FINDING BUGS AND SCALING YOUR SECURITY PROGRAM WITH SEMGREP

Brandon Wu

June 16, 2024

1 - Introduction

Link

The finest of parkour runners know the importance of guardrails.

Here at OWASP AppSec Days PNW, we are no different. In the absence of **secure guardrails**, the only difference between a parkour runner and a security engineer is in the time elapsed to hit the ground.[1]

Def The idea of **secure guardrails** is in making security the **default** behavior, rather than something that is **opted into**. In the lifecycle of software development, it champions guiding developers into secure practices **early**, rather than chasing down vulnerabilities once they have already occurred.

[1] He survived.

To understand the importance of secure guardrails, we must look into the mind of a developer. Fortunately, as a developer myself, I can shed some light into the thoughts, in decreasing priority, in a typical developer's head:

1. I wonder how I could solve this problem
2. I wonder which of those solutions is the most maintainable
3. I wonder which of those solutions is the most simple
4. I wonder how long I can take before product managers yell at me
5. I wonder when the climbing gym closes
6. I wonder how to make this feature have good UX
7. I wonder when lunch is

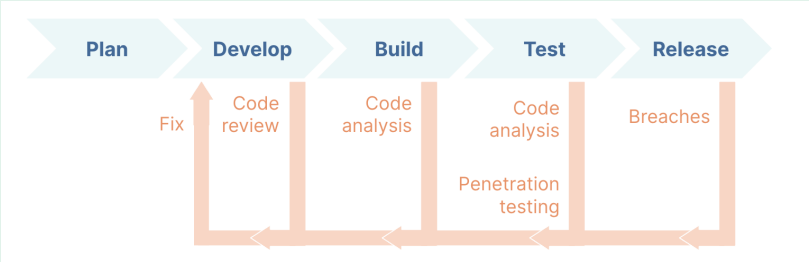
...

23. I wonder how to make this feature more secure

Developers care about many things, but security is not typically one of them.

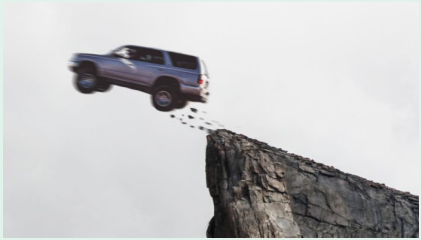
Dealing with a typical software engineer can be much the same as dealing with a rip current. It is futile to swim against it. Instead, we must guide the waves to a place they can cause a little less destruction.

The goal is to make security part of the developer's typical workflow – in their IDE, in their pre-commit, in their CI/CD pipeline. This is the idea of **shift left**.



Secure guardrails are not just for the convenience of developers – they are for *everyone*.

The game of security is a constant struggle of prioritization. Between audits, pentests, and triaging vulnerabilities, there is an endless stream of noise to sift through.



"A gloomy security engineer at a large bank once explained to me that if development stopped today, he had calculated it would take over 100 years for them to get through the vulnerability backlog." - Isaac Evans

The reality is that there is little time to cover an increasingly broad attack surface. Security teams literally cannot afford to spend their time firefighting when there are so many other things to do.[2]

[2] In the real world, there are very few things you can do once the car has already driven off the mountain.

They say that the definition of insanity is doing the same thing over and over and expecting a different result.[3]

It is not enough to keep chasing vulnerabilities as they come. We must come up with remediation strategies that extend into the *future*, because time is not on our side, and the backlog only grows larger.

"Knocking him down won the first fight. I wanted to win all the next ones, too." - Ender Wiggin, notable proponent of secure guardrails

The goal, then, is to simply stop the problem at its source. We must build the guardrails ourselves.

[3] Others say that the definition of insanity is continuously reusing this clichéd definition of insanity.

My name is Brandon Wu, and I am a program analysis engineer working at Semgrep.

I have been working at Semgrep for two years now, and I was educated in computer science at Carnegie Mellon University, where I previously lectured on the subject of functional programming.

Semgrep is a software security startup and application security platform that helps developers find and fix security vulnerabilities in their code, at minimal friction to their workflow.

Mission To profoundly improve software security.

@onefiftyman

LinkedIn

Make it **cheap** to make it **expensive** to write secure software.

or, less eloquently:

Prevent people from **shooting themselves in the foot**.

Menti code: 1226 8048

Go to menti.com to ask questions anonymously!

Some questions to get us started:

- What does everyone here do?
- Who here has heard of Semgrep before?
- Has anyone here written a Semgrep rule before?[4]
- Did anyone come to a previous Semgrep training?

[4] "No" is an acceptable answer.

2 - Static Analysis Essentials

We're here because we are interested in **software security**.

Broadly, we are interested in categories of undesirable behaviors in programs, and in particular, we are interested in minimizing them wherever possible. These may take the form of:

- **logic errors** - passing an ill-typed argument to a function
- **code smells** - using a deprecated utility function instead of a more up-to-date one
- **security vulnerabilities** - SQL injection, cross-site scripting, broken authentication

All of these are fair game for things that we want to find and fix. Our weapon of choice for doing so is **static analysis**.

Def **Static analysis** is the art of analyzing programs without ever running them.

This is in contrast to **dynamic analysis**, which involves running the program and observing its behavior at runtime. While an interesting field in its own right, dynamic analysis has its own disadvantages, so it is out of scope for this workshop.

You may think of the difference between static analysis and dynamic analysis by analogy to the problem of trying to ascertain if a gun is faulty.

Static analysis would be checking the gun for defects by taking it apart and examining its mechanical structure.

Dynamic analysis would be firing the gun and observing if it explodes in your hand. [5]

[5] This is only slightly a misrepresentation, but the problem is that a gun not exploding *now* does not mean it will not explode *later*.

Static analysis should be:

1. **fast** - Scanning code with a static analysis solution should run quickly, so that developers can get feedback and iterate as quickly as possible. [6]
2. **fitting** - Static analysis should be customized to *your* use case. Teams, codebases, and organizations are *not* one-size-fits-all. You should be able to scan your code in a **way that works for you**.
3. **friendly** - Static analysis tools should be simple to use, for both developers and security engineers. A tool which is not understood is a tool which will be disabled.

These are all things that Semgrep aims to achieve.

[6] The car's moving at 90mph. It's not going to wait for you to lay down some brickwork.

3 - Basics: Conceptual Fundamentals

Here's some things about Semgrep that are cool:

- **open-source**, with community-contributed language support and rules
- **customizable** - Semgrep scans for code patterns that you define, and provide as an argument to the tool. This means that Semgrep can be customized to your code base's idioms, and tuned to your organization's security needs.
- **multilingual** - Semgrep supports over 25+ languages
- **no building required**. Semgrep only needs source code to run, and doesn't require that the code it is analyzing can build
- **No DSL**. Semgrep patterns look like code in the language you're targeting, meaning no hours of browsing documentation and API references. All you need to be able to do is read and write code.[7]

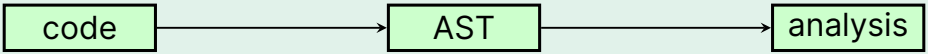
[7] This is really a killer, and will form the basis of this workshop as we move into Semgrep rule-writing.

Here are some useful Semgrep-related references, when it comes to utilizing the platform:

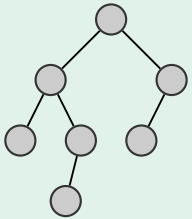
- The Semgrep open-source repository, where you can read the code I write[8]
- The Semgrep brew formula, where you can pull the Semgrep CLI to your computer via `brew install semgrep`
- The Semgrep VS Code Extension, which allows developers to scan code directory from their IDE, **before it is committed**.
- The Semgrep Rule Registry, which contains all the pre-written rule packs from Semgrep's security research team, as well as community-contributed rules
- The Semgrep docs page, which contains information on how to run Semgrep in your CI/CD pipeline, how to write rules, and other useful information[9]

[8] Please don't.

[9] This is mostly here as a reference for *after* the workshop. Leave that link blue for now.




```
b = 1
def foo(a):
  return a + b
```

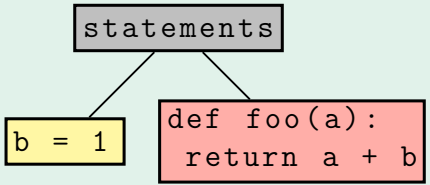


- pattern matching
- IL translation
- taint analysis
- constant propagation

Consider the simple program seen on the previous page. To analyze it, we could look at the text directly, or we could notice that the shape of the program forms a kind of hierarchical structure, called an **abstract syntax tree (AST)**.

For instance, we note that the program is made of two statements, which could be regarded as the children of a parent node.

```
b = 1
def foo(a):
  return a + b
```

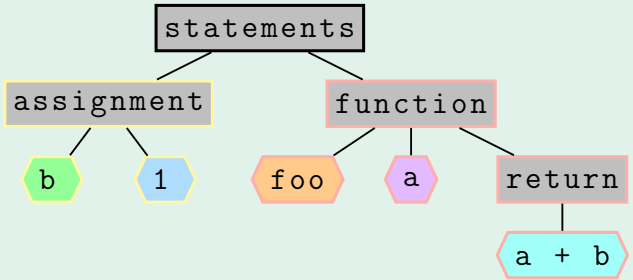


Important We say that the AST nodes of the assignment to `b` and the definition of the function `foo` have a **range**, which is the contiguous span of text in the source file that it covers. These are highlighted in yellow and red.

But, this isn't the farthest granularity we can go. We see that the assignment to `b = 1`, for instance, is made up of parts also. We could regard the identifier `b` and the literal `1` as children of yet another node, this time for an assignment.

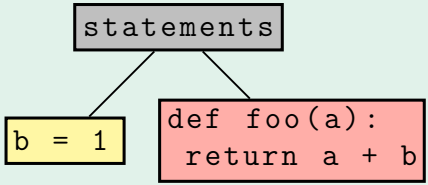
Our new tree will look like a more detailed version of the previous one:

```
b = 1
def foo(a):
    return a + b
```

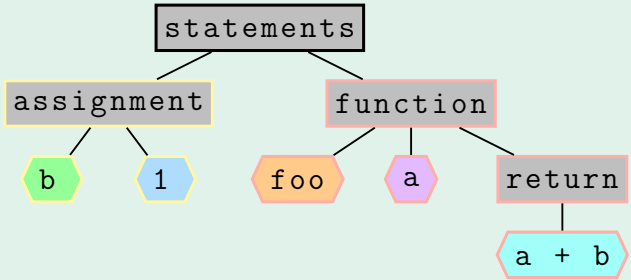


Keep in mind that the old assignment node still exists! It just comprises the entire subtree of the assignment, which is now highlighted in yellow. The same is true of the function node.

```
b = 1
def foo(a):
  return a + b
```



```
b = 1
def foo(a):
  return a + b
```



Semgrep is about **matching nodes** and **obtaining ranges**.

Suppose we were interested in matching calls to `print` in Python. We could be concerned with the following program:

```
# this line will print
print("Hello, world!")

isprinted(True)

print("printing")
```

We could use `grep` or `ctrl-f` to find these calls, but they won't work properly!

By grepping for print, we would get the following:

```
# this line will print
print('Hello, world!')

isprinted(True)

print('printing now')
```

We were interested in finding **calls** to print. We see that we have three matches we didn't intend – three **false positives**.

The reason is that grep is not aware of whether each contiguous print is a call or not. It just operates purely based on characters. Put another way, grep is not **semantic**.

Semgrep stands for **semantic grep**. It is a code-searching tool that is aware of the underlying structure of the code – that is, the AST.

Consider the following example:

```
print
```
Pattern

```
# this line will print
print("Hello, world!")
isprinted(True)
print("printing")
```
Target

With Semgrep, we will match the two calls to `print`, and nothing else.

```
print
```
Pattern

```
# this line will print
print("Hello, world!")
isprinted(True)
print("printing")
```
Target

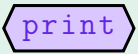
Note that it is not simply a matter of ignoring the comments and string literals – we also do not match the identifier `isprinted`, because it is not the same as the identifier `print`.

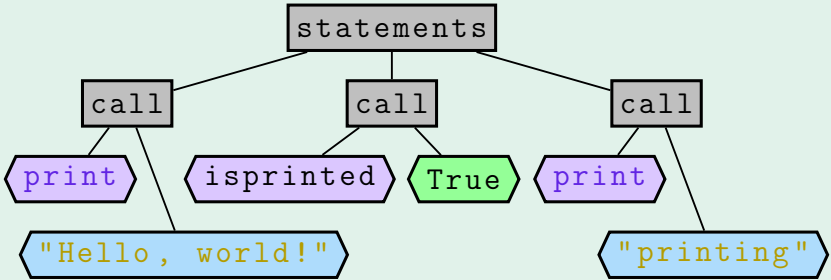
```
print
```

Pattern

```
# this line will print
print("Hello, world!")
isprinted(True)
print("printing")
```

Target



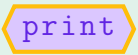


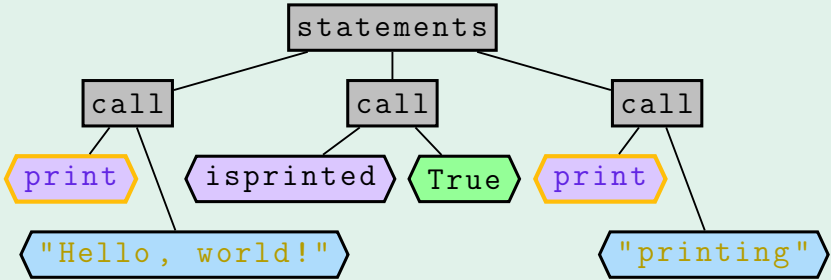
```
print
```

Pattern

```
# this line will print
print("Hello, world!")
isprinted(True)
print("printing")
```

Target





Semgrep works by doing three things:

1. parsing the **pattern** to an AST
2. parsing the **target** to an AST
3. **matching** both against each other

Since there are only two nodes on the right-hand side which look like the `print` node, they are the only ones which end up being printed.

Note that the `isprinted` node is **not** matched, because it contains different text!

4 - Basics: Patterns

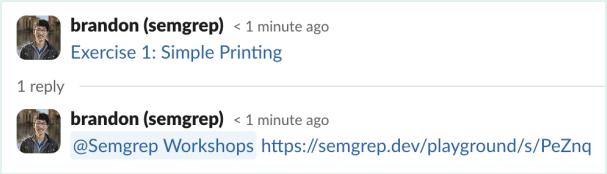
In today's workshop, we will be doing exercises via the Semgrep Playground, which is a web-based interface for writing and testing Semgrep rules.

You can go to it by going to, very simply, https://semgrep.dev/playground.

Important We are also going to implement an autograding schema and a little friendly competition, for funsies. To participate in the workshop, you will have to join the Semgrep Community Slack, which can be found at https://bit.ly/semgrepslack.

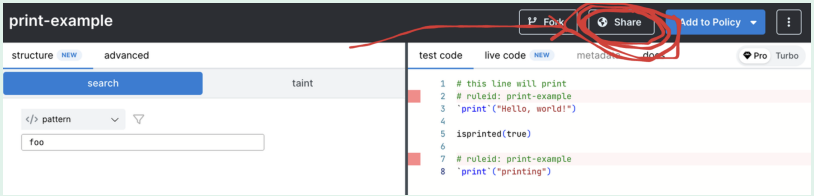
Once you are in the Slack, join the `2024-owasp-pnw-training` channel.

Every exercise, a new thread will be posted in the channel. To submit a solution, reply in the thread by @ing the Semgrep Workshops account, followed by the permalink to your solution.



This runs an autograding script, which awards points based on correctness and speed. **There are prizes at the end, so make each second count!**

You can generate a permalink for your rule by going to the "Share" button, and then clicking the "Permalink" toggle and copying the link.



Exercise: Try it for yourself!

Try writing the pattern we just saw within the Semgrep playground. [a]

```
# this line will print
# ruleid: print-example
print("Hello, world!")
isprinted(true)
# ruleid: print-example
print("printing")
```

Note In the Playground page, you will see a couple of tabs, notably ones labeled `advanced` and `taint`. Leave those alone for now.

[a] Solution

It's worth expanding on the strange comments that are present in the target source.

Feature In the Semgrep playground, you can add **test annotations** to your target, which will tell the Playground whether a match is expected or not on the next line:

```
# ruleid: rule-id-here
match_expected()
# ok: rule-id-here
match_unexpected()
```

This is useful for verifying that your rule is working as expected. We must obtain a match on line 2, and no match on line 4, or else the Playground will throw an error and tell you that tests are failing. This is also useful for conveying what you intend your rule to do!

Our solution has a bug in it, though! We only want to match **calls** to the **print** function, not the identifier `print` itself.

This causes a bug in the following example:

```
print
```
Pattern

```
# this code should be matched...
print("a call to print")
# but we also produce a match here!
alias = print
```
Target

Recall the mantra of Semgrep: **patterns look like code**. To write a pattern which matches call expressions, we will write a pattern which itself looks like a Python call.

To do this, we will need to somehow vary our pattern over any argument to the call. We do this with a special Semgrep syntax, the **ellipsis**.

Feature The **ellipsis** is a placeholder in a Semgrep pattern which matches zero or more occurrences of some element, for instance parameters, arguments, and statements.

```
print(...)
```

Pattern

```
# this code should be matched...
print("a call to print")
# now, no more match here:
alias = print
```

Target

Note The range has changed! Instead of matching just the identifier `print`, our match now covers the entire call.

When put as arguments to a function call, ellipses allow you to match *any number* of arguments, even 0.

Behold the following example showcasing the mighty ellipsis, paired with an equally mighty haiku[9]:

```
print(...)
```

Pattern

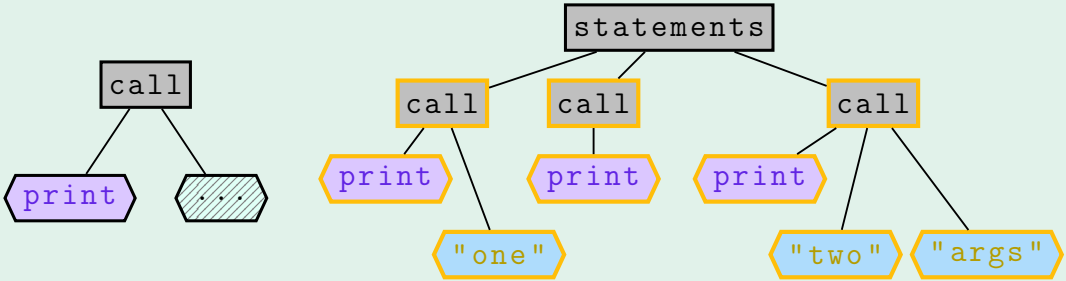
```
# no matter the count
# when up against arguments
# ellipsis matches
print("one")
print()
print("two", "args")
```

Target

[9] The slide author would like you to know this was not generated by GPT.

It's worth clarifying why it is that Semgrep is producing these matches!

When compiled to their respective ASTs, the pattern and target look something like this:



Here, we produce three matches, because the right tree has three subtrees which look the same as the left tree. They look slightly different, but the ellipsis node has special rules, and is able to match the rest of the call.

It is also worth clarifying another point: the reason why the match has expanded from just `print` to the entire call is because we match the entire `call` node, which has a range comprising the entire call, including the parentheses.

You don't need to be able to picture these trees in your head, you just need to know that ultimately, all we are doing is matching nodes which have certain ranges.

Exercise: Marking banned, deprecated, or dangerous functions

Let's try an example out using the ellipsis. [a]

```
import exec as definitely_safe_function

# ruleid: python-exec
definitely_safe_function(user_input)
# ruleid: python-exec
exec()
# ruleid: python-exec
exec("ls")
```

Keywords: ellipsis

[a] Solution

Something I feel that I must get across to you, that people often get wrong, is that **the ellipsis is not magic**.[9] Consider the following pattern:

```
def ...
```

Pattern

This looks like it should match any function definition, because we match a `def`, and then a sequence of things, which would match a function like:

```
def foo():
  return
```

Unfortunately, this isn't how Semgrep works at all, and will result in a pattern parse error.

[9] Friendship is, though.

In most cases, an ellipsis is **only admissible** wherever you could put a sequence of parameters, arguments, or statements.

Some might be tempted to think of it like `.*` in regex, but it's not the same! Recall that patterns are *parsed*, meaning that Semgrep patterns must *look like code* in the language they are a part of.

What Python construct looks like `def ...`, where `...` is a sequence of arguments, parameters, or statements? The answer is **none**, and so the pattern is invalid.

Above all else, remember: **patterns must parse**!

Sometimes, we are only interested in functions with just a single argument, though. In these cases, ellipsis is too permissive.

We need a construct which denotes a single element in the program, kind of like a placeholder. We will call this a **metavariable**.[10]

Feature The **metavariable** is a placeholder in a Semgrep pattern which matches exactly one occurrence of some element, for instance an argument, expression, or identifier.

The syntax for a metavariable is a dollar sign prefixing a capital identifier, for instance `$X` or `$EXPR`. Usually, you can put it anywhere in a pattern that you could ordinarily put an identifier.

[10] It's like a regular variable, but it drinks and goes to parties.

Using the ellipsis and metavariable we learned about earlier, we can try to write a pattern which matches any Python function.

```
def $FUNC(...):
  ...
```

Pattern

```
def foo():
  pass

def bar(a, b):
  return a + b
```

Target

Note that we only put the ellipsis in place of arguments and statements!

Exercise: Enforcing secure cookies

Let's use our newfound knowledge to tackle a real use case – determining if a cookie's `setSecure` method was called with a `false` value. [a]

Remember: Semgrep patterns look like code!

Keywords: metavariable

[a] Solution

Exercise: Enforcing secure named arguments

Finally, let's use the ellipsis operator to look at examples of functions called in an improper way.

In particular, we are concerned with the Python `requests.get` function, which is a potential security concern if called with `verify=False`. [a]

```
# ruleid: requests-get-verify
requests.get("https://example.com", verify=False)
```

Keywords: ellipsis

[a] Solution

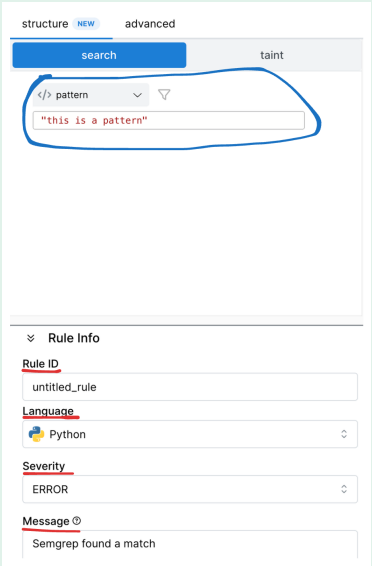
5 - Basics: Pattern Operators

Now that we have seen basic Semgrep patterns, we can go on to constructing actual Semgrep rules.

A Semgrep rule is a YAML file which contains a Semgrep pattern, and a few other things:

- a **message**, which is a string which will be displayed when a match is found
- a **severity** (`INFO`, `WARNING`, or `ERROR`), which describes how severe findings from the rule are
- a **rule ID**, which is the name of the rule
- a **languages** field, which specifies the languages the rule is written for

We will be using the Playground's Structure Mode, which will generate the YAML for us from an interface. Note the pattern in blue and other fields in red.



Sometimes, when we are trying to match something, we are interested in combining multiple matches in some way.

Usually, this we want to match something which is **one of several things**, or **all of several things**.

Feature The `all` and `any` operators are patterns which combine other patterns. `all` matches ranges where all of the children patterns match, and `any` matches ranges where at least one of the children patterns match.

```
any:
  - "exec(...)"
  - "print(...)"
```

Rule

This pattern matches **both** calls to `exec` and calls to `print`.

`any` operates under **set union** rules, where each of its children produces a set of matches, which are simply unioned together.

```
any:
  - "exec(...)"
  - "print(...)"
```

Rule

```
exec("ls")
print("hi")
```

Target

$\{ \square \} \cup \{ \square \} = \{ \square, \square \} \implies$ **2 matches**

```
any:
  - "exec(...)"
  - "print(...)"
```

Rule

```
exec("ls")
print("hi")
```

Target

`all`, on the other hand, operates under **set intersection** rules (or similar to it), where each of its ranges are intersected with each other, keeping the smaller ones.

```
all:
  - "exec(...)"
  - "print(...)"
```

Rule

```
exec("ls")
print("hi")
```

Target

$\{\ \square\ \} \cap \{\ \square\ \} = \{\ \} \implies$ **no matches**

```
all:
  - "exec(...)"
  - "print(...)"
```

Rule

```
exec("ls")
print("hi")
```

Target

So we could write the above rule, but it's not very useful – **it will match nothing**, because there is no way for a range to be both a call to `exec` *and* a call to `print`.

Instead, the `all` operator is most useful when combined with `not`.

Feature The `not` operator, when placed under an `all`, means the result should be anything which *does not* match the pattern under the `not`.

With this, we can express complementary logic, such as "all calls which are not the print function".

```
all:
  - $FUNC(...)
  - not: print(...)
```

Rule

```
print("hi")
foo()
exec("ls")
```

Target

Exercise: Order of API calls must be enforced

In this example, we are working in a financial trading application where we need to ensure that `verify_transaction` is called on any transaction, before we call `make_transaction`. [a]

```
// ruleid: find-unverified-transactions-exercise
public void bad(Transaction t) {
  make_transaction(t);
}
```

Keywords: ellipsis, not, all

[a] Solution

Exercise: Hardcoded secrets detection

We want to catch initializations of a `PasswordAuthenticator`, imported from a Python package `couchbase_core.cluster`. In particular, we are interested in when the password passed to the function is a hardcoded, nonempty literal string. [a]

Feature A pattern of the form `"..."`, with the quotes, is a **literal ellipsis**, and will match all instances of literal strings.

```
# ruleid: python-couchbase-hardcoded-secret
PasswordAuthenticator('username', "password")
```

Keywords: literal ellipsis, not, all

[a] Solution

So far, we have discussed four pattern operators in `pattern`, `not`, `all`, and `any`.

From the dropdown, we can see there are two more, `inside` and `regex`. These form the fundamental six pattern operators which are used to produce ranges.

Feature The `inside` operator is used for when you want to specify that a match should be found only when it is a subrange of the pattern under the `inside`.

```
all:
  - inside: |
      def $FUNC(...):
        ...
  - pattern: |
      structlog.get_logger()
```

Rule

```
logger = structlog.get_logger()

def foo():
  logger =
    structlog.get_logger()
```

Target

Feature The `regex` operator matches ranges of **text** which match the regular expression.

The `regex` operator is the sole exception when it comes to patterns matching AST nodes to obtain ranges. `regex` is like `grep`, and matches purely the text. It's useful as an escape hatch, but usually isn't the best solution.

The following rule matches all instances of the string "import" within functions.

Rule

```
all:
  - regex: "import"
  - inside: |
      def $FUNC(...):
        ...
```

Target

```
import structlog
import typing

def foo():
  import re
  imported_function = re.match
```

6 - Advanced: Conditions and Focusing

Now that we've seen all the fundamental pattern operators, we can move on to **conditions** and **focusing**, which let us alter and constrain existing ranges.

Sometimes we don't care about just producing a match, we care about the specific range that the match covers. For instance, we may want to match the name of a function, rather than the function entirely.

Feature The `focus` operator "focuses" a range to that of a metavariable it contains, making its range the same as what the metavariable matches.

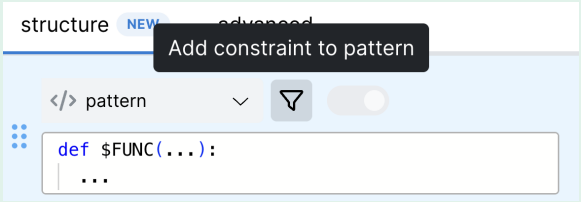
```
pattern: |
  def $FUNC(...):
    ...
where:
  - focus: $FUNC
```

Rule

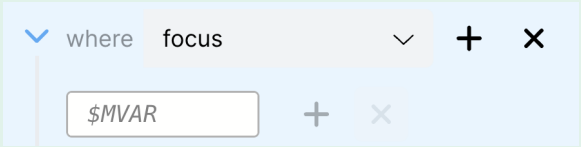
```
def foo():
  pass

def bar():
  pass
```

Target

To add a `focus` in the editor, click the filter button.



This will open a new row where you can enter the metavariable you want to focus on.



Exercise: Non-index routes should check fetch metadata

In this example, we are concerned with routes in a web application which do not check fetch metadata. For routes which are not index routes, we want to ensure that it has a parameter of type `SecFetchMetadata`. [a]

Write a Semgrep rule which well enforce that we properly check fetch metadata on appropriate routes!

Keywords: ellipsis, not, focus

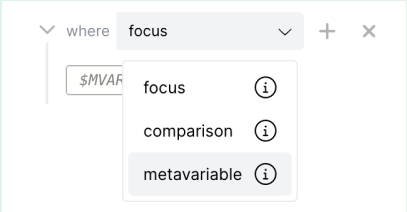
[a] Solution

Sometimes we want to constrain our matches to only ones which fulfill some conditions. Usually this is based on specific parts of the pattern, such as metavariables.

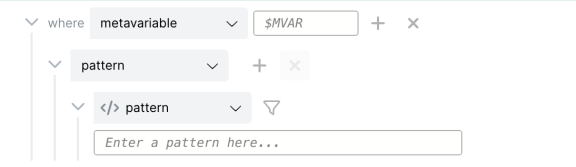
Feature **Metavariable conditions** are conditions that may be placed on a pattern, which filter out matches whose metavariables do not fulfill the condition. They come in three varieties:

- metavariable `pattern` constraints, which specify that a metavariable must contain a specified pattern inside of it
- metavariable `regex` constraints, which specify that a metavariable must match a specific regex
- metavariable `analyzer`s, which are specific kinds of validators that check for things such as high-entropy strings and ReDoS regexes
- metavariable `type` constraints, which specify that a metavariable can only match expressions with a certain type

To add a metavariable condition, we use the same drop-down that we would select a `focus` from:



Switching to `metavariable` will default to a metavariable `pattern` constraint. The drop-down below will let you select from `pattern`, `regex`, `analyzer`, and `type`.



Exercise: Enforcing controller conventions

We want to catch Ruby controllers which do not inherit from `ApplicationController` or `ActionController::Base`. The catch is that we don't necessarily know which Ruby classes are controllers, so we need to use regex on the name of the class to infer this. [a]

```
# Doesn't inherit from ApplicationController: warn!
class BarController < OtherController
end
```

Keywords: metavariable regex, not

[a] Solution

The final condition operator that we haven't talked about is `comparison`, which lets you verify that metavariables satisfy some programmatic condition.

Feature The `comparison` operator lets you write a simple Python expression, and only produces matches whose metavariables satisfy the expression.

```
pattern: |
  retention_in_days = $DAYS
where:
  - comparison: $DAYS <= 365
```

Rule

```
resource "aws_cloudwatch_log_group"
    "notifier" {
  name = "notifier-${var.cluster}"
  retention_in_days = 30
}
```

Target

Exercise: Flag weak RSA encryption keys

We want to ensure that generated RSA keys are of a sufficient strength, in our case greater than or equal to 2048. Let's write a Semgrep rule which can find instances where RSA keys are too weak. [a]

```
// ruleid: use-of-weak-rsa-key-solution
pvk, err := rsa.GenerateKey(rand.Reader, 1024)
// ruleid: use-of-weak-rsa-key-solution
pvk, err := rsa.MultiPrimeKey(rand.Reader, 3, 1024)
```

Keywords: metavariable comparison, any

[a] Solution

7 - Advanced: Taint Mode

Many of the problems in security are not quite so simple as can be phrased in a regular search mode rule.

Many problems in security come from the simple idea of sensitive data reaching a sensitive area, such as a network call or a file write. This problem is solvable via a classic technique in static analysis called **dataflow analysis**.

Def **Taint analysis** is a form of dataflow analysis which tracks whether some specified form of data (called the source) is capable of reaching some specified sensitive site (called the sink). We say that data which comes from the source is **tainted**.

For instance, consider the following code snippet:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

In this example, the input `args` to the program are a potential source of user input, which is sensitive data!

If this user input is allowed to reach the `Runtime.exec` method, for instance, an attacker could potentially cause dangerous arbitrary code execution.

- common sources: URL parameters, cookies, user input
- common sinks: SQL queries, shell commands

Let's trace the flow of taint through this code snippet:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

Let's trace the flow of taint through this code snippet:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

1. the source, the `args` argument to `main`

Let's trace the flow of taint through this code snippet:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

1. the source, the `args` argument to `main`
2. the expression `args[0]`

Let's trace the flow of taint through this code snippet:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

1. the source, the `args` argument to `main`
2. the expression `args[0]`
3. the newly declared variable `cmd`

Let's trace the flow of taint through this code snippet:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

1. the source, the `args` argument to `main`
2. the expression `args[0]`
3. the newly declared variable `cmd`
4. the usage of the tainted variable `cmd`

Let's trace the flow of taint through this code snippet:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

1. the source, the `args` argument to `main`
2. the expression `args[0]`
3. the newly declared variable `cmd`
4. the usage of the tainted variable `cmd`
5. we then land in the sink, the `exec` method

Let's trace the flow of taint through this code snippet:

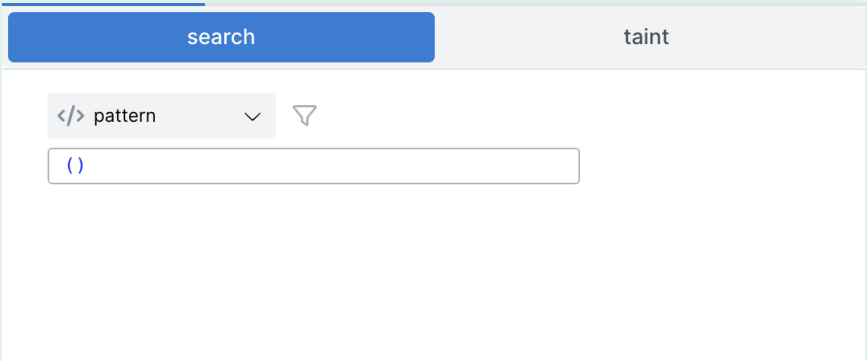
```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

That's a finding!

This reflects exactly how Semgrep works in taint mode:

1. sources of tainted data are specified via patterns
2. sinks of dangerous sites are specified via patterns
3. if a source is able to taint data which reaches a sink, a finding is produced at the sink

To write a taint mode rule, all you need to do is click on the tab labeled "taint" at the top of the Structure Mode editor.



Somewhat self-explanatory.

```yaml
taint:
  sources:
    - pattern: |
        void main(String $ARGS[]) {
          ...
        }
      where:
        - focus: $ARGS
  sinks:
    - pattern: |
        Runtime.getRuntime().exec(
    $CMD)
      where:
        - focus: $CMD
```

Rule

```java
public class DataflowExample {
  public static void main(String
    args[]) throws IOException {
    String cmd = args[0];
    Runtime.getRuntime().exec(cmd);
  }
}
```

Target

Note We have to be a little particular with the source and sink and `focus` accordingly, because we specifically want the argument `args` to be tainted, and specifically the argument to `exec` to be sensitive.

Let's look at another example:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String baseCmd = "rm";
    String sanitizedPath = filterFilePath(args[0])
    String cmd = baseCMD + " " + sanitizedPath;
    Runtime.getRuntime().exec(cmd);
  }
}
```

In this example, our sensitive user input still goes from `args` to `exec`. The difference is that on the path to the sink, it goes through a function `filterFilePath`, which verifies that the input is *not* dangerous.

This means that the call to `exec` is safe here, and should not produce a finding!

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String baseCmd = "rm";
    String sanitizedPath = filterFilePath(args[0])
    String cmd = baseCMD + " " + sanitizedPath;
    Runtime.getRuntime().exec(cmd);
  }
}
```

Here's the new trace, given appropriate source and sanitizer definitions:

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String baseCmd = "rm";
    String sanitizedPath = filterFilePath(args[0])
    String cmd = baseCMD + " " + sanitizedPath;
    Runtime.getRuntime().exec(cmd);
  }
}
```

Here's the new trace, given appropriate source and sanitizer definitions:

1. the source, the `args` argument to `main`

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String baseCmd = "rm";
    String sanitizedPath = filterFilePath(args[0])
    String cmd = baseCMD + " " + sanitizedPath;
    Runtime.getRuntime().exec(cmd);
  }
}
```

Here's the new trace, given appropriate source and sanitizer definitions:

1. the source, the `args` argument to `main`
2. the expression `args[0]`

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String baseCmd = "rm";
    String sanitizedPath = filterFilePath(args[0])
    String cmd = baseCMD + " " + sanitizedPath;
    Runtime.getRuntime().exec(cmd);
  }
}
```

Here's the new trace, given appropriate source and sanitizer definitions:

1. the source, the `args` argument to `main`
2. the expression `args[0]`
3. the sanitizer `filterFilePath(...)`, which ends the taint

```
public class DataflowExample {
  public static void main(String args[]) throws IOException {
    String baseCmd = "rm";
    String sanitizedPath = filterFilePath(args[0])
    String cmd = baseCMD + " " + sanitizedPath;
    Runtime.getRuntime().exec(cmd);
  }
}
```

Ultimately, no finding is produced.

Let's look at how we would produce this rule in Semgrep.

A sanitized rule

```
taint:
  sources:
    - pattern: |
        void main(String $ARGS[]) {
          ...
        }
      where:
        - focus: $ARGS
  sanitizers:
    - pattern: |
        filterFilePath(...)
  sinks:
    - pattern: |
        Runtime.getRuntime().exec($CMD)
      where:
        - focus: $CMD
```

Rule

Exercise: Do not pass request parameters to `Runtime.exec`

In this exercise, we're concerned about sensitive `HttpServletRequest` parameters reaching a `Runtime.exec` call. [a]

Let's use taint mode to flag all the ways that this could happen. You may find the metavariable `type` condition, as mentioned earlier, useful. It's worth noting that you may also write typed metavariables inline in the pattern, by using the syntax `(typename $METAVARNAME)`, in Java.

Keywords: taint analysis, sanitizers, metavariable type

[a] Solution

Exercise: Find XSS in ExpressJS

We have an ExpressJS app which is handling request data, and is potentially vulnerable to XSS attacks. We want to ensure that any user input which is received is properly sanitized before potentially reading a write or send. [a]

Let's write a taint mode rule which checks for XSS, while respecting the possible sanitization of the input.

Keywords: taint analysis, sanitizers, inside

[a] Solution

Exercise: Find path traversal in Go

A **path traversal** attack is a kind of attack which attempts to access files or directories that are outside of the application's root directory. We want to ensure that any user input which is received is properly sanitized before potentially opening a file.

Let's use taint mode to find instances of path traversal in a Go web server. [a]

Keywords: taint analysis, sanitizers, metavariable regex

[a] Solution

8 - Advanced: Autofix

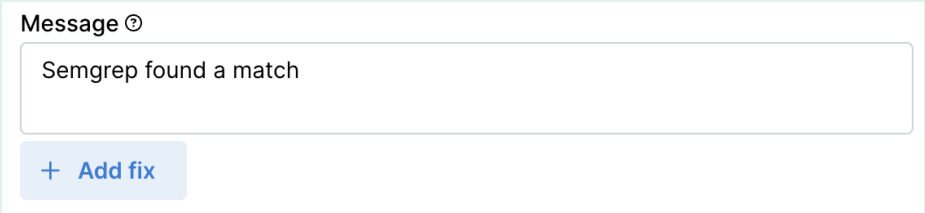
We now know how to write Semgrep rules to match patterns, but that is only one piece of the puzzle. Being able to *match* something doesn't necessarily tell you what you can *do* with it.

Remediation is a whole other half of the battle. How can we not just provide a guardrail around a dangerous behavior, but also tell developers what to do instead? We don't just build a guardrail to block others, we want to build a guardrail which will actively steer them towards better outcomes.

Feature **Autofix** is a Semgrep feature that allows you to write patterns that replace the matched code with safe, remediated code.

Luckily, it's as easy as adding onto the Semgrep rules we've already learned to write!

To add an autofix to your rule, simply click the "Add fix" button located below the rule message in the rule info panel, and enter your autofix.



Note While autofixes are strictly textual, they are **interpolated**, meaning that metavariables appearing in them will be substituted for their matches. This allows you to refactor code based on what was there before!

```
pattern:
  print($STR)
fix: |
  logger.info($STR)
```

Rule

```
print("hi")logger.info("hi")
print("there")logger.info("there")
```

Target

Here, we see that the metavariable `$STR` is shared between the two, so while the `print` is replaced, the content inside of the call is the same.

Fact Rules with autofixes have 50% higher fix rate than rules without autofix.

Exercise: Enforce use of `addSecureCookie` method

In this example, we are working in a codebase which has its own method for adding cookies, which is explicitly designed to be secure. We want to ensure that, instead of using the deprecated `Cookie.addCookie` method, we use the new `addSecureCookie` method instead. [a]

Let's use autofix to not just flag the finding, but provide guidance towards fixing it.

Keywords: autofix, not, inside

[a] Solution

Exercise: Secure subprocess `shell=True`

Here's an example of an Autofix rule, by making sure that we don't use `subprocess.check_output` with `shell` set to `True`!

Let's go through this one together, to make sure everyone is on the same page.

Keywords: autofix, focus, metavariable pattern

8 - Conclusions

Congratulations, you've braved the wilds and written your first Semgrep rules!

We've covered a lot of ground, but there's still more to learn. Semgrep is an application security platform with many different avenues from which to fortify a security program. Luckily, everything starts – and ends – with Semgrep rules.

Starting tomorrow, you will be able to pull Semgrep from `brew`, start scanning your codebases with default rulesets to find vulnerabilities, and even write your own org-specific rules to catch the things that are most important to you.

In a world with increasing technical (and security) debt, we cannot play catch-up. We cannot chase vulnerabilities. We must be **proactive**, and fight the fight from the beginning.

We've covered some of the fundamental rule-writing operators, but there are still more advanced features that we won't have time to cover. These include:

- taint propagators
- taint labels
- generic mode

Feel free to read about these, or approach with any questions on Semgrep rules or rule-writing.

Some excellent further resources include:

- **Semgrep Academy**, a free online learning platform for information security education, with courses on things like rule writing, secure coding, and functional programming (hey, that's me!)
- Pieter De Cremer's YouTube channel with helpful videos about security and Semgrep
- My website, where these slides will be hosted: https://brandonspark.github.io/files/finding_bugs.pdf

Thank you!