PortBunny

A kernel-based port-scanner
Hey, scanning ports is sports
  - It’s not religion
  - It’s competitive

Port scanning is fun for most people
  - Needs random scanning
  - Needs 1337 output
  - Needs 23 different scanning types

Port scanning is work for some people
  - Needs Accuracy
  - Needs Speed
    - Speed → Time → Money
  - Will use dedicated machines
Why not nmap?

- 3 * 255 Hosts in 30 days with nmap
  - I’m actually coming of age
  - Your scanner is not 1337 if it takes 13:37 per host!
  - No, `--disable-waiting-for-things-that-dont-happen` doesn’t cut it

- Professionals don’t scan hosts that are ...
  - ... powered off
  - ... disassembled
  - ... currently being carried around in the office

- Large scale network scanning is application stocktaking, not vulnerability identification
  - Little interest in the one fully filtered host with only port 23420 open
  - Much interest in how many systems in five Class B networks have port 12345 open
And on a more abstract level...

- All discovery methods depend on a single set of information: the list of open, closed and filtered TCP ports
  - OS Fingerprinting
  - Service probing
  - Banner grabbing
- Accordingly, we need this list first, and quickly at that
- While at it, have actual algorithms

Invent & Verify
Our Requirements

- TCP SYN Scanning only, no XMAS trees
- No UDP Scanning
  - UDP scanning is a negative scan method
  - Information value of a UDP scan of a properly firewalled host with UDP services is exactly zero
- Constant access to result data
  - Offloading fingerprinting tasks right when results become available
- Design for embedded use
- Engine design with variable front ends
- Bottom line: Do just one thing, but do it right.

Invent & Verify
Hello DefCon 😊

- Fabian “fabs” Yamaguchi
- 22 years old, lives in Berlin
- Works for Recurity Labs
- Studies Computer-Science/Electrical Engineering at the TUB
- Loves networking and reading/writing code
PortBunny

- **Goal:** Accurate results as fast as possible
- **Difficulties:**
  - Performance in data-networks is a complex topic
  - Large number of different setups
- **Approach:**
  - No “patch-work”-, “handle yet-another-condition”-scanner
  - Base algorithms on strong theoretical foundation
1. Port-Scanning - Basics

Identify open, closed and filtered ports by sending connection requests and observing responses.

(TCP-SYN or “half-open”-scanning)
Naive port-scanner

- Won’t quite do it.
- Sending as fast as possible may result in dropped packets or even congestion collapse.
  - Open/Closed ports will be falsely reported as being filtered.
- The optimal speed, which changes over time, needs to be determined.
... wait a minute...

... but this works just fine:

```python
sock.connect( .. )
while(is_data_left):
    sock.send(data_left)
sock.close()
```

- For TCP connections, you're right: Because TCP takes care of these issues for you!
- But only for data-transfers across established TCP-connections!
Tell us to slow down, please.

- **Q:** Will the network explicitly tell us that we should slow down?

**A:** In general, no.
- Exception: ICMP source-quenches,
- Exception: ECN.

- People want forms of “explicit congestion notification” but they aren’t widely used yet.
What info do we have?

- If a response is received, we have a round-trip-time.

- Packet-drops can be detected given that we know a certain packet should have provoked an answer.

- That's all.
Networks – what we’re dealing with

- Edges: Throughput (Delay), Reliability
- Nodes: Queuing-capacity
Simplification

- Model implicitly suggested by the term “bottleneck” and by experience from socket-programming.

$\text{MinimumThroughputOfNodesInvolved \text{ bps}}$
Optimal speed

- Speed is the number of packets sent per time-frame.

Find the optimal delay.
But don’t forget the queuing-capacity!

- “You can fire 10 packets at a delay of 0 but that doesn’t mean you can do the same with 100 packets.” Why?

- The network has limited ability to queue data.

  - This very important property of the network suggests a new model.
Think of each host as a bucket with a hole at the bottom. The optimal speed has been reached when buckets are at all times filled completely.
New model, new question

- Old question:
  “How long should I wait before sending the next packet”

- New question:
  “How much data can be out in the network at once?”

- “Self-clocked”! New data is inserted when old data leaves.
3. TCP Congestion Control

- TCP congestion control schemes ask that exact same question!
- Very active research-field.
- Let’s make use of those existing results!
## TCP vs. Port-Scanning

<table>
<thead>
<tr>
<th>TCP</th>
<th>Port-Scanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver acks packets.</td>
<td>Packets may not produce answers.</td>
</tr>
<tr>
<td>Timeouts are error-conditions</td>
<td>Timeouts are not error-conditions</td>
</tr>
<tr>
<td>Sequence-numbers are used</td>
<td>No sequence numbers</td>
</tr>
</tbody>
</table>
... in other words:

- The TCP-receiver is cooperative
- A port-scanned host is not cooperative.

- Of course, that doesn’t mean we can’t force it to be.
Triggers - forcing cooperation

- Before starting the scan, find one or more packets which trigger a response.

- **PortBunny tries the following:**
  - TCP-SYN Port 22/80/139/135 …
  - TCP-ACK Port …
  - ICMP-Echo Requests
  - ICMP Timestamp Requests
  - ICMP Address-Mask Requests
  - UDP Port …
  - IP-PROT Protocol …
Send probes in batches

- Terminate each batch with a trigger

- Probes are “attached” to the trigger as payload
- That way, each batch must produce an answer!
On trigger-response

- ... the probability that all probes were received as well, is very high!
- Reason: Drops are caused by
  - Overflowing queues
  - Physical transmission errors (wireless)
- In both cases, drops almost always occur in batches!
- Exception: Random Early Drop
On drop

- All probes of the batch, which did not produce answers must be resent.
- To detect probes, which were dropped although their trigger was not, rescan filtered ports if their number is smaller than 30%.
- This increases accuracy.
What’s that good for?

- Trigger-responses now play the same role Acknowledgments play in TCP’s congestion control!
- We receive constant information about the network’s performance no matter if it is largely filtered or not!
- A timeout is actually a signal of error!
Probe-based congestion control (NMAP)

NMAP on a responsive host

Drop detected

ssthresh has been divided by 2

Going into cong. avoidance
... and the same for a filtered host

NMAP scanning a mostly filtered host

An open port has been identified!
Port Scan Ping

If a host has not responded in 5 seconds, a ping is sent.

A response is then counted as 3 regular responses.

This is called the “port scan ping”-system.

/* When a successful ping response comes back, it counts as this many "normal" responses, because the fact that pings are necessary means we aren't getting much input. */
... and then there are filtered hosts 😊

- 65535 ports, mostly filtered, Internet.
Timeout-detection - probe-based

- Drops can only be detected after resending
- If a resent probe produces an answer, obviously, the initial probe was dropped.

/* A previous probe must have been lost ... */.
# Triggers vs. TCP

<table>
<thead>
<tr>
<th>TCP</th>
<th>Trigger-based scanning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver acks packets.</td>
<td>Triggers are acknowledged.</td>
</tr>
<tr>
<td>Timeouts are error-conditions</td>
<td>Trigger-Time-Outs are error-conditions.</td>
</tr>
<tr>
<td>Sequence-numbers are used</td>
<td>Sequence-numbers are used for all triggers.</td>
</tr>
</tbody>
</table>
Benefits of trigger-use

- Filtered hosts can be scanned properly.
- Packet-drops can be detected much earlier leading to better responsiveness to drops.
- Immediate probe resends are not necessary anymore which helps reduce useless extra traffic.
- Port-Scanning has been ported to the tcp-congestion control domain! We can implement any TCP-congestion-control scheme!

Invent & Verify
Problems with triggers

- Not all triggers have the same quality:
  - ICMP-triggers and UDP-triggers could be rate-limited while probes aren’t.
  - TCP-triggers are the best available triggers.
  - QoS might be a problem, some times

- A host may not respond to any supported trigger.
Fixes

- Try to find TCP-SYN-triggers first and use ICMP and UDP-triggers and others as a fallback-solution.

- If a TCP-SYN-trigger can be found at scan-time, add it to the list of triggers in use and discard fallback-triggers.
Problem solved? Not quite:

- Bucket-model is NOT valid for rate-limiting firewalls, instead, the pipe-model is valid!
  - using classical congestion-control algorithms in this case is not appropriate.

- We have implemented a number of congestion-control-schemes designed for TCP but how will the user know which one to choose?
We need detection

- The scanner needs to be able to interpret network-conditions and choose a timing-algorithm, which is most suited by itself.

- The scanner is the expert on these issues because it’s communicating with the target!
Detection of rate-limiting firewalls
NMAP’s rate-limit detection

/* If packet drops are particularly bad, enforce a delay between packet sends (useful for cases such as UDP scan where responses are frequently rate limited by dest machines or firewalls) */

- “Particularly bad” is vague => False positives are common
- Artificial delays mean that the algorithm is no longer self-clocked 😞 No more theoretical foundation for the timing-algo.
- CWND does no longer reflect the number of packets out at once.

Invent & Verify
Effect of false-positive
Let's take the RTT into account!
What does this look like during a scan?

- When scanning, we constantly change the network-load in reaction to network-events.
- How does that effect the rtt?
- Is the data clear enough to differentiate between normal congestion and the effects of a rate-limiter?
Yes, it is ☺
Working on this approach

- If \( n \) drops occur in a row and the RTT observed is relatively constant and close to the base-RTT, then we are dealing with a rate-limiting firewall.

- This is still being tested, but another nice “two-shot” approach has been implemented.
Observe: This is a packet...

Frame 161 (58 bytes on wire, 58 bytes captured)

Ethernet II, Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)

Internet Protocol, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)

Transmission Control Protocol, Src Port: 61373 (61373), Dst Port: tacacs-ds (65), Seq: 0, Len: 0

Source port: 61373 (61373)
Destination port: tacacs-ds (65)
Sequence number: 0 (relative sequence number)
Header length: 24 bytes

Flags: 0x02 (SYN)
Window size: 0
Checksum: 0x2a25 [correct]
Options: (4 bytes) Maximum segment size: 1460 bytes
... and this is, too.
Now if the bucket claims…

- … that 4 of these fit:

- … but 4 of those would, too:

Invent & Verify
... then it's not really a bucket.

Invent & Verify
Packet-size does not matter for rate-limiters!

- Rate-limitation limits **number of packets**, the packet-size does not matter!
- Congestion is caused by **too much data** in the network
- **Just enlarge the packet** (Add TCP-options)
- If still the same number of packets return, we're obviously dealing with rate-limitation.
Or create background-traffic

- In practice, enlarging the TCP-SYN by 40 byte doesn’t change much.
- In contrast, for ICMP-Echo-Request-triggers, the approach is feasible.
- For all other triggers, background-traffic can be generated instead.
- UDP-datagrams are a good option.
<table>
<thead>
<tr>
<th>Chain</th>
<th>policy</th>
<th>target</th>
<th>prot</th>
<th>opt</th>
<th>source</th>
<th>destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>ACCEPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORWARD</td>
<td>ACCEPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUTPUT</td>
<td>ACCEPT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
File Edit View Terminal Tabs Help
fabs@fabs-laptop:$ sudo ./rlimit_test.sh 193.239.165.73
Aug  8 00:07:46 fabs-laptop kernel: [ 5954.536000] nresponses_first_round: 49
Aug  8 00:07:46 fabs-laptop kernel: [ 5954.536000] nresponses_second_round: 28
```
<table>
<thead>
<tr>
<th>Source Port</th>
<th>Destination Port</th>
<th>Source IP</th>
<th>Destination IP</th>
<th>Protocol</th>
<th>Seq</th>
<th>Ack</th>
<th>Len</th>
<th>MSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61378</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61379</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61380</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61381</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61382</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61383</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61384</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61385</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61386</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61387</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61388</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61389</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61390</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61391</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
<tr>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>10.248.240.254</td>
<td>193.239.165.73</td>
<td>TCP</td>
<td>61392</td>
<td>81</td>
<td>MSS=1460</td>
</tr>
</tbody>
</table>
Invent & Verify
Using Bunny

- Download it at http://portbunny.recurity.com

- "# portbunny $host" to scan a host
- "# portbunny $network" to scan a network
- "# portbunny $network -d" to only trigger
  $network for discovery-purposes
- "# portbunny $host -l" to scan and generate a
detailed scan-log.

Invent & Verify
Send us your scan-log ;)
Contains all relevant info
One more cool trick ;)!

- … this “burst-response” just looks so pretty, is there really nothing we can do with it?
Burst-response on wired Ethernet

Error values:

- 0.0589
- 0.0737
- 0.0322
- 0.0439
- 0.0421
- 0.0422

Mean = 0.0488
... and on wireless 😊

error [] =

0.1706
0.0823
0.1154
0.1052
0.0935
0.1578

Mean =

0.1208
Problems with this approach

- Watch out! If the test-result is positive, we know that there is a rate-limiter. If it’s negative, we don’t know anything!

- One bucket, which cannot hold the entire burst at once could be the limit.
Two types of solutions

- **One/Few-Shot Solutions:**
  - Is it possible to detect the rate-limiter with a few shots of data?

- **At-Scan-Time Solutions**
  - Is it possible to detect the rate-limiter by observing the network’s behavior during the scan?
Burst-response

- Can we just send a burst of triggers, which is big enough and measure the RTT to classify the bottleneck?
- RTT-increase is exponential $\Rightarrow$ congestion
- Else: Firewall
Nope, doesn’t work for bursts
So, were the books wrong?

- Result is only valid if data is offered at a constant rate over a certain time.
- Scanning does mean offering data at a certain rate over a longer time-period.
  - Detection could be done during the scan!
  - But the rate is constantly changed by the timing-algorithm in use, will measurements be clear enough?
Consequence

- To stay responsive to drops, probes that may have just dropped must be resent straight away!
- This makes you extremely vulnerable to the “late-responses”-problem
- (... and to “port-cloaking”, btw)
“Late-responses” Problem

If the approximation of the timeout is too optimistic, responses arrive shortly after the resend has occurred.

→ Lots of unnecessary traffic which reduces the scanning-speed.
When integrating sequence-numbers into triggers, an algorithm similar to **fast-retransmit** can be implemented:

**Example:**
- Responses for 7, 8 and 9 have been received but there’s no response for 6.
- One can assume that 6 has been dropped even if its timeout-value has not been reached!