1. Che cosa sono gli automated security tests e quali sono i loro vantaggi?

Gli **automated security tests** sono test che verificano automaticamente la presenza di **vulnerabilità di sicurezza all'interno del software**. Questi test si basano sugli stessi principi dei **test funzionali tradizionali:** definiscono dei comportamenti attesi e verificano che, dopo una modifica al codice o alla configurazione, questi comportamenti vengano rispettati. Il problema principale della sicurezza tradizionale è che dipende molto da esperti umani, come penetration tester o auditor. Questo comporta costi elevati e un basso grado di frequenza: molte applicazioni non saranno mai oggetto di un vero penetration test.

L'automazione interviene proprio qui: permette di simulare piccoli test di penetrazione ogni volta che il codice viene modificato o rilasciato. Ad esempio, si possono automatizzare controlli su input malevoli (SQL injection, XSS), scanning delle dipendenze vulnerabili, port scanning, verifiche di configurazioni, o test di sicurezza delle API.

I vantaggi principali sono:

- **Early detection delle vulnerabilità**: il principio dello "shift-left", ovvero trovare problemi prima che arrivino in produzione.
- **Riduzione del lavoro manuale**: i test ripetitivi (input malevoli, scansioni, controlli basilari) non richiedono più un analista umano.
- **Coerenza e ripetibilità**: i test vengono eseguiti sempre nello stesso modo, riducendo l'errore umano.
- **Blocco automatico del deploy in caso di problemi**, migliorando la qualità del software rilasciato.
- **Supporto ai tester umani**: questi si possono concentrare su problemi complessi che non si possono automatizzare.

In pratica, l'automazione non sostituisce il penetration testing umano, ma lo potenzia liberandolo dai compiti noiosi e frequenti.

2. Perché gli automated tests sono ideali per test di sicurezza ripetitivi?

I test di sicurezza ripetitivi includono tutto ciò che deve essere controllato ad ogni rilascio, ma che non richiede creatività o giudizio umano. Sono quindi ideali per essere automatizzati.

Molti attacchi sono prevedibili e simili tra loro: injection, input validation, header mancanti, porte aperte che non dovrebbero esserlo, ecc. Dal punto di vista tecnico, si tratta di operazioni in cui un umano non aggiunge valore significativo dopo aver progettato il test. Per esempio:

- inviare input pericolosi a un form per cercare SQL injection è un'operazione ripetitiva;
- fare scansioni automatiche delle dipendenze note come vulnerabili;
- verificare che gli header di sicurezza HTTP siano presenti e corretti.

Queste verifiche sono spesso considerate "hygiene checks": esattamente come controllare se un'app risponde correttamente in un test funzionale, si controlla se passa determinati test di

sicurezza. **L'automazione permette di eseguire questi controlli in pipeline**, rapidamente e spesso, riducendo la probabilità che una modifica futura reintroduca una vulnerabilità già corretta.

Infine, poiché molte vulnerabilità emergono da errori banali o regressioni, il fatto che i test siano automatici garantisce che tali errori vengano intercettati anche se il team dimentica o trascura l'aspetto sicurezza.

3. Spiega come l'Infrastructure as Code (IaC) migliora la sicurezza.

L'I**nfrastructure as Code (IaC)** è un paradigma in cui l'infrastruttura (server, reti, firewall, load balancer) è definita in forma **dichiarativa tramite file di configurazione** versionabili, esattamente come il codice applicativo.

Dal punto di vista della sicurezza, IaC è un enorme passo avanti perché:

- **Elimina errori umani nella configurazione manuale**: configurare a mano un firewall o una rete introduce spesso errori difficili da tracciare. Con IaC, la configurazione è replicabile e deterministica.
- **Consente di testare l'infrastruttura in pre-produzione**: si può creare un ambiente identico a quello di produzione e applicare modifiche in modo sicuro.
- **Permette versionamento e tracciabilità**: ogni modifica è registrata. Se un cambiamento introduce vulnerabilità, si può risalire all'autore e al commit.
- **Facilita test automatici di sicurezza sull'infrastruttura**: si possono eseguire port scanning, verifiche di permessi, configurazioni di rete, ecc., prima del deploy.
- **Rende più semplice ripristinare configurazioni sicure**: in caso di errore, basta applicare la versione precedente dei file IaC.

4. Perché la disponibilità è spesso trascurata nella CIA Triad?

La triade CIA (Confidentiality, Integrity, Availability) è un riferimento classico per la sicurezza. Tuttavia, nell'immaginario comune, quando si parla di "sicurezza", ci si concentra quasi esclusivamente su:

- proteggere i dati (confidenzialità)
- evitare manipolazioni (integrità)

La disponibilità, invece, sembra più vicina al concetto di performance o affidabilità che non alla sicurezza. Questo porta molti a sottovalutarla. In realtà, la disponibilità è una caratteristica **critica** della sicurezza: un sistema non disponibile è, di fatto, un sistema compromesso. Esempi citati:

- Una centrale di emergenza con linee intasate è una minaccia alla vita umana.
- Un sito di biglietteria che collassa durante un evento importante può avere impatti economici enormi.

È inoltre difficile da testare: testare la confidenzialità si fa con controlli specifici, testare la disponibilità richiede simulazioni di carico e scenari complessi come attacchi DoS.

5. Come si simula un DoS e cosa si vuole misurare?

Un **attacco DoS** (Denial of Service) rende un servizio non disponibile tramite saturazione di risorse o abuso di funzionalità. Per testare la disponibilità bisogna simulare queste condizioni artificialmente.

Le slide spiegano che si parte da una stima dell'**headroom**, ovvero la capacità massima del sistema prima che inizi a rispondere male. Poi si aumenta il carico tramite strumenti di load testing (commerciali o open source, ad esempio "Bees with Machine Guns") inviando migliaia di richieste concorrenti.

Lo scopo non è solo vedere quando il sistema "crolla", ma:

- come si comporta **prima** del collasso (degradazione graduale/ improvvisa);
- quali risorse diventano colli di bottiglia (CPU, RAM, thread, DB);
- se si verificano condizioni pericolose (deadlock, memory leaks);
- come recupera dopo il carico.

Si vuole misurare la resilienza: ovvero la capacità del sistema di mantenere operatività accettabile anche sotto stress.

6. Cosa significa "estimating the headroom" e quali parametri si analizzano?

"Estimating the headroom" significa determinare quanto carico massimo un'applicazione può sostenere prima di iniziare a funzionare in modo insoddisfacente. Non si tratta di capire quando il sistema crolla completamente, ma quando la qualità del servizio degrada al punto da non essere più accettabile.

I parametri principali da monitorare sono:

- **Memoria**: molti sistemi falliscono per memory leaks o saturazione RAM.
- **CPU**: se la CPU è costantemente al 100%, le latenze aumentano.
- **Response times**: criticali per capire la percezione dell'utente; un tempo di risposta eccessivo equivale a una forma di indisponibilità.

Durante questi test si osservano anche:

- comportamento del database sotto carico,
- colli di bottiglia nelle connessioni,
- comportamento dei thread e dei pool di risorse,
- limiti dei load balancer e dei firewall.

L'obiettivo finale è capire "dove fa male" l'applicazione.

7. Che cos'è un domain DoS? Fai un esempio reale.

Il domain DoS è una forma sofisticata di Denial of Service che non sfrutta vulnerabilità tecniche ma **le regole del dominio applicativo**, usandole in modo perfettamente valido ma con intenti malevoli.

È molto pericoloso perché:

- non genera errori o richieste anomale,
- sembra un comportamento completamente legittimo,
- è quasi impossibile da rilevare con sistemi di sicurezza tradizionali.

Esempio classico delle slide:

➤ **Hotel con cancellazione gratuita entro le 16:00**

Un attaccante prenota tutte le camere e le cancella alle 15:59.
Il risultato è che nessun cliente reale può prenotare: l'hotel è "non disponibile" pur non avendo violazioni tecniche.

Esempio reale citato:

➤ **Uber vs Lyft a San Francisco**

Un gruppo associato a Uber prenotava e cancellava migliaia di corse Lyft per sottrarre clienti, creando indisponibilità del servizio.

Questi attacchi sono difficili da prevenire perché l'unica differenza tra utente normale e attaccante è l'**intenzione**, non il comportamento.

8. Elenca e spiega le tre cause delle vulnerabilità da configurazione.

Le slide identificano tre macro-cause:

1. Unintentional changes (modifiche non intenzionali)

Accadono quando qualcuno modifica la configurazione senza accorgersi di aver introdotto vulnerabilità.
Cause possibili:

- errori di merge
- typo
- cancellazioni involontarie di righe
- aggiornamento dipendenze non verificato

Poiché la configurazione spesso è "potente" (basta cambiare 1 riga per modificare il comportamento), questi errori possono compromettere la sicurezza senza che nessuno se ne accorga.

2. Intentional changes (modifiche intenzionali)

Succede quando un programmatore modifica una configurazione per introdurre un nuovo comportamento, ma senza sapere che quella modifica disabilita o altera una protezione esistente.

È frequente nei sistemi complessi, soprattutto quando:

- il codice ha pochi test,
- il team non conosce la storia delle scelte progettuali pregresse,
- la documentazione è scarsa.

Le modifiche sono corrette, ma gli effetti collaterali non vengono scoperti.

3. Misunderstood configuration (configurazione fraintesa)

Succede quando la configurazione **non fa quello che si crede** che faccia.
Il problema deriva spesso da API ambigue:

- interi come codici di configurazione,
- stringhe magiche,
- opzioni negate (es. disableSecurity = false),
- documentazione poco chiara.

Il risultato è che il programmatore crede di aver abilitato una funzionalità, ma in realtà la disattiva o la lascia in uno stato insicuro.

9. Perché i test sulle configurazioni sono importanti?

Molte funzionalità, soprattutto quelle legate alla sicurezza (HTTPS, autenticazione, ORM, sanitizzazione input), non sono più scritte a mano: vengono fornite da framework o librerie. Il programmatore non implementa la sicurezza, la **configura**.

Ciò significa che la sicurezza dell'applicazione dipende totalmente dalla correttezza di queste configurazioni.

I test sulle configurazioni sono importanti perché:

- permettono di rilevare modifiche accidentali,
- mantengono la "memoria storica" delle decisioni di sicurezza prese nel tempo,
- evitano di disattivare funzionalità di sicurezza senza accorgersene,
- proteggono dalle regressioni,
- verificano non il file di configurazione, ma il **comportamento risultante**.

Testare una configurazione non è testare un setter: è verificare che il sistema si comporti come dovrebbe quando il framework interpreta quella configurazione.

10. Perché è necessario testare anche i comportamenti impliciti (default)?

Molti framework hanno comportamenti di default che non vengono esplicitamente configurati. Questi defaults possono essere:

- sicuri oppure insicuri,

- modificati durante aggiornamenti della libreria,
- diversi da un ambiente all'altro.

Se non si conoscono i default, non si può sapere:

- se il sistema è sicuro,
- se una modifica indiretta li cambia,
- se un aggiornamento della libreria li ha modificati.

Esempio: un framework che di default abilita il CSRF protection potrebbe cambiarlo in una versione successiva. Se nessun test verifica la protezione, l'app potrebbe diventare vulnerabile dopo un update.

Testare i comportamenti impliciti permette una visione completa della sicurezza del sistema, non solo delle parti configurate esplicitamente.