SEMGREP RULE WRITING WORKSHOP

EXERCISES

JOSH GROSSMAN &
MICHAL KAMENSKY
BOUNCE SECURITY

Example 06 Exercise
Discovering weak comparisons

- Rule currently catches use of == and != instead of === and !==
- What if we want to change it to also catch accidental use of =
- Needs to be restricted to where this appears in an if statement
- Make changes/additions to the rule to address this
- You will know you have been successful if the unit tests pass

```
if (intNumber = intAdded)
{
    console.log("Appears
because code is incorrect")
}
```

https://semgrep.dev/playground/r/4bUrel/semgrep_bouncesecurity.ex06_dbg_pattern-either

josh@bouncesecurity.com michal@bouncesecurity.com

Bounce SECURITY

Example 07 Exercise
Discovering potential XSS

- Trying to search JavaScript files for potentially dangerous sinks
- We have a basic rule, but:
  - *It isn't finding all the unsafe code*
  - *It is also finding false positives*
- Make changes/additions to the rule to address this
- You will know you have been successful if the unit tests pass

https://semgrep.dev/playground/r/5rUbNq/semgrep_bouncesecurity.ex07_dbg_patterns

josh@bouncesecurity.com michal@bouncesecurity.com

Bounce SECURITY

Example 10 Exercise
Expanding entropy rule

- Looking for where hardcoded data is found
- We have a rule, but it currently shows where the high entropy variable is used.
- We want it to also highlight where the high entropy string is assigned to a variable.
- Make changes/additions to the rule to address this
- You will know you have been successful if the unit tests pass

https://semgrep.dev/playground/r/JDU8lz/semgrep_bouncesecurity.ex10_dbg_entropy

josh@bouncesecurity.com michal@bouncesecurity.com

Bounce SECURITY

Example 17 Exercise
Taint mode rule

- Similar to example 07, looking for flows from untrusted sources to dangerous sinks
- Current rule is missing the following sources:
  - *window.location.href*
  - *potentiallyDangerous*
- Also missing the following sanitizers:
  - *CustomSanitize*
  - *potentiallyDangerous.selfClean*
- You will know you have been successful if the unit tests pass
- (Hint: When is a sanitizer not a `pattern-sanitizers`...)

https://semgrep.dev/playground/r/qNU3Qg/semgrep_bouncesecurity.ex17_dbg_taint-sanitize-advanced

josh@bouncesecurity.com michal@bouncesecurity.com

Example 21 Exercise
Detecting missing decorators

- Python code using “decorators” for authorization
  - `@PD.require_update`
  - `@PD.require_read`
- We want to find the places in the SensitiveFunctions and the MoreSensitiveFunctions classes where they forgot
- Current rule finds where they *did* add a decorator
- Make changes/additions to the rule to address this
- You will know you have been successful if the unit tests pass

https://semgrep.dev/playground/r/yyUKNp/semgrep_bouncesecurity.ex21_dbg_decorator

josh@bouncesecurity.com michal@bouncesecurity.com

Example 22 Exercise
Detecting missing SQLi sanitization

- We have a quite complex rule to detect SQL injection in C#
- The code is using custom sanitization using **Replace** and **Utility.SQLSanitize.**
- Create a taint mode rule that will also find SQL injection but will not match places where the input was sanitized
- You will know you have been successful if the unit tests pass

https://semgrep.dev/playground/r/zdUEWd/semgrep_bouncesecurity.ex22_dbg_sqli

josh@bouncesecurity.com michal@bouncesecurity.com

Bounce SECURITY

Example 23-01 Exercise
Discovering part of XSS

- This code demonstrates some content being received from some function and being unsafely written to the page.
- We want to catch the line where the dangerous input enters the application before it is then used in the dangerous document.write sink.
- (Maybe use a simple rule, not taint mode.)
- You will know you have been successful if unit tests pass

https://semgrep.dev/playground/r/pKU5Ly/semgrep_bouncesecurity.ex23-01_dbg_sink_join

josh@bouncesecurity.com michal@bouncesecurity.com

Bounce SECURITY

Example 23-02 Exercise
Discovering another part of XSS

- This code the original place where this unsafe input is coming from.
- We want to catch the line declaring the function where the dangerous input is being returned.
- We don’t want to include a function declaration where data is not coming from an untrusted source.
- You will know you have been successful if unit tests pass

https://semgrep.dev/playground/r/X5Ur2z/semgrep_bouncesecurity.ex23-02_dbg_source_join

josh@bouncesecurity.com michal@bouncesecurity.com

Bounce SECURITY

Example 23-03 Exercise
Putting it all together with a join rule

- Whilst the previous rules returned a few results each, there is actually only one exploitable path.
- Use a join mode rule to link the previous two rules together and find the only actually exploitable path.
- Note: A join mode rule will return one line only, either from one file or the other. However, if you reverse the order of the “on w.x == y.z” line, it will return the relevant line in the other file.

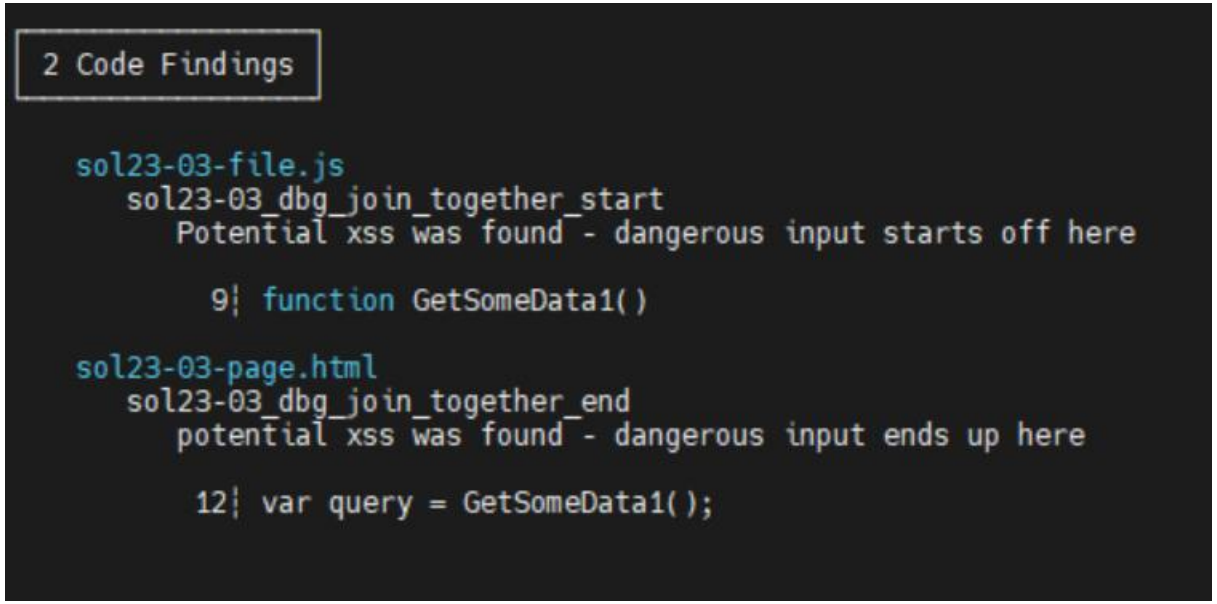
https://github.com/semgrep-bouncesecurity/exercises/tree/main/ex23-03-dbg_join_together

josh@bouncesecurity.com michal@bouncesecurity.com

Bounce SECURITY

Example 23-03 Exercise
Putting it all together with a join rule

- This is example output from running two copies of the same rule together with a different “on” order.
- These are the lines you should be finding.



josh@bouncesecurity.com michal@bouncesecurity.com
Bounce SECURITY

Example 25 Exercise
Running it on WebGoat

- Create a taint rule to find certain XSS issues in WebGoat.NET (a known vulnerable application)
- A results.html file will show you the results we are looking for.
- WebGoat.NET is already in the folder.
- You will need to figure out the relevant sources, sinks, and sanitizers and how to write the rule to look for them in order to get similar output to the results file.

https://github.com/semgrep-bouncesecurity/exercises/blob/main/ex25-dbg_webgoat_xss/ex25-dbg_webgoat_xss.yml