

Detection-as-Code with MongoDB Change Streams

Jacob Latonis - Proofpoint

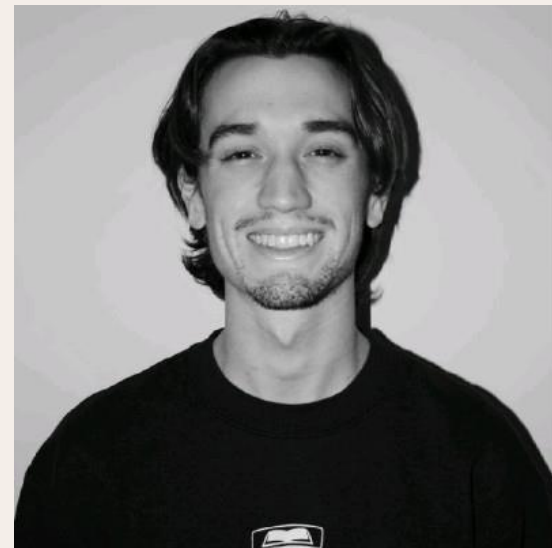




/usr/bin/whoami

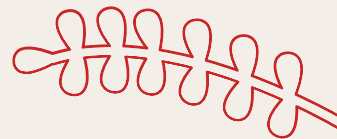
Jacob Latonis
Senior Threat Research Engineer @ Proofpoint

I run a lot, currently training for the Chicago Marathon.
I geek out about all things in the security world.
Ask me about walkable cities ☺.



Twitter: [@jacoblatonis](https://twitter.com/jacoblatonis)
BlueSky: [@jacoblatonis.bsky.social](https://bsky.app/profile/jacoblatonis.bsky.social)
LinkedIn: [Jacob-latonis](https://www.linkedin.com/in/jacob-latonis)

Table of Contents



01

Detection-as-Code

Versioned security
detections stored in Git

02

The Data

Where to get the data to run
detections against

03

Change Streams

Leveraging MongoDB's
Change Stream on certain
collections

04

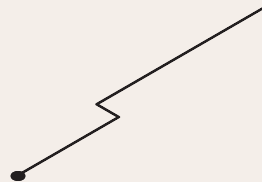
Listening

Using Change Streams to
stream the data into a
handler

05

Detection!

Catch and alert on the
activity

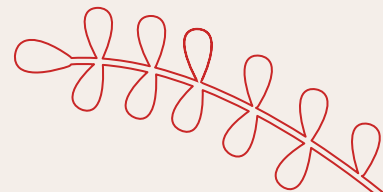




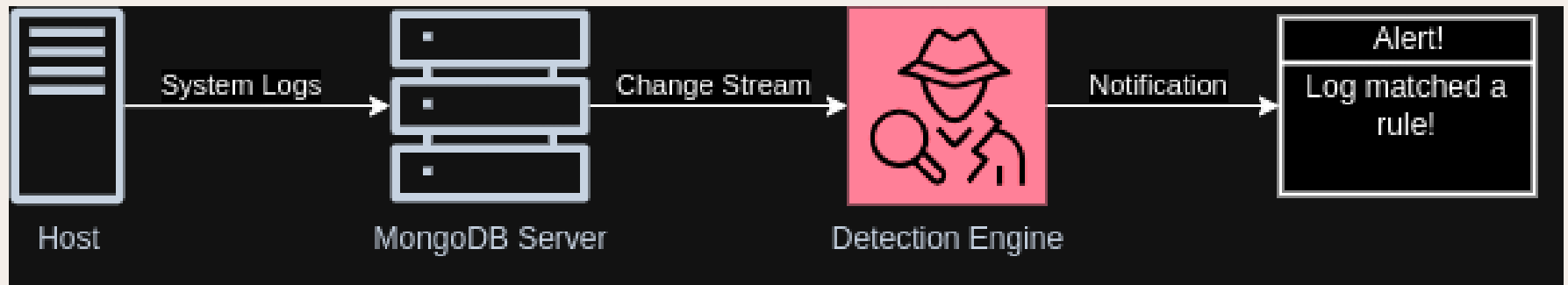
01

Detection-as-Code

01010101
01010101



General Architecture



A Use Case



- Organization A wants to monitor command line usage a little more closely to observe and alert on commands or actions they deem suspicious
- Organization A wants to store those detections in a standard, uniform way with version control
- Organization A has familiarity with MongoDB
- Organization A also has familiarity with a language that has a great MongoDB Driver, like Python or Go

Why Detection-as-Code



● Versioning!

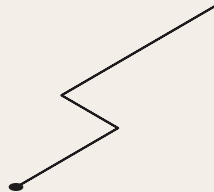
Track changes made to the detections with Git

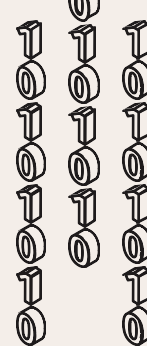
● Tests!

Run tests against the detections to ensure the detection works as intended

● Automated!

Automate the testing and deployment of the detections

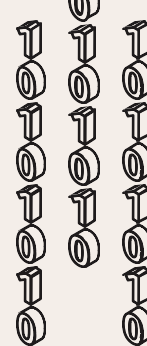




```
26 26      CommandLine|contains:
27 27      # Add more suspicious locations as you find them
28 28      # The space from the start is missing to cover append operations ">>"
29 -      - '> %USERPROFILE%\ '
30 -      - '> %APPDATA%\ '
31 29      - '> \Users\Public\ '
32 -      - '> C:\Users\Public\ '
33 30 +      - '> %APPDATA%\ '
33 31      - '> %TEMP%\ '
34 32      - '> %TMP%\ '
35 -      - '> C:\Windows\Temp\ '
36 33 +      - '> %USERPROFILE%\ '
36 34      - '> C:\Temp\ '
37 35 +      - '> C:\Users\Public\ '
38 36 +      - '> C:\Windows\Temp\ '
39 37 +      - '> \Users\Public\ '
40 38 +      - '> %APPDATA%\ '
41 39 +      - '> %TEMP%\ '
42 40 +      - '> %TMP%\ '
43 41 +      - '> %USERPROFILE%\ '
44 42 +      - '> C:\Temp\ '
45 43 +      - '> C:\Users\Public\ '
46 44 +      - '> C:\Windows\Temp\ '
```

Versioning

You can see when changes were made to what detections and likely why



```
[~/projects/detection-as-code-mongo]-[jacob@pop-os]-[0]-[781]
[(:)] % pytest
===== test session starts =====
platform linux -- Python 3.10.12, pytest-7.4.2, pluggy-1.3.0
rootdir: /home/jacob/projects/detection-as-code-mongo
configfile: pyproject.toml
collected 2 items

engine/tests/test_engine.py .. [100%]

===== 2 passed in 0.10s =====
```

Testing

Ensure the detection will fire when conditions are met





```
[~/projects/detection-as-code-mongo]-[jacob@pop-os]-[0]-[795]  
o [[:]] % python3 engine/detection.py  
Loading new rules...  
    loaded shadow.yaml!  
    loaded tmp_pipe.yaml!
```

Automated

Deploy new rules and test them
with each change



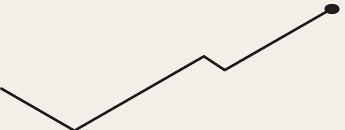
Writing a Rule



- Choose your favorite structured text language
 - YAML, JSON, etc.
- Have a structured template
- Write the logic

```
rules > ! shadow.yaml
```

```
1 name: Suspicious /etc/shadow Usage
2 uuid: 5e3566a3-9f19-4ac7-8710-ec5fb09260ed
3 detection_logic:
4   command_line:
5     - /etc/shadow
6
```




```
rules > ! tmp_output_redir.yaml
```

```
1 name: Output redirection to /tmp/ Directory
2 uuid: f825ad39-d4b1-41da-91ec-6438d8dc9ace
3 detection_logic:
4   command_line:
5     - "> /tmp/"
6
```

My Rule Structure



- **name:** the name of the rule
 - **uuid:** unique identifier for each rule to know what fired
 - **detection_logic:** dictionary of fields to search for 1+ specific values in each field
-
- These rules are expandable
 - These rules can be as complex or as simple as you want
 - Your detection as code can be actual code, not just structured text loaded into an engine
- 

Another Rule Format

- If you've had enough YAML in your life already, we can also define rules as code another way.
- Instead of an engine loading rule files from YAML, the engine can load modules of code that handle the event directly
- No need for YAML -> Rule class
- Code the Rule classes themselves

YAML <--> Python

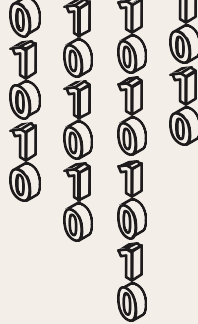
Python Rule

```
rules > shadow.py > ...  
1 def rule(event: dict) -> bool:  
2     if "/etc/shadow" in event.get("command_line"):  
3         return True  
4     return False  
5
```

YAML Rule

```
rules > ! shadow.yaml  
1 name: Suspicious /etc/shadow Usage  
2 uuid: 5e3566a3-9f19-4ac7-8710-ec5fb09260ed  
3 detection_logic:  
4     command_line:  
5         - /etc/shadow
```

Please test your detections to help avoid...



Alert Fatigue

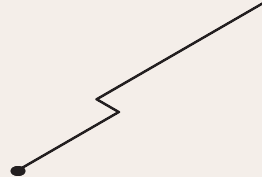
Too many events
overwhelm

False Positives

Events that are benign
but fired an alert

False Negatives

Events that should alert
but do not



Test Driven Development

- You've probably heard this term before
- It is similar; write the code ↔ test the code
- We can automate most of the testing with frameworks
 - nose
 - pytest
 - <insert your favorite here>

Breaking down a test

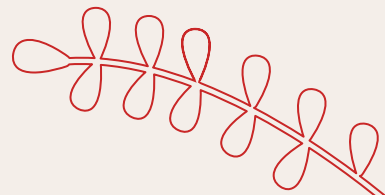
- Define a test log or a set of test logs
- Load the rules to be tested
- Run detection engine
- Compare output to what is expected from the logs given
- Fix/work on detections as needed

```
engine > tests > test_engine.py > ...
1  import detection
2
3
4  def test_shadow():
5      test_log = {
6          "hostname": "pop-os",
7          "ip_address": "127.0.1.1",
8          "parent_pid": 1,
9          "pid": 2,
10         "path": "/usr/bin/sudo",
11         "binary": "sudo",
12         "arguments": "cat /etc/shadow",
13         "command_line": "/usr/bin/sudo cat /etc/shadow",
14     }
15
16     engine = detection.DetectionEngine()
17     engine.add_rules("./rules/")
18
19     expected = ["5e3566a3-9f19-4ac7-8710-ec5fb09260ed"]
20
21     results = engine.process_log(test_log)
22
23     assert results == expected
```



02

The Data



Data Source Options

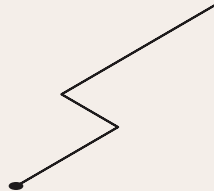


- **Windows Event Logs**

- **Linux logs**


- **eBPF**

This is the fun one 😊



Why eBPF?

A decorative line in the top right corner consisting of a horizontal segment followed by a series of connected diagonal segments forming a jagged, zig-zag pattern.

- First, if you're unfamiliar with eBPF, I highly recommend <https://ebpf.io/> and <https://www.brendangregg.com/>.
 - Great resources, great examples
 - eBPF provides monitoring at almost any level, network, process, and more via hooks into the kernel.
 - This is how we're going to get our data today
 - Process monitoring with eBPF and BCCTools [1]
- 
- A decorative line in the bottom left corner consisting of a series of connected diagonal segments forming a jagged, zig-zag pattern.

[1] <https://github.com/iovisor/bcc>

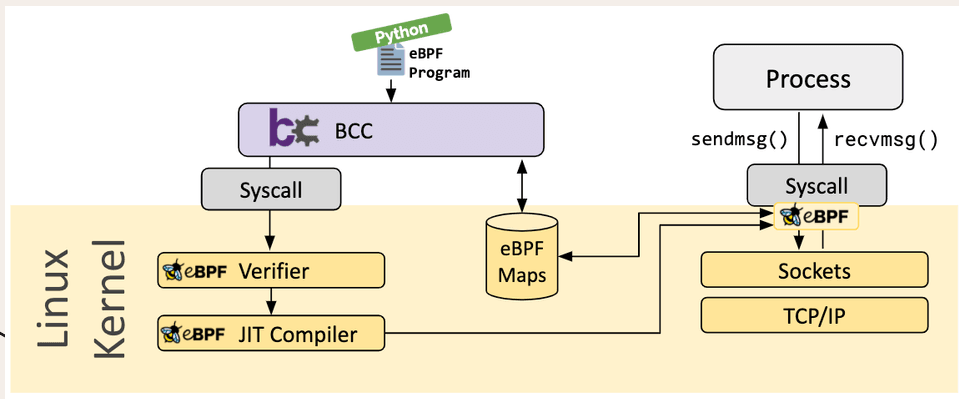
What's an eBPF Program look like?

- tldr; it's a bunch of C code
- create a struct to store the details we want
- hook the system call
- send that data back

```
41 struct pt_regs *bpf_get_current_pt_regs() {  
42     task = (struct task_struct *)bpf_get_current_task();  
43     // Some kernels, like Ubuntu 4.13.0-generic, return 0  
44     // as the real_parent->tgid.  
45     // We use the get_ppid function as a fallback in those cases. (#1883)  
46     data.ppid = task->real_parent->tgid;  
47     bpf_get_current_comm(&data.comm, sizeof(data.comm));  
48     data.type = EVENT_ARG;  
49     __submit_arg(ctx, (void *)filename, &data);  
50     // skip first arg, as we submitted filename  
51     #pragma unroll  
52     for (int i = 1; i < 20; i++) {  
53         if (submit_arg(ctx, (void *)&__argv[i], &data) == 0)  
54             goto out;  
55     }  
56     // handle truncated argument list  
57     char ellipsis[] = "...";  
58     __submit_arg(ctx, (void *)ellipsis, &data);  
59 out:  
60     return 0;  
61 }  
62 int do_ret_sys_execve(struct pt_regs *ctx)  
63 {  
64     struct data_t data = {};  
65     struct task_struct *task;  
66     data.pid = bpf_get_current_pid_tgid() >> 32;  
67     task = (struct task_struct *)bpf_get_current_task();  
68     // Some kernels, like Ubuntu 4.13.0-generic, return 0  
69     // as the real_parent->tgid.  
70     // We use the get_ppid function as a fallback in those cases. (#1883)  
71     data.ppid = task->real_parent->tgid;  
72     bpf_get_current_comm(&data.comm, sizeof(data.comm));  
73     data.type = EVENT_RET;  
74     data.retval = PT_REGS_RC(ctx);  
75     events.perf_submit(ctx, &data, sizeof(data));  
76     return 0;  
77 }
```

Collecting the Data

- Leverage BCC tools and Python to load an eBPF program
- Once loaded, collect the process events
- This is just an example; there's lots of room for improvement or customization to fit what your organization may want to collect or alert on



```
131 b = BPF(text=program, cflags=["-Wno-macro-redefined"])
132 eventt = b.get_syscall_fnname("execve")
133 b.attach_kprobe(event=eventt, fn_name="syscall_execve")
134 b.attach_kretprobe(event=eventt, fn_name="do_ret_sys_execve")
```

Sending the Data to MongoDB (plan)

1. Load the eBPF program into memory
2. Hook **execve()** calls
3. Monitor **execve()** calls
4. Log the **execve()** calls
5. Push the **execve()** logs to a MongoDB collection

Sending the Data to MongoDB (code)

- Load the eBPF program into memory
- Hook **execve()** calls
- Monitor **execve()** calls

```
def init_agent(self):
    self.b = BPF(text=self.program, cflags=["-Wno-macro-redefined"])
    eventt = self.b.get_syscall_fnname("execve")
    self.b.attach_kprobe(event=eventt, fn_name="syscall_execve")
    self.b.attach_kretprobe(event=eventt, fn_name="do_ret_sys_execve")

    self.b["events"].open_perf_buffer(self.parse_event)

    while 1:
        try:
            self.b.perf_buffer_poll()
```

Sending the Data to MongoDB (code)

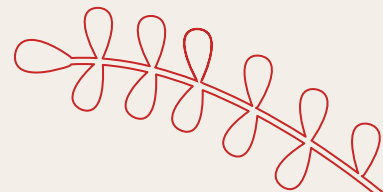
- Log the `execve()` calls
- Push the `execve()` logs to a MongoDB collection

```
def push_mongo(self, event):
    self.collection.insert_one(event)

def parse_event(self, cpu, data, size):
    event = self.b["events"].event(data)
    if event.type == EventType.EVENT_ARG:
        self.argv[event.pid].append(event.argv.decode("utf-8"))
    elif event.type == EventType.EVENT_RET:
        ppid = event.ppid
        pid = event.pid
        binary = str(event.comm.decode("utf-8"))
        path = ""
        arguments = " ".join(self.argv[pid][1:])
        if self.argv[pid]:
            path = self.argv[pid][0]
        entry = {
            "hostname": self.hostname,
            "ip_address": self.ip_addr,
            "parent_pid": ppid,
            "pid": pid,
            "path": path,
            "binary": binary,
            "arguments": arguments,
            "command_line": f"{path} {arguments}",
        }
        self.push_mongo(entry)
```



03 MongoDB Change Streams

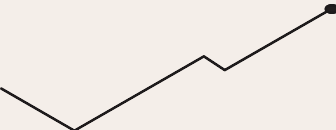


Why Change Streams?

- React to data changes in near real time
- Develop solutions in most (all?) languages with driver support for MongoDB
- Don't have to tail the oplog!
- Filter out the items you do not want with an aggregation pipeline

Prerequisites for Change Streams



- Change Streams are available for replica sets and sharded clusters.
 - Replica sets and sharded clusters must use replica set protocol version 1 (**pv1**).
 - replica sets and sharded clusters must use the **WiredTiger** storage engine
 - Default in Mongo 3.2+
 - Change Streams cannot be enabled for **system** collections or any collections in the **admin**, **local**, and **config** databases
- 

Change Stream Event

- What does it look like?
- Why do I care?
- What can we use from it?

```
1  {
2    "_id": "<resume token>",
3    "operationType": "insert",
4    "clusterTime": "Timestamp(1695255007, 1)",
5    "wallTime": "2023-09-21 00:10:07.378000",
6    "fullDocument": {
7      "_id": "650b89dfcf00592980c13266",
8      "hostname": "pop-os",
9      "ip_address": "127.0.1.1",
10     "parent_pid": 43729,
11     "pid": 43748,
12     "path": "/usr/bin/cat",
13     "binary": "cat",
14     "arguments": "/proc/43695/stat",
15     "command_line": "/usr/bin/cat /proc/43695/stat"
16   },
17   "ns": {
18     "db": "monitoring",
19     "coll": "process"
20   },
21   "documentKey": {
22     "_id": "650b89dfcf00592980c13266"
23   }
24 }
```

What can I filter on?

- tldr; a lot

Event	Description
<u>create</u>	Occurs on the creation of a collection. Requires that you set the <u>showExpandedEvents</u> option to true.
<u>createIndexes</u>	Occurs on the creation of indexes on the collection. Requires that you set the <u>showExpandedEvents</u> option to true.
<u>delete</u>	Occurs when a document is removed from the collection.
<u>drop</u>	Occurs when a collection is dropped from a database.
<u>dropDatabase</u>	Occurs when a database is dropped.
<u>dropIndexes</u>	Occurs when an index is dropped from the collection. Requires that you set the <u>showExpandedEvents</u> option to true.

What can I filter on?

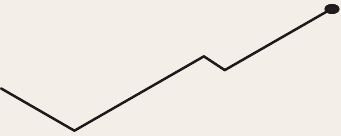
- tldr; a lot part 2

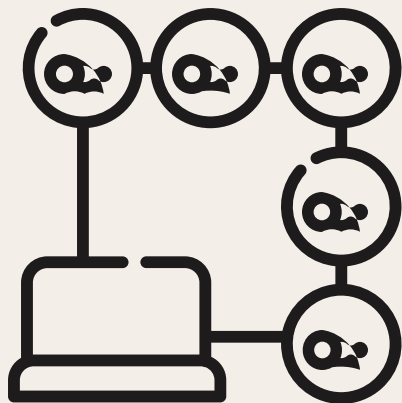
<u>invalidate</u>	Occurs when an operation renders the change stream invalid.
<u>modify</u>	Occurs when a collection is modified. Requires that you set the <u>showExpandedEvents</u> option to true.
<u>refineCollectionShardKey</u>	Occurs when a shard key is modified.
<u>rename</u>	Occurs when a collection is renamed.
<u>replace</u>	Occurs when an update operation removes a document from a collection and replaces it with a new document.
<u>reshardCollection</u>	Occurs when the shard key for a collection and the distribution of data changes.
<u>shardCollection</u>	Occurs when a collection is sharded. Requires that you set the <u>showExpandedEvents</u> option to true.
<u>update</u>	Occurs when an operation updates a document in a collection.

What we'll filter on today



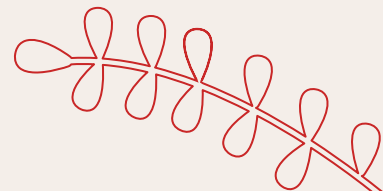
<u>insert</u>	Occurs when an operation adds documents to a collection.
---------------	--





04

Listening to the Events



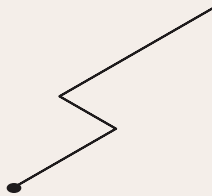
Watching a Change Stream

- Now that we know the prerequisites, let's get watching!
- We can get a cursor and iterate over the changes and monitor them.
- Most (all?) Mongo drivers implement a way to watch over the change streams
- We can define a **pipeline** to only see the operations we want
 - In this case, it's only insert operations

```
def get_change_stream(self) -> None:
    pipeline = [{"$match": {"operationType": {"$in": ["insert"]}}}]

    self.cursor = (
        self.mongo_client.get_database(os.getenv("CHANGE_DB_NAME", ""))
        .get_collection(os.getenv("CHANGE_COLLECTION_NAME", ""))
        .watch(pipeline=pipeline)
    )
```

Working with Resume Tokens



- If your detection engine crashes, needs to be restarted, etc., the engine needs to be restarted from the moment in time when it stopped processing events
- We can resume from that moment in time with a **resume_token**, accessible from the **change stream cursor**
- Keep track of the **resume_token** and save it somewhere

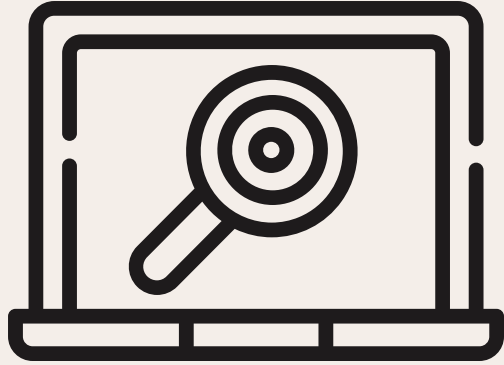
```
# watch the change stream
for document in engine.cursor:
    log = document.get("fullDocument")
    engine.process_log(log)
    engine.resume_token = engine.cursor.resume_token
```

Resume from a Resume Token

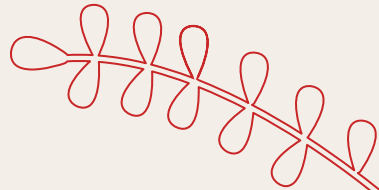
- Once you are ready to resume watching the change stream, use the **resume_token** and pass it when instantiating the **change stream**.

```
def get_change_stream(self) -> None:
    pipeline = [{"$match": {"operationType": {"$in": ["insert"]}}}]

    if self.resume_token:
        self.cursor = (
            self.mongo_client.get_database(os.getenv("CHANGE_DB_NAME", ""))
            .get_collection(os.getenv("CHANGE_COLLECTION_NAME", ""))
            .watch(pipeline=pipeline, resume_after=self.resume_token)
        )
    else:
        self.cursor = (
            self.mongo_client.get_database(os.getenv("CHANGE_DB_NAME", ""))
            .get_collection(os.getenv("CHANGE_COLLECTION_NAME", ""))
            .watch(pipeline=pipeline)
        )
```



05 Detection Time!



Running a suspicious command...

Testing

There's multiple ways to test a detection

- Insert logs that *should* fire
- Compare against known malicious logs
- Test on a host

For the sake of time and results, we'll be testing on a host directly by executing suspicious commands

The actual command



A terminal window titled 'jacob@pop-os:~' with search, menu, and window control icons in the title bar. The prompt is '[~]-[jacob@pop-os]-[0]-[674]'. The command ': sudo cat /etc/shadow' is entered, with a cursor at the end. A vertical scrollbar is on the right.

```
[~]-[jacob@pop-os]-[0]-[674]  
[: ] % sudo cat /etc/shadow
```


The actual detection!

Running the command from the previous slide, let's try out the detection!

```
alert!! 5e3566a3-9f19-4ac7-8710-ec5fb09260ed - Suspicious /etc/shadow Usage
{
  "_id": "64fc8ecf6c6d490f62497b7a",
  "hostname": "pop-os",
  "ip_address": "127.0.1.1",
  "parent_pid": 29297,
  "pid": 31454,
  "path": "/usr/bin/sudo",
  "binary": "sudo",
  "arguments": "cat /etc/shadow",
  "command_line": "/usr/bin/sudo cat /etc/shadow"
}
```

An alert fired, now what?

Store Them!

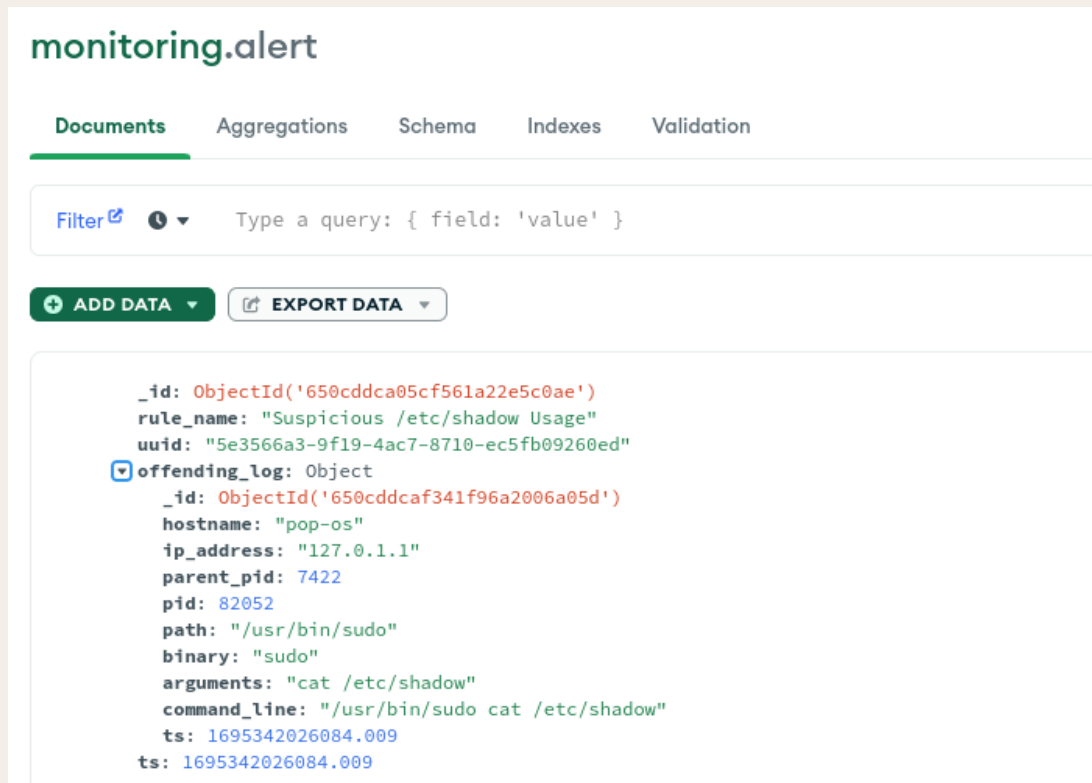
Over time, you'll likely amass quite a few detection alerts.

Investigate!

Use the other logs from the system in a +/- 5min time frame to see what was occurring.

Peeking into the alert collection

- What rule fired
- When did it fire
- Why did it fire
- Is it malicious



The screenshot shows the 'monitoring.alert' interface. The 'Documents' tab is selected, displaying a single document. The document contains metadata and an 'offending_log' field with details about a suspicious command execution.

```
monitoring.alert

Documents Aggregations Schema Indexes Validation

Filter [icon] [clock] Type a query: { field: 'value' }

[+ ADD DATA] [EXPORT DATA]

{
  _id: ObjectId('650cddca05cf561a22e5c0ae')
  rule_name: "Suspicious /etc/shadow Usage"
  uuid: "5e3566a3-9f19-4ac7-8710-ec5fb09260ed"
  offending_log: Object
    {
      _id: ObjectId('650cddcaf341f96a2006a05d')
      hostname: "pop-os"
      ip_address: "127.0.1.1"
      parent_pid: 7422
      pid: 82052
      path: "/usr/bin/sudo"
      binary: "sudo"
      arguments: "cat /etc/shadow"
      command_line: "/usr/bin/sudo cat /etc/shadow"
      ts: 1695342026084.009
    }
  ts: 1695342026084.009
}
```



Pivot from the original alert

Use your timestamps!

```
1 import datetime
2
3 timenow = datetime.datetime.now(datetime.timezone.utc)
4 five_before = (timenow - datetime.timedelta(minutes=5)).timestamp() * 1000
5 five_after = (timenow + datetime.timedelta(minutes=5)).timestamp() * 1000
6
```

Locate the process identifier (pid)

```
_id: ObjectId('650cddca05cf561a22e5c0ae')
rule_name: "Suspicious /etc/shadow Usage"
uuid: "5e3566a3-9f19-4ac7-8710-ec5fb09260ed"
☒ offending_log: Object
  _id: ObjectId('650cddcaf341f96a2006a05d')
  hostname: "pop-os"
  ip_address: "127.0.1.1"
  parent_pid: 7422
  pid: 82052
  path: "/usr/bin/sudo"
  binary: "sudo"
  arguments: "cat /etc/shadow"
  command_line: "/usr/bin/sudo cat /etc/shadow"
  ts: 1695342026084.009
ts: 1695342026084.009
```

Once you have the pid, build the process tree

```
jacob      12615      1975      1 08:19 ?          00:00:00 \_ /usr/libexec/gnome-terminal-server
jacob      12641      12615      0 08:19 pts/2        00:00:00 \_ zsh
jacob      12672      12641      0 08:19 pts/2        00:00:00 |   \_ python3
jacob      12939      12672      0 08:20 pts/2        00:00:00 |       \_ sh -c /bin/bash
jacob      12940      12939      0 08:20 pts/2        00:00:00 |           \_ /bin/bash
root       12967      12940      0 08:20 pts/2        00:00:00 |               \_ sudo su
root       12968      12967      0 08:20 pts/3        00:00:00 |                   \_ sudo su
root       12969      12968      0 08:20 pts/3        00:00:00 |                       \_ su
root       12970      12969      0 08:20 pts/3        00:00:00 |                           \_ bash
```

Validating the original Use Case



- Organization A wants to monitor command line usage a little more closely to observe and alert on commands or actions they deem suspicious
- Organization A wants to store those detections in a standard, uniform way with version control
- Organization A has familiarity with MongoDB
- Organization A also has familiarity with a language that has a great MongoDB Driver, like Python or Go



Questions?

Repository (slides + code)

- <https://github.com/latonis/detection-as-code-mongo/>



Thanks!

Socials (reach out!)

- Twitter: [@jacoblatonis](https://twitter.com/jacoblatonis)
- BlueSky: [@jacoblatonis.bsky.social](https://bsky.social/@jacoblatonis)
- LinkedIn: [Jacob-latonis](https://www.linkedin.com/in/jacob-latonis)

CREDITS: This presentation template was created by **Slidesgo**, including icons by **Flaticon** and infographics & images by **Freepik**

