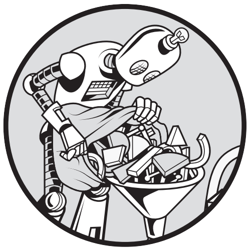
4 ENDPOINT AND NETWORK DATA



All effective monitoring and security infrastructures rely on the ability to collect data from diverse sources across an organization. We'll begin our discussion of log collection with Elastic Filebeat, which can extract dozens of log types from endpoints and the network.

You'll discover how to harvest local logfiles and listen to the network for incoming data. Then, you'll use Filebeat's modules and processors to convert data to the Elastic Common Schema (ECS) naming convention, extract relevant fields from a filestream, apply tags to events to aid later analysis, and read custom logs that Filebeat's modules don't support.

Once you've collected logs from endpoints, you'll likely want to transmit them to another tool. We'll explore ways of outputting data to tools covered in later chapters, including Kafka, Redis, Logstash, and text files.

Collecting Logs with Filebeat

You can install Filebeat on any device running Linux, Windows, or macOS to collect data from it. Filebeat reads logs on the host machine or from the network; converts them into JSON; uses its internal processors to add, modify, or delete the content of events; and then sends the logs downstream to Logstash, Kafka, Elasticsearch, or other tools. Its many modules allow it to interpret events from various vendor technologies, including commercial tools like Cisco and Palo Alto Networks and open source tools like Zeek.

Filebeat can also rename log values using the ECS naming convention, which is helpful because different tools use distinct names to label the same information. For example, they might include destination IP address fields called `dst_ip` or `dip`; ECS would convert both names into `destination.ip`, making searching for data from multiple sources much easier.

The tool provides several advantages over other log collectors we'll discuss later in this book, such as Elastic Agent and Rsyslog, covered in Chapters 6 and 7, respectively. Unlike Rsyslog, it automatically converts many log varieties into JSON and ECS by default, although Filebeat needs Elasticsearch or Logstash for complex ECS conversions. Filebeat can also prune and privatize data, and it requires relatively few configuration settings to do so. Additionally, Filebeat can output logs to more destinations than Elastic Agent can, including Redis and text files, and offers a lower barrier to entry than Elastic Agent.

Another advantage of Filebeat over other traditional logfile collectors is that it provides *backpressure support* when outputting events directly to Logstash and Elasticsearch. This means that if Logstash experiences a spike in data that it needs time to process, it can notify Filebeat to slow things down so it can catch up. Alternatives to Filebeat would continue to firehose data, causing queue backups or potential data loss.

Filebeat could even replace the need for Logstash in environments running technologies supported by Filebeat and Winlogbeat that consume JSON downstream. Written in Golang, Filebeat is more lightweight than Logstash, which is written in Java and Ruby. However, Filebeat supports only one output destination at a time, whereas Logstash can send data to any of several destinations by using conditional statements. Logstash also has a much more robust filtering capability than Filebeat, but if an organization needs only the features Filebeat can provide, Filebeat is the better option.

One downside to Filebeat is that it acts as a stand-alone agent rather than a centrally managed one. Operators must manually upgrade and reconfigure Filebeat, whereas they can modify and upgrade Elastic Agent from a central server. We'll explore Elastic Agent in Chapter 6 and compare its functions to those of Filebeat. We'll also use Ansible in Chapters 11 and 12 to demonstrate deploying Filebeat on multiple servers.

Another challenge is that Filebeat requires a unique socket for each enabled module when receiving events from other tools. For instance, it might receive some traffic on port 514/TCP, with the Cisco module listening on 5514/TCP for router and switch events and the Palo Alto module listening on 55514/TCP. If not managed properly, this can certainly lead to configuration clutter and a multitude of firewall holes across the network.

Installation

The majority of this chapter will focus on reading logs from Linux hosts. You can download Filebeat from the Elastic website in several file formats, including DEB, RPM, Apt, Yum, Windows, and macOS and Linux tarballs. Let's use the tarball package, which makes it easier to start and stop Filebeat frequently. The configurations you'll make for the tarball package are compatible with Apt and Yum installations.

On your Linux host, navigate to your user's home *Downloads* directory and download the *Linux x86_64* tarball from Elastic's Filebeat downloads web page. Replace the version number in the filename as appropriate:

```
$ cd ~/Downloads/
$ wget https://artifacts.elastic.co/downloads/
beats/filebeat/filebeat-X.Y.Z-linux-x86_64.tar.gz
$ tar xvzf filebeat-X.Y.Z-linux-x86_64.tar.gz
$ cd filebeat-X.Y.Z-linux-x86_64
```

We'll cover Windows installation and configuration instructions in "Filebeat for Windows" on page 78.

Enabling TLS

To protect our data, we need to create a TLS certificate for Filebeat so we can encrypt the data it sends. In Chapter 2, we created root and intermediate CAs. Let's use the intermediate CA to sign a new flex certificate request.

Creating a Configuration File

We'll use OpenSSL to create a new TLS flex certificate. Store this file, *openssl-flex-filebeat.local.cnf*, in the TLS configuration directory (*~/tls/configs*) you created in Chapter 2. Open your preferred text editor and enter the following:

```
###############################################################
[ req ]
prompt             = no
default_bits       = 4096
default_md         = sha512
❶ default_keyfile    = tls/keys/filebeat.local.flex.key.pem
distinguished_name = flex_distinguished_name
req_extensions     = flex_cert
###############################################################
❷ [ flex_distinguished_name ]
countryName                = US
stateOrProvinceName        = MO
```

```
  localityName               = St. Louis
  organizationName           = Business, Inc.
  organizationalUnitName     = Information Technology
❸ commonName                 = Filebeat Flex
  emailAddress               = none@localhost
  ###############################################################
  [ flex_cert ]
❹ nsComment               = OpenSSL Certificate for Clients or Servers
❺ subjectAltName          = @alternate_names
  ###############################################################
  [ alternate_names ]
❻ DNS.1  = filebeat
❼ DNS.2  = filebeat.local
```

OpenSSL will use this configuration to create a new private key ❶ in the *tls/keys* directory. Replace the location information in `flex_distinguished_name` with your own ❷ (unless you, like me, come from the land of toasted ravioli and pork steaks). We'll use the `commonName` ❸ and `nsComment` ❹ statements to indicate that this is a client certificate for initiating and receiving encrypted connections, although please note commercial certificate providers require an FQDN as the `commonName` value for backward browser compatibility. The subject alternative names ❺ include the base name ❻ and fully qualified domain name ❼ of the server running Filebeat.

If you haven't already done so, be sure to add your Filebeat IP address, hostname, and Logstash information to your DNS server or to */etc/hosts* so that the hostnames resolve to the proper destinations. These entries should look like the following; substitute your IP addresses as necessary:

```
--snip--
192.168.8.133  filebeat
192.168.8.133  filebeat.local
192.168.8.133  logstash
192.168.8.133  logstash.local
--snip--
```

Without the ability to resolve hostnames, Filebeat won't be able to establish TLS connections, since the certificate's subject alternative name doesn't include IP addresses. Next, we'll generate the private key and signing request.

Generating Certificate Signing Requests

Let's create the key pair and a certificate signing request. We'll also include the key passphrase on the command line to skip being prompted for it, but you shouldn't type passwords on the command line in production:

```
$ openssl req -config tls/configs/openssl-flex-filebeat.local.cnf -new
-out tls/csr/filebeat.local.flex.csr -outform PEM -passout pass:abcd1234
```

Next, let's use the intermediate CA from Chapter 2 to create the signed certificate. Pass in the private key's passphrase and specify the CA's `signing_policy` and `flex_cert` extensions:

```
$ openssl ca -batch -notext -config tls/configs/openssl-intermediateca.cnf -passin
pass:abcd1234 -policy signing_policy -extensions flex_cert -out
tls/certs/filebeat.local.flex.cert.pem -infiles tls/csr/filebeat.local.flex.csr
```

You should now have the signed certificate, *filebeat.local.flex.cert.pem*. View it to check its extensions:

```
$ openssl x509 -in tls/certs/filebeat.local.flex.cert.pem -text -noout
--snip--
    X509v3 Extended Key Usage:
    ❶ TLS Web Client Authentication, TLS Web Server Authentication
    X509v3 Subject Alternative Name:
    ❷ DNS:filebeat, DNS:filebeat.local
--snip--
```

The certificate indicates that it's for client and server connections ❶, and the DNS entries ❷ show both the base name and the fully qualified domain name (FQDN) we specified. Check that the certificate can be authenticated using the CA chain file:

```
$ openssl verify -CAfile tls/certs/ca-chain.cert.pem tls/certs/filebeat.local.flex.cert.pem
tls/certs/filebeat.local.flex.cert.pem: OK
```

As we have a valid certificate with both the *clientAuth* and *serverAuth* extensions, we can begin configuring Filebeat.

Configuration

Filebeat uses the *filebeat.yml* configuration file to centralize core settings. This is where you'll configure inputs, transformations not found in modules, and outputs, along with other settings.

The default outputs in *filebeat.yml* immediately try to connect to local Elasticsearch and Kibana instances. So, instead of using the file as provided, we'll slim it down and then fill it in over the course of this chapter to better understand how it works.

For the purposes of this example, let's say we used Project Discovery's network reconnaissance tools Subfinder and Httpx to acquire subdomain information about *https://owasp.org*, resulting in JSON output files. We'll edit *filebeat.yml* to provide a path to these files and an ID for troubleshooting, then explicitly enable the input.

First, copy the original file into a backup:

```
$ cp filebeat.yml filebeat.yml.original
```

Then, edit the filestream input statement in the file to match the following slimmed-down version, substituting your username for *j* as needed:

```
filebeat.inputs:

# Input that reads local files from custom recon tooling
❶ - type: filestream
  ❷ id: recon-logs
    enabled: true
  ❸ paths:
      - /home/j/example-logs/subfinder*.json
      - /home/j/example-logs/httpx*.json

  # Module location
❹ filebeat.config.modules:
    path: ${path.config}/modules.d/*.yml
❺ tags: [ "tags-for-everybody", "you-get-a-tag", "and-you-get-a-tag" ]

❻ output.logstash:
    enabled: true
    hosts: [ "logstash.local:5044" ]
  ❼ ssl.enabled: true
  ❽ ssl.verification_mode: full
  ❾ ssl.certificate: "/home/j/tls/certs/filebeat.local.flex.cert.pem"
    ssl.key: "/home/j/tls/keys/filebeat.local.flex.key.pem"
    ssl.key_passphrase: abcd1234
    ssl.certificate_authorities:
      - /home/j/tls/certs/ca-chain.cert.pem
```

The `filebeat.inputs` section contains one input, `filestream` ❶. Adding an `id` statement ❷ is a best practice for troubleshooting missing or unexpectedly high-volume logs, as it allows us to easily identify a problematic configuration block.

The `paths` statement ❸ provides the locations of the files containing the input. We've specified newline-delimited JSON (NDJSON) files that use wildcards to match any file beginning with *subfinder* or *httpx* and ending in the *.json* extension. This should let us read logs generated by the reconnaissance tools Subfinder and Httpx.

The `filebeat.config.modules` section ❹ tells Filebeat where to find its processing modules. We've left this section unchanged because these modules are part of the downloaded tarball. We add custom tags ❺ to every event to categorize them for later analysis. Finally, `output.logstash` ❻ describes where to send the data it outputs. Be sure that the IP address in the output statement matches that of the Logstash instance you configured in Chapter 2.

We enable TLS ❼ and set the verification mode to `full` ❽, requiring Filebeat to verify the downstream server's certificate against the CA and match the domain name against the one listed in the downstream certificate. We also specify the signed client certificate ❾, key file and passphrase, and our CA chain.

Let's copy this new *filebeat.yml* to its own backup, which we'll save in our remote Git repository at the end of this chapter:

```
$ cp filebeat.yml filebeat.yml.backup
```

Next, open a new terminal in which to run Logstash so it can receive data from Filebeat. Load the *beats-mtls.conf* Logstash pipeline we created in Chapter 2 in another terminal and keep the Logstash instance running to monitor the output on screen:

```
$ bin/logstash -f conf.d/beats-mtls.conf --config.reload.automatic
```

You can now view any output sent to Logstash on the screen in real time. But while this is ideal for testing purposes, it probably won't be useful in production unless you can read at *Matrix* speeds. Let's explore input and output types and use Logstash to view the data processed by Filebeat.

Input Sources

Filebeat extracts data by reading files on the host, listening to the network, or reaching out to external systems like Redis or Kafka. We configure these input sources in *filebeat.yml*, as shown in the previous section, or inside the individual module configuration files. Multiple inputs can run at once (though Filebeat can output data to only one location).

Reading Local Files

Filebeat will use the filestream inputs you configured in *filebeat.yml* to vacuum up local logs and send them downstream. If you're migrating to Filebeat from tools like Rsyslog and Syslog-ng, this process should look familiar; Filebeat reads the logs, parses them loosely into JSON and ECS when able, and ships them off.

The tool also keeps track of where it left off in each file using a registry of *offsets*, or byte positions. For the sake of testing, let's write a short bash script in the Filebeat directory, named *clean-and-reload-filebeat.sh*, that will delete this registry so we can reread the same logs multiple times, which will make it easier to generate test data. Open your editor of choice:

```
$ vim clean-and-reload-filebeat.sh
```

Add these lines and then save and close the file:

```
#!/bin/bash
rm -rf data/
./filebeat -e
```

Note that if you installed Filebeat using the DEB or RPM packages, you'll find your */data* directory in */var/lib/filebeat*, and you'll need `sudo` privileges to delete it.

The script will remove the cached positions and then restart Filebeat. Make the script executable:

```
$ chmod +x clean-and-reload-filebeat.sh
```

Invoke Filebeat for the first time by running the following command in its own terminal window. It should read the NDJSON files we listed as inputs during configuration:

```
$ ./filebeat -e
```

The `-e` option sends all output to the standard error stream. Also note the leading period before the slash.

NOTE *You might find it helpful to create a split terminal window using tools like Tmux or Terminator. By placing Filebeat on the left side of the screen and Logstash on the right side, you can watch both tools in action without navigating between terminals.*

As a result of running the command, Filebeat will process the logs configured earlier.

Applying Parsers and New Fields

Parsers interpret or translate an input format into another format, typically JSON unless otherwise specified. This may include reading a string containing JSON data and converting it into usable key-value pairs. Note that the NDJSON we read as input in the previous section appeared in the Logstash terminal as a giant blob in the `message` field:

```
{
           "log" => {
          "file" => {
          "path" => "/home/j/example-logs/httpx_owasp.org.json"
          },
        "offset" => 134706
           },
    "@timestamp" => 2040-04-28T20:29:57.884Z,
      "@version" => "1",
       "message" => "{\"timestamp\":\"2040-04-27T15:23:03.636525891-
                    05:00\",\"csp\":{\"domains\":[\"https://
--snip--
```

To properly format these key-value fields, let's add a parser inside of the *filebeat.yml* file's filestream statement. In this case, we'll use a parser called `ndjson`:

```
filebeat.inputs:
- type: filestream
  id: recon-logs
  enabled: true
  paths:
    - /home/j/example-logs/subfinder*.json
    - /home/j/example-logs/httpx*.json
❶ parsers:
  ❷ - ndjson:
    ❸ target: "processed"
    ❹ add_error_key: true
  fields_under_root: true
❺ fields:
    threat.tactic.name: "Reconnaissance"
    threat.tactic.id: "TA0043"
    threat.technique.name: "Gather Victim Network Information"
    threat.technique.id: "T1590"
```

We nest the parser under the `type: filestream` section ❶. We specify that it will read NDJSON from the event ❷, nest the new JSON under the `processed` key ❸, and add an error message ❹ if something unexpected happens.

We also add ECS-aligned custom fields ❺ to describe MITRE ATT&CK framework values present in these logfiles. MITRE ATT&CK is a popular system used to describe adversary behavior during threat modeling and analysis. You could use these fields downstream to categorize these logs as network reconnaissance activity.

Filebeat expects a dotted field name representing nested JSON objects, meaning a field named `abc.123.xyz` would have a top-level key of `abc` and a value of `123`, which itself is a key containing the final string value `xyz`.

Now that we can parse NDJSON into a nested field of its own, run the handy Filebeat cleaner script to wipe out the registry and reingest the files:

```
$ ./clean-and-reload-filebeat.sh
```

Much better! Filebeat can now parse the JSON fields, so other tools can reference them later. We'll also be able to use the custom threat fields for filtering or analysis:

```
{
  ❶ "@timestamp" => 2040-05-08T09:00:47.133Z,
           "log" => {
          "file" => {
          "path" => "/home/j/example-logs/httpx_owasp.org.json"
        },
        "offset" => 134706
        },
        "threat" => {
            "technique" => {
                   "id" => "T1590",
                 "name" => "Gather Victim Network Information"
            },
            "tactic" => {
                "id" => "TA0043",
              "name" => "Reconnaissance"
          }
      }
  },
❷ "processed" => {
              "timestamp" => "2040-04-27T15:23:03.636525891-05:00",
                 "method" => "GET",
         "content_length" => 74814,
              "webserver" => "Server",
                 "scheme" => "https",
            "status_code" => 200,
                   "host" => "...
--snip--
```

ECS uses `@timestamp` to label the time an event occurred, although additional time-related fields, such as `event.start` and `event.end`, can provide more granular time information.

Notice a problem in this example: The `@timestamp` value ❶ shows the time at which the logs were read by Filebeat, instead of the value in `processed.timestamp` ❷ representing the time of the actual event. The value is notably wrapped in quotes when displayed, as Filebeat treats it as a string object, not a date object. This is because we didn't tell Filebeat to use `processed.timestamp`. We'll discuss modifying this behavior in the "Processors" section on page 65.

Let's add another filestream input to ingest events from local files in */var/log*:

```
- type: filestream
❶ id: local-syslog-files
  enabled: true
  paths:
    - /var/log/*.log
❷ exclude_lines: ['.*UFW.*']
  parsers:
  ❸ - syslog:
      format: auto
      add_error_key: true
```

We set a new input ID ❶ and exclude lines ❷ that contain `UFW`, which stands for Uncomplicated Firewall, the default firewall on Ubuntu. We also use the `syslog` parser ❸ with the `auto` format to detect both RFC 3164 and 5424 formats. Note that Filebeat also has a `system` module for reading files in */var/log*.

For the remainder of this chapter, whenever we enable a new input, we'll disable the previous one. This will allow us to become familiar with the outputs from each new input without getting lost in the flow from other inputs.

Listening to the Network

Filebeat has a variety of network inputs that allow it to receive data from remote systems over TCP, User Datagram Protocol (UDP), and Unix sockets. It can also act as an HTTP application programming interface endpoint that accepts requests and sends HTTP requests at specified intervals.

To listen for incoming syslog messages over the network using a plaintext (unencrypted) listener, add a new input type to *filebeat.yml* below the filestream section:

```
❶ - type: syslog
    id: syslog-tcp-5514
    enabled: true
    # Can also specify rfc3164 and rfc5424
  ❷ format: auto
  ❸ tags: [ "forwarded" ]
  ❹ protocol.tcp:
    ❺ host: "localhost:5514"
    exclude_lines: ['.*UFW.*']
```

The `syslog` input type ❶ used with the `auto` format ❷ will detect both RFC 3164 and RFC 5424 syslog messages. Add the tag `forwarded` ❸ to indicate that the messages originated elsewhere. Filebeat will listen for TCP ❹ connections on localhost port 5514 ❺ because ports below 1024 (including the standard syslog port 514/TCP) require elevated permissions. Exclude all lines containing the string `UFW`, as it's best to track firewall logs separately due to the volume of traffic they generate.

We'll configure Rsyslog in Chapter 7, but if you already have it running, you can add the following line to a new configuration at */etc/rsyslog.d/send-to-filebeat.conf* and then restart Rsyslog to output logs to Filebeat:

```
*.*    @@localhost:5514
```

The double at sign (`@@`) represents TCP. (To represent UDP, use a single at sign.) Now run Filebeat again, and notice the `forwarded` tag present in the streaming data:

```
$ ./filebeat -e
--snip--
       "message" => "pam_unix(sudo:session): session closed for user root",
          "tags" => [
        [0] "tags-for-everybody",
        [1] "you-get-a-tag",
        [2] "and-you-get-a-tag",
        [3] "forwarded",
        [4] "beats_input_codec_plain_applied"
    ],
           "log" => {
        "source" => {
       "address" => "127.0.0.1:46474"
```

```
        }
    },
      "@version" => "1",
       "process" => {
       "program" => "sudo"
    },
        "syslog" => {
        "facility_label" => "security/authorization",
              "facility" => 10,
              "priority" => 86,
        "severity_label" => "Informational"
--snip--
```

Filebeat is now parsing events from a remote system and forwarding them.

You may have noticed that the process name `sudo` is now nested under `process.program` and that the filestream input puts it under `syslog.appname`. This is worth paying attention to, as you might want to copy the name into the preferred `process.name` field using Logstash or a Filebeat processor. Most modules in *filebeat/modules.d* that extract process information will correctly place its name in `process.name`, but inputs defined in *filebeat.yml* don't always align all fields to ECS.

Connecting to External Systems

Filebeat can reach out to external systems to pull data from them, which becomes useful when working with systems that cannot initiate the sending of data, as well as systems that need a middleman to cross network boundaries. For example, event stores such as Kafka don't send data to other systems directly; instead, producers send data into Kafka and consumers pull data out. Filebeat performs both functions, as you'll see shortly. We'll discuss Kafka in much greater depth in Chapter 10.

Filebeat's Kafka input needs a few settings for the connection to succeed. Add the following to *filebeat.yml*:

```
- type: kafka
  enabled: true
❶ hosts:
    - kafka01:9093
    - kafka02:9093
❷ topics: [ "filebeat" ]
❸ group_id: "filebeat"
❹ tags: [ "from-kafka" ]
  parsers:
  ❺ - ndjson:
      message_key: "message"
      target: "processed"
      overwrite_keys: true
      add_error_key: true
```

The `hosts` array ❶ contains two Kafka *brokers*, which are server nodes that process messages. Curiously, these might not represent the actual destinations to which Filebeat will send data. Instead, they're often hosts called *bootstrap servers*, which return metadata about the nodes inside of the Kafka cluster, describing where Filebeat will ultimately send data. Filebeat will read this metadata and then send data to the actual brokers.

Next, we specify the *topic*, which is a stream of events we want to subscribe to so we can read data from them. We define topics as an array of strings ❷. In this example, we include a single topic, named `filebeat`, containing data previously pushed into the cluster using the Kafka output we'll explore later in this chapter.

We set the consumer group using a group ID ❸. A *consumer group* is a collection of one or more subscribers that read the data in a topic as a single unit. This allows Kafka to load-balance the data it sends out, increasing total throughput. More importantly, it can avoid sending the same message more than once to a given group.

In this example, we use the group ID `filebeat`, which specifies a unique set of consumers that should receive data from a topic. If three Filebeat instances belong to one consumer group, Kafka will load-balance outgoing data by distributing it evenly among the instances. In this case, however, the single Filebeat will receive all the data from the topic, as it's the only group member.

We also specify a parser ❺ to read the `message` field and output the JSON structure into the `processed` field, as before. Note that the `from-kafka` tag ❹ exists only at the top level of the JSON structure and not inside of the nested `processed` fields we want to use downstream.

When Filebeat is running, the terminal should show output like the following:

```
--snip--
    "processed" => {
            "tags" => [
            [0] "tags-for-everybody",
            [1] "you-get-a-tag",
            [2] "and-you-get-a-tag",
            [3] "forwarded"
        ],
           "process" => {
           "program" => "sudo"
        },
           "message" => "pam_unix(sudo:session): session closed for user root",
            "syslog" => {
--snip--
```

You can see the tool sending the log to Kafka and then Logstash retrieving it and displaying it on the screen, including the `message` field containing the *sudo* event. Next, let's a new tag to the `processed` field.

Processors

Like parsers, processors modify incoming events, yet they offer more horsepower than parsers. For example, the processors enabled by default in *filebeat.yml* add host- or cloud-related data, such as machine type or instance ID, to events. Other processors might add details about the network direction, decode base64 strings, perform DNS lookups, or parse XML. A list of processors is available on Elastic's website at *https://www.elastic.co/guide/en/beats/filebeat/current/filtering-and-enhancing-data.html.*

Let's use a small `add_tags` processor to demonstrate adding a new array field inside a nested structure. This might come in handy if you eventually prune similar events down to just the `processed` fields, so you don't lose the array. The processor might look like this:

```
❶ processors:
  ❷ - add_tags:
    ❸ tags: [ "from-kafka" ]
    ❹ target: "processed.tags"
```

Under the `processors` level ❶, we use the `add_tags` processor ❷ to include a tag that indicates that the log came from Kafka ❸. We then nest the new field inside of `processed.tags` ❹.

Earlier in this chapter, we discussed another opportunity to use a processor: modifying the `processed.timestamp` field so that Filebeat can use it as an event's timestamp. To create such a processor, however, we'll have to write conditional statements.

Controlling Processors with Conditionals

Conditional statements allow us to specify the conditions in which to run processors. For example, we can use them to check whether a field contains a certain value (or explicitly doesn't contain it) and whether multiple fields are equal; then we can run a processor only if the condition is met.

We nest conditional statements under a processor's `when` statement and define them using operators such as `and`, `or`, and `not`. Table 4-1 lists Filebeat's conditional operators. You can use a single conditional statement or nest the statements to support complex comparisons.

Table 4-1: Filebeat's Conditional Operators

Conditional operator	Purpose
`equals`	Compares strings or integer equality
`and`	Meets all conditions in a list
`or`	Meets one or more conditions in a list
`contains`	Checks for a match in a string
`regexp`	Uses regular expression statements
`network`	Validates IP address membership in a network
`range`	Checks a number in upper/lower bounds
`has_fields`	Checks whether a field exists
`not`	Negates the following comparison

Let's use these conditionals to add processors that alter the `processed.timestamp` field from the custom JSON logs. Though we can read the field, we need to change its formatting so that Filebeat can understand its layout and use it as the event's timestamp:

```
2040-04-27T15:23:03.636525891-05:00
```

We must do two things to this value. First, we'll truncate the timestamp to millisecond resolution (the most granular resolution Logstash can handle). Then, we'll overwrite the top-level `@timestamp` field with this truncated timestamp.

We'll use the `script` processor to run custom JavaScript code that extracts the date, time, and time zone. (The `script` processor uses an implementation of ECMAScript written purely in Go.) Then we'll use the `timestamp` processor to overwrite the `@timestamp` field at the top of the JSON structure, making the event ready for database entry.

Add these lines to *filebeat.yml* inside of the `processor` block:

```
- script:
    lang: javascript
    # Example:
    # 2040-04-27T22:37:12.463504006-05:00
    #          | 10        |23    |29
    source: >
      function process(event) {
      ❶ var t = event.Get("processed.timestamp")
      ❷ var front = t.slice(0, 23)
        var back = t.slice(29)
      ❸ var combined = front+back
        event.Put("processed.timestamp_fixed", combined)
      }
  ❹ when:
      has_fields: [ "processed.timestamp" ]
```

First, we fetch the timestamp from the nested data ❶ and save it as the variable `t`. Next, we create slices of it up to and including the 23rd character ❷ (counting from position zero), saving it as `front`, and make a second slice from character 29 onward to capture the time zone offset, saving it as `back`.

Now that we've removed characters 24 through 28, we combine `front` and `back` into a single string named `combined` ❸, which we add to a new field, `process.timestamp_fixed`, for the timestamp processor to pick up. This processor runs only if the `when` conditional is met and the `processed.timestamp` field exists ❹. Note that I've intentionally written this code in a contrived manner to demonstrate running a code block inside of a processor; you might find more efficient ways to perform this example.

The following snippet shows the timestamp value before and after using the script processor to create the new timestamp_fixed field:

```
--snip--
      "processed" => {
      "timestamp" => "2040-04-27T15:23:03.636525891-05:00",
"timestamp_fixed" => "2040-04-27T15:23:03.636-05:00",
--snip--
```

The timestamp processor uses a reference time layout from the underlying Go time package to recognize timestamps:

```
01/02 03:04:05PM '06 -0700'
# Converted to the Filebeat-preferred layout
2006-01-02T15:04:05.999-07:00
```

Using this layout, it can understand the new string we just created with the script processor. It can also test whether the conversion will work and exit if it fails. The following snippet shows the processor in action:

```
- timestamp:
❶ field: "processed.timestamp_fixed"
❷ layouts:
    - '2006-01-02T15:04:05.999-07:00'
❸ test:
    - '2040-04-27T22:37:12.463-05:00'
❹ when:
    has_fields: [ "processed.timestamp_fixed" ]
```

The processor runs on the defined field we created (process.timestamp_fixed) ❶ only when the field exists ❹. The layout statement ❷ forms a template for the reference timestamp, and the unit test ❸ will force the module to exit if it fails during testing with an error message. Now Filebeat will overwrite the timestamp value with the one extracted from the nested JSON. You should see this output after running the *timestamp* processor:

```
--snip--
      "timestamp" => "2040-04-27T15:23:03.636525891-05:00",
"timestamp_fixed" => "2040-04-27T15:23:03.636-05:00",
     ...
     "@timestamp" => 2040-04-27T20:23:03.636Z
--snip--
```

We've updated the event timestamp to properly reflect the nested data, and Filebeat has converted it to the Greenwich Mean Time (GMT) time zone.

Screening Out Data

We can also use processors to screen out data, preventing unnecessary bytes from flowing across the network and eating up storage. One way to drop fields we don't need is by using multiple conditional statements:

```
processors:
  - drop_fields:
  ❶ fields: ["ecs", "agent.ephemeral_id"]
    when:
    ❷ or:
        - contains:
          log.file.path: "/home/j/example-logs/"
        - equals:
          input.type: "syslog"
        - contains:
          tags: "from-kafka"
```

We list fields to be dropped in an array ❶ and nest the required conditional logic ❷ to specify when to drop the fields. Here, we use the `equals` statement to match a field example, and we use `contains` to specify a portion of a name, allowing us to match multiple fields or names that might change.

Note that you can't drop the field `event.original` or any metadata fields beginning with `@` using the `drop_fields` processor. One way to remove these fields is to set them to an empty string or to another desired value using Filebeat processors or Logstash filters.

Modules

The true strength of Filebeat lies in its modules. Dozens of modules provide powerful transformation and parsing capabilities, such as reading complex firewall or cloud logs, and many of them contain submodules, or *filesets*, related to multiple technologies by the same vendor. Modules handle their own inputs; some can read files and network sockets, while others connect outbound to an API and request logs. You define processors directly in the module YML file to compartmentalize configurations and add tags where necessary.

NOTE *Many modules use Elasticsearch ingest pipelines to rename fields to ECS. We'll touch on ingest pipelines in Chapter 6 when we discuss Elasticsearch and Elastic Agent integrations.*

To explore Filebeat's modules, let's work with Zeek, a popular network traffic analyzer formerly known as Bro. Zeek provides metadata about happenings on the wire, and Filebeat's Zeek module converts local logs into ECS if Zeek is configured to write JSON data. Here is a Zeek log in raw JSON form:

```
{"ts":1688955554.952633,"uid":"CDYUvz2iep4cP5GKI6","id.orig_h":
"192.168.28.32","id.orig_p":62299,"id.resp_h":"239.255.255.250",
"id.resp_p":1900,"proto":"udp",
--snip--
```

To convert these hard-to-read logs into JSON and ECS, first enable the Zeek module:

```
$ ./filebeat modules enable zeek
```

Next, open the module configuration file, *modules.d/zeek.yml*. Zeek creates individual logs for its dozens of supported protocol analyzers, and Filebeat breaks them into individual statements that we can turn on or off:

```
- module: zeek
  capture_loss:
  ❶ enabled: true
  ❷ var.paths: ["/opt/zeek/spool/zeek/capture_loss.log"]
  connection:
    enabled: true
    var.paths: ["/opt/zeek/spool/zeek/conn.log"]
  dce_rpc:
    enabled: true
    var.paths: ["/opt/zeek/spool/zeek/dce_rpc.log"]
--snip--
```

We toggle each supported log type on or off by setting `enabled` to either `true` or `false` ❶. Also note that the configuration file specifies filename paths inside arrays ❷ in case Filebeat is running on a landing pad or middleman server and needs to read from multiple sources for the same log type.

Zeek requires elevated privileges to listen to the network interface, and it protects its logs accordingly. Add your username to the `zeek` group. The following adds my account, *j*:

```
$ sudo usermod -aG zeek j
```

Log out and back in for the change to take effect. If the change doesn't kick in for some reason, run the command **`groups`** and check the contents of */etc/group* to make sure your username is in the `zeek` group. Try the `newgrp` command if your username isn't there:

```
$ groups
$ grep zeek /etc/group
$ newgrp zeek
```

Now test and start Filebeat again, then watch the converted log populate in the Logstash terminal:

```
$ ./filebeat test config
$ ./filebeat test output
$ ./filebeat -e
```

The module converts the logs into lovely JSON, with most of the key fields converted to ECS:

```
     "source" => {
        "mac" => "11:50:56:c4:00:a2",
      "bytes" => 812,
    "packets" => 4,
       "port" => 62299,
    "address" => "192.168.28.32",
         "ip" => "192.168.28.32"
},
 "@timestamp" => 2040-07-10T02:20:22.191Z,
--snip--
```

Modules typically do a great job of converting fields to ECS, but there are gaps in what Filebeat converts locally versus downstream. For example, some Filebeat modules for technologies like Cisco and Palo Alto push ECS conversions to Elasticsearch ingest pipelines if they're deemed too complex for Filebeat.

We can use processors within each fileset so that the global configurations won't bloat to dozens or hundreds of lines. These module-based processors sit at the same level as the `enabled` and `var.*` statements and will run alongside the processors defined globally in *filebeat.yml*. If needed, we can add tags directly in the fileset using `var.tags`:

```
# modules.d/zeek.yml
❶ connection:
    enabled: true
    var.paths: ["/opt/zeek/spool/zeek/conn.log"]
  ❷ input:
      processors:
        - add_tags:
          tags: ["tag-in-connection-log"]
--snip--
```

Within the `connection` fileset ❶, we add an `input` field ❷ at the same level as the other options, then define a processor within it. As expected, this processor adds a tag to the output:

```
--snip--
       "tags" => [
    [0] "you-get-a-tag",
    [1] "zeek.connection",
    [2] "tag-in-connection-log",
--snip--
```

Here we see the new tag applied to this connection event using the `add_tags` processor, alongside tags you added earlier in the chapter.

Sending Outputs

Filebeat can define only one output at a time, leaving very little wiggle room for modifications after you've established a pipeline. In most cases, Filebeat sends data to Logstash or directly to Elasticsearch, but sending data to message brokers such as Kafka and Redis is common, too.

Publishing to Kafka

Sending output to Kafka works similarly to reading input from it: We rely on topics to determine where and how data gets pushed out.

We previously used consumer groups inside of *filebeat.yml* to define our identity when reading data from Kafka:

```
group_id: "filebeat"
```

For servers sending data to Kafka, set the `client_id` value to the hostname of the server running Filebeat or to some other value that will make troubleshooting easier if things go sideways. (This setting defaults to `beats`, which is unhelpful when you need to troubleshoot more than one server running Filebeat!)

Kafka stores JSON fields in a single string field. Logstash needs to decode the string back into usable JSON fields. Let's add a *conf.d/plain-kafka-consumer.conf* file in the Logstash directory to consume messages and display them on standard output:

```
# plain-kafka-consumer.conf
input {
    kafka {
      ❶ bootstrap_servers => "kafka01:9092,kafka02:9092"
      ❷ group_id => "logstash"
      ❸ topics => [ "filebeat" ]
    }
}

filter {
    # convert data back into JSON
  ❹ json {
        source => "[event][original]"
    }
}

output {
    stdout { codec => rubydebug { metadata => true }}
}
```

Note that we send this data in plaintext; we'll discuss configuring TLS for Kafka in Chapter 10.

The Kafka input for Logstash lists bootstrap servers ❶ used to acquire cluster metadata in a comma-separated string, the consumer group ID ❷, and the topic ❸ Logstash will pull from. The filter ❹ will expand the `event.original` field containing the string pulled from Kafka into the proper JSON structure.

Navigate to the Logstash directory in another terminal and run the Logstash pipeline:

```
$ bin/logstash -f conf.d/plain-kafka-consumer.conf --config.reload.automatic
```

Inside of *filebeat.yml*, define the new Kafka output and then restart Filebeat:

```
output.kafka:
  enabled: true
  hosts: [ "kafka01:9092", "kafka02:9092" ]
  topic: "filebeat"
❶ client_id: "my-awesome-server-running-filebeat"
❷ headers:
    - key: "category"
      value: "remoteaccess"
      when:
        equals:
          event.dataset: "zeek.ssh"
    - key: "category"
      value: "web"
      when:
        equals:
          event.dataset: "zeek.tls"
```

Like in the Kafka input, we define the bootstrap servers in an array, the topic to write to, the unique client identifier of the server running Filebeat ❶, and conditionally applied optional headers for Kafka ❷.

Kafka headers are key-value pairs that provide extra metadata for the purposes of routing data and analyzing performance. They're analogous to the HTTP headers that track the details of who sent a message, what was sent, when it was sent, and so on. You can add them to the outgoing message, and their syntax supports conditional logic. You can also specify multiple keys with the same name, as they don't need to be unique.

Specifying an array of hosts returns metadata about the cluster itself, and Filebeat uses this information to connect to the actual addresses where messages are sent. Kafka will do its best to load-balance itself by telling producers and consumers where to connect.

Unlike inputs, outputs don't directly support tags, which get applied globally in *filebeat.yml* or using modules. You might find it useful to add directional tagging in *filebeat.yml* when transmitting data in and out of Kafka. If Filebeat writes to a topic and then Logstash consumes and enriches that data before sending to a different topic, you might tag each step to capture this lineage. The following is just one tag you may add:

```
tags: [ "to-kafka-filebeat" ]
```

After you start Filebeat, data should flow, appearing as usable JSON fields:

```
--snip--
   "process" => {
   "program" => "sudo"
},
      "tags" => [
    [0] "tags-for-everybody",
    [1] "you-get-a-tag",
    [2] "and-you-get-a-tag",
    [3] "to-kafka-filebeat",
    [4] "forwarded"
],
   "message" => "pam_unix(sudo:session): session closed for user root",
     "input" => {
      "type" => "syslog"
},
--snip--
```

Logstash should display these JSON fields on the screen.

Publishing to Redis

Redis is an in-memory message broker that we can use as an intermediary between Filebeat and another system. Sending logs to Redis functions mostly the same way as sending them to Kafka: We specify a key, which will then store the values sent to it in a queue structure, along with a password and network details.

The key statement in the Redis output block supports conditional statements such as when, which is useful for routing per module or input type. The following example uses nested subconditionals to check for the Zeek module and either the SSH or the Telnet dataset:

```
output.redis:
  enabled: true
  hosts: ["localhost"]
  password: "YOUR-REDIS-PASSWORD"
❶ key: "filebeat"
❷ keys:
    - key: "remoteaccess"
   ❸ when:
     ❹ or:
       ❺ - contains:
              event.dataset: "zeek.ssh"
          - contains:
              event.dataset: "zeek.telnet"
        and:
          - equals:
              event.module: "zeek"
  ssl.enabled: true
  ssl.verification_mode: full
  ssl.certificate: "/home/j/tls/certs/filebeat.local.flex.cert.pem"
  ssl.key: "/home/j/tls/keys/filebeat.local.flex.key.pem"
  ssl.certificate_authorities:
    - /home/j/tls/certs/ca-chain.cert.pem
--snip--
```

In this statement, we funnel Zeek SSH and Telnet events to a Redis list named `remoteaccess` using the `keys` (plural) statement ❷ and conditionals, with a fallback to the `filebeat` list when `key` (singular) ❶ is also specified for logs that don't meet the conditional statements. The `when` ❸ conditional contains multiple checks ❹, which also have subconditions ❺ defined.

Simply checking `event.module` is a slimmed-down approach to data routing, as it pushes routing complexity downstream, away from endpoints. In the following snippet, we apply this approach using a single conditional:

```
output.redis:
  enabled: true
  hosts: ["localhost"]
  password: "YOUR-REDIS-PASSWORD"
  key: "filebeat"
  keys:
    - key: "zeek"
      when:
     ❶ equals:
          event.module: "zeek"
--snip--
```

Notice that the YML syntax changes when `equals` moves up a level ❶, as we don't use the hyphenated list structure. Here, Filebeat will send this network data to the Redis key `zeek` only when the `event.module` field is also `zeek`; everything else will go to the `filebeat` key.

Writing Output Data to a File

One straightforward way to output processed data is to write it to a file on disk that Filebeat has write access to. Unless we specify otherwise, Filebeat will name this file using the mandatory `path` value, followed by the program name and a timestamp, potentially with trailing numbers that increment, such as */var/log/filebeat-20400501-1.ndjson*. By default, files rotate every 10MB and roll over every seven files, or every time the Filebeat process starts. The default file permission mode is `0600`, meaning that only the owner of the Filebeat process can read or write to the file.

For example, the following output section would write 10 files that are approximately 100MB each before rotating, for a total of about 1GB:

```
output.file:
  enabled: true
  path: /var/log/filebeat/
  rotate_every_kb: 100000
  number_of_files: 10
```

Start Filebeat so we can monitor the creation of new output files by running the watch command. The following will list the new logfiles every second and will highlight where byte sizes change or new files appear:

```
$ watch -n 1 -d ls -l /var/log/filebeat/
-rw------- 1 j    j     33M May  1 22:53 filebeat-20400501-1.ndjson
-rw------- 1 j    j     49M May  1 22:52 filebeat-20400501.ndjson
```

We can watch files being written to, and aged off, in real time. Cancel the updating terminal with CTRL-C.

Sending Data to Logstash

Another output option is to send data to Logstash. Filebeat can receive status checks from Logstash telling it to slow down to reduce congestion or to apply *backpressure*, which results in smaller peaks and valleys in traffic volume across the network. Filebeat can also ramp up the level of compression when sending traffic to Logstash to reduce load on the network, but at the cost of slightly higher CPU usage by Logstash.

In this example, let's use two worker threads with no load balancing and crank the compression to 9 (the maximum). Use these lines for the Logstash output in *filebeat.yml*:

```
output.logstash:
  enabled: true
  hosts: ["192.168.8.132:5044"]
  workers: 2
  compression_level: 9
--snip--
```

Filebeat and other Beats add metadata fields that aren't typically displayed using the Logstash stdout output. To see this data in action, let's modify the *beats-mtls.conf* Logstash configuration by copying the invisible @metadata field into a new one named metadata:

```
--snip--
filter {
    mutate {
      ❶ copy => { "@metadata" => "metadata" }
    }
}
--snip--
```

The mutate filter copies the @metadata field, which we can't normally see, into a metadata field we can see ❶. The configuration should reload automatically, since Logstash is still running in another terminal.

You may also set metadata => true in the output, which we'll configure later in the book. For now, let's copy the metadata to get comfortable using filters. Downstream, Elasticsearch will use the @metadata.beat and @metadata .version to direct the indexing, or storage, of data in the database.

Although Filebeat sets fields such as `agent.name`, I've found that including the static IP address of the host at that point in time also helps troubleshoot issues, as seen in `metadata.beats.host.ip`:

```
  --snip--
  "metadata" => {
         "beat" => "filebeat",
      "version" => "8.7.0",
    "truncated" => false,
         "type" => "_doc",
        "input" => {
        "beats" => {
            "host" => {
              ❶ "ip" => "192.168.8.134"
   --snip--
   "message" => "pam_unix(sudo:session): session closed for
                 user root",
     "input" => {
      "type" => "syslog"
},
   "process" => {
   "program" => "sudo"
    --snip--
```

Having the IP address ❶ of the node running Filebeat provides insight if something in the pipeline goes awry.

Pruning and Privatizing Data

At some point, you'll need to privatize or remove unnecessary data from your logs. Perhaps an auditor is reviewing your company's logs for compliance reasons, or maybe a machine learning algorithm requires only certain fields and nothing more.

Earlier in this chapter, we covered the `drop_fields` and `script` processors. Let's now use these to remove sensitive data. Say an auditor needs to review metadata from a bank's servers; we could replace all `user.name` values with a generic term, such as `BankUser`. We could also drop everything irrelevant to the auditor, such as tags and agent names:

```
processors:
❶ - drop_fields:
    fields: ["agent", "ecs", "event.severity", "host",
      "input", "log", "syslog", "tags"]

❷ - script:
    lang: javascript
    source: >
      function process(event) {
      ❸ event.Put("user.name", "BankUser")
      }
    when:
      has_fields: [ "user.name" ]
```

First, we drop the fields we don't need to send ❶. Next, a short script overwrites all usernames ❷ with a single value ❸.

This results in an extremely truncated log (minus the otherwise invisible `@metadata` fields) to send downstream:

```
{
       "process" => {
       "program" => "sudo"
    },
          "user" => {
          "name" => "BankUser"
    },
      "@version" => "1",
       "message" => "pam_unix(sudo:session): session opened for
                    user root by (uid=0)",
    "@timestamp" => 2040-04-30T05:04:19.000Z,
      "hostname" => "server01"
}
```

This log meets the minimum requirements in our hypothetical audit situation. It may also be ideal for your daily needs; keep in mind that the bytes you store on disks come out of your budget!

Filebeat for Windows

Filebeat can read any logfile on Windows that isn't part of the Windows Event Log. (To read events from the Windows Event Log, we use Winlogbeat, discussed in Chapter 5.)

The fantastic web request tool cURL is now built into Windows, so you can download the Filebeat ZIP file for Windows directly using *curl.exe*. Don't skip the *.exe* portion, as plain `curl` is an alias for the `Invoke-WebRequest` PowerShell command. Download the latest version of Filebeat, substituting *`X.Y.Z`* for a version number:

```
PS> curl.exe -O https://artifacts.elastic.co/downloads/
beats/filebeat/filebeat-X.Y.Z-windows-x86_64.zip
```

Extract the ZIP somewhere and copy it into *C:\Program Files\Filebeat\* using an administrator PowerShell prompt. Then, change directories into that path:

```
PS> copy -r .\filebeat-8.x.x-windows-x86_64\ "C:\Program Files\Filebeat"
PS> cd "C:\Program Files\Filebeat"
```

Next, run the provided install script to create a Windows service:

```
PS>.\install-service-filebeat.ps1
```

If you receive an error while running the script stating that running scripts is disabled on the system, change the PowerShell execution policy temporarily:

```
PS> Get-ExecutionPolicy
PS> Set-ExecutionPolicy Bypass
```

Run the install script and then set the execution policy back to the previous value identified with `Get-ExecutionPolicy`.

This script should install the Windows service with the startup type `Automatic (Delayed Start)`, which means it may take around a minute to start. The Filebeat service will wait for other Windows services marked as `Automatic` to fully load after the next reboot. For now, don't manually start the service or execute *filebeat.exe*.

You should also use this *filebeat.yml* instead of the ones shown earlier in this chapter, as it contains a useful logfile for testing data found on most versions of Windows 10 and 11:

```
filebeat.inputs:

# Input that reads Edge update logs
- type: filestream
  enabled: true
  id: edgeupdate-logs
  paths:
    - C:/ProgramData/Microsoft/EdgeUpdate/Log/*.log
❶ encoding: utf-16le-bom

# Module locations
filebeat.config.modules:
  path: ${path.config}/modules.d/*.yml

tags: ["windows", "tags-for-everybody", "you-get-a-tag", "and-you-get-a-tag"]

output.logstash:
  enabled: true
  hosts: ["logstash.local:5044"]
--snip--
```

We use little-endian UTF-16 encoding with a required byte order mark ❶ to accurately read text files generated by Windows.

NOTE *Keep the following filepath handy for testing Filebeat on a Windows system:* C:/ProgramData/Microsoft/EdgeUpdate/Log/*.log. *When the Edge browser checks for or performs an update, it appends an entry to this log, making it an ideal data source when testing Filebeat on Windows.*

Before we wrap up, let's store our configurations in our Git repository.

SAVING CONFIGURATIONS WITH GIT

At the beginning of the chapter, we created Filebeat's TLS configurations; let's store them safely in the project repository we created in Chapter 3 along with any command notes and *filebeat.yml* examples you may have created. Change directories into your *book-data-pipelines* local repository and make a directory for Filebeat. Copy any configurations and notes you wish to preserve for reference into the new directory, and redact any passphrases you don't wish to commit. Track the new files in your local repository and check its status:

```
$ git add . && git status
On branch main
Your branch is up to date with 'origin/main'.
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
      new file:   filebeat/filebeat.yml
--snip--
```

Finally, commit the changes and push them to the remote repository:

```
$ git commit -a -m "added Filebeat configs and TLS files"
$ git push
```

Your Filebeat configurations should be safely backed up.

Summary

Filebeat is a simple yet powerful tool for extracting data, transforming it into something useful, and shipping it elsewhere. It formats data into JSON early in the data pipeline, then offloads further complex processing downstream. Filebeat's processors can also format, drop, or privatize data if needed, and its modules provide standardized field names by producing output in the ECS. Filebeat also supports industry-standard, enterprise-scale data engineering tools such as Kafka, Redis, and Logstash. In the next chapter, we'll use Elastic Winlogbeat to collect event logs from Windows systems.