SAST: uno strumento necessario per la strada verso il “secure by design”

L’incremento delle minacce provenienti da attacchi informatici, rende necessaria l’adozione di pratiche e metodologie ben definite, volte a ridurre i rischi di sicurezza provenienti dallo sviluppo di applicazioni non sicure.

Più nello specifico, il modello “legacy”, in cui i controlli di sicurezza dell’applicativo in sviluppo venivano affidati ad un team specifico di esperti di sicurezza e solo quando l’applicazione era praticamente prossima alle fasi di rilascio, ad oggi risulta essere un modello obsoleto e non sufficientemente adatto a prevenire e mitigare eventuali vulnerabilità del codice.

Al contrario, si è potuto osservare che per abbattere drasticamente i rischi di sicurezza durante lo sviluppo applicativo e pubblicare applicazioni più robuste in termini di sicurezza, è necessario includere i processi di security all’interno di tutto il ciclo di vita del software (SDLC) a partire dalle prime fasi di design architetturale, attraverso tutte le fasi di implementazione e test, fino alle fasi di rilascio e manutenzione del software. Solitamente si fa riferimento a questo processo con il termine Dev**Sec**Ops: la security si integra ai processi di DevOps in maniera automatizzata e senza avere un impatto negativo in termini di velocità di rilascio software e adattamento ai cambiamenti (pilastri portanti del DevOps).

Implementare correttamente la metodologia DevSecOps richiede diversi step (tra cui un vero e proprio cambiamento culturale relativo alla formazione di quello che potremmo definire

“security mindset” di tutte le figure coinvolte nello sviluppo software), ma in questo articolo analizzeremo più nel dettaglio una particolare categoria di tool che possono essere facilmente integrati nelle pipeline di sviluppo software al fine di ricercare vulnerabilità nel codice prima che la nostra applicazione venga rilasciata in produzione.

Identificare e correggere un problema di sicurezza nel nostro codice sorgente durante le fasi di sviluppo porta notevoli vantaggi in termini di facilità, velocità della correzione del problema e riduzione dei costi.

A cosa servono e come funzionano i tool SAST

SAST sta per **“Static Application Security Testing”**. Scannerizzando il codice sorgente di un’applicazione (“white box” testing), questi strumenti vanno alla ricerca di vulnerabilità all’interno del nostro codice senza bisogno di eseguire l’applicazione stessa (da cui il termine “static”).

Ci sono varie categorie di bug che possono essere identificate da questi tool, tra cui:

- utilizzo di funzioni pericolose
- mancata validazione dell’input
- mancato controllo dell’errore
- utilizzo di algoritmi di cifratura troppo deboli
- utilizzo di secrets/chiavi private/password all’interno del codice sorgente

Una funzionalità che rende molto potenti i tool SAST è chiamata **Taint Analysis**: attraverso questa tecnica è possibile tracciare i dati che sono stati modificati (“tainted”) dall’utente e che potenzialmente possono finire in qualche funzione vulnerabile. Se tali input (in questo contesto chiamati “source”) non vengono propriamente validati prima di finire nella funzione finale (chiamata “sink”) abbiamo

chiaramente trovato una vulnerabilità. Attraverso questo tipo di analisi è possibile identificare velocemente un elevato numero di vulnerabilità spesso riscontrate in applicazioni Web, tra le quali SQL injection, Cross-site scripting, Server Side Template Injection, Remote Code Execution.

Inoltre, molti di questi tool mettono a disposizione dell'utente la possibilità di scrivere le proprie regole di matching, che possono quindi essere specifiche per l'applicazione che stiamo sviluppando. Ad esempio, supponiamo che stiamo sviluppando un'applicazione Web che espone delle API: usando un valido tool SAST, potremmo facilmente implementare una regola custom che verifichi che uno specifico sottoinsieme di endpoint esposti dalla nostra API, implementano correttamente il controllo dell'accesso, ovvero che l'utente che sta richiedendo il servizio sia effettivamente autorizzato a farlo. Questa categoria di vulnerabilità è chiamata "Broken Access Control" ed è alla prima posizione dell'OWASP Top 10 del 2021.

In che modo questi tool operano sul codice sorgente alla ricerca di vulnerabilità?

Una semplice ed immediata soluzione potrebbe essere quella di usare delle espressioni regolari. Consideriamo ad esempio il seguente estratto di codice Python (usando il framework Flask):

Nella chiamata response.set_cookie() non sono esplicitamente settati i flag httponly, secure, samesite (che di default sono impostati a false). Il nostro obiettivo è quello di verificare che tali flag siano impostati nella maniera corretta, in modo da mitigare possibili vulnerabilità (ad esempio XSS diventa molto meno efficace se i cookie sono configurati con il flag httponly).

Potremmo semplicemente scrivere la seguente espressione regolare per verificare che ogni volta che utilizziamo la funzione set_cookie() nel nostro codice, abbiamo correttamente settato i flag di sicurezza del cookie:

```
grep -RE 'response\.set_cookie\('  path/to/code
```

Purtroppo, questa soluzione non è affatto completa ed affidabile. Infatti consideriamo i seguenti esempi:

Fondamentalmente dovremmo riscrivere la nostra espressione regolare per considerare una molteplicità di casi possibili:

- l'import della libreria è stato rinominato
- gli argomenti della funzione sono stati divisi su più righe
- il codice commentato non dovrebbe essere considerato
- etc.

È quindi evidente che l'approccio "regex" è molto limitato e non riesce a coprire in maniera significativa il nostro obiettivo.

Quello che invece viene fatto dai tool SAST più avanzati, è rappresentare il codice sorgente come un albero, più nello specifico un Abstract Syntax Tree, lo stesso tipo di struttura dati che viene utilizzata dai compilatori per rappresentare la struttura del codice di un programma.

Utilizzando questa rappresentazione formale del codice sorgente sotto forma di albero, è possibile considerare il codice da un punto di vista semantico, rimuovendo le ambiguità e ottenendo un matching molto più preciso dei pattern che stiamo ricercando all'interno del codice.

Semgrep

Semgrep è un tool SAST open source molto potente, versatile e facilmente estendibile. Supporta più di 20 linguaggi di programmazione e oltre a fornire il vero e proprio motore di parsing e analisi del codice sorgente, mette a disposizione una libreria di più di 1500 regole predefinite che coprono un gran numero di vulnerabilità per molti linguaggi (Java, Go, JavaScript, Python, Ruby, etc..).

A differenza di altri strumenti simili, Semgrep dà la possibilità di scrivere le proprie regole di matching utilizzando il semplice formato YAML. Non è quindi necessario conoscere uno specifico linguaggio di programmazione per estenderne le funzionalità.

L'utilizzo principale di questi tool è sicuramente l'integrazione nelle pipeline di CI/CD: possiamo eseguire Semgrep ad ogni "push" sul nostro repository in modo da bloccare l'introduzione di possibili bug fin dal principio. A seconda dell'exit code ritornato da Semgrep, possiamo immediatamente notificare lo sviluppatore di un possibile bug introdotto nella sua ultima modifica ed evitare di compilare un'applicazione vulnerabile.

È anche possibile integrare Semgrep sul proprio IDE preferito e avviare lo scan quando viene eseguito un commit locale, in modo da avere un riscontro immediato della qualità e sicurezza del codice su cui stiamo lavorando.

Per esplorare le potenzialità di Semgrep, consideriamo il seguente codice di esempio (Python):

```
1 import subprocess as sp
2 sp.call("ls -a .")  # OK perchè argomento è una costante
3 dir = "/tmp"
4 sp.call("ls -a " + dir) # OK perchè argomento è una costante
5 sp.call(dir, shell=True) # OK perchè argomento è una costante
```

```
6 sp.call(nonstring) # Possibile vulnerabilità
7 sp.call(nonstring, shell=True) # Possibile vulnerabilità
```

Vogliamo che Semgrep ci notifichi un errore nel caso in cui l'argomento passato alla funzione call() del modulo subprocess non sia una costante. Questo perchè potrebbe accadere che l'argomento della funzione possa essere controllato dall'utente, che ovviamente causerebbe una grave falla di sicurezza (un attaccante potrebbe eseguire comandi arbitrari).

Usando Semgrep possiamo scrivere la seguente regola per identificare questo errore:

```
rules:
  - id: subprocess-call
    patterns:
    - pattern: subprocess.call(...)
    - pattern-not: subprocess.call("...", ...)
```

Analizziamo più nel dettaglio la regola:

- **id** non è altro che l'identificativo della regola
- le condizioni definite sotto la parola chiave **patterns** sono in una relazione di AND logico
- la parola chiave **pattern** sta ad indicare che vogliamo matchare tutte le chiamate alla funzione call. Da notare che abbiamo usato l'operatore … come argomento della funzione. Questo operatore (chiamato ellipsis) sta ad indicare che vogliamo ottenere tutte le chiamate della funzione, a prescindere da quali siano i suoi argomenti
- la parola chiave **pattern-not** indica che vogliamo escludere le chiamate alla funzione call che hanno come primo argomento una qualsiasi costante (notare che abbiamo inserito l'operatore di ellipsis dentro doppi apici). Poichè la chiamata alla funzione può prevedere ulteriori argomenti, abbiamo anche specificato un altro operatore di ellipsis come secondo argomento
- notare che le righe 4 e 5 non vengono matchate dalla regola, in quanto Semgrep è in grado di "propagare"

l’informazione che la variabile dir è in realtà una costante (definita nella riga 3)
- notare che nella riga 1, l’import del modulo subprocess è stato rinominato in sp senza tuttavia avere nessun effetto nel modo in cui Semgrep esegue la ricerca (rinominare un elemento non ha nessun effetto “semantico” sul nostro codice)

Con questa semplice regola verremo notificati ogni volta che all’interno del nostro codice usiamo una funzione potenzialmente pericolosa (in questo caso la creazione di un nuovo processo) il cui argomento non è una costante predefinita.

Ovviamente, questo è un esempio molto semplice ma dimostra le grandi potenzialità dello strumento. Esistono molti altri operatori logici e filtri di pattern-matching a disposizione dell’utente che sono consultabili nella documentazione ufficiale del progetto.

Conclusioni

Includere strumenti SAST all’interno delle pipeline di CI/CD è un tassello fondamentale verso il raggiungimento dell’obiettivo finale: rendere la security una parte attiva del processo di sviluppo software, con il fine ultimo di sviluppare applicazioni più sicure e robuste.

Adottare questi strumenti può sicuramente dare benefici immediati, ma per produrre applicazioni più sicure è necessario seguire una metodologia ben definita e che include:

- Threat modeling e analisi dei rischi di sicurezza dell’applicazione
- Implementazione di strumenti SCA (Software Composition Analysis) per la verifica di vulnerabilità presenti su componenti open source di terze parti
- Analisti dinamica dell’applicazione (DAST) alla ricerca

di vulnerabilità verificabili a runtime
- Fornire security workshop/training agli sviluppatori allo scopo di formare un proprio “security mindset” ed accrescere le conoscenze in ambito security

Sorint.Lab, società specializzata in consulenza IT, ha recentemente definito un gruppo di lavoro incentrato proprio verso questa direzione, il cui nome è (non a caso) **SSL (Shift Security Left)**. Lo scopo del team è quello di supportare clienti che hanno necessità di sviluppare un applicativo che sia conforme ai moderni standard di sicurezza seguendo un approccio “secure by design”, in cui i processi di sicurezza sono integrati fin dal principio e attraversano l’intero ciclo di sviluppo del software.

Sorint.lab ha una specifica competenza nella security: è quindi spesso protagonista in contesti internazionali dove si parla di security. L’azienda è stata recentemente protagonista alla conferenza Insomnihack di Ginevra (una delle più famose a livello europeo sul tema security) dove ha presentato un software realizzato da Cesare Pizzi (Security Analyst di Sorint.lab), **REW-sploit**, per facilitare e semplificare l’analisi di codice malevolo. REW-sploit, attraverso la CPU emulation, può aitutare un analista e/o reverse engineer ad indirizzare più velocemente le investigation di codice malevolo basato su frameworks come Metasploit e CobaltStrike (ma non solo): grazie allo strumento è possibile estrarre informazioni vitali dal codice criptato e/o offuscato, permettendo di focalizzare più velocemente l’indagine e comprendere il comportamento generale dell’attacco.

Riferimenti:

A cura di **Luca Pierluigi Famà**, Security Consultant di Sorint.Lab

Condividi sui Social Network:
https://www.ictsecuritymagazine.com/articoli/sast-uno-strumento-necessario-per-la-strada-verso-il-secure-by-design/