# Certificate-based Monitoring with Icinga

BY BENEDICT REUSCHLING

Icinga is the successor to the popular Nagios host and service-monitoring software. With the aim of being a drop-in replacement with backwards compatibility to Nagios, Icinga also provides a couple of new features. This article describes how to set up a central Icinga host to monitor endpoint systems using certificates in a top-down configuration sync fashion.

Top-down describes the way that the checks (disk full, load-avg too high, etc.) execute on the remote machines. In top-down monitoring, the central Icinga host called parent is responsible for synchronizing the configuration files to the monitored nodes called child nodes. No manual restart after a configuration change is required on these child nodes, as syncing, validation, and restarts happen automatically. Checks execute directly on the child node scheduler in regular intervals. Hosts are organized in a global zone and each host (parent and children) is defined as an endpoint in it. Each child will have to verify that it is part of the monitoring zone by creating a certificate signing request in a ticket which the parent will then verify and sign. This creates trust that the machines communicate in an encrypted way and that the monitoring data received on the parent is not manipulated in transit.

**This article describes how to set up a central Icinga host to monitor endpoint systems using certificates.**

The setup described here uses FreeBSD as parent to monitor another FreeBSD host as child. Of course, other operating systems such as Linux or Windows are possible in the same way, but not covered as part of this already long description. Although the setup is described to run on the host itself to reduce complexity, the author has it happily running in a jail—so far without issues.

## Preparation

We assume that all our hosts are installed, can find each other on the same network, and have a basic SSH connection going on between them. Our central monitoring server will be called `monitor.example.org` and the client we're going to set up later is called `client.example.org` (you see my creative naming choices right there). The prompts used in front of the commands will indicate on which host this command is to be executed. Let's start with preparing our central monitoring host (the parent).

Ensure that clocks on the machines are synchronized. This is important for proper generation of certificates later. This is typically done using `monitor# ntpdate -b pool.ntp.org`.

On this central FreeBSD host, we want to use the latest version of Icinga and other pack-

ages, instead of the quarterly ones. Edit **/etc/pkg/FreeBSD.conf** and change the word quarterly to latest in the line starting with url:. Save and exit afterwards. To update the package repository with the newer packages, run:

```
monitor# pkg update
```

## PostgreSQL Setup

Before installing the required software packages (including PostgreSQL as the backend database and nginx for the webserver to host the Icingaweb2 monitoring interface), we're creating a ZFS dataset for the postgres database first to get populated when the packages extract. If you don't run ZFS, this also works fine with regular directories.

The following commands create a new dataset on our example pool called **mypool** a **/var/db/postgres/data**. Non-existent datasets on that path are also created using the **-p** parameter. Next, access time is deactivated as we don't need it here and it saves some I/O by not updating timestamps of files at every write. With newer ZFS 2.0, we also use the zstd compression on the dataset. As Postgres writes data in chunks of 8k, we set the ZFS **recordsize** to match for best performance. With **logbias** set to throughput, we instruct ZFS to optimize synchronous writes from the database for efficient resource use. The mountpoint is set to overlap the existing **/var/db/postgres** path. When the package gets installed, it is put on that dataset instead of the regular **/var/db** directory. Note: we don't focus on further tuning of the PostgreSQL database here. Go to https://pgtune.leopard.in.ua/ and enter the values of your PostgreSQL host to get configuration recommendations to put into the **postgresql.conf** file.

```
monitor# zfs create -p mypool/var/db/postgres/data
monitor# zfs set atime=off mypool/var/db
monitor# zfs set compression=zstd mypool/var/db
monitor# zfs set recordsize=8k mypool/var/db/postgres
monitor# zfs set logbias=throughput mypool/var/db/postgres
monitor# zfs set mountpoint=/var/db/postgres mypool/var/db/postgres
```

Now it is time to install the required packages. Easily done using pkg install, including automatic dependency resolution:

```
monitor# pkg install icinga2 icingaweb2-php74 postgresql13-server nginx \
ImageMagick7-nox11 php74-pecl-imagick-im7
```

Note: at the time of this writing, these package versions were the latest available. Check to make sure that is still the case by going to www.freshports.org to see if there are newer package versions listed. PHP, in particular, may have gotten a version bump in the meantime.

Some of these services need entries in **/etc/rc.conf** to start when the system boots. These include the following:

```
monitor# sysrc sshd_enable=yes
monitor# sysrc icinga2_enable=yes
monitor# sysrc postgresql_enable=yes
```

Note that the services have not started yet, as some configuration is required before that can happen. Along with the postgres package came the system user and group of the same name, which is why setting permissions on **/var/db/postgres** is possible now.

```
monitor# chown -R postgres:postgres /var/db/postgres
```

The PostgreSQL database cluster is initialized next by running **initdb** as the postgres user with UTF-8 as encoding. Note that the postgres user needs to execute these commands, although there is now a way to do this via the service command (sometimes I'm old fashioned).

```
monitor# su postgres
postgres@monitor$ initdb -D /var/db/postgres/data -E UTF8"
```

After a successful initialization of the database cluster, the server is started using **pg_ctl**:

```
postgres@monitor$ pg_ctl start -D /var/db/postgres/data
```

Next up is the creation of the Icinga role and database, which will later load some initial tables and sequences that form the monitoring backend.

```
postgres@monitor$ createuser -drs icinga
postgres@monitor$ createdb -O icinga -E UTF8 icinga
```

Entries in **pg_hba.conf** (in the data directory) like the following allow the just created Icinga user access to the database via localhost (no need to expose it to the network for Icinga to work properly):

```
local icinga icinga md5
host icinga icinga 127.0.0.1/32 md5
```

Load database schema definition for Icingaweb2 as well as those for the IDO (Icinga data objects) into the database now:

```
postgres@monitor$ psql -U icinga \
  -d icinga < /usr/local/share/icinga2-ido-pgsql/schema/pgsql.sql
```

Log out of the postgres user and continue the rest of the setup. On the Icinga side, features control the functionality of the monitoring system. This includes which database backend is used. To enable PostgreSQL as the backend for the IDO, run the following command:

```
monitor# icinga2 feature enable ido-pgsql
```

In some cases, not all files and directories are owned by the Icinga system user. Running **chown** over the main **icinga2** directory ensures the permissions are properly set.

```
monitor# chown -R icinga:icinga /usr/local/etc/icinga2
```

This concludes the database part of the setup. We'll continue with the webserver setup. Although nginx is used here, other webservers like Apache2 are also perfectly fine to use. The Icinga documentation shows the necessary steps for that, too.

## Nginx Setup

Icinga's web interface (aptly named Icingaweb2 since it is version 2) is a PHP application to manage hosts and services from the comfort of your browser. Events concerning any failed checks are also shown there and you can acknowledge problems or define downtimes at a central location. To configure the PHP fastCGI process manager (**php-fpm**) to handle requests coming from the webserver, enable the following options located in **/usr/local/etc/php-fpm.d/www.conf**:

```
monitor# cd /usr/local/etc/php-fpm.d
monitor# sed -i "" 's/^;listen = 127.0.0.1:9000/listen = /var/run/php5-fpm.sock/’
www.conf
monitor# sed -i "" 's/^;listen.owner/listen.owner/' www.conf
monitor# sed -i "" 's/^;listen.group/listen.group/' www.conf
monitor# sed -i "" 's/^;listen.mode/listen.mode/' www.conf
```

This basically uncomments lines that are in the file already to activate them and replaces the listen directive to use the local php5 socket instead of opening a port on the host for it. The major webserver configuration is done in the nginx.conf file located in **/usr/local/etc/nginx**, where we reference the fastCGI socket, among other things. The following configuration block is inserted after the commented **#access_log** line.

```
location ~ ^/icingaweb2/index\.php(.*)$ {
        fastcgi_pass unix:/var/run/php5-fpm.sock;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME /usr/local/www/icingaweb2/public/index.php;
        fastcgi_param ICINGAWEB_CONFIGDIR /usr/local/etc/icingaweb2;
        fastcgi_param REMOTE_USER $remote_user;
}

location ~ ^/icingaweb2(.+)? {
        alias /usr/local/www/icingaweb2/public;
        index index.php;
        try_files $1 $uri $uri/ /icingaweb2/index.php$is_args$args;
}
```

Note that this does not contain an SSL section to keep this tutorial simple. It is definitely recommended (and pretty much standard practice nowadays) to generate a certificate for the webserver and configure port 443 for the secure channel configuration. The web is full of tutorials for it and services like *Let's Encrypt* make the process easy and convenient.

When the packages were installed, a file containing PHP settings meant for production use was created in **/usr/local/etc/php.ini-production**. These settings are fine for our purposes, and we activate them by copying them to be our new **php.ini** file:

```
monitor# cp /usr/local/etc/php.ini-production /usr/local/etc/php.ini
```

With this file as a base PHP configuration, we only have to replace the timezone information in it. I use this sed one-liner to place my installation within central Europe. Use whatever fits your location best:

```
monitor# cd /usr/local/etc
monitor# sed -i "" s,;date.timezone =,date.timezone = Europe/Berlin, php.ini
```

We replaced the regular sed divider / with a comma here to not confuse it with the separator between region and city. See, I'll smuggle some sed tricks into my tutorials for you as well. Thank me later... Oh by the way, this is all that is needed for the basic serving of icingaweb2 to end-users. We will now focus our attention on the Icinga configuration.

## Icinga Monitoring Setup

Icinga uses various ways to monitor systems and is quite flexible about different monitoring environments and needs. For example, a host may not be available all the time (roaming user) or not have a direct connection to the central monitoring host. In the latter cases, satellite systems can relay monitoring data and check results from a different network or subnet to the central instance. In the setup that we use here, the central instance (called master) controls the execution of checks on the monitored systems. Trust is established by certificates exchanged between the master and clients. A zone defines either a geographic location (Europe, Africa, etc.) where monitoring happens or some other kind of logical grouping that makes sense in our monitoring context. For example, a whole plant, office, server room, rack, etc. could each form their own zone. Surely, a DNS zone is also possible, whatever is most useful to monitor as a whole or based on some common criteria.

First, we generate the master certificate, which we will use to monitor the central instance itself and as a base of trust for adding other monitored clients further down this tutorial. The setup is typically interactive, but here we pass all the necessary parameters on the command line:

```
monitor# icinga2 node setup --master --zone "my-zone" \
  --cn monitor.example.com \
  --listen monitor.example.com,5665 --disable-confd"
```

Here, the certificate for our central host **monitor.example.com** is generated and we instruct Icinga to not populate the **conf.d** subdirectory in **/usr/local/etc/icinga2**. We are going to create those files ourselves anyway.

Both the zone and cn are up to you to name based on your local requirements. Port 5665 should be open on your firewall to allow contacting the clients and sending the check result back. Next, we change into the directory **/usr/local/etc/icinga2** and create a couple of files and directories:

```
monitor# cd /usr/local/etc/icinga2
```

```
monitor# mkdir conf.d
```

We define an API user that has the permissions to generate a ticket through which the monitored clients request becoming part of the monitoring zone. The master will then allow or deny the ticket request and sign the client certificate with its own to establish a secure, trusted connection between the two.

The **api-users.file** contains the following:

```
object ApiUser "client-pki-ticket" {
        password = "randomstringthatmustbechanged"
        permissions = [ "actions/generate-ticket" ]
}
```

Definitely change the password line to a random string consisting of random numbers and characters, the longer the better. Next, a zones subdirectory in the **/usr/local/etc/icinga2** directory is created that holds all the information to distribute to all members of this zone. Typical examples are check information and monitoring intervals that the clients will receive from the central monitoring instance. That way, the clients need no extra local configuration, and the system administrator needs only to change the central zone configuration, which will then propagate securely to all the hosts. Since our zone is called my-zone (creativity is clearly my thing), we create a subdirectory that holds only the relevant information for clients in that zone. Other zones can be completely different, yet the monitoring configuration is located at a central place instead of each host that makes up the zone.

```
monitor# mkdir -p /usr/local/etc/icinga2/zones.d/my-zone
```

Our zones contain various information: the hosts to monitor, what to monitor (i.e., which checks to execute on each host), the monitoring intervals (how often), etc. We're starting with defining the central monitoring master in a file called **/usr/local/etc/icinga2/zones.d/my-zone/hosts.conf**. For our master, it looks like this:

```
object Host "monitor.example.com" {
      import "generic-host" // import generic settings for all hosts
      address = "monitor.example.org"
      vars.os = "FreeBSD"

      //follow convention that host name == endpoint name
       vars.agent_endpoint = name
}
```

Each host is defined as a host object and an address where it is reachable on the network. We also define a variable by which we can filter out specific hosts in Icingaweb2 for grouping purposes or define checks only for certain hosts matching these criteria. This is shown later.

The import **"generic-host"**-line is where we reference a template. Templates help us apply common settings to all hosts without having to redefine them for each host added to the file. For example, each host should have the same check interval (how often it is

checked) and other similar settings. It makes this file smaller by avoiding repetitions and different hosts may use other template settings or override them with their own that are only valid for this special system.

The **templates.conf** file is located within the **zones.d/my-zone** directory and looks like this:

```
template Host "generic-host" {
      max_check_attempts = 5
      check_interval = 2m
      retry_interval = 30s
      enable_flapping = true
      check_command = "hostalive" //check is executed on the master
}

template Service "generic-service" {
      max_check_attempts = 5 // re-check 5 times before HARD state
      check_interval = 2m
      retry_interval = 1m
      enable_flapping = true
}
```

Two templates are defined here for generic hosts and services. In total, hosts and services are checked five times before an alert is generated. This is to avoid occasional packet loss or slow reacting equipment or processes, but that are generally working. The **check_interval** defines how many times the checks are executed, while the **retry_interval** defines when to check again after one check did not return in an OK-state. Definitely play around with these intervals to fit your monitoring needs. Remember that the more often you monitor, the more traffic is generated, and the data returned by the checks needs to be stored in the database, gradually making it bigger the longer you monitor.

A flapping state can happen when a host or service seems to be available, then next time it is unavailable, then available again and so on (changing rapidly between states, without seeming to become stable). Icinga is capable of detecting those states by comparing the last known state with the current one over a period of time. These flapping states are not enabled by default but are valuable information for someone debugging a problem that only happens during certain load times or busy activity. An erratic host behaving that way shows up in Icinga and should be investigated further for the root cause. The problem may also originate in the network itself, so rule out any other influences that might be responsible. A **check_command** defines which check from the ones Icinga provides needs to run by default. The **hostalive** command is basically a ping in disguise, checking to see if a host is reachable. The reason that this is only defined in the generic-host template is because services usually define a different **check_command** that fits the service and can't be easily generalized with a template.

Now that we have templates for common functionality in place, it is time to define which checks our monitoring should run and on which hosts. They are defined in the **zones.d/my-zone/services.conf** file. Here is my definition to check the disk space:

```
apply Service "disk" {
      import "generic-service"
      check_command = "disk"

// Specify remote agent as command execution endpoint, fetch host custom variable
      command_endpoint = host.vars.agent_endpoint

// Only assign where a host is marked as agent endpoint
      assign where host.vars.agent_endpoint
}
```

Applying a service means assigning it to a particular target, either a host or the result of an expression. In this example, we define that all hosts that are defined as endpoints should have the disk check running. The execution of the check is happening on the host itself, called an active check. A passive check would be running on the central monitoring instance, trying to reach the remote system, run the check, and fetch the result. Both active and passive checks can be defined for a host or target. Both have their pros and cons, but in a simple monitoring setup such as this, it is a good start to use the services that come with Icinga.

Icinga provides these common services as checks: disk, load, users, swap, procs, ping, and ssh. These provide a good initial basis for monitoring to see if swap space is low, the disk is filling up, the load is extremely high, or that there are suddenly 200 users logged in (which may or may not be normal).

A special check is to test whether our remote endpoint system is still available within the defined zone. For that, we can extend the **`services.conf`** file we just created to contain the following agent health check:

```
apply Service "agent-health" {
      check_command = "cluster-zone"

      display_name = "cluster-health-" + host.name

// Follow convention: agent zone name is FQDN same as host object name.
      vars.cluster_zone = host.name

      assign where host.vars.agent_endpoint
}
```

In addition to running the cluster-zone check command (which we don't have to know too much about to use it), we also see how a different check description is displayed in the Icingaweb2 interface by defining **`display_name`**. With this, we can see at a glance the name of the monitored system, prefixed by the string "**`cluster-health`**".

The internal Icinga database (IDO) may also fail, so it is good to monitor it as well (remember to watch the watchers). Even though newer Icinga versions are doing away with the IDO altogether, replacing it with a database on its own, small installations are perfectly fine to still use the IDO. The checks for our IDO based on PostgreSQL are defined like this (again in **`services.conf`**):

```
object Service "ido" {
        check_command = "ido"
        vars.ido_type = "IdoPgsqlConnection"
        vars.ido_name = "ido-pgsql"
        host_name = NodeName
}
```

We don't even need to apply this to any host, as this check only runs where the Icinga IDO database is installed (the central monitoring instance). The assignment **`host_name = NodeName`** takes care of that, since **`NodeName`** is defined as the name of the host by default doing the checks and collecting the results. The plugin periodically checks the IDO database and emits (upon successful execution) information about the IDO:

```
Connected to the database server (Schema version: '1.14.3'). Queries per second:
4.633 Pending queries: 21.000. Last failover: 2022-03-23 16:05:05 +0100.
```

Moving on to a different file **`zones.d/my-zone/dependencies.conf`** is where we define (you guessed it) dependencies for a service. This allows us to say certain services depend on the functionality of other services (and their check results) and form a logical unit. A typical example would be a web application consisting of a database and a webserver. If the database fails, the application running on the webserver does not work properly, so it makes sense to define a dependency between the two. Thus, if the database checks fail, Icinga will also mark the webserver (or the application if that is also monitored somehow) as failed. This helps in determining the impact an outage has. If a service comes back online, other dependent services also need to be checked (or restarted) to ensure continued functionality. Otherwise, the checks may report all green again, but the application may have suffered from the loss of the database and may need manual intervention to fix.

Here, we show the dependency of the agent-health check for services only:

```
apply Dependency "agent-health-check" to Service {
      parent_service_name = "agent-health"

      states = [ OK ] // Fail if parent service state switches to NOT-OK
      disable_notifications = true

// Automatically assign all agent endpoint checks as child services on the
// matched host
      assign where host.vars.agent_endpoint

// Avoid self reference from child to parent
      ignore where service.name == "agent-health"
}
```

We see how flexible Icinga is with its domain specific language using common pieces like **`"apply"`** together with placeholders (like **`Host`**, **`Service`** or **`Dependency`**) to define what

and how the monitoring should take place. The **agent-health** checks trigger if a state other than OK (like or "FAILED" or "UNREACHABLE") is detected. To not define this for every single host we have, and not forget it for any new hosts added later, we use the assign keyword again to apply this to all hosts defined as an endpoint.

Groups of hosts or services help to keep an overview of systems with common tasks or criteria, like webservers, database servers, front-end hosts, firewalls, etc. This is what **groups.conf** defines, but is optional when the infrastructure to monitor is small or too diverse for any commonalities:

```
object HostGroup "FreeBSD-servers" {
      display_name = "FreeBSD Servers"
      assign where host.vars.os == "FreeBSD"
}
```

Remember the object Host **"monitor.example.com"** definition in **hosts.conf** above? We defined a local variable **vars.os**. We can now filter on the value of this variable using the **"assign where"** statement. Tools that automatically add entries for new hosts in the infrastructure to **hosts.conf** may also hold the information about what operating system is used (among others), hence Icinga groups these systems in the Icingaweb2 display. ServiceGroups are defined similarly. That way, a report may contain the number of systems that are periodically checked for certain services. Webservers may run different checks than database servers, but as service groups, it is easy to either apply them to new hosts as a whole or define a mixture of both to form a whole new monitoring target.

The last file that I want to show is the **users.conf** file that holds all the information about users that Icinga understands and notifies when some checks fail. A basic definition may look like this:

```
object UserGroup "icingaadmins" {
      display_name = "Icinga Admin Group"
}

object User "icingaadmin" {
      display_name = "Icinga 2 Admin"
      groups = [ "icingaadmins" ]
      email = "icinga@localhost"
}

object User "Helpdesk" {
      email = "ticket@example.org"
      display_name = "The Friendly Helpdesk Folks"
      groups = [ "icingaadmins" ]
}
```

Users may be part of other groups as in this example where the Helpdesk user is part of the Icinga Admin Group. Individual users may be assigned to only a certain host or a set of services (experts in their field), but not to the overall infrastructure that is monitored.

Notification rules define who is contacted when and by which method (email by default, but pagers, SMS, and even various instant messengers are possible). Escalations to a different group after a certain amount of time can happen when a problem has not been dealt with (or at least acknowledged), to define certain service-level agreements or for paying (our impatient) customers.

Other files make up the Icinga monitoring and all are well defined in the documentation. For now, let's start all the services to get our basic monitoring infrastructure going. Especially after all the extra files are added, Icinga needs to know about them, so we restart that particular service:

```
monitor# service postgresql restart
monitor# service php-fpm start
monitor# service nginx start
monitor# service icinga2 restart
```

The Icingaweb2 service is configured via the web browser for which a token is needed because we don't want a random stranger driving by our freshly installed monitoring by accident to misconfigure it. The token is generated and emitted with the following command:

```
monitor# icingacli setup token create --config=/usr/local/etc/icingaweb2
monitor# chown -R www:www /usr/local/etc/icingaweb2
```

The token is now readable from the browser and when pasted into the web form, the remaining setup steps for Icingaweb2 can happen. Fill in the details like the database users we created and other information like the admin user and its password. At the end, the Icingaweb2 login will be presented, and you can access all your monitored hosts and services from this central place.

## Adding New Host Endpoints

After the initial excitement about Icinga's functionality you may be wondering how to add more objects to monitor. We will demonstrate this with a new host and show all the steps necessary to include it into our monitoring.

On a freshly installed host (we use FreeBSD here) called `client.example.org`, install the icinga2 package.

```
client# pkg install icinga2
```

Since this is a certificate-based authentication between this host and the central Icinga monitoring instance, we need to ensure that the directory holding the certificates exists and has the right ownership:

```
client# mkdir /var/lib/icinga2/certs
client# chown icinga:icinga !$
client# chown -R icinga:icinga /usr/local/etc/icinga2
```

Next, we enable the icinga2 service to start at system bootup:

```
client# sysrc icinga2_enable=yes
```

A client certificate is generated next using the **"icinga2 pki"** subcommand. While this command is interactive, we can also provide all necessary parameters directly on the command line to ease automation later when adding hundreds of hosts. Note that this has to run on the central monitoring instance.

```
monitor# icinga2 pki new-cert --cn client.example.org \
  --key /var/lib/icinga2/certs/client.example.org.key \
  --csr /var/lib/icinga2/certs/client.example.org.csr"
```

The file ending in **.csr** is the certificate signing request, which is now used in combination with the previously generated master key to create a new signed client certificate (**example.org.crt**).

```
monitor# icinga2 pki sign-csr \
  --csr /var/lib/icinga2/certs/client.example.org.csr \
  --cert /var/lib/icinga2/certs/client.example.org.crt"
```

When we ran **"icinga2 node setup --master"** at the beginning of this article to generate the master certificate to sign the others, a file called **monitor.example.org.crt** was created in **/var/lib/icinga2/certs/**. Transferring this to the client in a secure way is necessary to validate the server certificate. There are various ways to do this, depending on how much you trust the client and any users connected to it, as well as the network (or medium) between the two.

```
monitor# scp /var/lib/icinga2/certs/monitor.example.org.crt \
  client.example.org:/var/lib/icinga2/certs/
```

Next, import the certificate into the client and tell Icinga to trust it.

```
client# icinga2 pki save-cert --trustedcert \
  /var/lib/icinga2/certs/monitor.example.org.crt \
  --host client.example.org"
```

A new ticket is created for the client on the monitoring server to establish a trust relationship. Essentially, the client asks to be part of the monitoring infrastructure. These requests may be generated automatically and signed at a later time (after a review by a human or third entity).

```
monitor# icinga2 pki ticket --cn client.example.org
```

Note the resulting ticket output on the commandline (in our case 4f76d2ecda535753e9180838ebffbcbca242fe61), we'll need it in this next step on the client. It will take the generated ticket from the central monitoring instance and generate configuration files just like we did manually when we set up our monitor. The zone relationship is established,

making the monitor a parent of the client, establishing trust between them. Additionally, we tell the client to accept commands and configuration changes sent to it by the monitor. This is optional and clients may also choose to make their own configuration choices, independent of the host. Having the configuration on the central server and controlling the configuration of each client there eases the burden of keeping them in sync on every monitored host. When changing a setting like a new monitoring interval, it only needs to be set once and the clients will apply the changes coming from the monitor locally.

```
client# icinga2 node setup --ticket 4f76d2ecda535753e9180838ebffbcbca242fe61 \
  --cn client.example.org --endpoint monitor --zone client.example.org \
  --parent_zone my-zone --parent_host monitor.example.org \
  --trustedcert /var/lib/icinga2/certs/monitor.example.org.crt \
  --accept-commands --accept-config --disable-confd"
```

Before we start our Icinga instance, we need to verify that all files were written correctly and conform to Icinga's logic. To do that, we tell the Icinga daemon to perform a configuration check with the following command:

```
client# icinga2 daemon -C
```

If there are any errors, Icinga tries to help pinpoint the file and line in question. Typical errors may be providing the wrong names for the parent or zone. Once the validation is complete, a new entry for this new client needs to be added on the monitoring server to include it in future check executions. On **monitor.example.org**, edit

```
/usr/local/etc/icinga2/zones.d/my-zone/hosts.conf
```

and add the following lines of configuration, ensuring that this is placed before the central hosts object definition:

```
object Host "client.example.org" {
      import "generic-host" // import generic settings for all hosts
      display_name = "My Client Host"
      address = "client.example.org"

      vars.os = "FreeBSD"
//follow convention that host name == endpoint name
      vars.agent_endpoint = name
}
```

Host objects are fairly simple to define and don't contain any new fields that we have not yet seen from our previous edits when we added the central server itself. As before, we also need to define that this host is an endpoint (no further monitoring clients are below it and that it is not a parent of another host) as well as the zone it belongs to. Typically, these entries are placed before the line containing **'object Zone "director-global" {'** and look like this:

```
object Endpoint "client.example.org" {
// client connects itself
      host = "client.example.org"
      log_duration = 0
}

object Zone "client.example.org" {
      endpoints = [ "client.example.org" ]
      parent = "my-zone" // Establish zone hierarchy
}
```

In the zone object, we only need to define the name of our endpoint, referencing the host definition we did earlier in the file. The parent zone is the one that was generated when we created the monitor certificate. There should already be entries for it in the file by the Icinga configuration. The log duration entry as endpoint attribute instructs the endpoint how long to store a replay log of all check results on the client if the connection to the parent is lost. Once the connection is reestablished, the client will replay the log and all the data will be sent to the parent. Since the parent schedules all the checks to be run on the monitored systems, setting this to zero is fine.

We're done wading through configuration files on both hosts. The only thing left is to start the icinga2 service on the client and on the server to read the configuration changes we made.

```
client# service icinga2 start
monitor# service icinga2 restart
```

The new client should now appear in the Icingaweb2 overview as pending. When the next scheduled check interval happens, the client is contacted in a secure way (since they exchanged certificates, remember?) checks are executed, and results delivered to the central host.

Congratulations, you can now enjoy monitoring your infrastructure for common services and add new hosts to it. Make sure to check out the Icinga configuration for various monitoring-related information and further ways of configuring your Icinga installation to fit your needs.

---

**BENEDICT REUSCHLING** is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.