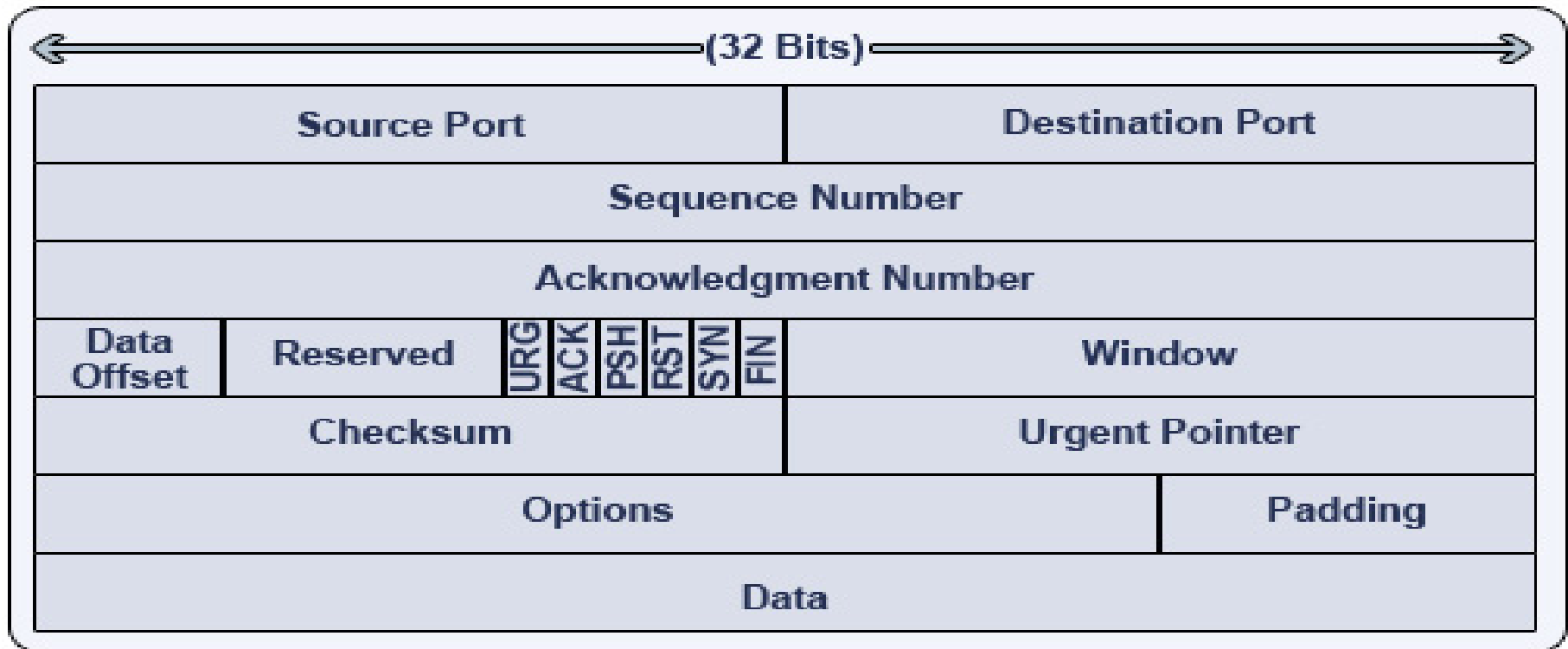
- version != (4 || 6) → reject
- IHL → always a multiple of 4 → max = 60 bytes
- TOS → often not used
- IP id → predictable = Idle scan
- DF → path mtu discovery
- Fragments: complex attacks (daemon9: teardrop.c for linux/win NT overlapping IP fragment / rhino9: nestea.c)

TCP/IP packet headers

Interesting remarks about IP header:

- Protocol → transport level
- TTL → traceroute/paratrace concept
- Checksum: kernel always fills it
- Options: Record Route (very limited), Source routing (traceroute -g (loose)), Timestamp, ICMP has better ways to provide this and even more functionality

TCP/IP packet headers



TCP/IP packet headers

Interesting remarks about TCP header:

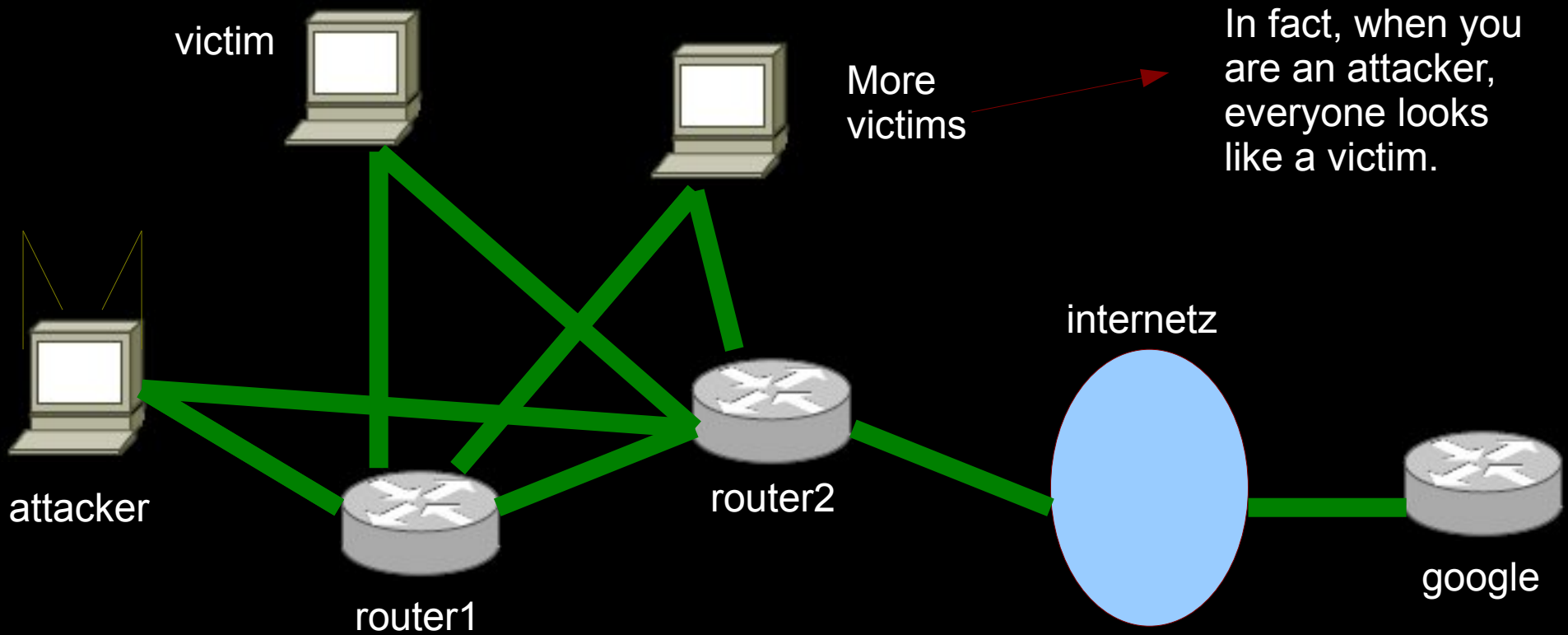
- Different combinations of flags UAPRSF → portscanning techniques
- Usual TCP options: MSS
- Sequence Numbers: predictable PRNG → blind spoofing/reset/injection
- Urgent Data: OOB data (unusual)
- Checksum: make sure it's correct (unless you don't want to ;))

Test case: ICMP redirection tool

Best way to learn is by example:

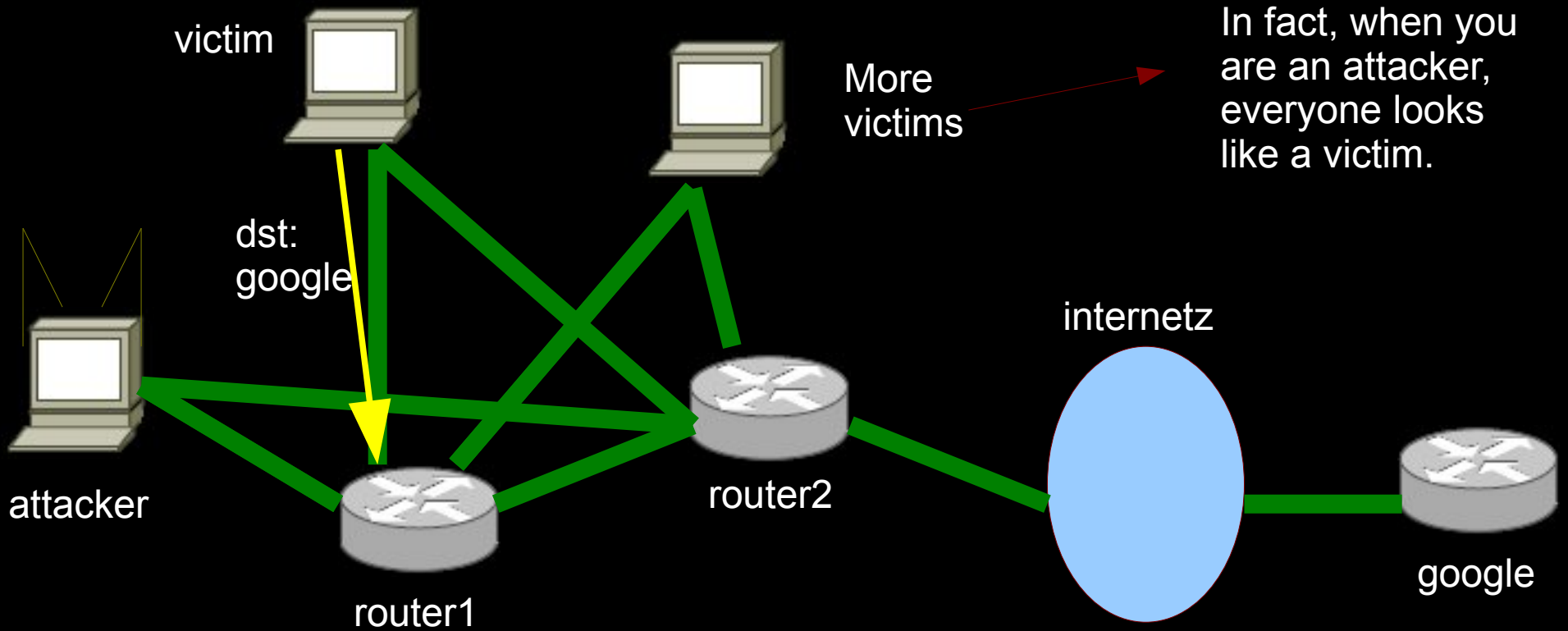
- ICMP redirection – old attack, still works
- very simple to implement:
 - craft our own ip header + icmp header
 - fiddle with the destination/sources
 - `/proc/sys/net/ipv4/conf/all/accept_redirects` (it's on by default ;)
 - Profit!

ICMP redirection tool



ICMP redirection tool

NORMAL SCENARIO



ICMP redirection tool

OK, but next time send to router2 for dst: google

victim



NORMAL SCENARIO



More victims

In fact, when you are an attacker, everyone looks like a victim.



attacker

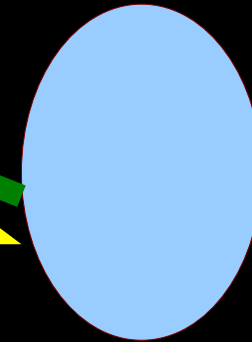


router1

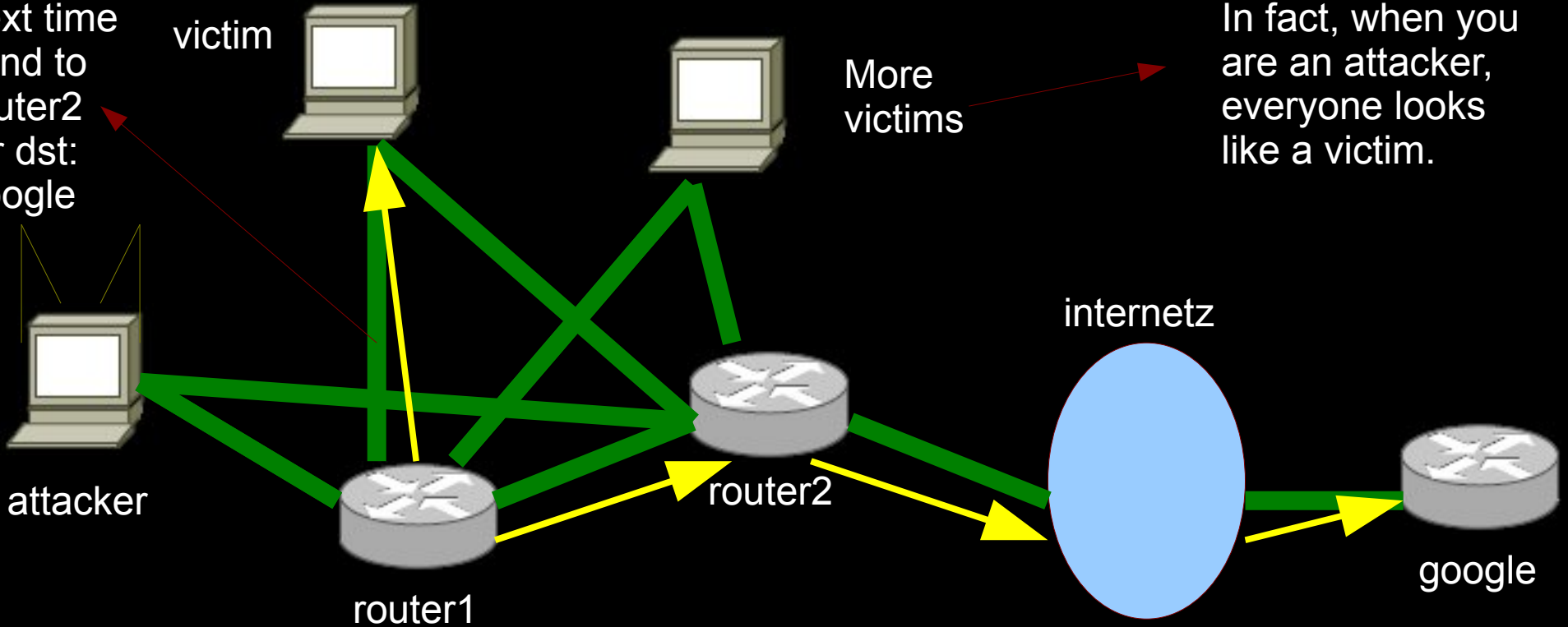


router2

internetz

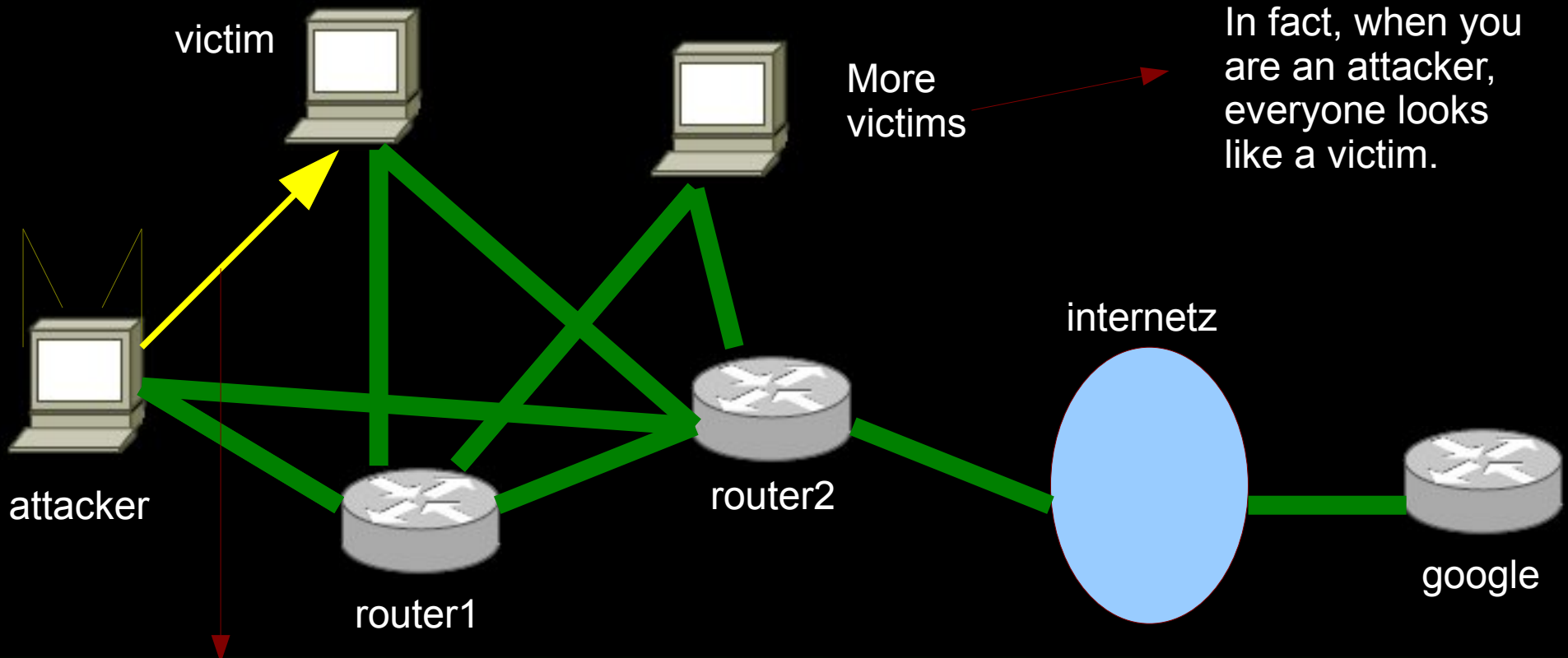


google



ICMP redirection tool

ATTACKING SCENARIO



In fact, when you are an attacker, everyone looks like a victim.

I am router1 speaking here. Look pal. We 've got a problem. Why don't you start sending everything normally destined for router2 to 'arbitrary' instead to 'fix' the situation?

ICMP redirection tool

ICMP redirect header

type code

5	0	cksum
IP address		
IP header + (optional) 8 bytes of orig. IP data		

ICMP redirection tool

```
struct raw_pkt {  
    struct iphdr ip;  
    struct icmphdr icmp;  
    struct iphdr encl_iphdr;  
    char encl_ip_data[8];  
};
```

```
struct raw_pkt* pkt;
```

```
sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);  
setsockopt(sock, IPPROTO_IP, IP_HDRINCL, (const char  
*)&on, sizeof(on));
```

ICMP redirection tool

```
pkt->ip.version = 4;
pkt->ip.ihl = sizeof(struct iphdr) >> 2;
pkt->ip.tos = 0;
pkt->ip.tot_len = sizeof(struct raw_pkt);
pkt->ip.id = htons(getpid() & 0xFFFF); // whatever
pkt->ip.frag_off = 0;
pkt->ip.ttl = 0x40;
pkt->ip.protocol = IPPROTO_ICMP; // layer 4 = ICMP
pkt->ip.check = 0; // kernel fills it
pkt->ip.saddr = faked_router_ip;
pkt->ip.daddr = victim_ip;
pkt->ip.check = checksum((unsigned short*)pkt, sizeof(struct
iphdr));
```

ICMP redirection tool

```
pkt->icmp.type = ICMP_REDIRECT;  
pkt->icmp.code = ICMP_REDIR_HOST;  
pkt->icmp.checksum = 0;  
pkt->icmp.un.gateway = new_ip; /* this is the ip to which  
the victim will start sending packets from now on when he  
wants to communicate with the host that appears in the  
ICMP enclosed IP header */
```

```
memcpy(&(pkt->encl_iphdr),pkt,sizeof(struct iphdr));  
pkt->encl_iphdr.protocol = IPPROTO_IP;  
pkt->encl_iphdr.saddr = victim_ip;  
pkt->encl_iphdr.daddr = host_to_communicate;
```

Raw socket kernel handlers

- Raw sockets are handled specially by each kernel.
- General idea → kernel avoids meddling with headers fields – user has total control
- Analysed thoroughly at:
sock-raw.homeunix.org/papers/sock_raw
- Linux and BSD have each different approaches
- careful to make portable code

Raw socket kernel handlers

- Reading network kernel code is a life-changing experience. No, really.
- No mind-explosions today. We'll just look into some basic things and give directions.
- `SOCK_RAW` is a fallback protocol on both Linux & *BSD → `socket(AF_INET, SOCK_RAW, XXX)` where `XXX` = protocol unspecified in kernel → raw wildcard entry used, does NOT fail

Raw socket kernel handlers

lookup_protocol:

```
err = -ESOCKTNOSUPPORT; ...
```

```
list_for_each_rcu(p, &inetsw[sock->type]) {
```

```
    answer = list_entry(p, struct inet_protosw, list);
```

```
    if (protocol == answer->protocol) {
```

```
        if (protocol != IPPROTO_IP)
```

```
            break;
```

/usr/src/linux/net/ipv4/af_inet.c

```
    } else {
```

```
        if (IPPROTO_IP == protocol) {
```

```
            protocol = answer->protocol;
```

```
            break;
```

```
        }
```

```
        if (IPPROTO_IP == answer->protocol)
```

```
            break;
```

```
    }
```

```
err = -EPROTONOSUPPORT;
```

```
answer = NULL;
```

```
} ...
```

```
}
```

Raw socket kernel handlers

- The code above infers the following:
 - `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);` → OK
 - `socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);` → OK
 - `socket(AF_INET, SOCK_STREAM, 0);` → OK
 - `socket(AF_INET, SOCK_DGRAM, 0);` → OK
 - `socket(AF_INET, SOCK_RAW, 0);` → `EPROTONOSUPPORT`
 - `socket(AF_INET, SOCK_STREAM, XXX);` where `XXX != IPPROTO_TCP`
→ `EPROTONOSUPPORT`
 - `socket(AF_INET, SOCK_DGRAM, XXX);` where `XXX != IPPROTO_UDP`
→ `EPROTONOSUPPORT`
 - `socket(AF_INET, SOCK_RAW, XXX);` where `XXX != 0` → OK
this is the fallback case

Things to watch out for

- Raw sockets need subtle care, since **many** things can actually go wrong:
 - Incorrect checksums → DROP
 - Malformed packets (invalid fields) → DROP
 - Network instability → enraged sysadmin!
 - Portability issues → see IP_HDRINCL and sock_raw paper
 - Linux kernel always fills ip->tot_len (unless customized of course)

Things to watch out for

- Interesting things:
 - Specially crafted packets → kernel/service exploitation
 - Use tcpdump -X and cmp output with what you thought you crafted
 - Usually, you'll need libpcap to sniff victim replies
 - Potent tools → nkiller: combination of raw sockets, libpcap, scanrand idea of reverse syn cookies and tcp request handling exploitation → generic DoS: sock-raw.homeunix.org/projects/nkiller.c

Things to watch out for

- More interesting things:
 - The most interesting thing about raw sockets are the **tools** that use them.
 - The API by itself is nothing without a good idea.
 - Good ideas come with experimentation and reading. Some good starting points:
 - nmap.org, sectools.org, seclists.org
 - TCP/IP Illustrated Vol 1,2 / UNP vol 1 / Linux Network Internals
 - phrack.org / packetstormsecurity.org / thc.org / phenoelit-us.org / awarenetwork.org

Part 0x03. Questions

That's it for today. Expand the knowledge.

QUESTIONS?

ithilgore
sock-raw.homeunix.org