General Platform Hardening Implementation Guidelines

RECOMMENDATIONS FOR ALL DNS OPERATORS

Network Security

1. **Practice 1:** ACLs **MUST** be implemented to restrict network traffic to your DNS servers.

   Note: The examples below are specified using the PF packet filter. We will be providing examples using other vendors/implementations in the Additional Information page.

   a. For authoritative operators, the ACLs **MUST** allow only DNS traffic and associated ICMP response codes to your authoritative DNS servers; access to all other services and ports on your network for DNS servers **MUST** be denied.

      Implementation:

      (Note: We only concern ourselves with which traffic should be allowed to publicly reachable authoritative DNS servers. If your filtering policy is a "default allow," you'll need to insert the correct rules to block other traffic flows. Also, administrative flows such as remote management, access to package sites, auxiliary services such as NTP, etc. are not covered.)

      Example (don't apply as is!)

```
# Define table with servers to be accessed
table <dns_servers> { 192.168.64.11, 192.0.2.1, \
          2001:DB8::192:0:2:1, 10.20.30.2, 10.10.0.8
}

# Allow DNS traffic over UDP and TCP to 53, 443 and
853
pass quick proto { tcp, udp } from any to \
     <dns_servers>  port { 53, 443, 853 } keep state
# Allow outgoing queries from the servers
pass quick proto { tcp, udp } from <dns_servers> to
any port { 53, 443, 853 } keep state

# Allow anyone to ping servers
pass quick inet proto icmp from any to <dns_servers>
\
     icmp-type echoreq keep state
```

```
pass quick inet6 proto icmp6 from any to
<dns_servers> \
     icmp6-type echoreq keep state

# Rules here would cover traffic initiated by the
servers,
# and administrative access to the servers (SSH, ...)
# ...

# Block all other traffic
block in log quick from any to <dns_servers>
```

b. For all DNS operator types (authoritative and recursive), incoming traffic from all so-called Bogon IP subnets **MUST** be blocked, including private RFC 1918 addresses, and possibly RFC 6598 (shared IPv4 address space) for v4. Recursive DNS operators should of course **NOT** block private/shared IP address space deployed within the organization. See https://ipgeolocation.io/resources/bogon.html, https://team-cymru.com/community-services/bogon-reference/bogon-reference-http/ and https://team-cymru.com/community-services/bogon-reference/bogon-bit-notation/

   Implementation:

   Example (don't apply as is!)

```
ext_if = eth0
table <bogons_v4> { 0.0.0.0/8, 10.0.0.0/8, \
          100.64.0.0/10, 127.0.0.0/8, \
          169.254.0.0/16, 172.16.0.0/12, \
          192.0.0.0/24, 192.0.2.0/24, \
          192.168.0.0/16, 198.18.0.0/15, \
          198.51.100.0/24, 203.0.113.0/24, \
          224.0.0.0/3 }
table <bogons_v6> {
          ::/128, ::1/128, \
          ::ffff:0:0/96, \
          100::/64, 2001:10::/28, \
          2001:db8::/32, fc00::/7, \
          fe80::/10, fec0::/10, \
          ff00::/8 \
}

block in quick on $ext_if from <bogons_v4> to any
block in quick on $ext_if from <bogons_v6> to any
```

```
            # Insert here rules restricting access to
client nets
# only if running a recursive, then rules to limit
access
            # to DNS service ports and ICMP (see
previous section)
```

Note that bogon lists may also include prefixes that have been allocated to RIRs but not yet assigned.

2. **Practice 2:** BCP38 egress filtering **MUST** be implemented so that no network traffic can leave your network with a source IP address that is not assigned to you or your customers.

   Note that this is not limited to your DNS servers - BCP38 type egress filtering should be applied at the edge/border of your network anyway.

   Implementation:

   This is a simplified example where rules are implemented on an edge device with two interfaces, one connected to an upstream provider (“outside”, eth0) and one facing the customers and local services (“inside”, eth1). Note that we only apply the rules on the inside interface - not all networks are this simple, and there are cases where traffic will not be symmetric. Also note the use of the “log” directive in the block rule, so that you can capture and analyze attempts to send traffic with an invalid source address from your network.

   Example (don’t apply as is!):

```
# Define table with client networks and local infra
table <client_local_nets> { 192.168.1.0/24,
2001:DB8:1:1::/48,
    192.0.2.0/24, 2001:DB8:1:192::/48, \
    2001:DB8:1:2::/48, 172.31.254.0/24 }

# Int
int_if=vtnet0

pass quick from 192.168.64.0/24 to any
pass quick from any to 192.168.64.0/24

block quick log on $int_if from !<client_local_nets> to
any
```

If there are many networks, or if they need to be stored in separate tables, it may be preferable to structure the rules differently–for instance, tagging all traffic entering the internal interface, and then applying policies as the traffic exits from the external interface(s). This is of course implementation-specific - what matters is that the sending of spoofed traffic, deliberate or accidental, towards the Internet and other customers shouldn't be permitted.

Host and Service Security

3. **Practice 3:** The configuration of each DNS server **MUST** be locked down.

    a. All services and software packages that are not required for offering DNS service on the system **MUST** be uninstalled or disabled.

    Implementation:

    On a Linux host running Ubuntu or Debian (for example), we would use the following command to list installed packages:

    ```
    # dpkg –list
    ```

    Since even the most basic installation of an operating system distribution has hundreds of packages, it is recommended to issue this command immediately after installing the OS. During installation, select only the most minimal set of packages (typically only SSH server, and no additional features).

    Another solution is to use container-based deployments, where *only* required software is installed.

    Some packages are essential and shouldn't be removed. If you're not sure whether a package is essential or not, you can issue:

    ```
    dpkg-query -Wf '${Package;-40}${Essential}\n' | grep
    yes
    ```

    and

    ```
    dpkg-query -Wf '${Package;-40}${Priority}\n' | grep
    -E "required"
    ```

    You can always see the impact of removing a package (once it's been identified as non-essential) with the following command:

    ```
    apt-get remove -s package_name
    ```

"-s" = simulate the operation, and see which other package dependencies would be affected, but do not actually remove the package.

Also, verify that there are no services open and listening on TCP and UDP, besides on the loopback interface (identified by 127.0.0.1 or ::1)

```
# netstat -tulpn |grep LISTEN
```

b. Hosts running DNS services **MUST** only run DNS software. In other words, DNS servers **MUST NOT** run other services, such as web or email servers.

Implementation:

While the previous point focused on identifying services that may have been enabled inadvertently or are no longer required, here we emphasize the importance of NOT running other services on the same server (whether it is physical, a virtual machine, or a container), even for convenience/resource conservation. Obviously, remote administration portals/consoles that make use of an HTTPS service to manage the host are excluded from this requirement. Likewise, the use of an SMTP service listening locally for mails to be sent *from* the server is also excluded.

Basically, the number of services exposed to the outside world that may potentially have security flaws in them should be reduced to the strict minimum, or disabled and uninstalled entirely.

c. All relevant logging channels and levels for the DNS subsystem **MUST** be enabled. Logs **MUST** be sent to a central location for archiving, inspection, and auditing, and they **MUST** be retained for a reasonable time in accordance with retention policies.

Implementation:

Each DNS implementation has its own set of controls to set what should be logged, and to which channel or file. In BIND, you can define *channels* and *categories.* Channels are either syslog facilities/levels or files. Categories correspond to different parts of the BIND internals/subsystems, for instance: queries, dnssec, and transfer.

In PowerDNS, logging is sent to the DAEMON syslog facility. This can be reconfigured to a dedicated facility (local0 to local7).

In all cases, it is critical that logs be duplicated to a remote location so that they may be analyzed/post-processed in the event of a system failure or security compromise where local logs may be lost or destroyed.

Depending on the syslog implementation, remote logging can be enabled in different ways. When using *rsyslog*, the syntax for sending messages in a given facility is:

```
facility.*      @hostname.of.log.server
```

(If the hostname is prefixed with @@, then logging will be done over TCP instead of the usual port 514/UDP.)

In syslog-ng, one method to configure remote logging from your clients could be:

```
destination remote { network("192.0.2.10" \
transport("udp") port (514)); };
```

Note that in modern distributions, logs might be set up to go to systemd's journald daemon. In this case, you can still decide to send logs to syslog by editing `/etc/systemd/journald.conf` and setting:

```
ForwardToSyslog=yes
```

and then restarting `journald` with:

```
systemctl restart systemd-journald
```

You will also need to enable reception of syslog on a centralized log server, and set up rules/filters to store the incoming logs in distinct directories (per host/date, for example).

4. **Practice 4:** User permissions and application access to system resources **MUST** be limited. File permissions and ownership restrictions **MUST** be set so that users and services not directly associated with management of the DNS subsystem do not have read or write access to DNS service configuration, data files, and database subsystems.

   Implementation:

   There is no specific implementation guideline because the details will be OS and software vendor dependent; however, you should inspect and correct any discrepancies in the file ownership/permissions, and verify that any privileges (setuid/setgid) that the DNS subsystem and processes may have are justified/required. Most packages in modern Linux distributions will ship with an AppArmor profile, which controls which

resources (files, sockets, …) a given running process may access, and what it may do to them (open, read, write, etc.).

i. Consider using AppArmor or another capabilities-based security mechanism to restrict which files and resources the DNS subsystem is allowed to access on the host OS.

ii. Consider placing the DNS service and associated support services in a containerized environment. In case of a service compromise (remote code/shell execution), the impact is limited to the container running the DNS service.

5. **Practice 5:** System and service configuration files **MUST** be versioned. For authoritative operators, zone files/data **MUST** also be versioned.

   Implementation:

   By using a revision/version control system (VCS) such as Git, changes to system configuration files and zone files/data can be quickly reviewed. This makes it easier to identify the point in time when corruption or unauthorized changes took place, and take corrective action.

   For DNS zones that are stored as text files, a tool such as gitwatch (https://github.com/gitwatch/gitwatch) will watch one or more directories and automatically commit files that have been updated/added to a Git repository. These updates can be periodically pushed/pulled to a remote repository for safekeeping.

   For configuration/system files that do not change often, consider using a tool like AIDE (https://aide.github.io/), which monitors changes to files, unauthorized or otherwise.

   If you use a filesystem that supports snapshots, it may offer the ability to “time travel” for easy inspection across multiple snapshots, and even compare files between them (for instance, ZFS).

6. **Practice 6:** Access to management services (e.g., SSH, web-based configuration tools) **MUST** be restricted. All services not needed for DNS or management **MUST** be disabled or uninstalled if possible, otherwise network access to the unnecessary services **MUST** be blocked.

   Implementation:

   See Practice 1 regarding ACLs for DNS. Rules must be implemented so that services such as SSH and any other remote management tools are only accessible from management subnets.

Example (don't apply as is!):

```
table <mgmt_net> { 192.168.64.0/24, 192.0.2.0/24 }
table <dns_servers> { 10.254.0.1/24, 2001:DB8::/64 }

pass in quick on vtnet0 proto tcp from <mgmt_net> to
<dns_servers> port 22 keep state

# Insert rules to allow DNS traffic (see Practice 1),
block other
# services

block log quick proto tcp from any to <dns_servers> port
22
```

7. **Practice 7:** Access to the system console **MUST** be secured using cryptographic keys, protected with a passphrase (e.g. SSH keys) or using suitable two-factor authentication (OTP generator or token-based).

   Implementation

   At the very least, password-enabled login to sensitive accounts (which can escalate privileges,) should be disabled, and preferably for all accounts). SSH or other remote administration mechanisms using strong encryption should always be used (HTTPS-based, for instance, if using web-based tools).

   To disable password authentication for all users, here's how to proceed on a Linux system, for instance. Edit the file `/etc/ssh/sshd_config`:

   ```
   # To disable tunneled clear text passwords, change to no
   here!
   #PasswordAuthentication yes
   ```

   Change the above line to:

   ```
   PasswordAuthentication no
   ```

   And restart sshd:

   ```
   systemctl restart ssh
   ```

   Make sure all users on the system have SSH public keys installed!

   It's also possible to selectively disable password authentication on a per-user basis, as described here.

Implementing two-factor authentication is somewhat outside the scope of these guidelines, but as a starting point, it is actually possible to tell SSH to require both a password AND the presence of a public key by setting this in the sshd_config file:

```
AuthenticationMethods "publickey,password"
```

Another solution that doesn't require hosting a dedicated authentication server or issuing tokens to users using Google Authenticator: https://ubuntu.com/tutorials/configure-ssh-2fa

Customer-Facing Portal and Service Security

8. **Practice 8:** Credentials for customer access (registrants and other domain contacts) **MUST** follow sound credential management practices, including offering two-factor authentication as an option.

   Implementation:

   There are no explicit configuration guidelines here, because each vendor will have their own specific solution.

   At the very least:

   - Customers/registrants MUST be given the option to enable multi-factor authentication (MFA), with a choice of a recovery mechanism should the device or mechanism used for 2FA/MFA be lost or compromised (pre-generated list of one-time passwords is one possibility).

   - Customers should be discouraged from picking passwords that are too short/simplistic, and a throttling mechanism should prevent brute-force attacks on registrant UI login.