Port Scanning improved
New ideas for old practices

http://www.recurity-labs.com
Why reinventing wheels?

- The world changes constantly
- The world is full of crappy software
- The requirements of software change
  - When the requirements changed sufficiently, the software no longer fits the purpose
  - Some software didn’t fit the purpose to begin with
- Reality is your measure
Redo-Software: When to start?

- Only for people who have a **realistic** chance to actually finish the project
  - The crappy original is still better than the unfinished sequel
- Extrapolate if the problem you are planning to solve is going to get better or worse in the future without your solution.
- Don’t make a schedule!
- Make it as good as you possibly can!

*Invent & Verify*
Redo-Software:
How do you start?

- Set your requirements
  - Remember, they are your requirements
  - Don’t try to please everyone you talk to, tell them to fsck off
- Don’t import requirements from the existing software
  - Do you really need to be portable?
  - Do you really have to have this feature?
- Don’t read too much of the “other” code
  - Think for yourself first
  - Compare your solution with the “other” code later
Warning: Redo-Software is uncool!

“But I want to research quantumcybercryptofeminism and its impact on onion-routed RFID Sex 2.0!”

Go ahead!
A Port Scanner? *Yawn*

- 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

Invent & Verify
My hat is off to Fyodor!

- nmap was the first general purpose port scanning tool available
  - Some of you might remember the times when you had to use synscan or similar
  - Nobody really misses them
- nmap introduced many important inventions
  - Granted, most do not belong into a port scanner
  - They are nice and useful anyway
- Redo-Software just doesn’t mean the original is bad, worthless or outdated
  - It just means you need something else

Invent & Verify
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.
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.
PortBunny
A kernel-based port-scanner

Invent & Verify
PortBunny

- Portbunny scans faster by sending more
- Portbunny builds a bridge between TCP congestion control and port-scanning.
- Portbunny shows that vanilla TCP-SYN port-scans already leave you with lots of room for research.
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.
- Optimal speed may change over time!
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 for IPv6
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.
2. A network model

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

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

$\text{MinimumThroughputOfNodesInvolved \ bps}$
Optimal speed

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

Find the optimal delay.

Invent & Verify
So much for theory...

- … but finding the optimal delay will fail in practice!
The round-trip-time problem

- Dropped packets can’t be detected before a complete round-trip-time has passed.
- At that time about rtt/delay other packets have already been sent to maintain the “optimal delay”.

Drop detected!

Drop detected, but way too late :/
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?”
TCP Congestion Control

- TCP congestion control schemes ask that exact same question!

- Note: NMAP’s timing-code is based on the classic TCP-congestion-control algorithm “TCP-Reno”.

Invent & Verify
Doesn’t that work automatically?

- Why do we have to implement congestion control at all?
- Doesn’t TCP provide congestion control to upper layers?
- No established TCP-connection
- Control the emission of IP-packets which happen to be TCP-SYNs.

Application-Layer (HTTP/FTP/SSH)

Transport-Layer (TCP/UDP/ICMP/IGMP)

Network-Layer (IPv4/IPv6/ARP...)

Data-Link-Layer (Ethernet/PPP/Token-ring)

Physical Layer
# 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 my 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>

Invent & Verify
... 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:
  - ICMP-Echo Requests
  - ICMP Timestamp Requests
  - ICMP Address-Mask Requests
  - TCP-SYN Port 22/80/139/135 …
  - UDP Port …

Invent & Verify
Inserting triggers into the probe-stream

- Insert these packets into the packet-stream and base your timing-code on the triggers

<table>
<thead>
<tr>
<th>SYN 10</th>
<th>SYN 140</th>
<th>TRIGGER</th>
<th>SYN 164</th>
<th>SYN 24</th>
<th>TRIGGER</th>
</tr>
</thead>
</table>

Invent & Verify
What’s that good for?

- Trigger-responses now play the same role that 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!
What NMAP Had in Mind

NMAP on a responsive host

- Drop detected
- ssthresh has been divided by 2

Going into congestion avoidance

Invent & Verify
What nmap forgot.

NMAP scanning a mostly filtered host

An open port has been identified!

Invent & Verify
But let’s be fair:

- If a host has not responded in **5 seconds**, a ping is sent.
- A response is then counted as **3 regular responses**.
- *g*

/* 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 😊

- 66535 ports, mostly filtered, Internet.
Why mention Sequence-Numbers?

- An Ack is sent by the receiver for each packet
- Duplicate Acks indicate packet-loss!
- Fast-retransmit

Next seq-num expected: 2

Out-of-order-queue
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!
NMAP – Timeout-detection

- NMAP can only detect drops after resending.
- If a resent probe produces an answer, obviously, the initial probe was dropped.
- Each probe has its own timeout-clock. That doesn’t scale well, so there are interesting hacks to solve this.
Consequence

- To stay responsive to drops, NMAP must resend the probe that may have just dropped straight away!

- This makes NMAP extremely vulnerable to the “late-responses”-problem
“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.
Defeating late-responses (with triggers)

PortBunny does not rely on immediate resends to detect packet-loss!

→ The probe can be resent after ALL other unknown ports have been probed!
## 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-Timeouts 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 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.
Racing on responsive hosts

- PortBunny sends 10% more data because of the triggers? Can it still compete with NMAP on responsive hosts?
Nothing's for free

- 65535 ports, mostly closed, WRT.
... but that doesn't count much.
... still PortBunny often wins this race
And then there are serious bugs

/* 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) */

Translates to: If packet-drops are particularly bad, break the entire timing-concept.

⇒ The CWND will not reflect the number of probes out at once anymore!

⇒ The self-clocking-property is being ignored!
Scanning the IPHONE

24:41.51 m

Invent & Verify

7:58.03 m
Scanning in parallel

- PortBunny can scan a large number of hosts in parallel but by default, it will scan one host at a time. Why?

- Is a parallel scan always faster than a sequential scan?
Bottlenecks

Will parallel scans win?
Shared bottleneck

- If there’s a bottleneck shared among all scan-jobs (common case), then there is no gain in scanning in parallel!

- … assuming that the congestion-control-scheme actually works correctly (even for filtered hosts!)

- In fact, more unpredicted drops will occur and they will slow us down!
Target-bottlenecks

- If the target is the bottleneck, there is a gain in parallel scanning.

- It’s possible to **do timing on a per-host basis** entirely: TCP-congestion-control-schemes were created with this scenario in mind!

- “Fairness” has been considered.

Invent & Verify
What does NMAP do?

- Implement the same timing-algorithm for a global system which is informed of all answers and packet-drops to address shared bottlenecks.

- A scan-job may only send a new packet if the per-host-timing AND the global timing allows that.
Problem with this solution

- A badly performing host (target-bottleneck) will keep good performing host from firing.
- This timing is biased towards the performance of the worst scan-job.
- CWND is not “the number of packets out” anymore => again, the concept was broken.
Portbunny’s solution

- Each scan-job performs its own timing based on a tcp-cc-scheme.
- This is similar to starting several independent http-downloads.
- You can only do that if the congestion-control-scheme actually works!
- By default: scan sequentially because single shared bottlenecks are the most common scenario.
Research in parallel scanning

- Old congestion control schemes must generate losses to find boundaries. Think wireless ;)
- Modern congestion control techniques are based on detecting changes in round-trip-time.
- Correlations between changes in round-trip-time can be used to detect shared bottlenecks!
… which is why Bunny is in the kernel.

- Timing is as precise as it can get.
- The “scanner-bottleneck”-issue for a large number of hosts is addressed not just algorithmically.
- We get a reliable sniffer for free.
Kernel-based sniffer

Port-bunny adds packet-handler by calling

```c
dev_add_pack(struct packet_type *pt)
```

from net/core/dev.c
The user’s perspective

- Chat with /dev/portbunny 😊
- The protocol is text-based and very simple.
- You can use portbunny with cat, echo and friends… but don’t worry, we have a UI.
Example input

- `# echo $command > /dev/portbunny`

- `$command:
  - `create_scanjob 192.168.1.1 FLOOD`
  - `set_ports_to_scan 192.168.1.1 FLOOD 1-500`
  - `execute_scanjob 192.168.1.1 FLOOD`

Invent & Verify
Example output

- `# cat /dev/portbunny`
- `SCAN_JOB_CREATED 192.168.1.1 FLOOD`
- `SCAN_JOB_EXECUTED 192.168.1.1 FLOOD`
- `...`
- `RESULT 192.168.1.1 FLOOD PORT_STATE 79 CLOSED`
- `RESULT 192.168.1.1 FLOOD PORT_STATE 80 OPEN`
- `...`
- `SCAN_JOB_REMOVED 192.168.1.1 FLOOD`
The PortBunny UI

- `$ portbunny host`
- `-p <port|port-range> ...` ports to scan
- `-d` discover-mode
- `-t <trigger> ...` triggers to try
- `-g` generate data for gnuplot

And that’s all.
Thank you!

Recurity Labs GmbH, Berlin, Germany
http://www.recurity-labs.com

Fabian ‘fabs’ Yamaguchi
fabs@recurity-labs.com

Felix ‘FX’ Lindner
fx@recurity-labs.com