



Sicherheitslücken mit SAST-Tools früh erkennen

Wegen immer kürzerer Releasezyklen können Sicherheitsteams mit manuellen Codereviews nicht mehr Schritt halten. Open-Source-SAST-Tools wie Semgrep und sein Fork Opengrep versprechen Abhilfe.

Von Timo Schäpe

■ Die Releasezyklen verkürzen sich kontinuierlich: Wo früher quartalsweise deploy wurde, sind heute tägliche oder stündliche Releases keine Seltenheit mehr – während gleichzeitig die Anforderungen an die Sicherheit wachsen. Wie soll man Code auf Sicherheitslücken prüfen, der sich schneller ändert, als man ihn lesen kann? Eine Lösung liegt in der Automatisierung von Securitytests direkt im Entwicklungsprozess. Hier setzen Tools für Static Application Security Testing (SAST) an. Anders als Dynamic Application Security Testing (DAST), das laufende Anwendungen von außen angreift und testet, analysiert SAST den Quellcode direkt und ist ein White-Box-Test.

Der Vorteil ist, dass Sicherheitslücken bereits während der Entwicklung erkannt werden und keine laufende Instanz der Anwendung nötig ist. SAST-Tests können nicht nur in einer CI/CD-Pipeline laufen, sondern sind auch auf Entwicklungsmaschinen oder in der IDE durchführbar. Die Integration von SAST

in CI/CD-Pipelines ermöglicht es, jeden Pull Request und Commit automatisch auf Sicherheitsprobleme zu überprüfen. Entwickler erhalten schnelles Feedback – idealerweise noch in ihrer IDE, spätestens aber während des Build-Prozesses. Das kontinuierliche Überprüfen verhindert, dass unsicherer Code in den Main-Branch gelangt.

SAST-Tools durchsuchen Code systematisch nach bekannten Schwachstel-

lenmustern. Ein typisches Beispiel ist SQL Injection: Sind Benutzereingaben ungefiltert in SQL-Queries eingebaut, öffnet das Angreifen Tür und Tor zur Datenbank. Ein SAST-Tool erkennt solche Muster im Code. Ein Beispiel in Python zeigt Listing 1.

Semgrep und Opengrep im Blickpunkt

Der Artikel betrachtet zwei der derzeit relevantesten Open-Source-SAST-Tools: Semgrep und seinen Fork Opengrep. Sie verfolgen einen modernen Ansatz, der sich von traditionellem SAST unterscheidet: Statt komplexer Datenflussanalysen verwenden sie Pattern Matching mit semantischem Verständnis des Codes. Das Besondere daran: Entwickler können Regeln in einer Syntax schreiben, die dem zu findenden Code sehr ähnlich ist. Das macht das Erstellen eigener Regeln intuitiv und senkt die Einstiegshürde.

SAST-Tools arbeiten nach einem mehrstufigen Analyseprozess: Sie parsen den Quellcode und überführen ihn in eine abstrakte Repräsentation, den Abstract Syntax Tree (AST). Die Baumstruktur bildet die logische Struktur des Codes ab – unabhängig von Formatierung und Kommentaren. Auch Semgrep nutzt dieses Konzept: Seine sprachspezifischen Parser erstellen ASTs, auf denen das Pattern Matching operiert. Im nächsten Schritt durchlaufen Analyse-Engines die Struktur. Pattern-Matching-Engines suchen nach bekannten unsicheren Codierungsmustern, um logische Fehler oder Sicherheitsprobleme zu identifizieren.

Der besondere Ansatz von Semgrep liegt in Semantic Grep: Es versteht die Semantik des AST-basierten Codes und erkennt, dass `password = "123456"` und `pwd = "123456"` konzeptionell dasselbe Sicherheitsproblem darstellen, obwohl die Variablennamen unterschiedlich sind. Die AST-Analyse erlaubt, Patterns zu schreiben, die wie echter Code aus-

IX-TRACT

- Weil die Anforderungen an die IT-Sicherheit gewachsen sind und Code sich durch verkürzte Releasezyklen immer schneller ändert, ist es sinnvoll, Securitytests zu automatisieren.
- SAST-Tools (Static Application Security Testing) analysieren Code während des Entwicklungsprozesses und fangen Sicherheitslücken früh ab.
- Semgrep und Opengrep sind Open Source und wenden eine moderne Herangehensweise an, die sich von der herkömmlichen SAST-Methodik abhebt: Anstelle von Datenflussanalysen nutzen sie Pattern Matching, das ein semantisches Verständnis des Codes ermöglicht.

sehen, aber mit verschiedenen Schreibweisen übereinstimmen.

Dennoch unterliegt SAST Einschränkungen. False Positives bleiben die größte Herausforderung. Erhalten Entwickler täglich Dutzende irrelevante Warnungen, entsteht Alert Fatigue (Alarmmüdigkeit), und sie übersehen Probleme. Tools wie Semgrep adressieren das durch präzisere Regeln und kontextsensitive Analyse, beseitigen das Problem aber nicht vollständig.

Auch das Kontextverständnis erweist sich als problematisch: SAST kann schwer einschätzen, ob eine potenzielle Schwachstelle tatsächlich ausnutzbar ist. In einem Adminbereich mit vorgelagerter Authentifizierung hat eine SQL Injection eine andere Kritikalität als in einem öffentlichen Formular.

Die Sprachvielfalt moderner Projekte stellt Tools vor Herausforderungen. Ein Full-Stack-Projekt nutzt JavaScript, Python, Go oder SQL. Da jede Sprache eigene Sicherheitsmuster und Idiome besitzt, müssen Tools jeweils spezifische Parser und Regeln bereitstellen.

Das Konzept: grep für Code

Die Idee von Semgrep ist einfach, aber effektiv: Was wäre, wenn man grep-ähnliche Suchmuster nicht nur für Text, sondern auch für Codestrukturen hätte? Während traditionelle Suchwerkzeuge auf Zeichenkettenebene arbeiten, versteht Semgrep die semantische Struktur von Code. Die Philosophie dahinter ist, Sicherheitsregeln so einfach schreiben zu können wie den Code selbst. Statt komplexer Abfragesprachen schreibt man Patterns, die wie echte Codebeispiele aussehen:

```
pattern: hashlib.md5(...)
```

Das Pattern findet jedes verwendete MD5-Hashing, unabhängig davon, welche Argumente übergeben werden. Die drei Punkte (...) sind eine Wildcard, die für beliebige Parameter steht. Anders als bei einer Textsuche versteht Semgrep, dass `hashlib.md5($data)` und `hashlib.md5(user_input.encode())` auf das Pattern passen. Der entscheidende Unterschied zu regulären Ausdrücken ist, dass Semgrep die Programmiersprache versteht. Es erkennt, dass `password="geheim"` und `password = "geheim"` das selbe bedeuten, auch wenn die Leerzeichen unterschiedlich sind.

Eine solche semantische Analyse macht Semgrep-Regeln robust gegenüber Formatierungsunterschieden und Refactorings. Man kann den Code umfor-

Listing 1: SQL Injection: unsicherer und sicherer Code

```
# Unsicherer Code – SQL Injection
def get_user(user_id):
    query = f"SELECT * FROM users WHERE id = {user_id}"
    return db.execute(query)

# Sicherer Code mit Parametrisierung
def get_user_safe(user_id):
    query = "SELECT * FROM users WHERE id = ?"
    return db.execute(query, [user_id])
```

Listing 2: Scanaufrufe

```
# Scan mit automatischer
# Regelauswahl
semgrep scan --config=auto /path/to/code

# Scan mit spezifischem Ruleset
semgrep scan --config=p/security-audit /path/to/code
```

Listing 3: Direkte Beispielaufufe für Python

```
# Direktes Pattern ohne Regeldatei
semgrep scan -e 'os.system($CMD)' --lang python /path/to/code

# Mehrere Patterns kombinieren
semgrep scan -e 'eval($X)' -e 'exec($X)' --lang python /path/to/code
```

Listing 4: Aufrufe mit Filter und dem Flag --error

```
# Nach Schweregrad filtern
semgrep scan --severity ERROR /path/to/code # Nur kritische Findings
semgrep scan --severity WARNING,ERROR /path/to/code # Mehrere Level

# CI-freundliche Exit-Codes
semgrep scan --error /path/to/code # Exit 1 bei Findings
```

Listing 5: Beispielaufufe mit Extraparametern

```
# Beispiele zu Include/Exclude-Patterns
semgrep scan --include="*.py" --include="*.js" /path/to/code
semgrep scan --exclude="test_*.py" --exclude="node_modules" /path/to/code
semgrep scan --exclude-dir .git --exclude-dir vendor /path/to/code

# Symlinks, grosse oder versteckte Dateien
semgrep scan --no-git-ignore /path/to/code

semgrep scan --skip-unknown-extensions /path/to/code

semgrep scan --max-target-bytes 3000000 /path/to/code

# .gitignore nicht
# beachten
# Unbekannte
# Dateitypen überspringen
# Grosse Dateien
# einbeziehen
```

Listing 6: Konfigurationsdatei

```
# .semgrep.yml
rules:
  - id: custom-rule
    pattern: ...

paths:
  include:
    - src/
    - lib/
  exclude:
    - test/
    - vendor/
```

matieren, Variablen umbenennen und Funktionen extrahieren – die Semgrep-Regeln finden trotzdem die relevanten Patterns.

Technik: OCaml und Tree-sitter

Semgrep besteht aus mehreren Komponenten: Die Pattern-Matching-Engine ist in OCaml geschrieben, einer schnellen und multiparadigmatischen Programmiersprache, die sich gut für Pattern Matching eignet. Die meisten Sprachparser basieren auf der inkrementellen Parsingbibliothek Tree-sitter. Ursprünglich für Syntax-Highlighting in Editoren entwickelt, ist sie extrem schnell. Semgrep nutzt diese Parser, um aus Quellcode einen Syntaxbaum zu erstellen, auf dem dann das Pattern Matching stattfindet.

Das CLI-Interface ist in Python implementiert und kümmert sich um die Orchestrierung: Dateien finden, Regeln laden, die OCaml-Engine aufrufen und Ergebnisse formatieren. Durch die Trennung lässt sich die performancekritische Analyse in OCaml implementieren, während die Benutzeroberfläche in der zu-

gänglicheren Sprache Python geschrieben ist.

Mit mehr als dreißig Programmiersprachen deckt Semgrep die meisten modernen Tech-Stacks ab. Die Bandbreite reicht von Mainstreamsprachen bis zu spezialisierten Domänen. Zu den vollständig unterstützten Sprachen gehören unter anderem:

- Python, JavaScript/TypeScript, Java, C#, Go, Ruby, PHP;
- C/C++, Swift, Kotlin, Rust, Scala;
- JSON und HCL (Terraform).

Experimentell berücksichtigt sind unter anderem Julia, R, Lua, Dart, Clojure, YAML, XML sowie verschiedene Config-Formate und DSLs.

Semgrep CE versus Pro: die wichtigsten Unterschiede

Die Semgrep Community Edition (CE) ist Open Source und für die meisten Anwendungsfälle ausreichend. Es bietet Single-File-Analyse mit Taint-Tracking in einer Datei, Communityregeln, alle Corefeatures für Pattern Matching sowie die Integration in CI/CD-Pipelines. Semgrep Pro erweitert das um eine Cross-File- und Cross-Function-Analyse, einen Zugriff auf Pro-Regeln mit höherer Präzision sowie einen KI-Assistenten für Triage und Remediation in der Cloud. Für Teams, die gerade mit SAST beginnen, ist die Community Edition meist der beste Einstieg. Ein Upgrade auf Pro ist jederzeit möglich. Die Regelsyntax bleibt dabei identisch, Investitionen in eigene Regeln sind daher zukunftssicher.

Im Dezember 2024 veröffentlichte Semgrep Inc. überraschend eine Lizenzänderung: Es führte die Semgrep Rules License v1.0 ein, die die kommerzielle Nutzung der offiziellen Regelsets einschränkte. Gleichzeitig reservierte es Metadaten in den Scanergebnissen für die kommerzielle Engine (siehe Abschnitt „Metadatenbeschränkungen“). Die Antwort ließ nicht lange auf sich warten: Ein Konsortium aus Securityunternehmen forkte das Projekt als Opengrep, das keine künstlichen Beschränkungen enthalten sollte. Veröffentlicht wurde es im Januar 2025. Um die Tragweite einzuschätzen, muss man verstehen, was sich geändert hat:

Metadatenbeschränkungen: Semgrep CE versteckt in neueren Versionen bestimmte Felder in der JSON- und der SARIF-Ausgabe (Static Analysis Results Interchange Format). Dazu gehören Fingerprints für Finding-Deduplizierung, erweiterte Metriken und einige experi-

mentelle Features. Diese Daten sind nur noch in der kommerziellen Version verfügbar.

Regellizenzen: Die offiziellen Semgrep-Regeln dürfen nicht mehr in kommerziellen Securityprodukten verwendet werden. Wer einen SAST-Service anbieten will, muss eigene Regeln schreiben oder auf Communitybeiträge zurückgreifen.

Featureverschiebungen: Einige Features, die zuvor als experimentell in der Open-Source-Version verfügbar waren, sind in die Pro-Version gewandert. Dazu gehören bestimmte Arten der Datenflussanalyse und neue Patterntypen.

Opengrep: technische Unterschiede

Opengrep basiert auf dem letzten freien Semgrep-Release vor den Lizenzänderungen. Das Entwicklerteam hat aber nicht einfach nur geforkt, sondern auch die Metadaten und den Windows-Support verbessert. Es sind nun alle Scannergebnisse ohne Einschränkungen verfügbar: Fingerprints, Metriken und experimentelle Daten liegen vor. Während Semgrep CE Windows nur über WSL unterstützt, läuft Opengrep nativ unter Windows. Das mag trivial klingen, ist aber für viele Enterprise-Umgebungen ein wichtiges Kriterium.

Opengrep ist vollständig mit Semgrep-Regeln kompatibel. Der Fork erfolgte spät genug, sodass alle wichtigen Features der Regelsprache implementiert sind. Eine Migration ist denkbar einfach. Auch eigene Regeln funktionieren ohne Änderungen. Wegen der Kompatibilität bezieht der Artikel Opengrep mit ein, auch wenn nur von Semgrep die Rede ist. Im Folgenden funktionieren daher sämtliche Aufrufe von Semgrep auch mit Opengrep.

Semgrep lässt sich auf verschiedene Arten installieren und ausführen, entweder mit pip oder pipx:

```
python3 -m pipx install semgrep
```

Über Homebrew (macOS/Linux):

```
brew install semgrep
```

oder Docker:

```
docker run --rm -v "${PWD}:/src" \
  semgrep/semgrep semgrep scan \
  --config=auto /src
```

Nach der Installation ist Semgrep sofort einsatzbereit. Listing 2 zeigt einen ersten Testscan.

Die Option `--config=auto` wählt automatisch passende Regeln anhand der gefundenen Dateitypen aus. Für einen ersten Überblick ist das ideal, in der Praxis lassen sich jedoch spezifische Regelsätze aussuchen.

Die Semgrep-Registry bietet viele vorgefertigte Regeln, die in Kategorien organisiert sind (siehe Registry-Seite von Semgrep unter ix.de/zgqv), wie etwa:

- `p/security-audit`: umfassende Sicherheitsanalyse;
- `p/owasp-top-ten`: OWASP-Top-Ten-Schwachstellen;
- `p/jwt`: JWT-spezifische Sicherheitsprobleme;
- `p/secrets`: Erkennen von Hardcoded Secrets.

Beispiele für sprachspezifische Rulesets sind:

```
p/python
p/javascript
p/java
```

Die Regeln sind beliebig kombinierbar:

```
semgrep scan --config=p/security-audit \
  audit --config=p/python --config=
  my-rules.yml /path/to/code
```

Das Command-Line-Interface meistern

Die praktische Arbeit mit Semgrep beginnt am Command Line Interface. Das grundlegende Kommando folgt dem vertrauten Muster `semgrep <OPTIONS> <TARGETS>`, wobei die Targets Dateien oder Verzeichnisse sein können. Für den Einstieg empfiehlt sich der Automodus mit `semgrep scan --config=auto`, der passende Regeln basierend auf den gefundenen Programmiersprachen auswählt. In der Praxis zeigt sich jedoch schnell, dass eine gezielte Regelauswahl zu präziseren Ergebnissen führt. Kombiniert man verschiedene Regelsets wie `--config=p/security-audit --config=p/python`, kann man sowohl allgemeine Sicherheits-

Listing 7: Aufruf zur Codeanalyse

```
semgrep --config=p/security-audit \
  --config=p/owasp-top-ten \
  --severity ERROR,WARNING \
  --json \
  -o security-report-$(date +%Y%m%d).json \
  /path/to/code
```

Listing 8: Menschlich lesbare Ausgabe einer TLS-Schwachstelle

```
nginx/conf.d/prod.conf
>> generic.nginx.security.insecure-ssl-version.insecure-ssl-version
  Detected use of an insecure SSL version. Secure SSL versions are
  TLSv1.2 and TLS1.3; older versions are known to be broken and are
  susceptible to attacks. Prefer use of TLSv1.2 or later.
  Details: https://sg.run/gLKy

19| ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
```

probleme als auch sprachspezifische Anti-patterns feststellen:

```
semgrep scan --config=p/security-  
audit --config=p/python /path/to/code
```

Es ist möglich, Patterns direkt auf der Kommandozeile zu definieren. Der Befehl

```
semgrep scan -e 'os.system($CMD)' --lang python
```

führt Ad-hoc-Suchen durch, ohne dass man erst eine Regeldatei erstellen muss (Listing 3).

Die Severity-Filterung mit Flags wie `--severity ERROR` oder `--severity WARNING`, `ERROR` erlaubt es Teams, sich auf die kritischsten Probleme zu konzentrieren (Listing 4). Das Flag `--error` transformiert Semgrep in ein Gating-Tool für CI/CD-Pipelines: Bei kritischen Findings wird der Exit-Code ungleich null gesetzt, was Build-Prozesse automatisch stoppt.

Include- und Exclude-Patterns ermöglichen zu kontrollieren, welche Dateien zu analysieren sind. Das Beachten von `.gitignore` ist standardmäßig aktiviert, lässt sich aber mit `--no-git-ignore` ausschalten, was wichtig für vollständige Securityaudits ist, die auch generierte oder temporäre Dateien einbeziehen sollen. Die Option `--max-target-bytes` spricht ein praktisches Problem an: Sehr große Dateien, oft generiert oder Vendor-Code, können die Analyse verlangsamen, ohne relevante Findings zu liefern (Listing 5).

Konfigurationsdateien in Form von `.semgrep.yml` bringen Struktur und Reproduzierbarkeit in den Scanprozess. Statt Kommandozeilenparameter zu duplizieren, definiert man Pfade, Ausschlüsse und Custom-Regeln zentral. Diese Dateien werden versioniert und stellen sicher, dass alle Teammitglieder und CI/CD-Systeme identische Konfigurationen verwenden (Listing 6).

Semgrep im praktischen Einsatz

Der Aufruf in Listing 7 startet die Ausführung einer statischen Codeanalyse, die den Code im angegebenen Pfad nach Sicherheitslücken und OWASP-Top-Ten-Schwachstellen durchsucht. Er speichert die Ergebnisse als JSON-Report mit aktuellem Datum. Die Ergebnisse enthalten nur Findings der Schweregrade `ERROR` und `WARNING`.

Die Kombination von `git diff` und `xargs` bewirkt, dass Semgrep nur geänderte Dateien im Scan analysiert:

```
git diff --name-only | xargs --  
semgrep scan --config=auto
```

Listing 9: JSON-Ausgabe der TLS-Schwachstelle

```
{  
  "check_id": "generic.nginx.security.insecure-ssl-version.insecure-ssl-version",  
  "path": "nginx/conf.d/prod.conf",  
  "start": {  
    "line": 19,  
    "col": 5,  
    "offset": 455  
  },  
  "end": {  
    "line": 19,  
    "col": 42,  
    "offset": 492  
  },  
  "extra": {  
    "metavars": {},  
    "message": "Detected use of an insecure SSL version. Secure SSL versions are TLSv1.2 and TLS1.3; older versions are known to be broken and are susceptible to attacks. Prefer use of TLSv1.2 or later.",  
    "metadata": {  
      "cwe": ["CWE-326: Inadequate Encryption Strength"],  
      "references": [...],  
      "category": "security",  
      "technology": ["nginx"],  
      "confidence": "HIGH",  
      "owasp": [  
        "A03:2017 - Sensitive Data Exposure",  
        "A02:2021 - Cryptographic Failures"  
      ],  
      "subcategory": ["audit"],  
      "likelihood": "MEDIUM",  
      "impact": "MEDIUM",  
      "license": "...",  
      "vulnerability_class": ["Cryptographic Issues"],  
      "source": "...",  
      [...]  
    },  
    "severity": "WARNING",  
    "fingerprint": "...",  
    "lines": "    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;",  
    "is_ignored": false,  
    "validation_state": "NO_VALIDATOR",  
    "engine_kind": "OSS"  
  },  
}
```

Listing 10: SARIF-Ausgabe

```
{  
  "fingerprints": {  
    "matchBasedId/v1": "..."  
  },  
  "locations": [  
    {  
      "physicalLocation": {  
        "artifactLocation": {  
          "uri": "nginx/conf.d/prod.conf",  
          "uriBaseId": "%SRCROOT%"  
        },  
        "region": {  
          "endColumn": 42,  
          "endLine": 19,  
          "snippet": {  
            "text": "    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;"  
          },  
          "startColumn": 5,  
          "startLine": 19  
        }  
      }  
    }  
  ],  
  "message": {  
    "text": "Detected use of an insecure SSL version. Secure SSL versions are TLSv1.2 and TLS1.3; older versions are known to be broken and are susceptible to attacks. Prefer use of TLSv1.2 or later."  
  },  
  "properties": {},  
  "ruleId": "generic.nginx.security.insecure-ssl-version.insecure-ssl-version"  
}
```

Die vielen Outputformate machen Semgrep zu einem flexiblen Werkzeug, das sich in unterschiedlichste Workflows und Toolchains integrieren lässt. Das Standardtextformat ist für menschliche Reviewer optimiert (Listing 8). Es präsentiert Findings mit Kontext, zeigt die betroffenen Codezeilen und liefert die Regelbeschreibung. Die Gruppierung nach Dateien und Regeln ermöglicht es, auch bei vielen Findings schnell zu navigieren.

Der JSON-Output (`--json`) bietet sich an, wenn eine weitere programmatische Verarbeitung der Findings gewünscht ist (Listing 9). Der Code gibt jedes Finding als strukturiertes Objekt mit allen Metadaten aus: Dateipfad, Zeilennummern, Regel-ID, Severity und sogar gematchtem Code. Daher lassen sich eigene Dashboards bauen und Findings in Issue-Tracking-Systeme importieren. Tools wie jq filtern und transformieren effizient JSON-Reports.

Das SARIF-Format ist ein Standard für Toolinteroperabilität (Listing 10). GitHub versteht SARIF nativ. Für VS Code gibt es eine Extension von Microsoft, um das Ergebnis darzustellen und zu den referenzierten Dateien zu springen. Die Integration wird damit trivial: `semgrep --sarif` genügt, um Findings direkt in GitHub-Security-Alerts oder VS Code anzuzeigen.

Ein weiteres Outputformat ist JUnit XML (`--junit-xml`). CI/CD-Systeme wie Jenkins, GitLab CI und CircleCI berücksichtigen JUnit-Reports und stellen Security-Findings als fehlgeschlagene Tests dar (Listing 11).

Die editorspezifischen Formate (`--vim`, `--emacs`) erzeugen einen Output, der sich direkt in die Quickfix-Listen der Editoren laden lässt. Vim-User können mit `:cfile semgrep-results.txt` zu den Findings navigieren. Die Integration macht Secu-

rity-Fixes zu einem natürlichen Teil des Entwicklungsworkflows und erleichtert die Akzeptanz bei Entwicklern (Vim-Ausgabe siehe Listing 12).

Weniger bekannt, aber nützlich ist das GitLab-SAST-Outputformat für native GitLab-Integration. Da der Output nach stderr trennbar ist (`--quiet` unterdrückt Progress-Meldungen), sind saubere Pipes und Automation möglich. Mit `--max-chars-per-line` lässt sich sogar die Zeilenlänge für Terminals optimieren.

Eigene Regeln – das Regelsystem im Detail

Das Herzstück von Semgrep sind seine Regeln. Sie sind in YAML-Dateien definiert und folgen einer klaren Struktur (siehe [ix.de/zgqv](#) und Listing 13). Jede Regel beginnt mit Metadaten: einer eindeutigen ID, einer aussagekräftigen Beschreibung für Entwickler, einem Schweregrad (INFO, WARNING, ERROR) und der Liste unterstützter Programmiersprachen. Optional kann man Metadaten wie CWE-Nummern oder OWASP-Kategorien hinzufügen, was die Integration in Complianceprozesse erleichtert. Ein besonders praktisches Feature ist die Direktive `fix`, die automatische Codekorrekturen vorschlägt.

Die eigentliche Macht von Semgrep liegt in seinem flexiblen Patternsystem. Der einfachste Patterntyp ist `pattern`, der einzelne Codemuster matcht. Komplexere Analysen erfordern es jedoch oft, mehrere Bedingungen zu kombinieren. Hier kommt `patterns` ins Spiel, das als logisches AND fungiert – alle enthaltenen Subpatterns müssen zutreffen. Das Gegenstück ist `pattern-either`, das als OR-Operator arbeitet und Findings generiert, wenn mindestens eines der Subpatterns passt. Mit `pattern-not` lassen sich explizite Ausnahmen definieren, was

die Präzision der Regeln erheblich erhöht und False Positives reduziert.

Ein besonders elegantes Feature ist `pattern-inside`, das den Kontext eines Matches einschränkt. Damit lässt sich beispielsweise spezifizieren, dass ein unsicheres Pattern nur in einer bestimmten Funktion oder Klasse als problematisch gelten soll.

Metavariablen sind die Bindeglieder zwischen abstrakten Patterns und Code. Sie beginnen mit einem Dollarzeichen und können beliebige Codefragmente repräsentieren. Eine Metavariable wie `$X` kann sowohl mit einer einfachen Variablen als auch mit einem komplexen Funktionsaufruf übereinstimmen. Besonders nützlich ist es, dieselbe Metavariable mehrfach in einem Pattern zu benutzen, um beispielsweise Fälle zu finden, in denen derselbe Wert mehrfach verwendet wird:

```
pattern: |
    $X = get_user_input()
    process($X)
```

Die Ellipse (`...`) ist ein Wildcardkonstrukt, das beliebig viele Statements, Argumente oder andere Syntaxelemente matchen kann:

```
pattern: |
    if (...):
        ...
        eval($USER_INPUT)
        ...
```

In einem Pattern wie `func($X, ..., $Y)` deckt die Ellipse alle Argumente zwischen `$X` und `$Y` ab, unabhängig von deren Anzahl oder Typ. Dadurch sind Patterns robust gegenüber Codeänderungen und verschiedenen Programmierstilen.

Erweiterte Features für genauere Analysen

Für noch präzisere Analysen gibt es erweiterte Features. Mit `metavariable-regex` lassen sich Regular Expressions auf den Inhalt von Metavariablen anwenden. Das ermöglicht es beispielsweise, nur Funktionsaufrufe auszuwählen, deren Namen mit `unsafe` beginnen, oder Strings zu identifizieren, die bestimmte Muster wie IP-Adressen enthalten. Die Funktion `metavariable-pattern` geht noch einen Schritt weiter und erlaubt Nested Pattern Matching, bei dem man prüfen kann, ob der Inhalt einer Metavariablen einem bestimmten Pattern entspricht.

Durch die Kombination dieser Features kann man spezifische Regeln erstellen, die weit über einfaches Pattern Matching hinausgehen. Um False Positives zu reduzieren, kann eine Regel beispielsweise

Listing 11: JUnit-XML-Ausgabe

```
<testcase name="generic.nginx.security.insecure-ssl-version.insecure-ssl-version"
  classname="nginx/conf.d/prod.conf" file="nginx/conf.d/prod.conf" line="19">
  <failure type="WARNING"
    message="Detected use of an insecure SSL version. Secure SSL versions
      are TLSv1.2 and TLS1.3; older versions are known to be broken and are
      susceptible to attacks. Prefer use of TLSv1.2 or later.">
    ssl_protocols TLSv1 TLSv1.1 TLSv1.2;</failure>
</testcase>
```

Listing 12: Vim-Ausgabe

```
nginx/conf.d/prod.conf:19:5:W:generic.nginx.security.insecure-ssl-version.insecure-ssl-version:Detected use of an insecure SSL version. Secure SSL versions are TLSv1.2 and TLS1.3; older versions are known to be broken and are susceptible to attacks. Prefer use of TLSv1.2 or later.
```

Listing 13: Beispielregeln

```
rules:
- id: insecure-hash-function
  message: MD5 is cryptographically broken. Use SHA256 instead.
  severity: WARNING
  languages: [python]
  pattern: hashlib.md5(...)
  fix: hashlib.sha256(...)
  metadata:
    cwe: CWE-328
    owasp: A02:2021
```

Listing 14: Beispielregel: hartcodiertes Passwort

```
rules:
- id: hardcoded-password
  pattern: |
    $FUNC(..., password="...", ...)
  message: "Hardcoded password gefunden"
  languages: [python, javascript, java]
  severity: ERROR
```

se SQL-Injection-Schwachstellen identifizieren, indem sie nach Stringkonkatenation in SQL-Querys sucht, aber gleichzeitig Fälle ausschließt, bei denen die Eingaben durch eine Allowlist validiert wurden.

Die Besonderheit bei den Scanregeln ist, dass die Patterns wie der Code aussehen, den sie finden sollen. Wer schon einmal mit grep nach Text gesucht hat, wird sich zurechtfinden – nur dass Semgrep eben nicht nur nach Text sucht, sondern auch die semantische Struktur des Codes versteht.

Die Regel in Listing 14 findet jeden Funktionsaufruf, bei dem ein Parameter mit dem Namen password mit einem String-Literal belegt wird – unabhängig von der Position des Parameters oder dem Namen der Funktion.

Das Schreiben eigener Semgrep-Regeln beginnt meist mit einem konkreten

Problem: einem Sicherheitsvorfall, einem wiederkehrenden Bug oder einer neuen Complianceanforderung. Die Stärke des Tools liegt darin, die Erkenntnisse direkt in ausführbare Regeln übersetzen zu können, ohne komplexe Abfragesprachen lernen zu müssen.

Die Grundstruktur einer Semgrep-Regel ist bewusst einfach gehalten. Jede Regel erfordert eine eindeutige ID. Das Messagefeld sollte nicht nur das Problem beschreiben, sondern außerdem auch Kontext und Lösungsansätze bieten. Ein gutes Beispiel hierfür ist: „Unsicherer Zufallsgenerator: random.random() ist nicht kryptografisch sicher. Es empfiehlt sich, secrets.SystemRandom() für sicherheitskritische Anwendungen zu verwenden.“

Ein zentrales Element jeder Regel ist das Pattern. Die Kunst besteht darin, die

richtige Balance zwischen Spezifität und Generalität zu finden. Ein zu spezifisches Pattern wie requests.get("http://api.example.com/users") findet nur exakte Kopien, während ein zu generisches Pattern wie requests.\$METHOD(...) zu viele False Positives generiert.

Die Entwicklung beginnt mit einem Codebeispiel des Problems (hier SQL Injection): Der unsichere Code

```
"SELECT * FROM users WHERE id = " + user_input
```

wird zum Pattern

```
"SELECT * FROM users WHERE id = " + $USER_INPUT
```

Doch schnell zeigt sich, dass es Variationen gibt. Die Tabelle könnte anders heißen und die WHERE-Klausel verschiedene Felder nutzen. Die Evolution führt zu

```
"SELECT * FROM $TABLE WHERE ↵
    $FIELD = " + $USER_INPUT
```

Doch auch das greift zu kurz. Der vermeintliche Durchbruch kommt mit der Erkenntnis, dass das eigentliche Problem die Stringkonkatenation in SQL-Kontexten ist. Ein Pattern wie \$CURSOR.execute(\$QUERY + \$USER_INPUT) erfasst das Problem präziser. Aber auch hier verpasst man Findings, wenn \$QUERY schon vorab im Code mit \$USER_INPUT konkateniert wird:

```
query = "SELECT * FROM users WHERE ↵
    id = " + user_input
...
cursor.execute(query)
```

Um auch diesen Fall abzufangen, braucht man ein weiteres Pattern:

```
- pattern: |
...
$INPUT = $QUERY + $USER_INPUT
...
$CONNECTION.execute($INPUT)
```

Eine fertige Regel für diese Art der SQL Injection zeigt Listing 15. Auch sie deckt nicht alle möglichen SQL Injections ab, soll aber als Beispiel für eine Regelentwicklung genügen.

Die Regelentwicklung ist ein iterativer Prozess, den Semgrep mit eingebauten Testmechanismen berücksichtigt. Für jede Regel sollten positive und negative Testfälle vorliegen. Die Konvention platziert sie mit speziellen Kommentaren in Dateien wie rule-id.py: # ruleid: <rule-id> markiert Code, der übereinstimmen

soll, und # ok: <rule-id> markiert legitimen Code. Die Option --validate prüft die Regelsyntax, während --test die Testfälle ausführt.

In CI/CD-Pipelines integrieren

Die Integration von Semgrep in CI/CD-Pipelines folgt einem plattformübergreifenden Ansatz. Semgrep bietet ein fertiges Docker-Image an, das sich in CI/CD-Umgebungen verwenden lässt. Man kann es in GitHub Actions, GitLab CI, Jenkins, CircleCI, Azure DevOps, Buildkite und weitere Plattformen einbinden.

Ein Beispiel zeigt, wie sich Semgrep in eine GitLab-CI/CD-Pipeline einbinden lässt (Listing 16). Die Konfiguration von .gitlab-ci.yml folgt dem GitLab-Jobparadigma. Ein minimales Set-up braucht nur wenige Zeilen. Mit Semgrep-Docker-Image als Basis sind reproduzierbare Scans ohne lokale Installation möglich. Das GitLab-Feature artifacts persistiert Scannergebnisse über Pipeline-Runs hinweg und erlaubt historische Vergleiche.

Ein Beispielpattern ist, zwischen schnellen Merge-Requests-Scans und umfassenden Komplettskans zu trennen. Die Pipelineregel merge_requests_event triggert nur bei Merge Requests und scannt ausschließlich geänderte Dateien. Die geplante Pipeline führt nächtlich vollständige Repository-Scans durch.

Wer die Ultimate-Lizenz von GitLab besitzt, profitiert von GitLabs eingebauter Unterstützung für SAST: Während andere CI/CD-Plattformen generische Integrationen erfordern, versteht GitLab den Output von Semgrep nativ und integriert Findings direkt in Merge Requests, Security-Dashboards und Compliance reports.

Die Semgrep-Cloud-Plattform erweitert das Kommandozeilentool um eine webbasierte Schaltzentrale für Securityteams. Das Dashboard visualisiert Securitytrends über alle verbundenen Repositories hinweg und macht abstrakte Metriken greifbar: Findings über Zeit entwickeln, Hotspots im Code und Teamperformance beim Beheben von Issues.

Das Policy-Management wandelt die Regelverwaltung von verteilten YAML-Dateien in zentralisierte Richtlinien. Teams können Regelsets kuratieren, Schweregrade anpassen und Ausnahmen definieren. Die Finding-Triage kann kollaborativ erfolgen: Security-Engineers markieren False Positives, Entwickler kommentieren Findings und Team Leads tracken den Remediation Progress. Integrierte Issue-Tracker wie Jira oder GitHub Issues schaffen nahtlose Workflows. Das Filtern nach Repository, Regel

Listing 15: Beispielregel SQL Injection

```
rules:
- id: security.sql-injection.string-concatenation
  message: |
    Potenzielle SQL Injection durch Stringkonkatenation erkannt.
    Nutzereingaben werden unsicher in SQL-Query eingefügt.
    Empfehlung: Parametrisierte Querys (Prepared Statements) statt ↵
    Stringkonkatenation verwenden.
    Beispiel: cursor.execute("SELECT * FROM users WHERE id = ?", ↵
    (user_id,))

  severity: ERROR
  languages: [python]
  patterns:
  - pattern-either:
    - pattern: $CONNECTION.execute(..., $QUERY + $USER_INPUT, ...)
    - pattern: |
      ...
      $INPUT = $QUERY + $USER_INPUT
      ...
      $CONNECTION.execute($INPUT)

  metadata:
    cwe: CWE-89
    owasp:
    - A03:2021 - Injection
    - A01:2017 - Injection
    references: ...
    likelihood: HIGH
    impact: HIGH
    confidence: HIGH
```

Listing 16: GitLab-Integration

```
semgrep-mr:
  stage: test
  image: semgrep/semgrep:latest
  script:
    - git fetch origin main
    - git diff --name-only origin/main...HEAD | xargs semgrep scan ↵
    --config=auto --json --output=semgrep.json

  artifacts:
    reports:
      sast: semgrep.json

  rules:
  - if: $CI_PIPELINE_SOURCE == "merge_request_event"

semgrep-full:
  stage: test
  image: semgrep/semgrep:latest
  script:
    - semgrep scan --config=p/security-audit --sarif -o semgrep-full.sarif .
  artifacts:
    reports:
      sast: semgrep-full.sarif

  rules:
  - if: $CI_PIPELINE_SOURCE == "schedule"
```

oder Entwickler ermöglicht fokussierte Reviews.

Semgrep Cloud bietet eine komfortable SaaS-Lösung. Aus Compliance-, Datenschutz- oder Kostengründen bevorzugen viele Anwender aber selbst gehostete Alternativen. DefectDojo füllt diese Lücke als Open Source Vulnerability Management Plattform, die eine vollständige Kontrolle über die Securitydaten gibt.

Die Integration von Semgrep-Findings in DefectDojo verlegt isolierte Scanergebnisse in ein Vulnerability-Management-System (zur Integration siehe ix.de/zgqv). Statt Findings in CI-Logs zu verlieren oder in Spreadsheets zu tracken, werden sie in übersichtliche, priorisierte Work-Items mit vollständigem Lifecycle umgewandelt. Zum Einsatz kommen der generische JSON-Importer oder die REST-API von DefectDojo. Der JSON-Output von Semgrep mappt nahezu nahtlos auf DefectDojos Finding-Modell: Severity, CWE-IDs, Dateipfade und Line Numbers werden automatisch übernommen. Ein einfaches Python-Skript oder ein CI-Job genügt, um den Datenfluss zu automatisieren. Durch die API sind zudem inkrementelle Updates möglich – neue Fin-

dings werden hinzugefügt, behobene automatisch geschlossen.

Fazit

SAST-Tools wie Semgrep und Opengrep verkörpern das Prinzip von Shift-Left-Security: Sie verlagern Sicherheitstests vom Ende des Entwicklungszyklus direkt in die Entwicklungsphase hinein. Statt Schwachstellen erst bei Penetrationstests oder im Produktivbetrieb zu entdecken, identifizieren sie Sicherheitsprobleme bereits während des Schreibens von Code – in der IDE, bei jedem Commit oder Pull Request. Die frühe Integration macht Sicherheit zu einem natürlichen Bestandteil des Entwicklungsworkflows und reduziert Kosten und Risiken späteren Nachbesserns. Dabei ergänzen SAST-Tools andere Sicherheitstests wie Penetrationstests und Laufzeitanalysen, ersetzen sie aber nicht. Sie bilden vielmehr die erste Verteidigungslinie in einem mehrschichtigen Sicherheitskonzept. Der Vorteil beider Tools liegt in ihrer niedrigen Einstiegshürde: Während kommerzielle SAST-Lösungen oft hohe Lizenzkosten verursachen und komplexe Set-ups erfor-

dern, lassen sich Semgrep und Opengrep binnen Minuten installieren und produktiv einsetzen.

Besonders wertvoll ist die Unabhängigkeit von Cloud-Diensten: Beide Tools funktionieren vollständig lokal und sind mit selbst gehosteten Lösungen wie DefectDojo kombinierbar, was gerade für Unternehmen mit strengen Datenschutzanforderungen essenziell ist. Die intuitive, codeähnliche Regelsyntax ermöglicht es Entwicklungsteams, schnell und ohne große Einarbeitung eigene Sicherheitsregeln zu erstellen. (nb@ix.de)

Quellen

Dokumentation, Rulesets und Syntax: ix.de/zgqv

TIMO SCHÄPE

arbeitet als Cybersicherheitsberater und SAST-Fachverantwortlicher bei der secuvera GmbH. Er beschäftigt sich mit Softwareentwicklung, Pentesting und SAST.

