An overview of Splunk .conf files

If you have spent any length of time in the filesystem investigating Splunk, you must have seen many different files ending in .conf. In this section, we will give you a quick overview of the most common `.conf` files. The official documentation is the best place to look for a complete reference to files and attributes.

The quickest way to find the official documentation is with your favorite search engine by searching for splunk `filename.conf`. For example, a search for splunk `props.conf` pulled up (and will pull up) the Splunk documentation for `props.conf` first in every search engine I tested.

props.conf

The stanzas in `props.conf` define which events to match based on host, source, and sourcetype. These stanzas are merged into the master configuration based on the uniqueness of stanza and attribute names, as with any other configuration, but there are specific rules governing when each stanza is applied to an event and in what order. Stated as simply as possible, attributes are sorted by type, then by priority, and then by the ASCII value.

We'll cover those rules under the *Stanza types* section. First, let's look at common attributes.

COMMON ATTRIBUTES

The full set of attributes allowed in `props.conf` is vast. Let's look at the most common attributes and try to break them down by the time when they are applied.

Search-time attributes

The most common attributes that users will make in `props.conf` are field extractions. When a user defines an extraction through the web interface, it ends up in `props.conf`, as shown here:

```
[my_source_type]
EXTRACT-foo = \s(?<bar>\d+)ms
EXTRACT-cat = \s(?<dog>\d+)s
```

This configuration defines the fields bar and dog for the `my_source_type` source type. Extracts are the most common search-time configurations. Any of the stanza types listed under the *Stanza types* section can be used, but the source type is definitely the most common one.

Other common search-time attributes include:

- `REPORT-foo = bar`: This attribute is a way to reference stanzas in `transforms.conf`, but apply them at search time instead of index time. This approach predates `EXTRACT` and is still useful in a few special cases.

  We will cover this case later under the *transforms.conf* section.

- `KV_MODE = auto`: This attribute allows you to specify whether Splunk should automatically extract fields in the form of `key=value` from events. The default value is auto. The most common change is to disable automatic field extraction for performance reasons by setting the value to none. Other possibilities are multi, JSON, and XML.
- `LOOKUP-foo = mylookup barfield`: This attribute lets you wire up a lookup to automatically run for a set of events. The lookup itself is defined in `transforms.conf`.

Index-time attributes

As discussed in Chapter 3, *Tables, Charts, and Fields*, it is possible to add fields to the metadata of events. This is accomplished by specifying a transform in `transforms.conf`, and an attribute in `props.conf`, to tie the transformation to specific events.

The attribute in `props.conf` looks as follows: `TRANSFORMS-foo = bar1,bar2`.

This attribute references stanzas in `transforms.conf` by name, in this case, `bar1` and `bar2`. These transform stanzas are then applied to the events matched by the stanza in `props.conf`.

Parse-time attributes

Most of the attributes in `props.conf` actually have to do with parsing events. To successfully parse events, a few questions need to be answered, such as these:

- When does a new event begin? Are events multiline? Splunk will make fairly intelligent guesses, but it is best to specify an exact setting. Attributes that help with this include:
  - `SHOULD_LINEMERGE = false`: If you know that your events will never contain the newline character, setting this to `false` will eliminate a lot of processing.
  - `BREAK_ONLY_BEFORE = ^\d\d\d\d-\d\d-\d\d`: If you know that new events always start with a particular pattern, you can specify it using this attribute.
  - `TRUNCATE = 1024`: If you are certain you only care about the first *n*characters of an event, you can instruct Splunk to truncate each line. What is considered a line can be changed with the next attribute.
  - `LINE_BREAKER = ([\r\n]+)(?=\d{4}-\d\d-\d\d)`: This is the most efficient approach to multiline events is to redefine what Splunk considers a line. This example says that a line is broken on any number of newlines followed by a date of the form `1111-11-11`. The big disadvantage to this approach is that, if your log changes, you will end up with garbage in your index until you update your configuration. Try the props helper app available at Splunkbase for help in making this kind of configuration.
- Where is the date? If there is no date, see `DATETIME_CONFIG` further down this bullet list. The relevant attributes are as follows:
  - `TIME_PREFIX = ^\[`: By default, dates are assumed to fall at the beginning of the line. If this is not true, give Splunk some help and move the cursor past the characters preceding the date. This pattern is applied to each line, so if you have

redefined `LINE_BREAKER`correctly, you can be sure only the beginnings of actual multiline events are being tested.
    - `MAX_TIMESTAMP_LOOKAHEAD = 30`: Even if you change no other setting, you should change this one. This setting says how far after `TIME_PREFIX` to test for dates. With no help, Splunk will take the first 150 characters of each line and then test regular expressions to find anything that looks like a date. The default regular expressions are pretty lax, so what it finds may look more like a date than the actual date. If you know that your date is never more than *n* characters long, set this value to `n` or `n+2`. Remember that the characters retrieved come after `TIME_PREFIX`.
- What does the date look like? These attributes will be of assistance here:
    - `TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N %:z`: If this attribute is specified, Splunk will apply `strptime` to the characters immediately following `TIME_PREFIX`. If this matches, then you're done. This is, by far, the most efficient and least error-prone approach. Without this attribute, Splunk actually applies a series of regular expressions until it finds something that looks like a date.
    - `DATETIME_CONFIG = /etc/apps/a/custom_datetime.xml`: As mentioned, Splunk uses a set of regular expressions to determine the date. If `TIME_FORMAT` is not specified, or won't work for some strange reason, you can specify a different set of regular expressions or disable time extraction completely by setting this attribute to `CURRENT`(the indexer clock time) or `NONE` (file modification time, or if there is no file, clock time). I personally have never had to resort to a custom `datetime.xml` file, though I have heard of it being done.
- The Data preview function available when you are adding data through the manager interface builds a good, usable configuration. The configuration generated does not use `LINE_BREAKER`, which is definitely safer but less efficient.

  Here is a sample stanza that uses `LINE_BREAKER` for efficiency:

```
[mysourcetype]
TIME_FORMAT = %Y-%m-%d %H:%M:%S.%3N %:z
```

```
MAX_TIMESTAMP_LOOKAHEAD = 32
TIME_PREFIX = ^\[
SHOULD_LINEMERGE = False
LINE_BREAKER = ([\r\n]+)(?=\[\d{4}-\d{1,2}-\d{1,2}\s+\
d{1,2}:\d{1,2}:\d{1,2})
TRUNCATE = 1024000
This configuration would apply to log messages that looked like
this:
[2011-10-13 13:55:36.132 -07:00] ERROR Interesting message.
More information.
And another line.
[2011-10-13 13:55:36.138 -07:00] INFO All better.
[2011-10-13 13:55:37.010 -07:00] INFO More data
and another line.
```

Let's walk through how these settings affect the first line of this sample configuration:

- `LINE_BREAKER` states that a new event starts when one or more newline characters are followed by a bracket and series of numbers and dashes in the `[1111-11-11 11:11:11]` pattern.
- `SHOULD_LINEMERGE=False` tells Splunk not to bother trying to recombine multiple lines.
- `TIME_PREFIX` moves the cursor to the character after the `[` character.
- `TIME_FORMAT` is tested against the characters at the current cursor location. If it succeeds, we are done.
- If `TIME_FORMAT` fails, `MAX_TIMESTAMP_LOOKAHEAD` characters are read from the cursor position (after `TIME_PREFIX`) and the regular expressions from `DATE_CONFIG` are tested.
- If the regular expressions fail against the characters returned, the time last parsed from an event is used. If there is no last time parsed, the modification date from the file would be used, if known; otherwise, the current time would be used.

This is the most efficient and precise way to parse events in Splunk, but also the most brittle. If your date format changes, you will almost certainly have junk data in your index. Only use this approach if you are confident the format of your logs will not change without your knowledge.

Input-time attributes

There are only a couple of attributes in `props.conf` that matter at the input stage, but they are generally not needed:

- `CHARSET = UTF-16LE`: When reading data, Splunk has to know the character set used in the log. Most applications write their logs using either `ISO-8859-1 or UTF-8` is handled by the default settings just fine. Some Windows applications write logs in 2-byte Little Endian, which is indexed as garbage. `Setting CHARSET = UTF-16LE` takes care of the problem. Check the official documentation for a list of supported encodings.
- `NO_BINARY_CHECK = true`: If Splunk believes that a file is binary, it will not index the file at all. If you find that you have to change this setting to convince Splunk to read your files, it is likely that the file is in an unexpected character set. You might try other `CHARSET` settings before enabling this setting.

STANZA TYPES

Now that we have looked at common attributes, let's talk about the different types of stanzas in `props.conf`. Stanza definitions can take the following three forms:

- `[foo]`
  - This is the exact name of a source type and is the most common type of stanza to be used; the source type of an event is usually defined in `inputs.conf`
  - Wildcards are not allowed
- `[source::/logs/.../*.log]`
  - This matches the source attribute, which is usually the path to the log where the event came from
  - `*` matches a file or directory name
  - `...` matches any part of a path
- `[host::*nyc*]`
  - This matches the host attribute, which is usually the value of the hostname on a machine running Splunk Forwarder
  - `*` is allowed

Types follow this order in taking precedence:

1. Source.
2. Host.
3. Source type.

For instance, say an event has the following fields:

```
sourcetype=foo_type
source=/logs/abc/def/gh.log
host=dns4.nyc.mycompany.com
```

Given this configuration snippet and our preceding event, we have the following code:

```
[foo_type]
TZ = UTC
[source::/logs/.../*.log]
TZ = MST
[host::*nyc*]
TZ = EDT
```

`TZ = MST` would be used during parsing because the source stanza takes precedence.

To extend this example, say we have this snippet:

```
[foo_type]
TZ = UTC
TRANSFORMS-a = from_sourcetype
[source::/logs/.../*.log]
TZ = MST
BREAK_ONLY_BEFORE_DATE = True
TRANSFORMS-b = from_source
[host::*nyc*]
TZ = EDT
BREAK_ONLY_BEFORE_DATE = False
TRANSFORMS-c = from_host
```

The attributes applied to our event would, therefore, be as shown here:

```
TZ = MST
BREAK_ONLY_BEFORE_DATE = True
TRANSFORMS-a = from_sourcetype
TRANSFORMS-b = from_source
TRANSFORMS-c = from_host
```

PRIORITIES INSIDE A TYPE

If there are multiple source or host stanzas that match a given event, the order in which settings are applied also comes into play. A stanza with a pattern has a priority of `0`, while an exact stanza has a priority of `100`. Higher priorities win. For instance, say we have the following stanza:

```
[source::/logs/abc/def/gh.log]
TZ = UTC
[source::/logs/.../*.log]
TZ = CDT
```

Our `TZ` value will be `UTC` since the exact match of `source::/logs/abc/def/gh.log` has a higher priority.

When priorities are identical, stanzas are applied by the ASCII order. For instance, say we have this configuration snippet:

```
[source::/logs/abc/.../*.log]
TZ = MST
[source::/logs/.../*.log]
TZ = CDT
```

The attribute `TZ=CDT` would win because `/logs/.../*.log` is first in the ASCII order.

This may seem counterintuitive since `/logs/abc/.../*.log` is arguably a better match. The logic for determining what makes a better match, however, can quickly become fantastically complex, so the ASCII order is a reasonable approach.

You can also set your own value of priority, but luckily, it is rarely needed.

ATTRIBUTES WITH CLASS

As you dig into configurations, you will see attribute names of the FOO-bar form.

The word after the dash is generally referred to as the class. These attributes are special in a few ways:

- Attributes merge across files as with any other attribute
- Only one instance of each class will be applied according to the rules described previously
- The final set of attributes is applied in the ASCII order by the value of the class. Once again, say we are presented with an event with the following fields:
  - `sourcetype=foo_type`
  - `source=/logs/abc/def/gh.log`

    `host=dns4.nyc.mycompany.com`

And, say that this is the configuration snippet:

```
[foo_type]
TRANSFORMS-a = from_sourcetype1, from_sourcetype2
[source::/logs/.../*.log]
TRANSFORMS-c = from_source_b
[source::/logs/abc/.../*.log]
TRANSFORMS-b = from_source_c
[host::*nyc*]
TRANSFORMS-c = from_host
The surviving transforms would then be:
TRANSFORMS-c = from_source_b
TRANSFORMS-b = from_source_c
TRANSFORMS-a = from_sourcetype1, from_sourcetype2
```

To determine the order in which the transforms are applied to our event, we will sort the stanzas according to the values of their classes, in this case, `c`, `b`, and `a`. This gives us:

```
TRANSFORMS-a = from_sourcetype1, from_sourcetype2
TRANSFORMS-b = from_source_c
TRANSFORMS-c = from_source_b
```

The transforms are then combined into a single list and executed in this order:

```
from_sourcetype1, from_sourcetype2, from_source_c,
from_source_b
```

The order of transforms usually doesn't matter, but it is important to understand it if you want to chain transforms and create one field from another. We'll try this later, in the *transforms.conf* section.

inputs.conf

This configuration, as you might guess, controls how data makes it into Splunk.

By the time this data leaves the input stage, it still isn't an event but has some basic metadata associated with it: `host`, `source`, `sourcetype`, and optionally `index`. This basic metadata is then used by the parsing stage to break the data into events according to the rules defined in `props.conf`:

Input types can be broken down into files, network ports, and scripts. First, we will look at attributes that are common to all inputs.

COMMON INPUT ATTRIBUTES

These common bits of metadata are used in the parsing stage to pick the appropriate stanzas in `props.conf`.

- `host`: By default, `host` will be set to the hostname of the machine producing the event. This is usually the correct value, but it can be overridden when appropriate.

- `source`: This field is usually set to the path, file, or network port that an event came from, but this value can be hardcoded.
- `sourcetype`: This field is almost always set in `inputs.conf` and is the primary field to determine which set of parsing rules in `props.conf` to apply to these events.

  ### NOTE

  It is very important to set `sourcetype`. In the absence of a value, Splunk will create automatic values based on the source, which can easily result in an explosion of `sourcetype` values.

- `index`: This field says what index to write events to. If it is omitted, the default `index` will be used.

All of these values can be modified using transforms, the only caveat being that these transforms are applied after the parsing step. The practical consequence of this is that you cannot apply different parsing rules to different events in the same file, for instance, different time formats on different lines.

FILES AS INPUTS

The vast majority of events in Splunk come from files. Usually, these events are read from the machine where they are produced and as the logs are written. Very often, the entire input's stanza will look as follows:

```
[monitor:///logs/interesting.log*]
sourcetype=interesting
```

This is often all that is needed. This stanza says:

- Read all logs that match the `/logs/interesting.log*` pattern, and going forward, watch them for new data
- Name the source type "interesting"
- Set the source to the name of the file in which the log entry was found
- Default the host to the machine where the logs originated
- Write the events to the default index

These are usually perfectly acceptable defaults. If `sourcetype` is omitted, Splunk will pick a default source type based on the filename, which you don't want—your source type list will get very messy very fast.

Using patterns to select rolled logs

You may notice that the previous stanza ended in `*`. This is important because it gives Splunk a chance to find events that were written to a log that has recently rolled. If we simply watch `/logs/interesting.log`, it is likely that events will be missed at the end of the log when it rolls, particularly on a busy server.

There are specific cases where Splunk can get confused, but in the vast majority of cases, the default mechanisms do exactly what you would hope for. See the *When to use crcSalt* section further on for a discussion about special cases.

Using blacklist and whitelist

It is also possible to use a blacklist and whitelist pattern for more complicated patterns. The most common use case is to blacklist files that should not be indexed, for instance, `gz` and `zip` files. This can be done as follows:

```
[monitor:///opt/B/logs/access.log*]
sourcetype=access
blacklist=.*.gz
```

This stanza would still match `access.log.2012-08-30`, but if we had a script that compressed older logs, Splunk would not try to read `access.log.2012-07-30.gz`.

Conversely, you can use a whitelist to apply very specific patterns, as shown here:

```
[monitor:///opt/applicationserver/logs]
sourcetype=application_logs
whitelist=(app|application|legacy|foo)\.log(\.\d{4})?
blacklist=.*.gz
```

This whitelist would match `app.log`, `application.log`, `legacy.log.2012-08-13`, and `foo.log`, among others. The blacklist will negate any `gz` files.

Since a log is a directory, the default behavior will be to recursively scan that directory.

Selecting files recursively

The layout of your logs or your application may dictate a recursive approach.

For instance, say we have these stanzas:

```
[monitor:///opt/*/logs/access.log*]
sourcetype=access
[monitor:///opt/.../important.log*]
sourcetype=important
```

The character `*` will match a single file or directory, while `...` will match any depth. This will match the files you want, with the caveat that all of `/opt` will continually be scanned.

Splunk will continually scan all directories from the first wildcard in a monitor path.

If `/opt` contains many files and directories, which it almost certainly does, Splunk will use an unfortunate amount of resources scanning all directories for matching files, constantly using memory and CPU. I have seen a single Splunk process watching a large directory structure use 2 gigabytes of memory. A little creativity can take care of this, but it is something to be aware of.

The takeaway is that if you know the possible values for `*`, you are better off writing multiple stanzas. For instance, assuming our directories in `/opt` are `A` and `B`, the following stanzas will be far more efficient:

```
[monitor:///opt/A/logs/access.log*]
sourcetype=access
[monitor:///opt/B/logs/access.log*]
```

```
sourcetype=access
```

It is also perfectly acceptable to have stanzas matching files and directories that simply don't exist. This causes no errors, but be careful not to include patterns that are so broad that they match unintended files.

Following symbolic links

When scanning directories recursively, the default behavior is to follow symbolic links. Often this is very useful, but it can cause problems if a symbolic link points to a large or slow filesystem. To control this behavior, simply do this:

```
followSymlink = false
```

It's probably a good idea to put this on all of your monitor stanzas until you know you need to follow a symbolic link.

Setting the value of the host from the source

The default behavior of using the hostname from the machine forwarding the logs is almost always what you want. If, however, you are reading logs for a number of hosts, you can extract the hostname from the source using `host_regex` or `host_segment`. For instance, say we have the path:

```
/nfs/logs/webserver1/access.log
```

To set host to `webserver1`, you could use:

```
[monitor:///nfs/logs/*/access.log*]
sourcetype=access
host_segment=3
```

You could also use:

```
[monitor:///nfs/logs/*/access.log*]
sourcetype=access
host_regex=/(.*?)/access\.log
```

The `host_regex` variable could also be used to extract the value of the host from the filename. It is also possible to reset the host using a transform, with the caveat that this will occur after parsing, which means any settings in `props.conf` that rely on matching the host will already have been applied.

Ignoring old data at installation

It is often the case that, when Splunk is installed, months or years of logs are sitting in a directory where logs are currently being written. Logs that are appended to infrequently may also have months or years of events that are no longer interesting and would be wasteful to index.

The best solution is to set up archive scripts to compress any logs older than a few days, but in a large environment, this may be difficult to do. Splunk has two settings that help ignore older data, but be forewarned: once these files have been ignored, there is no simple way to change your mind later. If, instead, you compress older logs and blacklist the compressed files as explained in the *Using blacklist and whitelist* section, you can simply decompress, at a later stage, any files you would like to index. Let's look at a sample stanza:

```
[monitor:///opt/B/logs/access.log*]
sourcetype = access
ignoreOlderThan = 14d
```

In this case, `ignoreOlderThan` says to ignore, forever, all events in any files, the modification date of which is older than `14` days. If the file is updated in the future, any new events will be indexed.

The `followTail` attribute lets us ignore all events written until now, instead starting at the end of each file. Let's look at an example:

```
[monitor:///opt/B/logs/access.log*]
sourcetype = access
followTail = 1
```

Splunk will note the length of files matching the pattern, but `TailfollowTail` instructs Splunk to ignore everything currently in these

files. Any new events written to the files will be indexed. Remember that there is no easy way to alter this if you change your mind later.

It is not currently possible to say *ignore all events older than x*, but since most logs roll on a daily basis, this is not commonly a problem.

When to use crcSalt

To keep track of what files have been seen before, Splunk stores a checksum of the first 256 bytes of each file it sees. This is usually plenty as most files start with a log message, which is almost guaranteed to be unique. This breaks down when the first 256 bytes are not unique on the same server.

I have seen two cases where this happens, as follows:

1. The first case is when logs start with a common header containing information about the product verion, for instance:

```
==========================================================
=
== Great product version 1.2 brought to you by Great company ==
== Server kernel version 3.2.1 ==
```

4. The second case is when a server writes many thousands of files with low time resolution, for instance:

```
12:13:12 Session created
12:13:12 Starting session
```

To deal with these cases, we can add the path to the log to the checksum, or salt our crc. This is accomplished as shown here:

```
[monitor:///opt/B/logs/application.log*]
sourcetype = access
crcSalt = <SOURCE>
```

It says to include the full path to this log in the checksum.

This method will only work if your logs have a unique name. The easiest way to accomplish this is to include the current date in the name of the

log when it is created. You may need to change the pattern for your log names so that the date is always included and the log is not renamed.

Do not use **crcSalt** if your logs change names!

If you enable crcSalt in an input where it was not already enabled, you will re-index all the data! You need to ensure that the old logs are moved aside or uncompressed and blacklisted before enabling this setting in an existing configuration.

Destructively indexing files

If you receive logfiles in batches, you can use the batch input to consume `logs` and then delete them. This should only be used against a copy of the logs.

See the following example:

```
[batch:///var/batch/logs/*/access.log*]
sourcetype=access
host_segment=4
move_policy = sinkhole
```

This stanza would index the files in the given directory and then delete the files. Make sure this is what you want to do!

NETWORK INPUTS

In addition to reading files, Splunk can listen to network ports. The stanzas take the following form:

```
[protocol://<remote host>:<local port>]
```

The remote host portion is rarely used, but the idea is that you can specify different input configurations for specific hosts. The usual stanzas look as follows:

- `[tcp://1234]`: This specifies that we will listen to port 1234 for TCP connections. Anything can connect to this port and send data in.

- `[tcp-ssl://importanthost:1234]`: This listens on TCP using SSL, and we can apply this stanza to the `importanthost` host. Splunk will generate self-signed certificates the first time it is launched.
- `[udp://514]`: This is generally used to receive `syslog` events. While this does work, it is generally considered a best practice to use a dedicated syslog receiver, such as `rsyslog` or `syslogng`. See Chapter 12, *Advanced Deployments*, for a discussion on this subject.
- `[splunktcp://9997] or [splunktcp-ssl://9997]`: In a distributed environment, your indexers will receive events on the specified port. It is a custom protocol used between Splunk instances. This stanza is created for you when you use the **Manager** page at **Manager** | **Forwarding and receiving** | **Receive data**.

For TCP and UDP inputs, the following attributes apply:

- `source`: If it is not specified, the source will default to `protocol:port`, for instance, `udp:514`.
- `sourcetype`: If it is not specified, `sourcetype` will also default to `protocol:port`, but this is generally not what you want. It is best to specify a source type and create a corresponding stanza in `props.conf`.
- `connection_host`: With network inputs, what value to capture for `host`is somewhat tricky. Your options essentially are:
    - `connection_host = dns` uses reverse DNS to determine the hostname from the incoming connection. When reverse DNS is configured properly, this is usually your best bet. This is the default setting.
    - `connection_host = ip` sets the host field to the IP address of the remote machine. This is your best choice when reverse DNS is unreliable.
    - `connection_host = none` uses the hostname of the Splunk instance receiving the data. This option can make sense when all traffic is going to an interim host.
    - `host = foo` sets the hostname statically.
    - It is also common to reset the value of the host using a transform, for instance, with syslog events. This happens after parsing, though, so it is too late to change things such as time zone based on the host.

- `queueSize`: This value specifies how much memory Splunk is allowed to set aside for an input queue. A common use for a queue is to capture spikey data until the indexers can catch up.
- `persistentQueueSize`: This value specifies a persistent queue that can be used to capture data to the disk if the in-memory queue fills up. If you find yourself building a particularly complicated setup around network ports, I would encourage you to talk to Splunk support as there may be a better way to accomplish your goals.

NATIVE WINDOWS INPUTS

One nice thing about Windows is that system logs and many application logs go to the same place.

Unfortunately, that place is not a file, so native hooks are required to access these events. Splunk makes those inputs available using stanzas of the `[WinEventLog:LogName]`. form. For example, to index the `Security` log, the stanza simply looks like this:

```
[WinEventLog:Security]
```

There are a number of supported attributes, but the defaults are reasonable. The only attribute I have personally used is `current_only`, which is the equivalent of `followTail` for monitor stanzas. For instance, this stanza says to monitor the `Application` log, but also to start reading from now:

```
[WinEventLog:Application]
current_only = 1
```

This is useful when there are many historical events on the server.

The other input available is **Windows Management Instrumentation**(**WMI**). With WMI, you can accomplish the following:

- Monitor native performance metrics as you would find in Windows Performance Monitor
- Monitor the Windows Event Log API
- Run custom queries against the database behind WMI
- Query remote machines

Even though it is theoretically possible to monitor many Windows servers using WMI and a few Splunk forwarders, this is not advised. The configuration is complicated, does not scale well, introduces complicated security implications, and is not thoroughly tested. Also, reading Windows Event Logs via WMI produces different output than the native input, and most apps that expect Windows events will not function as expected.

The simplest way to generate the `inputs.conf` and `wmi.conf`configurations needed for Windows Event Logs and WMI is to install Splunk for Windows on a Windows host and then configure the desired inputs through the web interface. See the official Splunk documentation for more examples.

SCRIPTS AS INPUTS

Splunk will periodically execute processes and capture the output. For example, here is input from the `ImplementingSplunkDataGenerator`app:

```
[script://./bin/implSplunkGen.py 2]
interval=60
sourcetype=impl_splunk_gen_sourcetype2
source=impl_splunk_gen_src2
host=host2
index=implSplunk
```

Things to notice in this example are as follows:

- The present working directory is the root of the app that contains `inputs.conf`.
- Files that end with `.py` will be executed using the Python interpreter included with Splunk. This means the Splunk Python modules are available.

  To use a different Python module, specify the path to Python in the stanza.

- Any arguments specified in the stanza will be handed to the script as if it was executed at the command line.

- The interval specifies how often, in seconds, this script should be run:
    - If the script is still running, it will not be launched again.
    - Long-running scripts are fine. Since only one copy of a script will run at a time, the interval will instead indicate how often to check whether the script is still running.
    - This value can also be specified in the `cron` format.

Any programming language can be used as long as it can be executed at the command line. Splunk simply captures the standard output from whatever is executed.

Included with Splunk for Windows are scripts to query WMI. One sample stanza looks as follows:

```
[script://$SPLUNK_HOME\bin\scripts\splunk-wmi.path]
```

The things to note are as follows:

- Windows paths require backslashes instead of slashes
- `$SPLUNK_HOME` will expand properly

transforms.conf

The `transforms.conf` configuration is where we specify transformationsand lookups that can then be applied to any event. These transforms and lookups are referenced by name in `props.conf`.

For our examples in the later subsections, we will use this event:

```
2012-09-24T00:21:35.925+0000 DEBUG [MBX] Password reset
called.
[old=1234, new=secret, req_time=5346]
```

We will use it with these metadata values:

```
sourcetype=myapp
source=/logs/myapp.session_foo-jA5MDkyMjEwMTIK.log
host=vlbmba.local
```

CREATING INDEXED FIELDS

One common task accomplished with `transforms.conf` is the creation of new indexed fields. Indexed fields are different than extracted fields in that they must be created at index time and can be searched for whether the value is in the raw text of the event or not. It is usually preferable to create extracted fields instead of indexed fields. See Chapter 3, *Tables, Charts, and Fields*, for a deeper discussion about when indexed fields are beneficial.

Indexed fields are only applied to events that are indexed after the definition is created. There is no way to backfill a field without re-indexing.

Creating a loglevel field

The format of a typical stanza in `transforms.conf` looks as follows:

```
[myapp_loglevel]
REGEX = \s([A-Z]+)\s
FORMAT = loglevel::$1
WRITE_META = True
```

This will add to our events the field `loglevel=DEBUG`. This is a good idea if the values of `loglevel` are common words outside of this location, for instance `ERROR`.

Walking through this stanza, we have the following:

- `[myapp_loglevel]`: The stanza can be any unique value, but it is in your best interest to make the name meaningful. This is the name referenced in `props.conf`.
- `REGEX = \s([A-Z]+)\s`: This is the pattern to test against each event that is handed to us. If this pattern does not match, this transform will not be applied.
- `FORMAT = loglevel::$1`: Create the `loglevel`. Under the hood, all indexed fields are stored using a :: delimiter, so we have to follow that form.

- `WRITE_META = True`: Without this attribute, the transform won't actually create an indexed field and store it with the event.

Creating a session field from the source

Using our event, let's create another field, session, which appears only to be in the value of the source:

```
[myapp_session]
SOURCE_KEY = MetaData:Source
REGEX = session_(.*?)\.log
FORMAT = session::$1
WRITE_META = True
```

Note the `SOURCE_KEY`.attribute. The value of this field can be any existing metadata field or another indexed field that has already been created. See the *Attributes with class* subsection within the *props.conf* section for a discussion about the transform execution order. We will discuss these fields in the `Modifying metadata fields` subsection.

Creating a tag field

It is also possible to create fields simply to tag events that would be difficult to search for otherwise. For example, if we wanted to find all events that were slow, we could search for:

```
sourcetype=myapp req_time>999
```

Without an indexed field, this query would require parsing every event that matches `sourcetype=myapp` over the time that we are interested in. The query would then discard all events whose `req_time` value was 999 or less.

If we know ahead of time that a value of `req_time>999` is bad, and we can come up with a regular expression to specify what "bad" is, we can tag these events for quicker retrieval. Say we have this `transforms.conf` stanza:

```
[myapp_slow]
```

```
REGEX = req_time=\d{4,}
FORMAT = slow_request::1
WRITE_META = True
```

This `REGEX` will match any event containing `req_time=` followed by four or more digits.

After adding `slow_request` to `fields.conf` (see the *fields.conf* section), we can search for `slow_request=1` and find all slow events very efficiently. This will not apply to events that were indexed before this transform existed. If the events that are slow are uncommon, this query will be much faster.

Creating host categorization fields

It is common to have parts of a hostname mean something in particular. If this pattern is well known and predictable, it may be worthwhile to pull the value out into fields. Working from our fictitious host value, `vlbmba.local`(which happens to be my laptop), we might want to create fields for the owner and the host type. Our stanza might look similar to this:

```
[host_parts]
SOURCE_KEY = MetaData:Host
REGEX = (...)(...)\.
FORMAT = host_owner::$1 host_type::$2
WRITE_META = True
```

With our new fields, we can now easily categorize errors by whatever information is encoded into the hostname. Another approach would be to use a lookup, which has the advantage of being retroactive. This approach has the advantage of faster searches for the specific fields.

MODIFYING METADATA FIELDS

It is sometimes convenient to override the main metadata fields. We will look at one possible reason for overriding each base metadata value.

Remember that transforms are applied after parsing, so changing metadata fields via transforms cannot be used to affect which `props.conf` stanzas are applied for date parsing or line breaking.

For instance, with `syslog` events that contain the hostname, you cannot change the time zone because the date has already been parsed before the transforms are applied. The keys provided by Splunk include:

- `_raw` (this is the default value for `SOURCE_KEY`)
- `MetaData:Source`
- `MetaData:Sourcetype`
- `MetaData:Host`
- `_MetaData:Index`

Overriding the host

If your hostnames are appearing differently from different sources: for instance, `syslog` versus Splunk Forwarders, you can use a transform to normalize these values. Given our hostname `vlbmba.local`, we may want to only keep the portion to the left of the first period. The stanza would look as follows:

```
[normalize_host]
SOURCE_KEY = MetaData:Host
DEST_KEY = MetaData:Host
REGEX = (.*?)\.
FORMAT = host::$1
```

This will replace our hostname with `vlbmba`. Note these two things:

- `WRITE_META` is not included because we are not adding to the metadata of this event; we are instead overwriting the value of a core metadata field
- `host::` must be included at the beginning of the format

Overriding the source

Some applications will write a log for each session, conversation, or transaction. One problem this introduces is an explosion of source values. The values of the source will end up

in `$SPLUNK_HOME/var/lib/splunk/*/db/Sources.data`—one line per unique value of the source. This file will eventually grow to a huge size, and Splunk will waste a lot of time updating it, causing unexplained pauses. A new setting in `indexes.conf`, called `disableGlobalMetadata`, can also eliminate this problem.

To flatten this value, we could use a stanza such as this:

```
[myapp_flatten_source]
SOURCE_KEY = MetaData:Source
DEST_KEY = MetaData:Source
REGEX = (.*session_).*.log
FORMAT = source::$1x.log
```

This would set the value of source to `/logs/myapp.session_x.log`, which would eliminate our growing source problem. If the value of session is useful, the transform in the *Creating a session field from source* section could be run before this transform to capture the value. Likewise, a transform could capture the entire value of the source and place it into a different metadata field.

A huge number of logfiles on a filesystem introduces a few problems, including running out of nodes and the memory used by the Splunk process of tracking all of the files. As a general rule, a cleanup process should be designed to archive older logs.

Overriding sourcetype

It is not uncommon to change the `sourcetype` field of an event based on the contents of the event, particularly from syslog. In our fictitious example, we want a different source type for events that contain `[MBX]` after the log level so that we can apply different extracts to these events. The following examples will do this work:

```
[mbx_sourcetype]
DEST_KEY = MetaData:Sourcetype
REGEX = \d+\s[A-Z]+\s\([MBX\])
FORMAT = sourcetype::mbx
```

Use this functionality carefully as it easy to go conceptually wrong, and this is difficult to fix later.

Routing events to a different index

At times, you may want to send events to a different index, either because they need to live longer than other events or because they contain sensitive information that should not be seen by all users. This can be applied to any type of event from any source, whether it be a file, network, or script.

All that we have to do is match the event and reset the index.

```
[contains_password_1]
DEST_KEY = _MetaData:Index
REGEX = Password reset called
FORMAT = sensitive
```

The things to note are as follows:

- In this scenario, you will probably make multiple transforms, so make sure to make the name unique
- `DEST_KEY` starts with an underscore
- `FORMAT` does not start with `index::`
- The index `sensitive` must exist on the machine indexing the data, or else the event will be lost

LOOKUP DEFINITIONS

A simple lookup simply needs to specify a filename in `transforms.conf`, as shown here:

```
[testlookup]
filename = test.csv
```

Assuming that `test.csv` contains the `user` and `group` columns and our events contain the field user, we can reference this lookup using the `lookup`command in search, as follows:

```
* | lookup testlookup user
```

Otherwise, we can wire this lookup to run automatically in `props.conf`, as follows:

```
[mysourcetype]
LOOKUP-testlookup = testlookup user
```

That's all you need to get started, and this probably covers most cases. See the *Using lookups to enrich data* section in Chapter 7, *Extending Search*, for instructions on creating lookups.

Wildcard lookups

In Chapter 10, *Summary Indexes and CSV Files*, we edited `transforms.conf` but did not explain what was happening. Let's take another look. Our transform stanza looks as follows:

```
[flatten_summary_lookup]
filename = flatten_summary_lookup.csv
match_type = WILDCARD(url)
max_matches = 1
```

Walking through what we added, we have the following terms and their descriptions:

- `match_type = WILDCARD(url)`: This says that the value of the `url`field in the lookup file may contain wildcards. In our example, the URL might look like `/ contact/*` in our CSV file.
- `max_matches = 1`: By default, up to 10 entries that match in the lookup file will be added to an event, with the values in each field being added to a multivalue field. In this case, we only want the first match to be applied.

CIDR wildcard lookups

CIDR wildcards look very similar to text-based wildcards but use Classless Inter-Domain Routing (CIDR) rules to match lookup rows against an IP address.

Let's try an example.

Say we have this lookup file:

```
ip_range,network,datacenter
10.1.0.0/16,qa,east
10.2.0.0/16,prod,east
10.128.0.0/16,qa,west
10.129.0.0/16,prod,west
```

It has this corresponding definition in `transforms.conf`:

```
[ip_address_lookup]
filename = ip_address_lookup.csv
match_type = CIDR(ip_range)
max_matches = 1
```

And, there are a few events such as these:

```
src_ip=10.2.1.3 user=mary
src_ip=10.128.88.33 user=bob
src_ip=10.1.35.248 user=bob
```

We could use `lookup` to enrich these events as follows:

```
src_ip="*"
| lookup ip_address_lookup ip_range as src_ip
| table src_ip user datacenter network
```

This would match the appropriate IP address and give us a table such as this one:

	src_ip	user	datacenter	network
1	10.2.1.3	mary	east	prod
2	10.128.88.33	bob	west	qa
3	10.1.35.248	bob	east	qa

The query also shows that you could use the same lookup for different fields using the `as` keyword in the `lookup` call.

Using time in lookups

A temporal lookup is used to enrich events based on when the event happened. To accomplish this, we specify the beginning of a time range in the lookup source and then specify a format for this time in our lookup configuration. Using this mechanism, `lookup` values can change over time, even retroactively.

Here is a very simple example to attach a version field based on time. Say we have the following CSV file:

```
sourcetype,version,time
impl_splunk_gen,1.0,2012-09-19 02:56:30 UTC
impl_splunk_gen,1.1,2012-09-22 12:01:45 UTC
impl_splunk_gen,1.2,2012-09-23 18:12:12 UTC
```

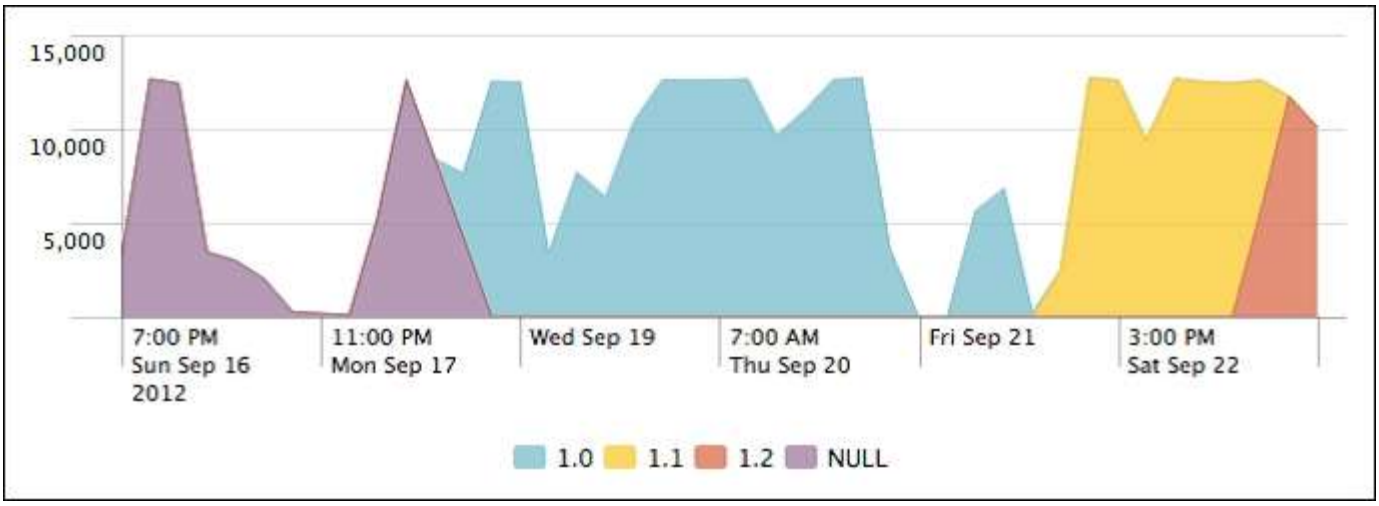
We then use the lookup configuration in `transforms.conf` to specify which field in our lookup will be tested against the time in each event, and what the format of the time field will be:

```
[versions]
filename = versions.csv
time_field = time
time_format = %Y-%m-%d %H:%M:%S %Z
```

With this in place, we can now use our lookup in search, as shown here:

```
sourcetype=impl_splunk_gen error
| lookup versions sourcetype
| timechart count by version
```

This would give us a chart of errors (by version) over time, as shown here:



Other use cases include tracking deployments across environments and tracking activity from disabled accounts.

USING REPORT

Attributes of the format `REPORT-foo` in `props.conf` call stanzas in `transforms.conf` at search time, which means that they cannot affect metadata fields. `EXTRACT` definitions are more convenient to write as they live entirely in a single attribute in `props.conf`, but there are a couple of things that can only be done using a `REPORT` attribute paired with a transform defined in `transforms.conf`.

Creating multivalue fields

Assuming some value might occur multiple times in a given event, an `EXTRACT` definition can only match the first occurrence. For example, say we have the event:

```
2012-08-25T20:18:09 action=send a@b.com c@d.com e@f.com
```

We could pull the first e-mail address using the following extraction:

```
EXTRACT-email = (?i)(?P<email>[a-zA-Z0-9._]+@[a-zA-Z0-9._]+)
```

This would set the field `email` to `a@b.com`. Using a `REPORT` attribute and the transform stanza, we can capture all of the e-mail addresses using the `MV_ADD` attribute. The props stanza would look as follows:

```
REPORT-mvemail = mvemail
```

The `transforms.conf` stanza would then look as follows:

```
[mvemail]
REGEX = (?i)([a-zA-Z0-9._]+@[a-zA-Z0-9._]+)
FORMAT = email::$1
MV_ADD = true
```

The `MV_ADD` attribute also has the effect that, if some other configuration has already created the email field, all values that match will be added to the event.

Creating dynamic fields

Sometimes, it can be useful to dynamically create fields from an event. For instance, say we have an event, such as:

```
2012-08-25T20:18:09 action=send from_335353("a@b.com")
to_223523("c@d.com") cc_39393("e@f.com") cc_39394("g@h.com")
```

It would be nice to pull from, to, and cc as fields, but we may not know all of the possible field names. This stanza in `transforms.conf` would create the fields we want, dynamically:

```
[dynamic_address_fields]
REGEX=\s(\S+)_\S+\("(.*?)"\)
FORMAT = $1::$2
MV_ADD=true
```

While we're at it, let's put the numeric value after the field name into a value:

```
[dynamic_address_ids]
REGEX=\s(\S+)_(\S+)\("
FORMAT = $1::$2
MV_ADD=true
```

This gives us multivalue fields such as the ones in the following screenshot:

action	cc	from	to
send	e@f.com g@h.com 39393 39394	a@b.com 335353	c@d.com 223523

One thing that we cannot do is add extra text to the `FORMAT` attribute. For instance, in the second case, it would be nice to use a `FORMAT` attribute such as this one:

```
FORMAT = $1_id::$2
```

Unfortunately, this will not function as we hope and will instead create the field `id`.

CHAINING TRANSFORMS

As covered before in the *Attributes with class* section, transforms are executed in a particular order. In most cases, this order does not matter, but there are occasions when you might want to chain transforms together, with one transform relying on a field created by a previous transform.

A good example is the source flattening that we used previously in the *Overriding source* section. If this transform happened before our transform in the *Creating a session field from source* section, our session field would always have the value `x`.

Let's reuse two transforms from the previous sections and then create one more transform. We will chain them to pull the first part of session into yet another field. Say we have these transforms:

```
[myapp_session]
SOURCE_KEY = MetaData:Source
REGEX = session_(.*?)\.log
FORMAT = session::$1
WRITE_META = True
[myapp_flatten_source]
SOURCE_KEY = MetaData:Source
DEST_KEY = MetaData:Source
```

```
REGEX = (.*session_).*.log
FORMAT = source::$1x.log
[session_type]
SOURCE_KEY = session
REGEX = (.*?)-
FORMAT = session_type::$1
WRITE_META = True
```

To ensure that these transforms run in order, the simplest thing would be to place them in a single `TRANSFORMS` attribute in `props.conf`, as shown here:

```
[source:*session_*.log]
TRANSFORMS-s =
myapp_session,myapp_flatten_source,session_type
```

We can use the source from our sample event specified inside `tranforms.conf` as follows:

```
source=/logs/myapp.session_foo-jA5MDkyMjEwMTIK.log
```

Walking though the transforms, we have the following terms and their descriptions:

- `myapp_session`: Reading from the metadata field, source, this creates the indexed field session with the `foo-jA5MDkyMjEwMTIK` value
- `myapp_flatten_source`: This resets the metadata field, source, to `/logs/myapp.session_x.log`
- `session_type`: Reading from our newly indexed field, session, this creates the `session_type` field with the value `foo`

This same ordering logic can be applied at search time using the `EXTRACT`and `REPORT` stanzas. This particular case needs to be calculated as indexed fields if we want to search for these values since the values are part of a metadata field.

DROPPING EVENTS

Some events are simply not worth indexing. The hard part is figuring out which ones these are and making very sure you're not wrong. Dropping

too many events can make you blind to real problems at critical times and can introduce more problems than tuning Splunk to deal with the greater volume of data in the first place.

With that warning stated, if you know what events you do not need, the procedure for dropping events is pretty simple. Say we have an event such as this one:

```
2012-02-02 12:24:23 UTC TRACE Database call 1 of 1,000.
[...]
```

I know absolutely that, in this case and for this particular source type, I do not want to index `TRACE` level events.

In `props.conf`, I will create a stanza for my source type, as shown here:

```
[mysourcetype]
TRANSFORMS-droptrace=droptrace
```

Then, I will create the following transform in `transforms.conf`:

```
[droptrace]
REGEX=^\d{4}-\d{2}-\d{2}\s+\d{1,2}:\d{2}:\d{1,2}\s+[A-
Z]+\sTRACE
DEST_KEY=queue
FORMAT=nullQueue
```

Splunk compares nullQueue to `nulldevice`, which (according to the product documentation) tells Splunk not to forward or index the filtered data.

This `REGEX` attribute is purposely as strict as I can make it. It is vital that I do not accidentally drop other events, and it is better for this brittle pattern to start failing and to let through `TRACE` events rather than for it to do the opposite.

fields.conf

We need to add to `fields.conf` any indexed fields we create, or else they will not be searched efficiently, or may even not function at all. For

our examples in the *transforms.conf* section, `fields.conf` would look as follows:

```
[session_type]
INDEXED = true
[session]
INDEXED = true
[host_owner]
INDEXED = true
[host_type]
INDEXED = true
[slow_request]
INDEXED = true
[loglevel]
INDEXED = true
```

These stanzas instruct Splunk not to look in the body of the events for the value being queried. Take, for instance, the following search:

```
host_owner=vlb
```

Without this entry, the actual query would essentially be:

```
vlb | search host_owner=vlb
```

With the expectation that the value `vlb` is in the body of the event, this query simply won't work. Adding the entry to `fields.conf` fixes this. In the case of loglevel, since the value is in the body, the query will work, but it will not take advantage of the indexed field, instead only using it to filter events after finding all events that contain the bare word.

outputs.conf

This configuration controls how Splunk will forward events. In the vastmajority of cases, this configuration exists on Splunk Forwarders, which send their events to Splunk indexers. An example would look similar to this:

```
[tcpout]
defaultGroup = nyc
[tcpout:nyc]
autoLB = true
server = 1.2.3.4:9997,1.2.3.6:9997
```

It is possible to use transforms to route events to different server groups, but it is not commonly used as it introduces a lot of complexity that is generally not needed.

indexes.conf

Put simply, `indexes.conf` determines where data is stored on the disk, how much is kept, and for how long. An index is simply a named directory with a specific structure. Inside this directory structure, there are a few metadata files and subdirectories; the subdirectories are called buckets and actually contain the indexed data.

A simple stanza looks as follows:

```
[implSplunk]
homePath = $SPLUNK_DB/implSplunk/db
coldPath = $SPLUNK_DB/implSplunk/colddb
thawedPath = $SPLUNK_DB/implSplunk/thaweddb
```

Let's walk through these attributes:

- `homePath`: This is the location for recent data.
- `coldPath`: This is the location for older data.
- `thawedPath`: This is a directory where buckets can be restored. It is an unmanaged location. This attribute must be defined, but I for one, have never actually used it.

An aside about the terminology of buckets is probably in order. It is as follows:

- `hot`: This is a bucket that is currently open for writing. It lives in `homePath`.

- `warm`: This is a bucket that was created recently but is no longer open for writing. It also lives in `homePath`.
- `cold`: This is an older bucket that has been moved to `coldPath`. It is moved when `maxWarmDBCount` has been exceeded.
- `frozen`: For most installations, this simply means deleted. For customers who want to archive buckets, `coldToFrozenScript` or `coldToFrozenDir` can be specified to save buckets.
- `thawed`: A thawed bucket is a frozen bucket that has been brought back. It is special in that it is not managed, and it is not included in all time queries. When using `coldToFrozenDir`, only the raw data is typically kept, so Splunk rebuild will need to be used to make the bucket searchable again.

How long data stays in an index is controlled by these attributes:

- `frozenTimePeriodInSecs`: This setting dictates the oldest data to keep in an index. A bucket will be removed when its newest event is older than this value. The default value is approximately 6 years.
- `maxTotalDataSizeMB`: This setting dictates how large an index can be. The total space used across all hot, warm, and cold buckets will not exceed this value. The oldest bucket is always frozen first. The default value is 500 gigabytes. It is generally a good idea to set both of these attributes. `frozenTimePeriodInSecs` should match what users expect. `maxTotalDataSizeMB` should protect your system from running out of disk space.

Less commonly used attributes include:

- `coldToFrozenDir`: If specified, buckets will be moved to this directory instead of being deleted. This directory is not managed by Splunk, so it is up to the administrator to make sure that the disk does not fill up.
- `maxHotBuckets`: A bucket represents a slice of time and will ideally span as small a slice of time as is practical. I would never set this value to less than 3, but ideally, it should be set to 10.
- `maxDataSize`: This is the maximum size for an individual bucket. The default value is set by the processor type and is generally acceptable. The larger a bucket, the fewer the buckets to be

opened to complete a search, but the more the disk space needed before a bucket can be frozen. The default is auto, which will never top 750 MB. The setting `auto_high_volume`, which equals 1 GB on 32-bit systems and 10 GB on 64-bit systems, should be used for indexes that receive more than 10 GB a day.

We will discuss sizing multiple indexes in Chapter 12, *Advanced Deployments*.

authorize.conf

This configuration stores definitions of capabilities and roles. These settings affect search functions and the web interface. They are generally managed through the interface at **Manager** | **Access controls**, but a quick look at the configuration itself may be useful.

A role stanza looks as follows:

```
[role_power]
importRoles = user
schedule_search = enabled
rtsearch = enabled
srchIndexesAllowed = *
srchIndexesDefault = main
srchDiskQuota = 500
srchJobsQuota = 10
rtSrchJobsQuota = 20
```

Let's walk through these settings:

- `importRoles`: This is a list of roles to import capabilities from. The set of capabilities will be the merging of capabilities from imported roles and added capabilities.
- `schedule_search` and `rtsearch`: These are two capabilities enabled for the role power that were not necessarily enabled for the imported roles.
- `srchIndexesAllowed`: This determines which indexes this role is allowed to search. In this case, all are allowed.

- `srchIndexesDefault`: This determines the indexes to search by default. This setting also affects the data shown in **Search** | **Summary**. If you have installed the `ImplementingSplunkDataGenerator` app, you will see the `impl_splunk_*` source types on this page even though this data is actually stored in the `implsplunk` index.
- `srchDiskQuota`: Whenever a search is run, the results are stored on the disk until they expire. The expiration can be set explicitly when creating a saved search, but the expiration is automatically set for interactive searches. Users can delete old results from the **Jobs** view.
- `srchJobsQuota`: Each user is limited to a certain number of concurrently running searches. The default is three. Users with the power role are allowed 10, while those with the admin role are allowed 50.
- `rtSrchJobsQuota`: Similarly, this is the maximum number of concurrently running real-time searches. The default is six.

savedsearches.conf

This configuration contains saved searches and is rarely modified by hand.

times.conf

This configuration holds definitions for time ranges that appear in the time picker.

commands.conf

This configuration specifies commands provided by an app. We will use this in Chapter 13, *Extending Splunk*.

web.conf

The main settings changed in this file are the port for the web server, the SSL certificates, and whether to start the web server at all.

User interface resources

Most Splunk apps consist mainly of resources for the web application. The app layout for these resources is completely different from all other configurations.

Views and navigation

Like `.conf` files, view and navigation documents take precedence in the following order:

- `$SPLUNK_HOME/etc/users/$username/$appname/local`: When a new dashboard is created, it lands here. It will remain here until the permissions are changed to **App** or **Global**.
- `$SPLUNK_HOME/etc/apps/$appname/local`: Once a document is shared, it will be moved to this directory.
- `$SPLUNK_HOME/etc/apps/$appname/default`: Documents can only be placed here manually. You should do this if you are going to share an app. Unlike `.conf` files, these documents do not merge.

Within each of these directories, views and navigation end up under the directories `data/ui/views` and `data/ui/nav`, respectively. So, given a view `foo`, for the user `bob`, in the app `app1`, the initial location for the document will be as follows:

```
$SPLUNK_HOME/etc/users/bob/app1/local/data/ui/views/foo.xml
```

Once the document is shared, it will be moved to the following location:

```
$SPLUNK_HOME/etc/apps/app1/local/data/ui/views/foo.xml
```

Navigation follows the same structure, but the only navigation document that is ever used is called `default.xml`, for instance:

```
$SPLUNK_HOME/etc/apps/app1/local/data/ui/nav/default.xml
```

You can edit these files directly on the disk instead of through the web interface, but Splunk will probably not realize the changes without a restart—unless you use a little trick. To reload changes to views or

navigation made directly on the disk, load the URL `http://mysplunkserver:8000/debug/refresh`, replacing `mysplunkserver` appropriately. If all else fails, restart Splunk.

Appserver resources

Outside of views and navigation, there are a number of resources that the web application will use. For instance, applications and dashboards can reference CSS and images, as we did in Chapter 8, *Working with Apps*. These resources are stored under `$SPLUNK_HOME/etc/apps/$appname/appserver/`. There are a few directories that appear under this directory, as follows:

- `static`: Any static files that you would like to use in your application are stored here. There are a few magic documents that Splunk itself will use, for instance, `appIcon.png`, `screenshot.png`, `application.css`, and `application.js`. Other files can be referenced using includes or templates. See the *Using ServerSideInclude in a complex dashboard*section in Chapter 8, *Working with Apps*, for an example of referencing includes and static images.
- `event_renderers`: Event renderers allow you to run special display code for specific event types. We will write an event renderer in Chapter 13, *Extending Splunk*.
- `templates`: It is possible to create special templates using the *mako*template language. It is not commonly done.
- `modules`: This is where new modules that are provided by apps are stored. Examples of this include the Google Maps and Sideview Utils modules. See http://dev.splunk.com for more information about building your own modules or use existing modules as an example.

Metadata

Object permissions are stored in files located at $`SPLUNK_HOME/etc/apps/$appname/metadata/`. The two possible files are `default.meta` and `local.meta`.

These files have certain properties:

- They are only relevant to the resources in the app where they are contained
- They do merge, with entries in `local.meta` taking precedence
- They are generally controlled by the admin interface
- They can contain rules that affect all configurations of a particular type, but this entry must be made manually

In the absence of these files, resources are limited to the current app.

Let's look at `default.meta` for `is_app_one`, as created by Splunk:

```
# Application-level permissions
[]
access = read : [ * ], write : [ admin, power ]
### EVENT TYPES
[eventtypes]
export = system
### PROPS
[props]
export = system
### TRANSFORMS
[transforms]
export = system
### LOOKUPS
[lookups]
Chapter 10
[ 329 ]
export = system
### VIEWSTATES: even normal users should be able to create
shared
viewstates
[viewstates]
access = read : [ * ], write : [ * ]
export = system
```

Walking through this snippet, we have the following terms and their descriptions:

- The `[]` stanza states that all users should be able to read everything in this app but that only users with the admin or power roles should be able to write to this app.
- The `[eventtypes]`, `[props]`, `[transforms]`, and `[lookups]` states say that all configurations of each type in this app should be shared by all users in all apps, by default. `export=system` is equivalent to `Global` in the user interface.
- The `[viewstates]` stanza gives all users the right to share `viewstates` globally. A viewstate contains information about dashboard settings made through the web application, for instance, chart settings. Without this, chart settings applied to a dashboard or saved search would not be available.

Looking at `local.meta`, we see settings produced by the web application for the configurations we created through the web application.

```
[indexes/summary_impl_splunk]
access = read : [ * ], write : [ admin, power ]
[views/errors]
access = read : [ * ], write : [ admin, power ]
export = system
owner = admin
version = 4.3
modtime = 1339296668.151105000
[savedsearches/top%20user%20errors%20pie%20chart]
export = none
owner = admin
version = 4.3
modtime = 1338420710.720786000
[viewstates/flashtimeline%3Ah2v14xkb]
owner = nobody
version = 4.3
modtime = 1338420715.753642000
[props/impl_splunk_web/LOOKUP-web_section]
```

```
access = read : [ * ]
export = none
owner = admin
version = 4.3
modtime = 1346013505.279379000
```

Hopefully, you get the idea. The web application will make very specific entries for each object created. When distributing an application, it is generally easier to make blanket permissions in `metadata/default.meta` as appropriate for the resources in your application.

For an application that simply provides dashboards, no metadata at all will be needed as the default for all resources (apps) will be acceptable. If your application provides resources to be used by other applications, for instance, lookups or extracts, your `default.meta` file might look like this:

```
### PROPS
[props]
export = system
### TRANSFORMS
[transforms]
export = system
### LOOKUPS
[lookups]
export = system
```

This states that everything in your `props.conf` and `transforms.conf` files, and all lookup definitions, are merged into the logical configuration of every search.