Scan-detection Internals: clusterization and netcontrol for active-response

Aashish Sharma

14th September 2016

#BroCon2016
Managed and operated by UC for the U.S. Department of Energy
>200 University of California faculty on staff at LBNL
4200 Employees, ~$820M/year Budget
13 Nobel Prizes
63 members of the National Academy of Sciences (~3% of the Academy)
18 members of the National Academy of Engineering,
2 of the Institute of Medicine
Birthplace of Bro
Over 10,000 visiting scientists (~2/3 from universities) use Berkeley Lab research facilities each year.
Overview

- A case for scan-detection
- Internals of scan-detection
  - what is a scan
- Clusterization and its problems
- Scan-NG features and how are those implemented
- What's in for the future
Philosophically a scan is an attribution or an intentionality problem but operationally we want to make it a measurement problem.

- Partha Banerjee, LBL
Recon

• We want to know if scans are coordinated, distributed*
• What is the scale of a recon?
• what is intention of a recon?
• No clear success criteria of a recon
• Don't even know what attackers found out, although the traffic went through your network

Q. How many incidents are detected at Scan Phase?

Ans: We might not even have an incident yet (at the scan phase)

Q. Of all the incidents we detect, for how many can we go back to and find the scan-phase that might have caused it?

Q. How many incidents happen without any scan-phase/recon?

Why scan-detection?

• Important to know about malicious activity early and quickly
• Attention to recon is as important as any other defense mechanism
 Characteristics of network traffic

Connection States

- **S0**: Connection attempt seen, no reply.
- **S1**: Connection established, not terminated.
- **SF**: Normal establishment and termination.
- **REJ**: Connection attempt rejected.
- **S2**: Connection established and close attempt by originator seen (but no reply from responder).
- **S3**: Connection established and close attempt by responder seen (but no reply from originator).
- **RSTO**: Connection established, originator aborted (sent a RST).
- **RSTR**: Established, responder aborted.
- **RSTOS0**: Originator sent a SYN followed by a RST, we never saw a SYN-ACK from the responder.
- **RSTRH**: Responder sent a SYN ACK followed by a RST, we never saw a SYN from the (purported) originator.
- **SH**: Originator sent a SYN followed by a FIN, we never saw a SYN ACK from the responder (hence the connection was "half" open).
- **SHR**: Responder sent a SYN ACK followed by a FIN, we never saw a SYN from the originator.
- **OTH**: No SYN seen, just midstream traffic (a "partial connection" that was not later closed).
Strategies for scan-detection

• Summary statistics
  – “N” IP or port in “t” time
• Signature Based
  – eg. Metasploit signature
• Behavior Based
  – Nmap scans start with 80/tcp, 443/tcp + icmp
• Probabilistic methods
  – Threshold Random Walk
• know_your_network_approach
  – Knockknock and Landmine
Overly simplified OldScan-1.5.3

- skip sources
- skip services
- skip nets
- skip ports
- UDP check

- Established?
  - Yes
    - PortScan
    - LowPortTroll
  - No
    - backscatter
    - distinct_peers/shutdown
    - Threshold
    - Landmine
    - AddressScan

- $Id: scan.bro 7073 2010-09-13 00:45:02Z vern $  
- 720 lines of code  
- Need to clusterize  
- &sync not useful anymore
scan.bro - One pill to cure all?

Scan detection needs to be broken into many sub-parts

• TCP
• UDP
• ICMP
• IPv4
• IPv6
• external
• internal scanners
Scan-detection: Underlying Reasoning...

- WE KNOW WHAT THEY DON’T KNOW
- WE DON’T KNOW WHAT THEY FOUND OUT
- WE WANT TO KNOW WHAT IS THEY WANT TO KNOW (hopefully before they find it out)
## Heuristics

<table>
<thead>
<tr>
<th>KnockKnock</th>
<th>LandMine</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Incoming remote IP connection and checks it against <strong>table of known-services</strong> for the LBNL IP and accesses if that's a good or bad connection.</td>
<td>• Policy - ingests the list of allocated subnets from a text-file using input-framework</td>
</tr>
<tr>
<td>• Policy is adaptive based on popularity of ports.</td>
<td>• Any connection not in the above list is a Darknet Connection</td>
</tr>
<tr>
<td></td>
<td>• “N” such connections lead to a conclusion that this is a scanner</td>
</tr>
<tr>
<td></td>
<td>• Block the IP.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AddressScan &amp; LowPortTrolling</th>
<th>Backscatter</th>
</tr>
</thead>
<tbody>
<tr>
<td>• “Bro treats connections differently depending on application protocol.</td>
<td>• Generally Victims of DoS attacks</td>
</tr>
<tr>
<td>• Bro only performs bookkeeping if the connection attempt failed (was either unanswered, or elicited a TCP RST response).</td>
<td>• result of address spoofing</td>
</tr>
<tr>
<td>• For others, it considers all connections, whether or not they failed. It then tallies the number of distinct destination addresses to which such connections (attempts) were made.</td>
<td>• Not really scanners</td>
</tr>
<tr>
<td>• If the number reaches a configurable parameter N, then Bro flags the source address as a scanner. By default, Bro sets N = 100”</td>
<td></td>
</tr>
</tbody>
</table>
Potential issues with clusterization of scan-detection

- Communication overhead - Scan detection is kind of the worst-case for distributed analysis: one needs to count across *all* connections.
- In a cluster we split things up via load-balancing, but for scan detection we need to essentially revert that through communication.
- Timely state synchronization across the workers
- Scans are unpredictable rates so cannot employ epochs – need to detect fast and slow scanners both
- How to implement dynamic thresholds
- Detection needs to run in both cluster and standalone setup
<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection_attempt</td>
<td>This event is raised when an originator unsuccessfully attempted to establish a connection. “Unsuccessful” is defined as at least <strong>tcp_attempt_delay</strong> seconds having elapsed since the originator first sent a connection establishment packet to the destination without seeing a reply.</td>
</tr>
<tr>
<td>connection_established</td>
<td>Generated when seeing a SYN-ACK packet from the responder in a TCP handshake. An associated SYN packet was not seen from the originator side if its state is not set to TCP_ESTABLISHED. The final ACK of the handshake in response to SYN-ACK may or may not occur later, one way to tell is to check the history field of connection to see if the originator sent an ACK, indicated by ‘A’ in the history string.</td>
</tr>
<tr>
<td>connection_half_finished</td>
<td>Generated when one endpoint of a TCP connection attempted to gracefully close the connection, but the other endpoint is in the TCP_INACTIVE state. This can happen due to split routing, in which Bro only sees one side of a connection.</td>
</tr>
<tr>
<td>connection_pending</td>
<td>Generated for each still-open TCP connection when Bro terminates.</td>
</tr>
<tr>
<td>connection_rejected</td>
<td>Generated for a rejected TCP connection. This event is raised when an originator attempted to setup a TCP connection but the responder replied with a RST packet denying it.</td>
</tr>
<tr>
<td>connection_reset</td>
<td>Generated when an endpoint aborted a TCP connection. The event is raised when one endpoint of an established TCP connection aborted by sending a RST packet.</td>
</tr>
<tr>
<td>connection_state_remove</td>
<td>Generated when a connection’s internal state is about to be removed from memory. Bro generates this event reliably once for every connection when it is about to delete the internal state. As such, the event is well-suited for script-level cleanup that needs to be performed for every connection. This event is generated not only for TCP sessions but also for UDP and ICMP flows.</td>
</tr>
<tr>
<td>new_connection</td>
<td>Generated for every new connection. This event is raised with the first packet of a previously unknown connection. Bro uses a flow-based definition of “connection” here that includes not only TCP sessions but also UDP and ICMP flows.</td>
</tr>
<tr>
<td>partial_connection</td>
<td>Generated for a new active TCP connection if Bro did not see the initial handshake. This event is raised when Bro ( observed traffic from each endpoint, but the activity did not begin with the usual connection establishment</td>
</tr>
</tbody>
</table>
ARCHITECTURE DIAGRAM - 1

- con-attempt
- established
- half-fin
- pending
- rejected
- reset
- state-em
- partial-cont

1. IS-Scanner
   - IP-white-list
   - subnet WL
   - skip:scan-source
   - skip:services
   - skip:outbound-serv
   - skip:scan-nets
   - skip:dst:serv:ports
   - known:scanner

2. Activate
   - Check
     - Address Scan
     - Port Scan
     - LandMine
     - Backscatter
     - Check LandMine
     - Check Backscatter
     - Knock
     - Knock
     - TRW
     - TRW
### Filtration – what qualifies (or not qualifies) as a potential scan candidate

<table>
<thead>
<tr>
<th>Desc</th>
<th>KnockKnock</th>
<th>LandMine</th>
<th>BackScatter</th>
<th>AddressScan</th>
</tr>
</thead>
<tbody>
<tr>
<td>$proto == TCP</td>
<td>Only TCP connections</td>
<td>Only TCP and ICMP</td>
<td>Only TCP connections</td>
<td>TCP and ICMP (UDP disabled by default )</td>
</tr>
<tr>
<td>Internal Scanners handled separately</td>
<td>Internal scanners handled separately</td>
<td>- NA -</td>
<td>Internal host scanning handled separately</td>
<td>Internal host scanning handled separately</td>
</tr>
<tr>
<td>DARKNET</td>
<td>Fast-track Darknet</td>
<td>Fast-track Darknet</td>
<td>-</td>
<td>we ignore all darknet connections since LandMine will take care of it</td>
</tr>
</tbody>
</table>

(c$resp$state == TCP_ESTABLISHED) OR (if (isF in c$conn$state))

<table>
<thead>
<tr>
<th>Desc</th>
<th>KnockKnock</th>
<th>LandMine</th>
<th>BackScatter</th>
<th>AddressScan</th>
</tr>
</thead>
<tbody>
<tr>
<td>full established conns not interesting</td>
<td>full established conns not interesting</td>
<td>Full established conns not interesting</td>
<td>Full established conns not interesting</td>
<td></td>
</tr>
</tbody>
</table>

Min_Subnet_check

if ( (|Site::subnet_table| < MIN_SUBNET_CHECK) return F ;

(state == "OTH" &
resp_bytes >0 )

# mid stream traffic - ignore

<table>
<thead>
<tr>
<th>Desc</th>
<th>KnockKnock</th>
<th>LandMine</th>
<th>BackScatter</th>
<th>AddressScan</th>
</tr>
</thead>
<tbody>
<tr>
<td># mid stream traffic - ignore</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pass/fail criteria

ignore traffic to host/port this is primarily whitelisting

if ( (is_failed(c) | is_reverse_failed(c) )

(c$orig$state == TCP_SYN_ACK_SENT && c$resp$state == TCP_INACTIVE) OR
(c$orig$state == TCP_SYN_SENT && c$resp$state == TCP_INACTIVE) OR
(c$history == "F" || c$history == "R" ) OR
(c$history == "H" && /s|a/ lin c$history )

Ignore if :
1) outbound && service in skip_outbound_services
2) local_address
3) orig in skin_scan_sources
4) orig in skip_scan_nets
5) outbound and [resp, service] in skip_dst_server_ports

---

![Berkeley Lab Logo](image-url)
module Clus;

expose {
    global m_w_add: event (ip: addr);
    global w_m_new: event (ip: addr);
    global add_to_cache: function(ip: addr);
    global intermediate_cache: table [addr] of string &redefine;
}

@if ( Cluster::is_enabled() )
@load base/frameworks/cluster
redefine Cluster::manager2worker_events += /Clus::m_w_add/;
redefine Cluster::worker2manager_events += /Clus::w_m_new/;
@endif

function log_reporter(msg: string)
{
    event reporter_info(current_time(), msg, peer_description);
}

event new_connection(c: connection)
{
    local ip = c$id$orig_h;
    if (ip !in intermediate_cache)
    {
        add_to_cache(ip) ;
    }
}

function add_to_cache(ip: addr)
{
    log_reporter(fmt("add_to_cache %s", ip));
    intermediate_cache[ip] = fmt("%s",peer_description);
    @if ( Cluster::is_enabled() )
    event Clus::w_m_new(ip);
    @endif
}

@if ( Cluster::is_enabled() && Cluster::local_node_type() == Cluster::MANAGER )
event Clus::w_m_new(ip: addr)
{
    log_reporter(fmt("w_m_new: %s", ip));
    if (ip in intermediate_cache)
        return;
    intermediate_cache[ip] = fmt("%s",peer_description);
    event Clus::m_w_add(ip);
}
@endif

@if ( Cluster::is_enabled() && Cluster::local_node_type() != Cluster::MANAGER )
event Clus::m_w_add(ip: addr)
{
    log_reporter(fmt("m_w_add: %s", ip));
    intermediate_cache[ip] = fmt("%s",peer_description);
}@endif
<table>
<thead>
<tr>
<th>Heuristic</th>
<th>OldScan</th>
<th>scan-NG</th>
</tr>
</thead>
<tbody>
<tr>
<td>LandMine</td>
<td>Limited: Manual define Landmine addresses</td>
<td>Extensive - derives allocated vs unallocated subnets</td>
</tr>
<tr>
<td></td>
<td><code>const landmine_address: set[addr] &amp;redef;</code></td>
<td><code>if (resp in Site::local_nets &amp;&amp; resp !in Site::subnet_table)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Extended feature</strong></td>
</tr>
<tr>
<td>AddressScan</td>
<td>Same</td>
<td>No Change</td>
</tr>
<tr>
<td></td>
<td><code>global distinct_peers: table[addr] of set[addr]</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Consistent</strong></td>
</tr>
<tr>
<td>Shutdown Threshold</td>
<td>Same &gt; N failures</td>
<td>No change</td>
</tr>
<tr>
<td>Backscatter</td>
<td>Limited to a few ports</td>
<td>Port Agnostic</td>
</tr>
<tr>
<td></td>
<td><code>const backscatter_ports = {80/tcp, 53/tcp, 53/udp, 179/tcp, 6666/tcp, 6667/tcp, } &amp;redef;</code></td>
<td><code>Extended feature</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relies on a new logic to infer reflection attacks and static src ports</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>if (|distinct_backscatter_peers[orig][orig_p]| &lt; 2)</code></td>
</tr>
<tr>
<td>Knockknock</td>
<td>Did not exist</td>
<td>Maintains list of valid services in the network</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tracks failed connections to non-existing services</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses really low and dynamic thresholds</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>clusterized</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>New</strong></td>
</tr>
<tr>
<td>false +ve</td>
<td>Plenty due to directionality problems due to content_gaps</td>
<td>Very few overall - still testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Improvement</strong></td>
</tr>
<tr>
<td>Memory</td>
<td>tables and sets</td>
<td>use hyperloglog (opaque of cardinality) resulting in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80% less memory usage</td>
</tr>
</tbody>
</table>
Performance and features

- Memory mgmt
- Speed detection
- Accuracy
- dynamic thresholds
- Realtime whitelists
- FP identification
event new_connection(c: connection) {
    # for new connections we just want C to the darknet spaces
    # to speed up reaction time and to avoid tcp_expire_delays of 5.0 sec
    if (gather_statistics) {
        s_counters$event_peer = fmt ("%s", peer_description);
        s_counters$new_conn_counter += 1;
    }
}

function is_catch_release_active(cid: conn_id): bool {
    if (gather_statistics)
        s_counters$is_catch_release_active += 1;
}

function check_scan(c: connection, established: bool, reverse: bool) {
    local orig=c$id$orig_h;
    ### already a known_scanner
    if (orig in Scan::known_scanners && Scan::known_scanners[orig]$status) {
        if (gather_statistics)
            s_counters$already_scanner_counter += 1;
        return ;
    }
    if (not_scanner(c$id)) {
        if (gather_statistics)
            s_counters$not_scanner += 1;
        return ;
    }
}
<table>
<thead>
<tr>
<th>Counter Name</th>
<th>Counters ~ 1 day</th>
<th>Counters ~ 7 days</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_conn_counter</td>
<td>184,772,975</td>
<td>1,569,935,400 (100%)</td>
</tr>
<tr>
<td>is_catch_release_active</td>
<td>273578054 (148%)</td>
<td>2,382,883,254 (151.78%)</td>
</tr>
<tr>
<td>not_scanner</td>
<td>170877124 (92.47%)</td>
<td>797,378,521 (50.79%)</td>
</tr>
<tr>
<td>darknet_counter</td>
<td>62747298 (33.95%)</td>
<td>103,620,129 (6.60%)</td>
</tr>
<tr>
<td>not_darknet_counter</td>
<td>13601622 (7.36%)</td>
<td>320,578,718 (20.41%)</td>
</tr>
<tr>
<td>already_scanner_counter</td>
<td>79308450 (42.92%)</td>
<td>435,007,325 (27.70%)</td>
</tr>
<tr>
<td>filter_entry</td>
<td>58024703 (31.40%)</td>
<td>384,651,055 (24.5%)</td>
</tr>
<tr>
<td>filter_success</td>
<td>27135590 (14.68%)</td>
<td>185,705,196 (11.82%)</td>
</tr>
<tr>
<td>c_knock_filter</td>
<td>58024703 (31.40%)</td>
<td>384,651,055 (24.5%)</td>
</tr>
<tr>
<td>c_knock_checkscan</td>
<td>21936393 (11.87%)</td>
<td>151,338,638 (9.63%)</td>
</tr>
<tr>
<td>c_land_filter</td>
<td>21392978 (11.57%)</td>
<td>384,651,055 (24.5%)</td>
</tr>
<tr>
<td>c_land_checkscan</td>
<td>19848677 (10.74%)</td>
<td>32,029,192 (2.04%)</td>
</tr>
<tr>
<td>c_backscat_filter</td>
<td>58024703 (31.40%)</td>
<td>384,651,055 (24.5%)</td>
</tr>
<tr>
<td>c_backscat_checkscan</td>
<td>2005200 (1.08%)</td>
<td>121,802,144 (7.75%)</td>
</tr>
<tr>
<td>c_addressscan_filter</td>
<td>58024703 (31.40%)</td>
<td>384,651,055 (24.5%)</td>
</tr>
<tr>
<td>c_addressscan_checkscan</td>
<td>4510730 (2.44%)</td>
<td>139,784,051 (8.9%)</td>
</tr>
<tr>
<td>worker_to_manager_counter</td>
<td>27133670 (14.68%)</td>
<td>176,982,937 (11.27%)</td>
</tr>
<tr>
<td>run_scan_detection</td>
<td>24965156 (13.51%)</td>
<td>173,071,224 (11.02%)</td>
</tr>
</tbody>
</table>
Hyperloglog and state table memory

```java
if (enable_big_tables)
{
    if ( orig !in distinct_peers )
        distinct_peers[orig] = set &mergeable;
    if ( resp !in distinct_peers[orig] )
        add distinct_peers[orig][resp];
    local n = |distinct_peers[orig]|;
    local address_scan_result = check_address_scan_thresholds(ori, resp, outbound, n);
}
if (orig !in c_distinct_peers)
{
    local cp: opaque of cardinality = hll_cardinality_init(0.1, 0.99);
    c_distinct_peers[orig] = cp;
}

hll_cardinality_add(c_distinct_peers[orig], resp);
local d_val = double_to_count(hll_cardinality_estimate(c_distinct_peers[orig]));
```
hyperloglog instead of traditional sets

- Gains of about 80% reduction in memory usage using hyperloglog in tables for cardinality estimation

```c
# scan storage containers
global distinct_peers: table[addr] of set[addr]
    &read_expire = 1 days &expire_func=scan_sum &redef;

global c_distinct_peers: table[addr] of opaque of cardinality
    &default = function(n: any): opaque of cardinality { return hll_cardinality_init(0.1, 0.99); }
    &read_expire = 1 day ; # &expire_func=scan_sum &redef;
```
Detection Latency

<table>
<thead>
<tr>
<th>Detection Time</th>
<th>Avg. time between connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1461742286.588579 5.800498</td>
<td>0.002949</td>
</tr>
<tr>
<td>1461742287.861258 5.000498</td>
<td>0.002949</td>
</tr>
<tr>
<td>1461742287.139465 5.000498</td>
<td>0.002949</td>
</tr>
<tr>
<td>1461742288.132537 5.278427</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742288.985863 5.009176</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742289.018803 5.833051</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742289.472211 5.278427</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742289.708529 5.833051</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742289.818178 5.821672</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742290.334705 5.832686</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742291.482814 5.278427</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742294.405399 6.101757</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742295.183003 5.833051</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742296.217585 5.021672</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742297.893336 5.962774</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742298.895237 5.032666</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742298.896497 5.000581</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742299.848051 5.000581</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742300.143831 5.000581</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742301.634689 5.081757</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742302.649987 5.821672</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742303.885889 5.832666</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742304.885889 5.000581</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742305.154756 5.000581</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742306.538238 5.000581</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742307.639197 5.080282</td>
<td>0.007612</td>
</tr>
<tr>
<td>1461742308.632175 5.000261</td>
<td>0.007612</td>
</tr>
</tbody>
</table>
Increasing detection speed

• Problem
  – all events use conn expiration timers as in the table

<table>
<thead>
<tr>
<th>conn_expiration_timer</th>
<th>Interval</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp_SYN_timeout</td>
<td>5.0 secs</td>
<td>Check up on the result of an initial SYN after this much time.</td>
</tr>
<tr>
<td>tcp_attempt_delay</td>
<td>5.0 secs</td>
<td>Wait this long upon seeing an initial SYN before timing out the connection attempt.</td>
</tr>
<tr>
<td>tcp_close_delay</td>
<td>5.0 secs</td>
<td>Upon seeing a normal connection close, flush state after this much time.</td>
</tr>
<tr>
<td>tcp_connection_linger</td>
<td>5.0 secs</td>
<td>When checking a closed connection for further activity, consider it inactive if there (n’t been any for this long. Complain if the connection is reused before this much time ( elapsed.</td>
</tr>
</tbody>
</table>

• This basically means that all events trigger after 5.0 secs of actual activity on the wire
Solution: speed up detection

- Not changing expiration_timers: haven’t studied the effect – could be drastic

- Leverage on “insider-information” - We know our darknet/unallocated spaces

- Use new_connection event and fast-track the connections going to darknet to scan-detection module instead of waiting for other events to kick in post timer-expirations

```cpp
### speed up landmine and knock_knock for darknet space
event new_connection(c: connection) {
    # we just want to supply c to check only for darknet spaces
    # to speed up reaction time and to avoid tcp_expire_delays of 5.0 sec issue
    local tp = get_port_transport_proto(c$id$resp_p);
    if (tp == tcp && c$id$orig_h !in Site::local_nets & is_darknet(c$id$resp_h)) {
        Scan::check_scan(c, F, F);
    }
}
```
### Detection Time

<table>
<thead>
<tr>
<th>ts</th>
<th>scanner state</th>
<th>detection</th>
<th>start_ts</th>
<th>detect_ts</th>
<th>detect_latency</th>
<th>total_conn</th>
<th>total_hosts_scanned</th>
<th>duration</th>
<th>scan_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1464900975.699798</td>
<td>203.193.173.41</td>
<td>Scan::DETECT</td>
<td>KnockKnockScan</td>
<td>1464900842.468613</td>
<td>1464900975.699798</td>
<td>1464900975.699798</td>
<td>3</td>
<td>133.231185</td>
<td>44.410395</td>
</tr>
</tbody>
</table>

### Avg. time between connections

<table>
<thead>
<tr>
<th>ts</th>
<th>scanner state</th>
<th>detection</th>
<th>start_ts</th>
<th>detect_ts</th>
<th>detect_latency</th>
<th>total_conn</th>
<th>total_hosts_scanned</th>
<th>duration</th>
<th>scan_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1464900975.699798</td>
<td>203.193.173.41</td>
<td>Scan::DETECT</td>
<td>KnockKnockScan</td>
<td>1464900842.468613</td>
<td>1464900975.699798</td>
<td>1464900975.699798</td>
<td>3</td>
<td>133.231185</td>
<td>44.410395</td>
</tr>
</tbody>
</table>

### CONN.LOG

<table>
<thead>
<tr>
<th>ts</th>
<th>scanner state</th>
<th>detection</th>
<th>start_ts</th>
<th>detect_ts</th>
<th>detect_latency</th>
<th>total_conn</th>
<th>total_hosts_scanned</th>
<th>duration</th>
<th>scan_rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1464900975.699798</td>
<td>203.193.173.41</td>
<td>Scan::DETECT</td>
<td>KnockKnockScan</td>
<td>1464900842.468613</td>
<td>1464900975.699798</td>
<td>1464900975.699798</td>
<td>3</td>
<td>133.231185</td>
<td>44.410395</td>
</tr>
</tbody>
</table>

### Worker Logs

- worker-2: CUXzwp1vSPtEpzsxES
- worker-15: CCvnu2NMu6Mxe70i8
- worker-7: CcWrUM1SaCoDPquwo5
Whitelist Mgmt

- IP and Subnet based whitelist
- Clusterized
- Self-cleaning
  - when IP or subnet is added to the whitelist bro purges it from the scan tables *and*
  - removes the nullzero blocks using netcontrol/acld
- Saves restarts
  - saves problem of many IPs from a subnet being blocked and we removed only one (facebook example)
Whitelist in action

1473416025.833145  Scan::KnockKnockScan  **108.61.123.72 scanned a total of 12 hosts**: [80/tcp] (port-flux-density: 6) (origin: FR distance: 5528.29 miles) - 108.61.123.72  --
manager Notice::ACTION_DROP, Notice::ACTION_LOG  60.000000  F  -

1473419748.634896  Scan::WebCrawler  **108.61.123.72 crawler is seen**: yacybot (/global; amd64 FreeBSD 10.3-RELEASE-p7; java 1.8.0_92; GMT/en) http://yacy.net/bot.html -
108.61.123.72  worker-11  Notice::ACTION_LOG

1473419748.634896  Scan::PurgeOnWhitelist  **108.61.123.72 is removed from known_scanners after whitelist**: [scanner=108.61.123.72, status=T, detection=KnockKnockScan, detect_ts=1473416025.886353, event_peer=worker-11, expire=F] 108.61.123.72  worker-11
Notice::ACTION_LOG  3600.000000  F

- removes from known_scanners
- removes from catch-n-release hell
- removes ACLD blocks on the router, if any
HotSubnets

- Often scanners can origin from the same subnet - ie identify bad neighborhoods
- Subnet-escalation advice and capabilities
  - Scan::HotSubnet 41.67.117.0/24 (10 scanners originating)
### notice.log: Scan::HotSubnets

<table>
<thead>
<tr>
<th>S.no</th>
<th>Uniq scanners in /24</th>
<th>How many such /24</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>55634</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>4141</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td>913</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>8</td>
</tr>
</tbody>
</table>

1473750000.009613
1473750313.190990
1473750328.685138
1473750405.154414
1473750526.342934
1473750530.967447
1473750568.814431
1473750666.635887
1473750932.757024
1473751070.188362
1473751299.735644
1473751362.930388
1473751426.632496
1473751455.651451
1473751469.627464
1473751604.053208
1473751723.227480
1473751724.706633
1473751974.212308
1473752182.718434

Scan::HotSubnet 178.175.114.0/24 (3 scanners originating) 178.175.114.79 manager Notice::ACTION_LOG
Scan::HotSubnet 195.34.28.0/24 (10 scanners originating) 195.34.28.26 manager Notice::ACTION_LOG
Scan::HotSubnet 195.43.67.0/24 (10 scanners originating) 195.43.67.32 manager Notice::ACTION_LOG
Scan::HotSubnet 109.185.63.0/24 (3 scanners originating) 109.185.63.181 manager Notice::ACTION_LOG
Scan::HotSubnet 36.37.136.0/24 (3 scanners originating) 36.37.136.17 manager Notice::ACTION_LOG
Scan::HotSubnet 183.129.235.0/24 (3 scanners originating) 183.129.235.181 manager Notice::ACTION_LOG
Scan::HotSubnet 117.202.192.0/24 (3 scanners originating) 117.202.192.243 manager Notice::ACTION_LOG
Scan::HotSubnet 49.32.72.0/24 (10 scanners originating) 49.32.72.36 manager Notice::ACTION_LOG
Scan::HotSubnet 42.117.114.0/24 (3 scanners originating) 42.117.114.36 manager Notice::ACTION_LOG
Scan::HotSubnet 93.116.84.0/24 (3 scanners originating) 93.116.84.207 manager Notice::ACTION_LOG
Scan::HotSubnet 177.105.121.0/24 (10 scanners originating) 177.105.121.141 manager Notice::ACTION_LOG
Scan::HotSubnet 31.173.240.0/24 (10 scanners originating) 31.173.240.35 manager Notice::ACTION_LOG
Scan::HotSubnet 177.137.125.0/24 (10 scanners originating) 177.137.125.173 manager Notice::ACTION_LOG
Scan::HotSubnet 37.237.212.0/24 (3 scanners originating) 37.237.212.26 manager Notice::ACTION_LOG
Scan::HotSubnet 178.249.209.0/24 (3 scanners originating) 178.249.209.197 manager Notice::ACTION_LOG
Scan::HotSubnet 41.252.61.0/24 (3 scanners originating) 41.252.61.218 manager Notice::ACTION_LOG
Scan::HotSubnet 117.248.197.0/24 (3 scanners originating) 117.248.197.119 manager Notice::ACTION_LOG
Scan::HotSubnet 188.113.198.0/24 (3 scanners originating) 188.113.198.8 manager Notice::ACTION_LOG
Scan::HotSubnet 85.119.243.0/24 (10 scanners originating) 85.119.243.191 manager Notice::ACTION_LOG
Scan::HotSubnet 178.175.6.0/24 (3 scanners originating) 178.175.6.238 manager Notice::ACTION_LOG
## HotSubnet /24 with > 200 scanners

<table>
<thead>
<tr>
<th>S.no</th>
<th>ASN</th>
<th>Subnet</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>262355</td>
<td>177.125.216.0 /24</td>
<td>VESX Networks, BR</td>
</tr>
<tr>
<td>2</td>
<td>262355</td>
<td>177.125.217.0 /24</td>
<td>VESX Networks, BR</td>
</tr>
<tr>
<td>3</td>
<td>262355</td>
<td>177.125.218.0 /24</td>
<td>VESX Networks, BR</td>
</tr>
<tr>
<td>4</td>
<td>262355</td>
<td>177.125.219.0 /24</td>
<td>VESX Networks, BR</td>
</tr>
<tr>
<td>5</td>
<td>50676</td>
<td>91.236.204.0 /24</td>
<td>TELCOMNET , RU</td>
</tr>
<tr>
<td>6</td>
<td>6461</td>
<td>64.125.239.0 /24</td>
<td>ZAYO-6461 - Zayo Bandwidth Inc, US</td>
</tr>
<tr>
<td>7</td>
<td>9808</td>
<td>112.5.236.0 /24</td>
<td>CMNET-GD Guangdong Mobile Communication Co.Ltd., CN</td>
</tr>
<tr>
<td>8</td>
<td>42570</td>
<td>185.35.62.0 /24</td>
<td>KS-ASN1 This ASN is used for Internet security research. Internet-scale port scanning activities are launched from it. Don’t hesitate to contact <a href="mailto:portscan@nagra.com">portscan@nagra.com</a> would you have any question., CH</td>
</tr>
</tbody>
</table>
May 8 08:08:35  Scan::KnockKnockScan  112.74.135.36 scanned a total of 3 hosts: [21/tcp]  
(port-flux-density: 6) (origin: CN distance: 0.00 miles) on  128.3.28.64  128.3.20.30  128.3.28.110   112.74.135.36  
manager Notice::ACTION_LOG,Notice::ACTION_DROP  3600.000000     F

May 8 08:08:35  History::SF_to_Scanner  outgoing SF to scanner 112.74.135.36   112.74.135.36  
Notice::ACTION_LOG

Conn.log :

<table>
<thead>
<tr>
<th>Date</th>
<th>Source IP</th>
<th>Source Port</th>
<th>Destination IP</th>
<th>Destination Port</th>
<th>Protocol</th>
<th>Duration</th>
<th>MSS</th>
<th>Window</th>
<th>Bytes</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 8 03:49:46</td>
<td>112.74.135.36</td>
<td>61291</td>
<td>128.3.28.110</td>
<td>21</td>
<td>tcp</td>
<td>3.059543</td>
<td>0</td>
<td>0</td>
<td>S0</td>
<td></td>
</tr>
<tr>
<td>May 8 03:49:55</td>
<td>112.74.135.36</td>
<td>61291</td>
<td>128.3.28.110</td>
<td>21</td>
<td>tcp</td>
<td>S0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>May 8 03:49:46</td>
<td>128.3.28.110</td>
<td>3</td>
<td>112.74.135.36</td>
<td>10</td>
<td>icmp</td>
<td>9.073815</td>
<td>152</td>
<td>0</td>
<td>OTH</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:23</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>56755</td>
<td>tcp</td>
<td>0.789239</td>
<td>520</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:26</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>57266</td>
<td>tcp</td>
<td>0.656309</td>
<td>0</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:29</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>57735</td>
<td>tcp</td>
<td>0.672116</td>
<td>0</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:31</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>58196</td>
<td>tcp</td>
<td>0.381356</td>
<td>0</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:34</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>58595</td>
<td>tcp</td>
<td>0.722489</td>
<td>0</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:37</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>59047</td>
<td>tcp</td>
<td>0.378877</td>
<td>0</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:40</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>59431</td>
<td>tcp</td>
<td>0.543354</td>
<td>0</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:46</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>60295</td>
<td>tcp</td>
<td>0.783772</td>
<td>0</td>
<td>0</td>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>May 8 03:51:48</td>
<td>131.243.2.64</td>
<td>20</td>
<td>112.74.135.36</td>
<td>60692</td>
<td>tcp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
event connection_established(c: connection) &priority=-5
{
  local src = c$id$orig_h;
  local dst = c$id$resp_h;
  
  # ignore remote originating connections
  if (src !in Site::local_nets)
    return;
  
  if (c$resp$state == TCP_ESTABLISHED)
  {
    add_to_bloom(dst);
  }
}

function check_conn_history(ip: addr): bool
{
  local result = F;
  local seen = bloomfilter_lookup(History::tcp_outgoing_SF, ip);
  if (seen == 1)
  {
    NOTICE([Snote=History::SF_to.Scanner, $src=ip, $msg=fmt("outgoing SF to scanner %s", ip)])
  }

  global tcp_outgoing_SF : opaque of bloomfilter;
  global tcp_conn_duration_bloom : opaque of bloomfilter;

  event connection_state_remove(c: connection) &priority=-5
  {
    local src = c$id$orig_h;
    local dst = c$id$resp_h;
    
    # ignore remote originating connections
    if (src !in Site::local_nets)
      return;
    
    # only worry about TCP connections
    # we deal with udp and icmp scanners differently
    if (c$conn$proto == udp || c$conn$proto == icmp)
      return;
    
    if (c$duration > 60 secs)
    {
      bloomfilter_add(tcp_conn_duration_bloom, src);
    }
  }
Identifying Legitimate Scanners

• Web crawlers, spiders, search engine indexers
• Yes, we’d like to be top hit on google
• Automatically identify web-crawlers and not flag them as scanners
Dynamic Thresholds

- High and medium threshold ports
- port flux density - basically a popularity function of a given port - less popular == higher threshold
identify-search-engines
Tap into http_request and http_header events

```c
event http_request(c: connection, method: string, original_URI: string, unescaped_URI: string, version: string) &priority=3 {
    if (ok_robots in original_URI)
    {
        local orig=c$id$orig_h ;
        if (orig !in Scan::whitelist_ip_table)
        {
            local _msg = fmt("web-spider seeking %s", original_URI);
            NOTICE([$note=WebCrawler, $src=orig, $msg=fmt("%s", _msg)]);
            event Scan::m_w_add_ip(orig, _msg);
        }
    }
}
```

```c
event http_header(c: connection, is_orig: bool, name: string, value: string) &priority=2 {
    if ( name == "USER-AGENT" && ok_web_bots in value )
    {
        local orig=c$id$orig_h ;
        if (orig !in Scan::whitelist_ip_table)
        {
            local _msg = fmt("%s crawler is seen: %s", orig, value);
            NOTICE([$note=WebCrawler, $src=orig, $msg=fmt("%s", _msg)]);
            event Scan::m_w_add_ip(orig, _msg);
        }
    }
}
```
## Scan-Summary

- Provides summary of:
  - when scan started,
  - when it ended,
  - when was it detected
  - how many connections were made by the scanner
  - how many uniq hosts did it touch
  - latency of detection
  - total duration of the scan
- Clusterized
- Memory efficient - relies on opaque of cardinality
- Incremental scan-summary for the lifetime of the scanner

### Table:

<table>
<thead>
<tr>
<th>fields</th>
<th>ts</th>
<th>scanner state</th>
<th>detection</th>
<th>start_ts</th>
<th>end_ts</th>
<th>detect_ts</th>
<th>detect_latency</th>
<th>total_conn</th>
<th>total_hosts_scanned</th>
<th>duration</th>
<th>scan_rate</th>
<th>country_code</th>
<th>region</th>
<th>city</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>KnockKnockScan</td>
<td>1473358239.225943</td>
<td>KD</td>
<td>KnockKnockScan</td>
<td>1473358239.225943</td>
<td>1473358239.225943</td>
<td>1473358239.225943</td>
<td>1473358239.225943</td>
<td>1473358239.225943</td>
<td>1473358239.225943</td>
<td>1785.912972</td>
<td>3</td>
<td>3</td>
<td>1785.912972</td>
<td>568.637991</td>
<td>US</td>
</tr>
<tr>
<td>KnockKnockScan</td>
<td>1473358239.580965</td>
<td>KD</td>
<td>KnockKnockScan</td>
<td>1473358239.580965</td>
<td>1473358239.580965</td>
<td>1473358239.580965</td>
<td>1473358239.580965</td>
<td>1473358239.580965</td>
<td>1473358239.580965</td>
<td>1785.912972</td>
<td>23</td>
<td>23</td>
<td>4522.761665</td>
<td>196.641812</td>
<td>US</td>
</tr>
<tr>
<td>KnockKnockScan</td>
<td>1473358239.389015</td>
<td>KD</td>
<td>KnockKnockScan</td>
<td>1473358239.389015</td>
<td>1473358239.389015</td>
<td>1473358239.389015</td>
<td>1473358239.389015</td>
<td>1473358239.389015</td>
<td>1473358239.389015</td>
<td>1785.912972</td>
<td>23</td>
<td>23</td>
<td>4522.761665</td>
<td>196.641812</td>
<td>US</td>
</tr>
<tr>
<td>KnockKnockScan</td>
<td>1473358239.187065</td>
<td>KD</td>
<td>KnockKnockScan</td>
<td>1473358239.187065</td>
<td>1473358239.187065</td>
<td>1473358239.187065</td>
<td>1473358239.187065</td>
<td>1473358239.187065</td>
<td>1473358239.187065</td>
<td>1785.912972</td>
<td>23</td>
<td>23</td>
<td>4522.761665</td>
<td>196.641812</td>
<td>US</td>
</tr>
</tbody>
</table>
Scan-Summary Architecture

- Conn_table
  - 10 mins - keep building state for ALL connections
    - expire
      - check if known_scanner
        - Yes
          - Populate Scan_summary on worker
            - keep updating counts on workers
              - manager_update_scan_summary

- 20 Mins
- 60 Mins
**Blocking speed**

Detection Time

**Broker:** Netcontrol acld_add_rule

**ACLD:** Arrival timestamp

**Broker:** Sending to ACLD:

**ACLD:** Completion timestamp

**Broker:** Sending to BRO:

1473400423.893674 detected by: Quagga (Netcontrol acld_add_rule), detected a total of 3 hosts: [2323/tcp] (port-flux-density: 6) (origin: VN distance: 7665.18 miles)

**ACLD:** 800 μs

**system_time()**

bro network_time()
## drop times

<table>
<thead>
<tr>
<th>timestamp</th>
<th>Action</th>
<th>Delta ((t_n - t_{n-1}))</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>1473663871.195220</td>
<td>Scan::KnockKnockScan</td>
<td>(t=0s)</td>
<td>notice.log</td>
</tr>
<tr>
<td>1473663871.195220</td>
<td>NetControl::REQUESTED</td>
<td>(t=0s)</td>
<td>netcontrol.log</td>
</tr>
<tr>
<td>1473663871.226191</td>
<td>Brokerlisten: Got event</td>
<td>30.9 ms</td>
<td>broker.log</td>
</tr>
<tr>
<td>1473663871.226378</td>
<td>brokerlisten:INFO:Sending to ACLD</td>
<td>187 (\mu s)</td>
<td>broker.log</td>
</tr>
<tr>
<td>1473663871.226359</td>
<td>ACLD Arrival timestamp</td>
<td>-0.19 (\mu s)</td>
<td>acld.log</td>
</tr>
<tr>
<td>1473663871.226420</td>
<td>ACLD Complete Timestamp</td>
<td>61 (\mu s)</td>
<td>acld.log</td>
</tr>
<tr>
<td>1473663871.227030</td>
<td>brokerlisten:INFO:Received from ACLD</td>
<td>610 (\mu s)</td>
<td>broker.log</td>
</tr>
</tbody>
</table>
Usability

- Plug-n-play
- Works with netcontrol-framework
- All configuration knobs moved to one single file
- Accompanying whitelist allows for addressing false-positives in real-time
- No need to restart Bro
- Dynamic thresholds and post-detection vetting reduces false positives significantly
- GeoIP inclusion in blocking threshold heuristics
## files and description

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>check-scan.bro</td>
<td>first file which taps into events and calls function check-scan</td>
</tr>
<tr>
<td>check-scan-impl</td>
<td>functions which enables clusterization</td>
</tr>
<tr>
<td>scan-config</td>
<td>ALL user configuration settings are redef variables centrally located here. No need to go into any other policy to tweak</td>
</tr>
<tr>
<td>scan-base</td>
<td>important core functions – I can actually move a bunch of functions from check-scan and check-scan-impl here but will wait</td>
</tr>
<tr>
<td>scan-summary</td>
<td>add-on code which generates scan_summary.log (pretty good log actually)</td>
</tr>
<tr>
<td>check-*</td>
<td>heuristics for knockknock, landmine, addressScan, BackScatter etc. All files are basically – 2 or 3 functions – validate_* , check-* , check-thresholds (this name varies)</td>
</tr>
<tr>
<td>scan-*</td>
<td>additional supporting scripts for input data, whitelists, host-profiling data, subnet-info for landmine etc etc</td>
</tr>
</tbody>
</table>
Reliability

• What if subnets file is Empty or incomplete
  – accuracy of functions like - is_darknet or
    is.Scanner or validation_func for heuristics
• Typos in the whitelist entry
  – catch reporter_error for all input files
• co-ordination with netcontrol
  – Any Bro shall not unblock what it did not block
• Memory and CPU on Manager
Pass Fail Criteria

• Not miss anything existing infrastructure flags
  – More accurate than existing policy
• Find more badness
• Speed
• Practical False +ve rate
• Pass peer review
• Bro runs stable for > 1 month
• Key to success is to be able to count failures correctly
• We should know what they know
<table>
<thead>
<tr>
<th>Users</th>
<th>Developers</th>
<th>Bro People</th>
</tr>
</thead>
<tbody>
<tr>
<td>notice.log and scan-summary.log</td>
<td>access to known_scanners table</td>
<td>how to make table persistent</td>
</tr>
<tr>
<td>memory efficient</td>
<td>use of hyperloglog and bloom-filters</td>
<td>hard to find data-structure sizes/usage</td>
</tr>
<tr>
<td>whitelist capability/ Dynamic darknets and configs on fly</td>
<td>input-framework + tap into reporter_error event</td>
<td>dealing failures in input-files due to lame typos</td>
</tr>
<tr>
<td>stable code</td>
<td>extendable and modular</td>
<td>Manager CPU is mystery</td>
</tr>
<tr>
<td>plug &amp; play</td>
<td>clusterization insights</td>
<td>ability to account of w2m and m2w events</td>
</tr>
<tr>
<td>speed &amp; accuracy</td>
<td>you can fix scan.bro</td>
<td>Should scan-detection be in C++ land instead of policy land ?</td>
</tr>
</tbody>
</table>
Must and Should Requirement checklist -
or feature list of scan detection

- Accuracy ✔
- Reaction
  - Must block really fast scanners ✔
  - Must block really slow scanners ✔
- Smart ACL mgmt - keep scanners blocked only until active - no unnecessary acl consumption ✔
- State management ✔
- Block sooner if they come back (catch-n-release) ✗
- Very very long state management (bloom filters) ✔
- Variety - Should be able to Block based on different events (AddressScan, PortScan, deep block, vuln-signature - ntp monlist or data feeds such as tor) ✔
- Should be able to Handle redundancy in infrastructure - ie avoid race conditions in blocking and unblocking independently ✔
  - Atomicity in blocking and unblocking ✔
  - Accountability in blocking and unblocking ✔
- Whitelisting mechanism ✔
- Outsmart attackers over attackers so that they cannot easily guess/defeat block thresholds (Dynamic thresholds) ✔
Must and Should Requirement checklist - aka feature list of scan detection

- Ability to add new heuristics very quickly
- Identify and Remove false positives quickly and suppress them in future
  - .gov, US .edu or foreign .edu etc
- Optimize ACLs, don’t block what's already blocked somewhere else
  - eg. icmp timestamp query is blocked on border router so no need to block those offending IP’s any more or port 135, 137, 445 scanners
- Do not block what’s blocked by the border router
- Watchdog processes to account for functionality
  - alert if too many failures on blocking
  - alert if too many success on blocking
  - alert if rate of blocking changes etc etc
- Verification capabilities
  - are blocking working as expected. Router ACLs are functional - violations of policies should alert (hey I am seeing SF on 445 )
- Prioritize a list of ports/IPs/nets to be aggressively blocked
- Careful and slow in blocking a certain set
- Mechanisms to handle established connection scanners/bruteforcers (RDP, SSH)
what does it not do

- Smart Defenses against spoofing udp
- Persistence - restarts should not matter
- Dynamic responses based on situation - eg. Change from acld to nullzero on thresholds
- Expire blocks based on priorities (icmp sooner than ssh for example)
- If possible figure out intentions why this scan specifically
- Who responded and why and what they sent?
- Highlight new trends
- Big limitation - this is on TCP
Availability

https://github.com/initconf/scan-NG/

Or

use bro-pkg install initconf/scan-NG

Alternatively, try Justin Azoff’s unified-scan policy which is significant improvement over stock misc/scan.bro
Questions and comments

security@lbl.gov
asharma@lbl.gov