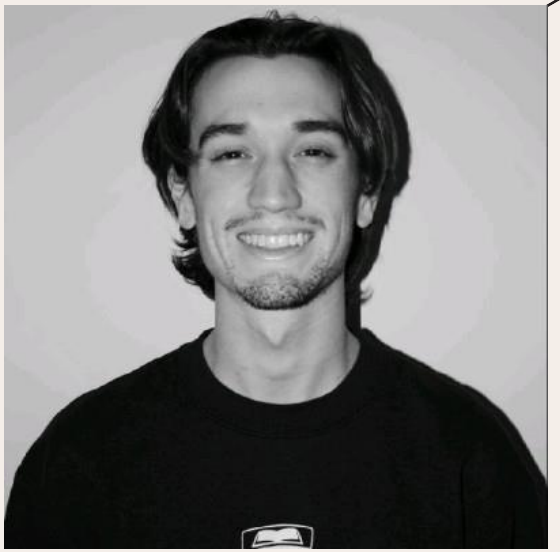
Detection-as-Code with MongoDB Change Streams

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 ☺.

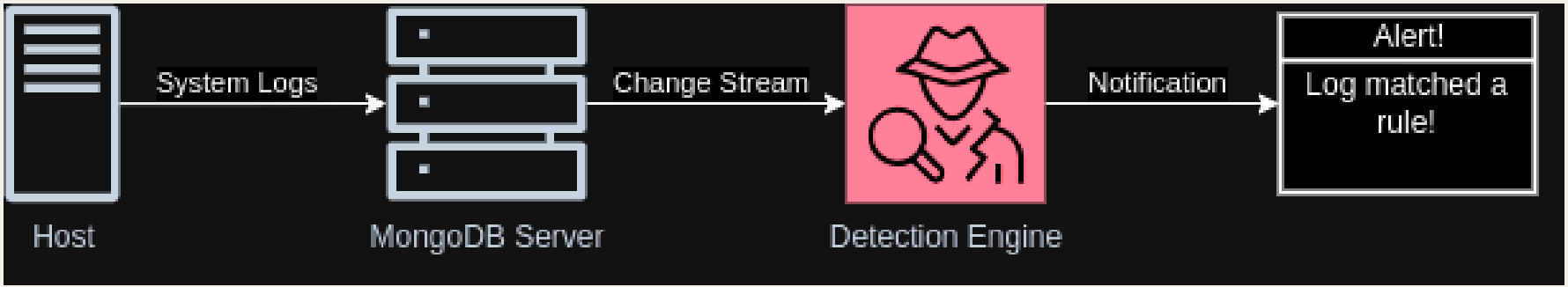
Twitter: @jacoblatonis
BlueSky: @jacoblatonis.bsky.social
LinkedIn: Jacob-latonis

Table of Contents

01

Detection-as-Code

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!

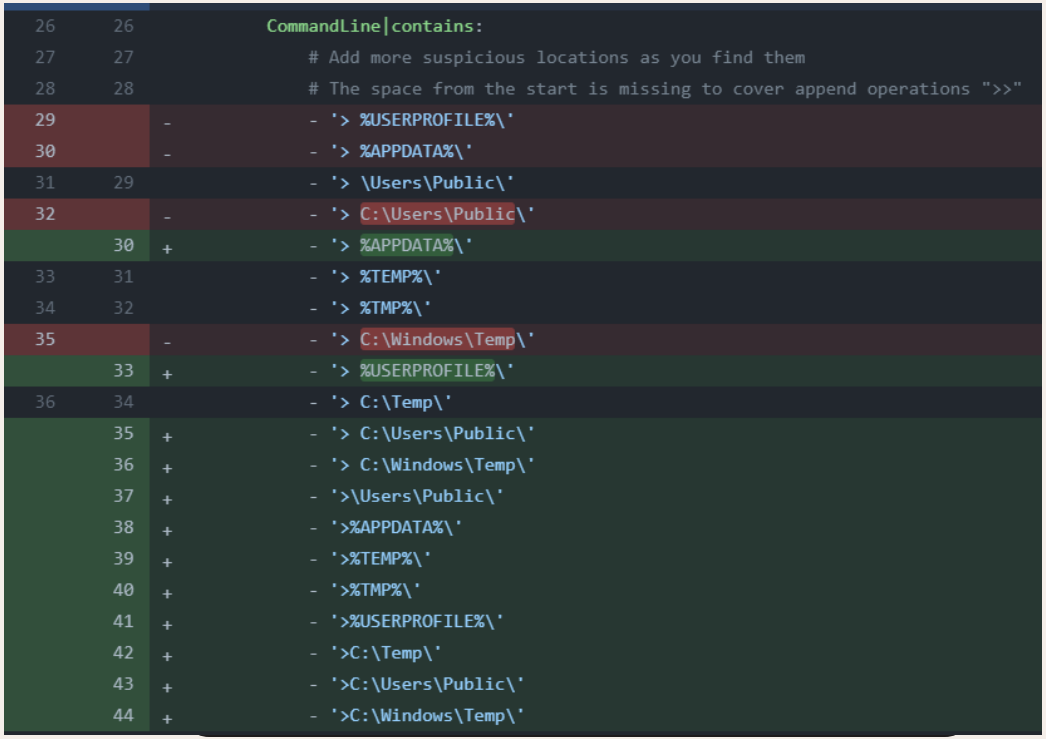
Track changes made to the detections with Git

Tests!

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

Automated!

Automate the testing and deployment of the detections

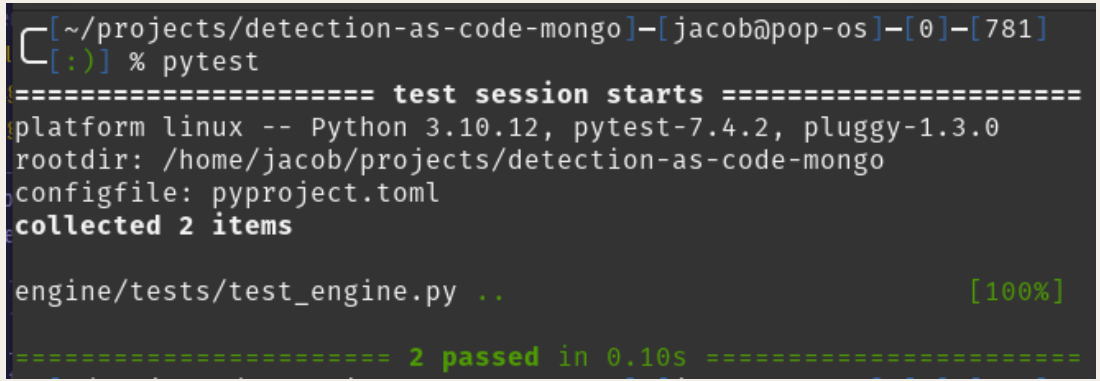


```
26  26      CommandLine|contains:
27  27          # Add more suspicious locations as you find them
28  28          # The space from the start is missing to cover append operations ">>"
29      -       - '> %USERPROFILE%\'
30      -       - '> %APPDATA%\'
31  29          - '> \Users\Public\'
32  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

Versioning

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

[1] https://github.com/SigmaHQ/sigma/commit/62caac47086968cc524860284c4d9b43447ba02d



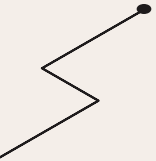
Testing

Ensure the detection will fire when conditions are met

```
┌─[~/projects/detection-as-code-mongo]─[jacob@pop-os]─[0]─[795]
└─[:)] % 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

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

```

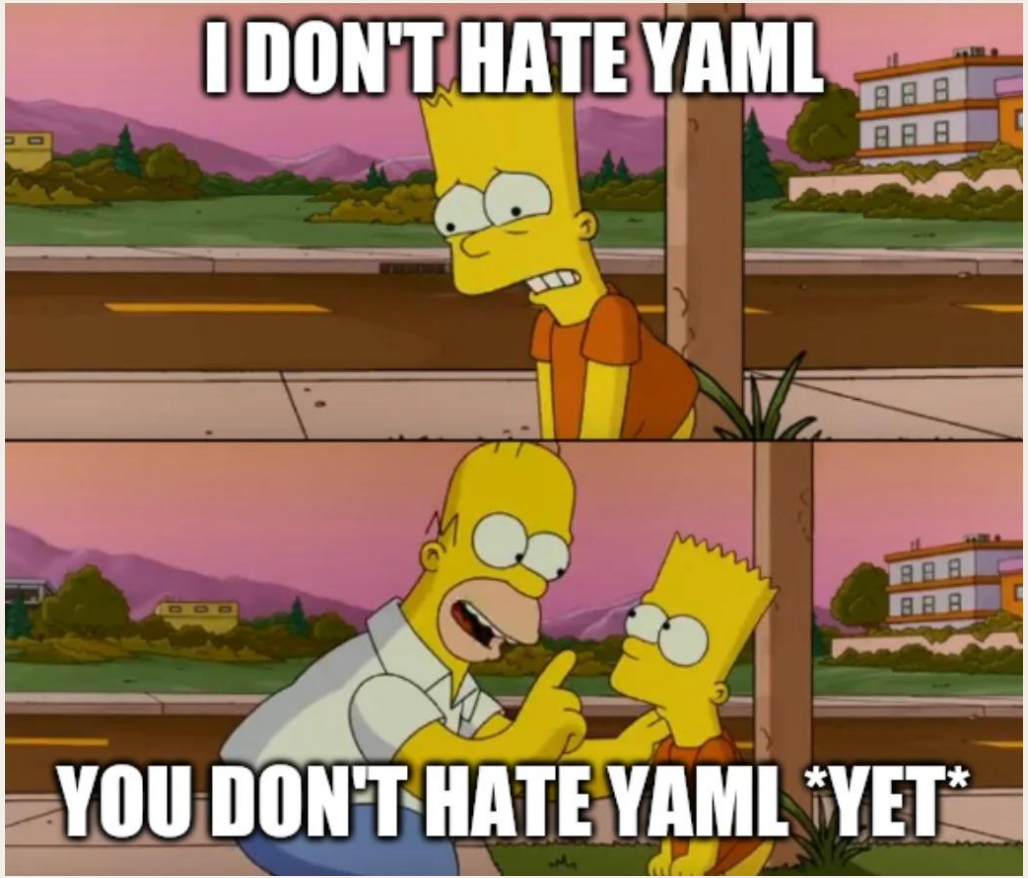
rules > ! tmp_output_redir.yaml

```yaml
name: Output redirection to /tmp/ Directory
uuid: f825ad39-d4b1-41da-91ec-6438d8dc9ace
detection_logic:
  command_line:
    - "> /tmp/"

```

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


I DON'T HATE YAML
YOU DON'T HATE YAML *YET*

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 > ...

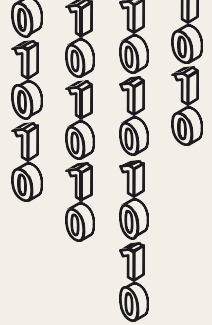
```python
def rule(event: dict) -> bool:
    if "/etc/shadow" in event.get("command_line"):
        return True
    return False

```

YAML Rule

rules > ! shadow.yaml

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

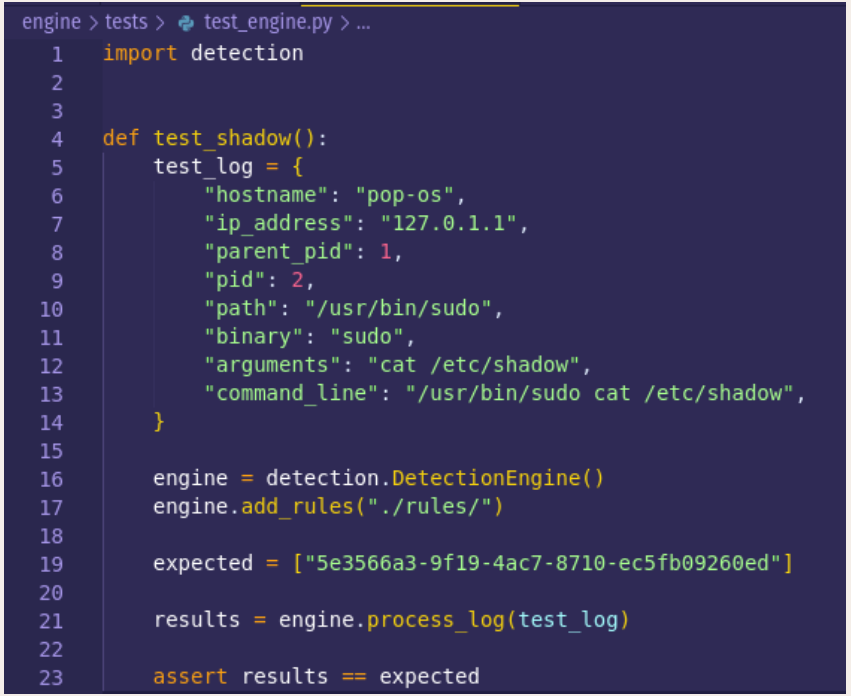
Events that should alert but do not

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

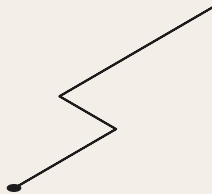


02
The Data

Data Source Options

- **Windows Event Logs**
- **Linux logs**
- **eBPF**

  This is the fun one ☺



Why eBPF?

- 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]

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

What's an eBPF Program look like?

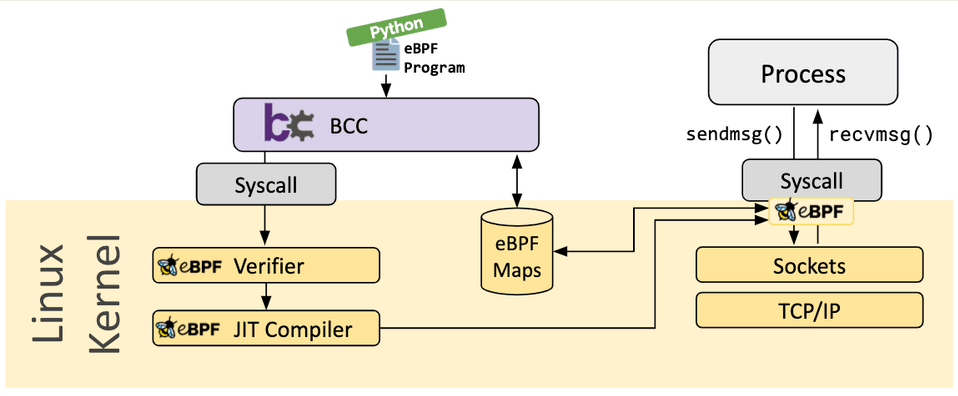
- tldr; it's a bunch of C code
- create a struct to store the details we want
- hook the system call
- send that data back

```
        task = (struct task_struct *)bpf_get_current_task();
        // Some kernels, like Ubuntu 4.13.0-generic, return 0
        // as the real_parent->tgid.
        // We use the get_ppid function as a fallback in those cases. (#1883)
        data.ppid = task->real_parent->tgid;
        bpf_get_current_comm(&data.comm, sizeof(data.comm));
        data.type = EVENT_ARG;
        __submit_arg(ctx, (void *)filename, &data);
        // skip first arg, as we submitted filename
        #pragma unroll
        for (int i = 1; i < 20; i++) {
            if (submit_arg(ctx, (void *)&__argv[i], &data) == 0)
                goto out;
        }
        // handle truncated argument list
        char ellipsis[] = "...";
        __submit_arg(ctx, (void *)ellipsis, &data);
    out:
        return 0;
    }
    int do_ret_sys_execve(struct pt_regs *ctx)
    {
        struct data_t data = {};
        struct task_struct *task;
        data.pid = bpf_get_current_pid_tgid() >> 32;
        task = (struct task_struct *)bpf_get_current_task();
        // Some kernels, like Ubuntu 4.13.0-generic, return 0
        // as the real_parent->tgid.
        // We use the get_ppid function as a fallback in those cases. (#1883)
        data.ppid = task->real_parent->tgid;
        bpf_get_current_comm(&data.comm, sizeof(data.comm));
        data.type = EVENT_RET;
        data.retval = PT_REGS_RC(ctx);
        events.perf_submit(ctx, &data, sizeof(data));
        return 0;
    }
```

[1] https://github.com/iovisor/bcc/blob/master/tools/execsnoop.py

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

```python
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

[1] https://www.mongodb.com/docs/manual/changeStreams/

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

[1] https://www.mongodb.com/docs/manual/changeStreams/

Change Stream Event

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

```
{
    "_id": "<resume token>",
    "operationType": "insert",
    "clusterTime": "Timestamp(1695255007, 1)",
    "wallTime": "2023-09-21 00:10:07.378000",
    "fullDocument": {
        "_id": "650b89dfcf00592980c13266",
        "hostname": "pop-os",
        "ip_address": "127.0.1.1",
        "parent_pid": 43729,
        "pid": 43748,
        "path": "/usr/bin/cat",
        "binary": "cat",
        "arguments": "/proc/43695/stat",
        "command_line": "/usr/bin/cat /proc/43695/stat"
    },
    "ns": {
        "db": "monitoring",
        "coll": "process"
    },
    "documentKey": {
        "_id": "650b89dfcf00592980c13266"
    }
}
```

What can I filter on?

- tldr; a lot

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

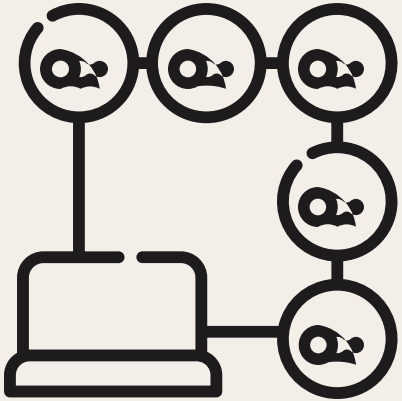
What can I filter on?

- tldr; a lot part 2

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

What we'll filter on today

insert	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

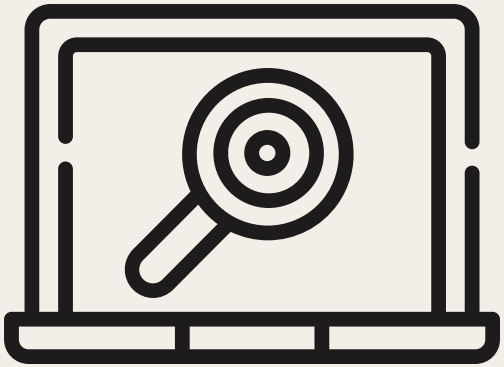
- If your detection engine crashes, needs to be restarted, etc., the engine needs to be restarted from the moment in time when it stopped processing events
- We can resume from that moment in time with a **resume_token**, accessible from the **change stream cursor**
- Keep track of the **resume_token**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

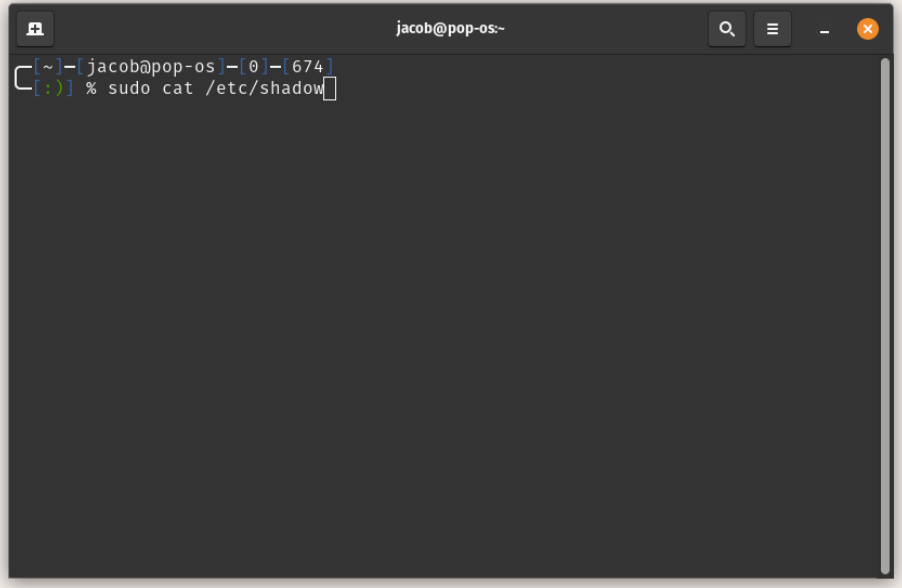
Testing

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



The actual detection!

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

An alert fired, now what?

Store Them!

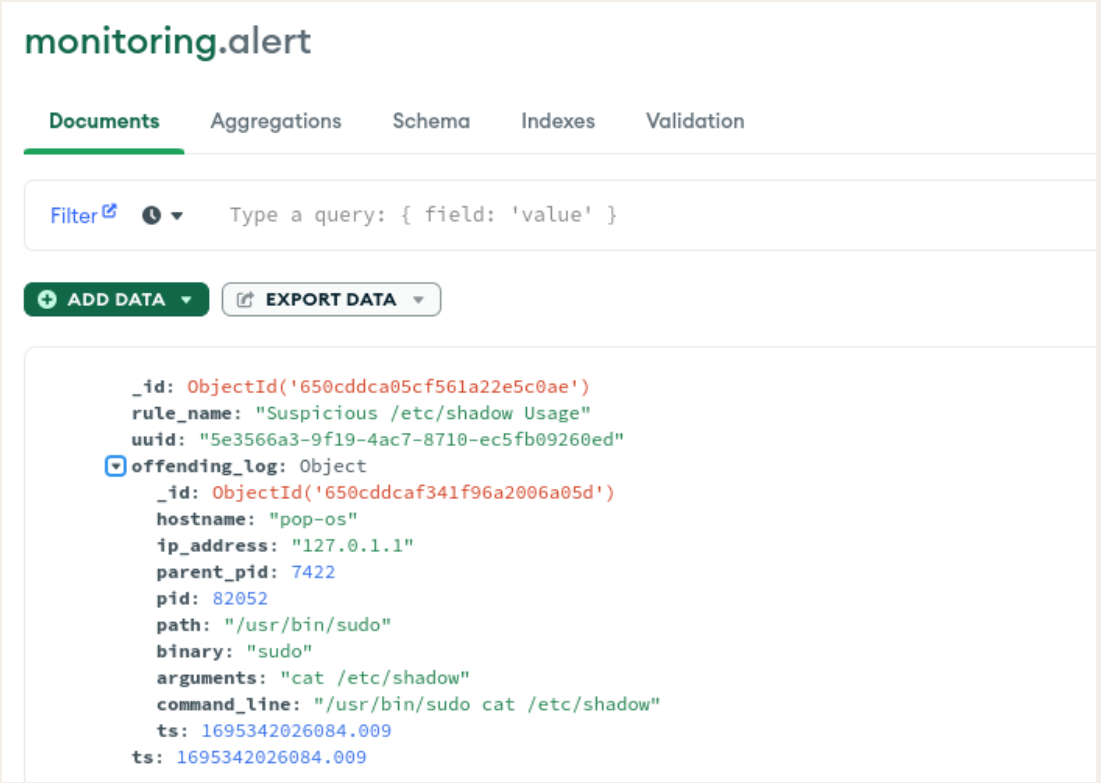
Over time, you'll likely amass quite a few detection alerts.

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



Pivot from the original alert

Use your timestamps!

```python
import datetime

timenow = datetime.datetime.now(datetime.timezone.utc)
five_before = (timenow - datetime.timedelta(minutes=5)).timestamp() * 1000
five_after = (timenow + datetime.timedelta(minutes=5)).timestamp() * 1000

```

Locate the process identifier (pid)

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
- BlueSky: @jacoblatonis.bsky.social
- LinkedIn: Jacob-latonis