CODE SMELL

La creación de una regla personalizada en SonarQube puede ser un proceso complejo y depende de la naturaleza de la regla que se quiera crear. Sin embargo, a continuación se presenta un ejemplo sencillo de una regla personalizada en Python para detectar funciones con un número excesivo de parámetros.

Para crear esta regla, es necesario seguir los siguientes pasos:

1. Crear un archivo de definición de regla personalizada en Python. Este archivo debe incluir la siguiente información:

```python
from sonarqube_rules_python.checks import Check
from sonarqube_rules_python.types import (
    BugPattern,
    Category,
    Priority,
    Severity,
)

class ExcessiveParametersCheck(Check):
    def __init__(self):
        super().__init__(
            id='my_rule_id',
            name='Excessive Parameters Check',
            description='Detect functions with too many parameters',
            category=Category.CODE_SMELL,
            priority=Priority.MINOR,
            severity=Severity.INFO,
            bug_pattern=BugPattern(
                pattern_id='ExcessiveParameters',
                description='Function has too many parameters',
```

```
            ),
        )

    def visit_FunctionDef(self, node):
        if len(node.args.args) > 5:
            self.add_issue(
                node,
                self.bug_pattern.format(node.name),
            )
```

Este código define la regla personalizada `ExcessiveParametersCheck`, que verifica la cantidad de parámetros en cada definición de función en el código Python y agrega un problema (issue) si la cantidad de parámetros excede el valor máximo de 5.

2. Cargar la regla personalizada en SonarQube. Para hacer esto, debe compilar el archivo Python que contiene la definición de la regla personalizada y cargarlo en SonarQube a través de la interfaz web.

3. Ejecutar un análisis de código en su proyecto de Python en SonarQube para probar la regla y verificar si está funcionando correctamente.

Ahora, para crear un ejemplo de fichero donde esta regla saltaría, podemos crear un archivo `example.py` con la siguiente definición de función:

```python
def my_function(param1, param2, param3, param4, param5, param6):
    # Function body
```

En este caso, la definición de la función tiene 6 parámetros, por lo que la regla personalizada `ExcessiveParametersCheck` detectaría un problema (issue) en esta definición de función. Al ejecutar un análisis de código en este archivo utilizando la regla personalizada cargada en SonarQube, se debería generar un problema (issue) asociado a la definición de la función `my_function`.

BUG

Aquí se presenta un ejemplo de cómo crear una regla personalizada en SonarQube para detectar un bug en código Python. En este caso, se detectará un error en la llamada a una función `open()` que no maneja correctamente las excepciones:

1. Crear un archivo de definición de regla personalizada en Python. Este archivo debe incluir la siguiente información:

```python
from sonarqube_rules_python.checks import Check
from sonarqube_rules_python.types import (
    BugPattern,
    Category,
    Priority,
    Severity,
)
class OpenFileCheck(Check):
    def __init__(self):
        super().__init__(
            id='my_bug_rule_id',
            name='Open File Check',
            description='Detect errors in file opening and closing',
            category=Category.BUG,
            priority=Priority.BLOCKER,
            severity=Severity.CRITICAL,
```

```
            bug_pattern=BugPattern(
                pattern_id='OpenFileError',
                description='Error in opening or closing file',
            ),
        )
    def visit_Call(self, node):
        if (
            isinstance(node.func, ast.Name) and
            node.func.id == 'open' and
            len(node.args) > 1 and
            not any(isinstance(arg, ast.NameConstant) and arg.value is None for arg in node.args[1:])
        ):
            self.add_issue(
                node,
                self.bug_pattern.format(),
            )
```

Este código define la regla personalizada `OpenFileCheck`, que verifica todas las llamadas a la función `open()` en el código Python y agrega un problema (issue) si la llamada no maneja adecuadamente las excepciones.

2. Cargar la regla personalizada en SonarQube. Para hacer esto, debe compilar el archivo Python que contiene la definición de la regla personalizada y cargarlo en SonarQube a través de la interfaz web.

3. Ejecutar un análisis de código en su proyecto de Python en SonarQube para probar la regla y verificar si está funcionando correctamente.

Ahora, para crear un ejemplo de fichero donde esta regla saltaría, podemos crear un archivo `example.py` con la siguiente línea de código:

```python
f = open("myfile.txt", "w")
```

En este caso, la llamada a la función `open()` no maneja excepciones correctamente, por lo que la regla personalizada `OpenFileCheck` detectaría un problema (issue) en esta línea de código. Al ejecutar un análisis de código en este archivo utilizando la regla personalizada cargada en SonarQube, se debería generar un problema (issue) asociado a la línea de código con la llamada a `open()`.

VULNERABILITY

Aquí se presenta un ejemplo de cómo crear una regla personalizada en SonarQube para detectar una vulnerabilidad en código Python. En este caso, se detectará el uso de una función de hash débil, que puede dejar el código vulnerable a ataques de fuerza bruta.

1. Crear un archivo de definición de regla personalizada en Python. Este archivo debe incluir la siguiente información:

```python
from sonarqube_rules_python.checks import Check
from sonarqube_rules_python.types import (
    Category,
    Priority,
    SecurityFamily,
    Severity,
    Vulnerability,
)
class WeakHashCheck(Check):
    def __init__(self):
        super().__init__(
            id='my_vuln_rule_id',
            name='Weak Hash Check',
            description='Detect the use of weak hash functions',
```

```
            category=Category.VULNERABILITY,
            priority=Priority.CRITICAL,
            severity=Severity.BLOCKER,
            security_family=SecurityFamily.PYTHON,
            vulnerability=Vulnerability(
                vulnerability_id='WeakHash',
                description='Weak hash function used',
                cwe='330',
                owasp_top_10='A3:2017-Sensitive Data Exposure',
            ),
        )
    def visit_Call(self, node):
        if (
            isinstance(node.func, ast.Name) and
            node.func.id in {'md5', 'sha1'}
        ):
            self.add_vulnerability(
                node,
                self.vulnerability.format(),
            )
```

Este código define la regla personalizada `WeakHashCheck`, que verifica todas las llamadas a las funciones de hash `md5()` y `sha1()` en el código Python y agrega una vulnerabilidad si se utiliza una función de hash débil.

2. Cargar la regla personalizada en SonarQube. Para hacer esto, debe compilar el archivo Python que contiene la definición de la regla personalizada y cargarlo en SonarQube a través de la interfaz web.

3. Ejecutar un análisis de código en su proyecto de Python en SonarQube para probar la regla y verificar si está funcionando correctamente.

Ahora, para crear un ejemplo de fichero donde esta regla saltaría, podemos crear un archivo `example.py` con la siguiente línea de código:

```python
import hashlib
hashlib.md5(b"password").hexdigest()
```

En este caso, la llamada a la función `md5()` utiliza una función de hash débil, por lo que la regla personalizada `WeakHashCheck` detectaría una vulnerabilidad en esta línea de código. Al ejecutar un análisis de código en este archivo utilizando la regla personalizada cargada en SonarQube, se debería generar una vulnerabilidad asociada a la línea de código con la llamada a `md5()`.

HOTSPOT

Aquí se presenta un ejemplo de cómo crear una regla personalizada en SonarQube para detectar un hotspot de seguridad en código Python. En este caso, se detectará la presencia de una contraseña en texto plano en el código.

1. Crear un archivo de definición de regla personalizada en Python. Este archivo debe incluir la siguiente información:

```python
from sonarqube_rules_python.checks import Check
from sonarqube_rules_python.types import (
    Category,
    Priority,
    SecurityFamily,
    Severity,
    SecurityHotspot,
)
class PasswordInCodeCheck(Check):
    def __init__(self):
        super().__init__(
```

```
            id='my_hotspot_rule_id',
            name='Password In Code Check',
            description='Detect the presence of passwords in plaintext in code',
            category=Category.SECURITY_HOTSPOT,
            priority=Priority.CRITICAL,
            severity=Severity.BLOCKER,
            security_family=SecurityFamily.PYTHON,
            security_hotspot=SecurityHotspot(
                hotspot_id='PasswordInCode',
                message='Sensitive information (password) in code',
            ),
        )
    def visit_Str(self, node):
        if 'password' in node.s.lower():
            self.add_security_hotspot(
                node,
                self.security_hotspot.format(),
            )
```

Este código define la regla personalizada `PasswordInCodeCheck`, que verifica todas las cadenas de texto en el código Python y agrega un hotspot de seguridad si se encuentra la cadena "password" en texto plano.

2. Cargar la regla personalizada en SonarQube. Para hacer esto, debe compilar el archivo Python que contiene la definición de la regla personalizada y cargarlo en SonarQube a través de la interfaz web.

3. Ejecutar un análisis de código en su proyecto de Python en SonarQube para probar la regla y verificar si está funcionando correctamente.

Ahora, para crear un ejemplo de fichero donde esta regla saltaría, podemos crear un archivo `example.py` con la siguiente línea de código:

```python
password = '1234'
```

En este caso, la cadena `password` se encuentra en texto plano en el código, por lo que la regla personalizada `PasswordInCodeCheck` detectaría un hotspot de seguridad en esta línea de código. Al ejecutar un análisis de código en este archivo utilizando la regla personalizada cargada en SonarQube, se debería generar un hotspot de seguridad asociado a la línea de código que define la variable `password`.