Multitool Deduplicaton without LLMs

A story from the trenches

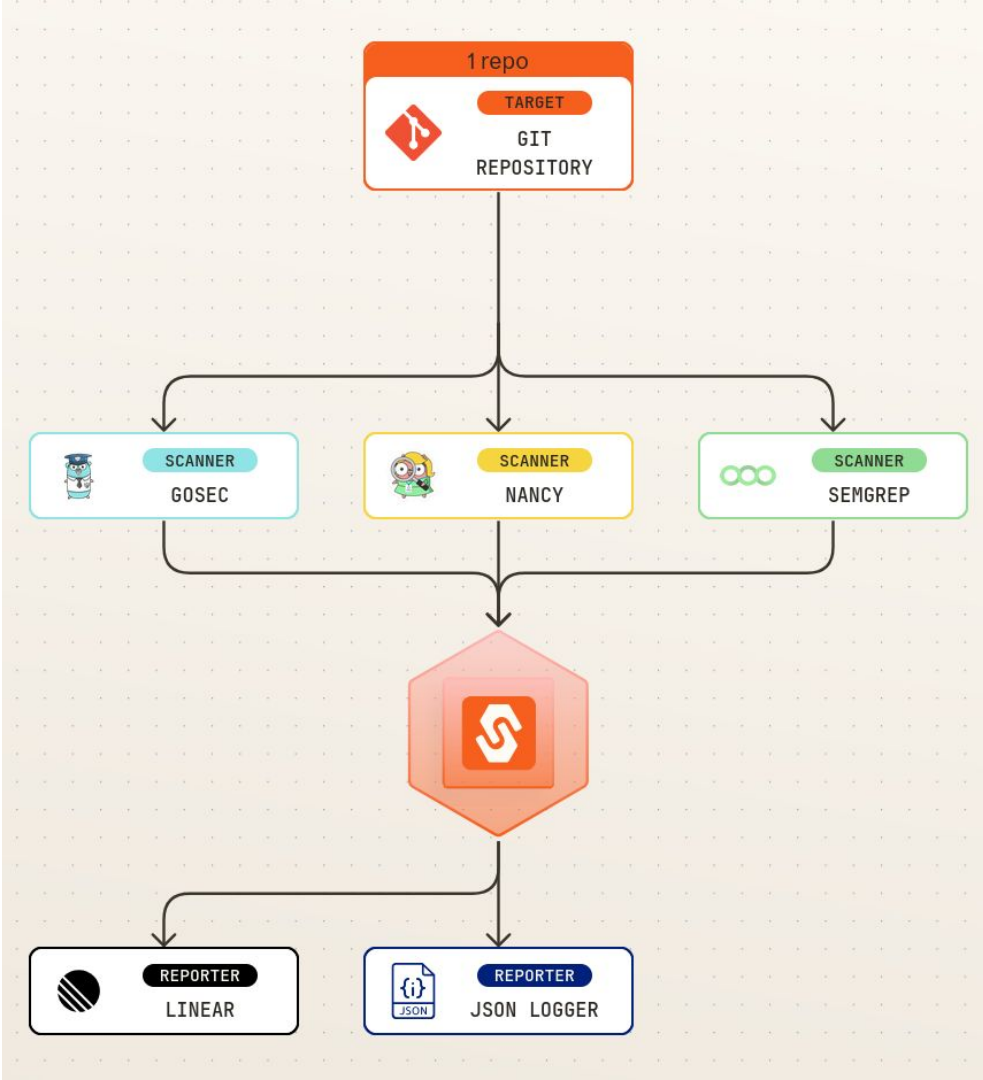
A story from the trenches

About me

- Started in the cybersecurity industry a decade ago
- Fell in love with data and working in small teams and startups
- Founding Data Engineer at Smithy Security

The Problem

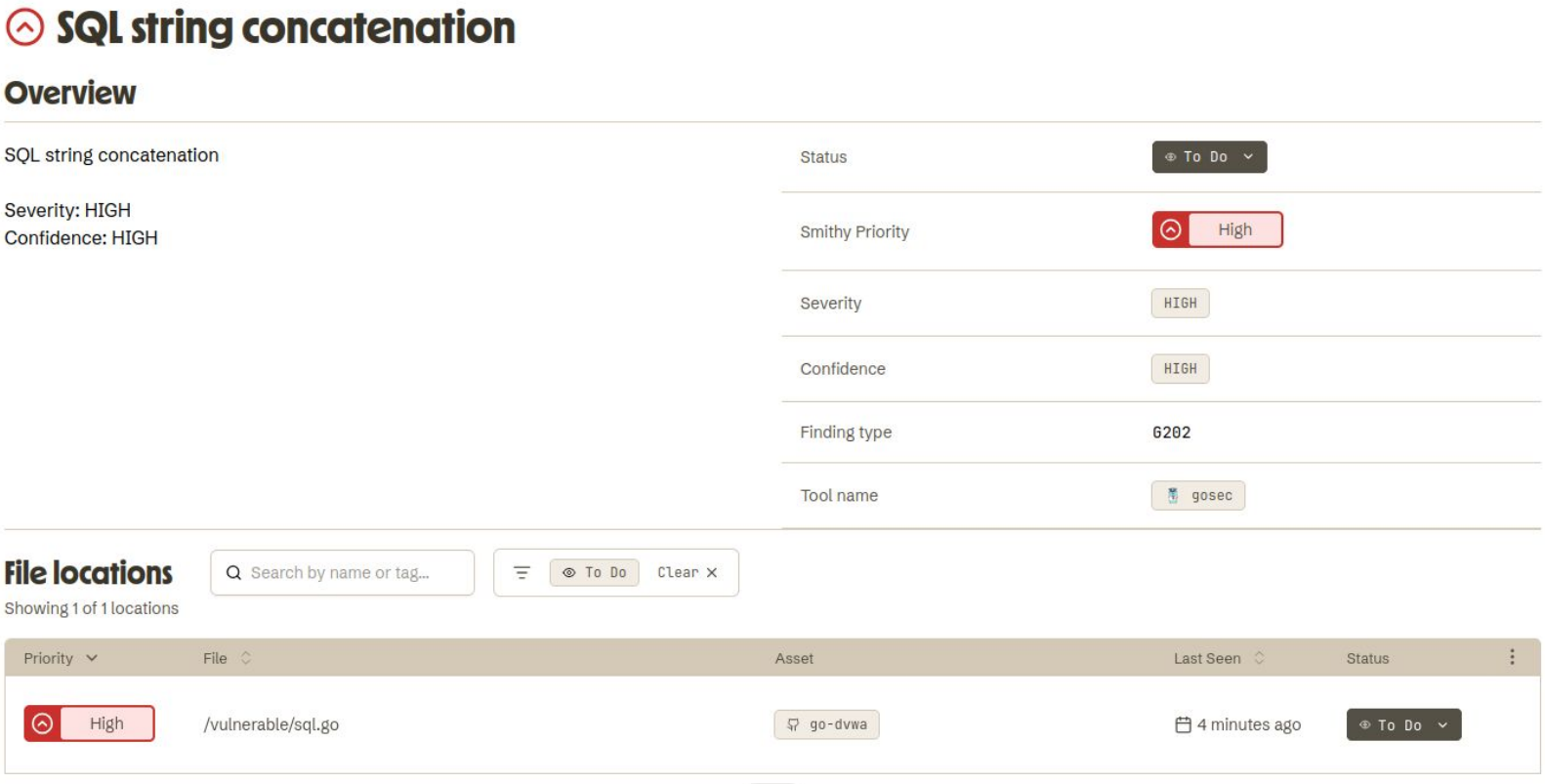
- Smithy Security is a workflow engine where we can scan assets with any number of tools
- Each tool independently raises findings on the same vulnerable piece of code
- We want to combine these findings automatically, so we're not raising duplicate tickets or notifications



How can we identify duplicates?

- Location
  - If the finding exists in the same place, there's a high chance it's related
  - It's possible that a single line or function has multiple different vulnerabilities
- CVEs
  - A good way of matching, but most tools don't report CVEs
- CWEs
  - Useful for relevance, but more general than a lot of the reported findings.
  - Most tools don't report CWEs
- Descriptions
  - Natural Language Processing required, and can be hard to give confident results
- 👷 Human in the loop?
  - We could use people to check and verify suggested results, but would be incredibly manual

A glance at the depth of the problem



A glance at the depth of the problem

Semgrep Finding: go.lang.security.audit.sqli.gosql-sqli.gosql-sqli

Overview

Detected string concatenation with a non-literal variable in a "database/sql" Go SQL statement. This could lead to SQL injection if the variable is user-controlled and not properly sanitized. In order to prevent SQL injection, use parameterized queries or prepared statements instead. You can use prepared statements with the 'Prepare' and 'PrepareContext' calls.

https://semgrep.dev/r/go.lang.security.audit.sqli.gosql-sqli.gosql-sqli

Status	To Do
Smithy Priority	Medium
Severity	MEDIUM
Confidence	UNKNOWN
Finding type	go.lang.security.audit.sqli.gosql-sqli.gosql-sqli
Tool name	Semgrep OSS

File locations

Search by name or tag... | To Do | Clear

Showing 1 of 1 locations

Priority	File	Asset	Last Seen	Status
Medium	/vulnerable/sql.go	go-dvwa	2 minutes ago	To Do

A glance at the depth of the problem

Status	To Do
Smithy Priority	High
Severity	HIGH
Confidence	HIGH
Finding type	G202
Tool name	gosec

Status	To Do
Smithy Priority	Medium
Severity	MEDIUM
Confidence	UNKNOWN
Finding type	go.lang.security.audit.sqli.gosql-sqli.gosql-sqli
Tool name	Semgrep OSS

Diving into NLP

- Use Python and spaCy (Python NLP module) to compare similarity
- Here we translate each individual word into a semantic vector, and average them to get the semantic vector of the whole sentence. We can then compare these to get the similarity of the sentences.
- Comparing our two previous descriptions:

```
Similarity between
'String-formatted SQL query detected. This could lead to SQL injection if the string is not sanitized properly. '
'Audit this call to ensure the SQL is not manipulable by external data.'
and
'SQL string concatenation'
: 0.4887
```

- That’s pretty good right?

No.

Comparing unrelated Gosec findings

```
Similarity between
'String-formatted SQL query detected. This could lead to SQL injection if the string is not sanitized properly. '
'Audit this call to ensure the SQL is not manipulable by external data.'
and
'Use of weak random number generator (math/rand instead of crypto/rand)'
: 0.7753
```

- Wordiness in the cybersecurity context is more strongly rewarded than the actual meaning

Preprocessing?

- Tidy up the text so it’s easier for the model to parse
- Extract noun phrases, lemmatise, remove stop words, and remove duplicate words in the noun phrases

```
Extracted keywords:
'SQL injection, string, external datum, string format SQL query, SQL'
'SQL string concatenation'
--> Chunked similarity: 0.7047

Extracted keywords:
'SQL injection, string, external datum, string format SQL query, SQL'
'crypto rand, Use, math rand, weak random number generator'
--> Chunked similarity: 0.7159
```

- Moving in the right direction, but still useless

A different approach

- We could instead use Sentence Transformers - these are LLMs optimised for semantic encoding of the entire text rather than individual words.
- As the entire sentence is understood, the average meaning of “cybersecurity” should be less significant in the output
- Comparing our SQL injection description: “String-formatted SQL query detected. This could lead to SQL injection if the string is not sanitized properly. Audit this call to ensure the SQL is not manipulable by external data.”

```
'SQL string concatenation': 0.4263
'Use of weak random number generator (math/rand instead of crypto/rand)': -0.0560
```

- This is actually a positive match, but not hugely confident
- This is *slow*. A corpus of 8 examples takes ~1 minute on a laptop

LLMs aren’t suitable

LLMs aren’t suitable

- Different models give worse results
- Models are hugely expensive and slow
- We have to scale to 100,000,000 findings per day, optimisations matter

Back to the basics

- After previous attempts at pre-processing, we had these results

```
Extracted keywords:
'SQL injection, string, external datum, string format SQL query, SQL'
'SQL string concatenation'
--> Chunked similarity: 0.7047

Extracted keywords:
'SQL injection, string, external datum, string format SQL query, SQL'
'crypto rand, Use, math rand, weak random number generator'
--> Chunked similarity: 0.7159
```

- These are obvious to a person, but the model can’t see it
- We can use Bag of Words instead 💰
  - This counts the frequency of each individual word and weights them across the corpus
  - The more the same word turns up, the more similar the results are

Testing it…

- We can build a corpus

```
corpus = [
    "String-formatted SQL query detected. This could lead to SQL injection if the
    "Use of net/http serve function that has no support for setting timeouts",
    "Potential file inclusion via variable",
    "By not specifying a USER, a program in the container may run as 'root'. This
    "This tag is missing an 'integrity' subresource integrity attribute. The 'inte
    "Do not use `math/rand`. Use `crypto/rand` instead."
]
```

- And given an unseen example, find the best match above a threshold

```
Query: 'Use of weak random number generator (math/rand instead of crypto/rand)'
Best match found: 'Do not use `math/rand`. Use `crypto/rand` instead.'
Similarity Score: 0.7301

Applying threshold of 0.35...
✅ Confident match found.
```

Further examples

```
Query: 'SQL string concatenation'
Best match found: 'String-formatted SQL query detected. This could lead to SQL inj
Similarity Score: 0.4954

Applying threshold of 0.35..
✅ Confident match found..
```

- Less confident but still accurate

```
Query: 'An attacker can cause excessive memory growth in a Go server accepting HT
Best match found: 'Use of net/http serve function that has no support for setting
Similarity Score: 0.1024

Applying threshold of 0.35...
❌ No confident match found. The best score was too low.
```

- Doesn’t flag on unrelated findings

Advantages

✅ Consistently gives the best match

✅ Doesn’t flag false positives

✅ Is lightning fast

✅ Can be trained and applied in separate steps

✅ Is explainable

The finished deduplicated issue

SQL string concatenation

Overview

Detected string concatenation with a non-literal variable in a "database/sql" Go SQL statement. This could lead to SQL injection if the variable is user-controlled and not properly sanitized. In order to prevent SQL injection, use parameterized queries or prepared statements instead. You can use prepared statements with the 'Prepare' and 'PrepareContext' calls.

https://semgrep.dev/r/go.lang.security.audit.sqli.gosql-sqli.gosql-sqli

Status	To Do
Smithy Priority	HIGH
Severity	High
Confidence	High
Finding types	G202 go.lang.security.aud… gosql-sqli show less
Tool names	gosec semgrep show less

Thank you!