КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (29), 2025
ISSN 2663 - 4023
DOI 10.28925/2663-4023.2025.29.887
УДК 004.056

Пархоменко Іван Іванович
кандидат технічних наук, доцент, завідуючий кафедри КБЗІ
Факультет інформаційних технологій
Київський національний університет ім. Тараса Шевченка, Київ, Україна
ORCID ID: 0000-0001-6889-9284
ivan.parkhomenko@knu.ua

Савонік Михайло Володимирович
аспірант кафедри КБЗІ
Факультет інформаційних технологій
Київський національний університет ім. Тараса Шевченка, Київ, Україна
ORCID ID: 0009-0001-7622-978X
savonikm@fit.knu.ua

ЗАПОБІГАННЯ ЗАГРОЗАМ ХИБНИХ КОНФІГУРАЦІЙ AWS ІНФРАСТРУКТУРИ ЯК КОДУ НА ОСНОВІ ТЕСТУВАННЯ

Анотація. Хибні конфігурації хмарної інфраструктури стали поширеним та впливовим вектором загроз у сфері кібербезпеки. Особливо це стосується організацій, які використовують сервіси Amazon Web Services (AWS), де неправильно налаштовані засоби контролю доступу, недостатнє ведення журналів подій, слабка сегментація мережі та відсутність критично важливих захисних засобів, таких як міжмережеві екрани вебдодатків (WAF), створюють значні ризики безпеки. Незважаючи на широке впровадження інструментів інфраструктури як коду (IaC), наприклад Terraform та AWS CloudFormation, що забезпечують передбачувані розгортання з контролем версій, ці конфігурації IaC зазвичай не проходять систематичного тестування на безпеку. На відміну від прикладного коду, який регулярно проходить модульні, інтеграційні та безпекові тести, інфраструктурний код рідко тестується за межами базового статичного аналізу чи моніторингу після розгортання. Як наслідок, критичні хибні конфігурації можуть залишатися непоміченими до моменту їх експлуатації зловмисниками. Для вирішення цієї проблеми в роботі запропоновано новий підхід, який названо «інфраструктура як протестований код». Шляхом застосування перевірених технік тестування програмного забезпечення (наприклад, тестових тверджень та робочих процесів безперервної інтеграції) до IaC, запропонована методика дозволяє проводити валідацію стану безпеки AWS-середовища ще до його розгортання. Розроблено та оцінено експериментальний прототип, який автоматизує перевірки безпеки для ресурсів AWS, визначених за допомогою Terraform, з фокусом на ключові конфігурації, такі як правила WAF, політики управління ідентифікацією та доступом (IAM), дозволи на сховища S3 та правила груп безпеки. Тестовий набір побудований на основі відкритих інструментів (Chef InSpec та AWS SDK) і виконується у CI/CD-конвеєрі із використанням LocalStack для емулювання сервісів AWS. Такий підхід дозволяє розробникам та DevSecOps-командам виявляти та виправляти хибні конфігурації на ранніх етапах життєвого циклу розробки, задовго до розгортання інфраструктури у продуктивному середовищі. Експериментальні результати демонструють, що інтеграція автоматизованих тестів безпеки у DevOps-конвеєр суттєво підвищує захист хмарних середовищ та знижує вразливість через неправильні конфігурації. Порівняно з традиційними статичними інструментами аналізу, запропонований підхід забезпечує більшу гнучкість, підтримує специфічні для середовища політики та дозволяє розробникам кодифікувати власні, придатні до тестування безпекові твердження. Навіть мінімальний набір тестів показав ефективність у виявленні критично важливих хибних конфігурацій, які не були зафіксовані статичними перевірками. Ця парадигма доповнює існуючі інструменти хмарної безпеки (AWS Config, Checkov та інші фреймворки політик як коду) і легко інтегрується у процеси DevSecOps. На завершення робота пропонує детальний опис реалізації, реальний практичний приклад застосування та аналіз практичних компромісів. Зроблено висновок, що аналогічно до тест-орієнтованої розробки програмного забезпечення, застосування тест-орієнтованого підходу до інфраструктури може стати критично важливою стратегією для активного забезпечення безпеки

хмарних середовищ. Ця праця закладає основу для майбутніх досліджень щодо формалізації практик тестування безпеки IaC, порівняльного аналізу покриття тестами та розробки бібліотек безпекових тверджень для AWS.

Ключові слова: хмарна безпека; інфраструктура як код (IaC); запобігання хибним конфігураціям; DevSecOps; безпека AWS; автоматизоване тестування безпеки; політики як код; валідація безпечних конфігурацій.

ВСТУП

У сучасних умовах стрімкої цифровізації підприємства все частіше впроваджують хмарні обчислення для підвищення масштабованості, гнучкості та ефективності IT-інфраструктури. Проте цей перехід супроводжується новими викликами у сфері кібербезпеки, зокрема — зростанням кількості інцидентів, спричинених неправильними конфігураціями хмарних ресурсів. Одним із ключових факторів таких інцидентів є те, що інфраструктурний код, попри свою автоматизовану природу, часто не охоплюється належними перевірками безпеки.

Постановка проблеми. Традиційні підходи до безпеки інфраструктури базуються переважно на ручному аудиті або післяфактумному моніторингу, що не дозволяє своєчасно виявляти критичні вразливості. Водночас методи забезпечення якості, давно усталені в розробці програмного забезпечення — зокрема тестування та валідація — лише частково або взагалі не застосовуються до інфраструктури як коду (IaC). Така асиметрія створює прогалину між надійністю застосунків і їхньої інфраструктури, яку можуть використовувати зловмисники. Саме тому виникає потреба в нових підходах до забезпечення безпеки хмарної інфраструктури, які інтегрують перевірки безпеки безпосередньо в процес розробки та розгортання.ибні конфігурації хмарної інфраструктури стали критичною загрозою безпеці в сучасних IT-середовищах. Зокрема, згідно з опитуванням, проведеним у 2020 році, саме хибна конфігурація була визначена головною причиною витоків даних у хмарних сервісах [1]. Аналогічно, у глобальному дослідженні 2024 року повідомляється, що майже третина нещодавніх інцидентів порушення безпеки в хмарі була спричинена саме помилками конфігурації або людським фактором [2]. Подібні помилки можуть мати катастрофічні наслідки: наприклад, витік даних у Capital One у 2019 році був згодом пов’язаний із неправильно налаштованим міжмережевим екраном у середовищі AWS цієї фінансової установи [3]. Із зростанням поширення практики керування інфраструктурою через код (Infrastructure as Code, IaC) одна помилка в конфігураційному скрипті може поширитися на безліч ресурсів, збільшуючи масштаб загрози. Отже, завдання зменшення ризиків, пов’язаних із хибними конфігураціями в хмарі, стало надзвичайно актуальним — як для захисту чутливих даних, так і для забезпечення стійкості хмарних систем.

Аналіз останніх досліджень і публікацій. Проблема хибної конфігурації розглядається як у галузевих ініціативах, так і в академічних дослідженнях. Постачальники хмарних послуг і розробники засобів безпеки пропонують інструменти автоматичного сканування IaC-шаблонів, які дають змогу виявляти небезпечні конфігурації ще до їх розгортання. Такі інструменти статичного аналізу, як-от Checkov чи TFSec, аналізують інфраструктурний код на наявність порушень політик або ризикованих налаштувань, ефективно вбудовуючи перевірки безпеки в процес розробки [11]. Крім того, спільнота OWASP опублікувала довідник з безпеки інфраструктури як коду (OWASP Infrastructure as Code Security Cheat Sheet, 2021), який описує найкращі практики: принцип найменших привілеїв, управління секретами, безпекові перевірки в CI/CD тощо [6].

З боку наукових досліджень було проведено кілька емпіричних робіт, що пояснюють причини збереження хибних конфігурацій. Наприклад, одне масштабне дослідження 2025 року проаналізувало понад 1600 публічних IaC-репозиторіїв і виявило поширені порушення безпеки в скриптах [4]. Вразливості, пов'язані з хибними конфігураціями, охоплювали всі категорії OWASP Top 10, що свідчить про наявність серйозних недоліків у IaC-шаблонах. Дослідники зазначили, що навіть за наявності сканерів безпеки команди не завжди послідовно інтегрують їх у свої автоматизовані конвеєри — виправлення часто відбувається в ручному режимі, без системного підходу [4]. Інше дослідження 2025 року оцінило рівень впровадження найкращих практик безпеки в конфігураціях Terraform і виявило нерівномірне застосування політик [5]. Хоча деякі заходи контролю (наприклад, обмеження доступу) впроваджуються регулярно, інші критично важливі захисти, як-от шифрування даних у стані спокою, здебільшого ігноруються [5]. Ці висновки свідчать, що саме використання IaC ще не гарантує безпечності — необхідне свідоме дотримання чітких політик та перевірок [5].

У результаті аналізу стає очевидним, що залишаються нерозв'язані аспекти проблеми. Багато організацій не використовують повною мірою наявні засоби захисту IaC, що призводить до відсутності важливих захисних механізмів у їхніх хмарних конфігураціях. Сучасні інструменти здебільшого орієнтовані на відомі шаблони помилок і можуть не виявляти контекстно-залежні загрози. Відсутність безшовної інтеграції перевірок безпеки у DevOps-процеси також призводить до того, що хибні конфігурації прослизають у продакшн [12]. Саме ці недоліки пояснюють, чому хмарні середовища залишаються вразливими до інцидентів, спричинених помилками налаштувань, попри наявність численних інструментів і рекомендацій.

Метою статті є представлення нового підходу до підвищення безпеки інфраструктури як коду та запобігання хибним конфігураціям у хмарних середовищах. Зокрема, запропоновано фреймворк методів і інструментів, які можна інтегрувати у життєвий цикл розробки хмарної інфраструктури з метою проактивного виявлення помилок конфігурації та впровадження найкращих практик. Шляхом усунення виявлених прогалин — покращення дотримання політик (наприклад, забезпечення шифрування та правильного контролю доступу в IaC), вдосконалення виявлення складних вразливостей конфігурації та інтеграції безперервного аудиту IaC у DevSecOps — запропонований підхід спрямований на суттєве зниження ризику порушень безпеки, спричинених хибними конфігураціями.

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Парадигма «Інфраструктура як протестований код» (Infrastructure as Tested Code) тісно інтегрує забезпечення хмарної інфраструктури з автоматизованим тестуванням безпеки. Цей підхід розширює практики Infrastructure-as-Code (IaC), поєднуючи кожне визначення інфраструктури з набором тестів безпеки, які постійно перевіряють конфігурації на відповідність очікуваним безпечним станам.

Парадигма «Інфраструктура як протестований код»

«Інфраструктура як протестований код» означає, що кожен критичний хмарний ресурс або конфігурація супроводжується тестами, які підтверджують його правильний і безпечний стан. На практиці це переносить добре відомі концепції тестування програмного забезпечення (модульні тести, інтеграційні тести, інтеграція з конвеєрами розгортання) у сферу управління конфігурацією хмар. Наприклад, замість того, щоб довірливо розгортати S3-бакет, спочатку створюється тест, який перевіряє, чи увімкнено шифрування та чи

відсутній публічний доступ; лише якщо конфігурація бакета відповідає цьому тесту, вона проходить перевірку CI/CD. Як ілюстровано на рис. 1, процес інтеграції тестів безпеки передбачає розгортання інфраструктури у середовищі емулятора хмарного провайдера, що забезпечує виконання статичних і динамічних перевірок конфігурацій у рамках CI/CD-конвеєра. Код і конфігурації розгортаються у середовищі емулятора хмарного провайдера, що дозволяє виконувати автоматизовані перевірки політик та активні тести безпеки ще до розгортання у реальному хмарному середовищі.

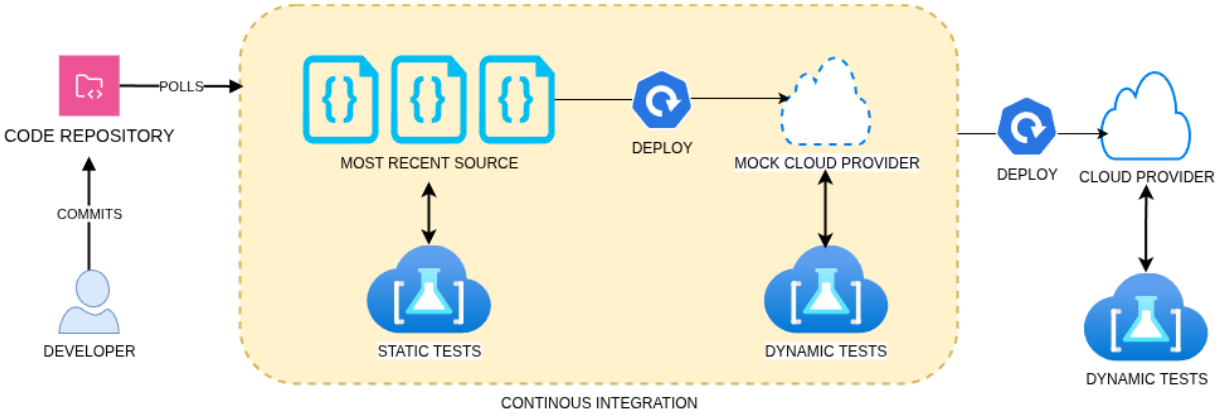

Рис. 1. Схема методу інтеграції статичних і динамічних тестів безпеки в CI/CD-конвеєр для перевірки інфраструктури як коду

Формалізуючи вимоги до безпеки у вигляді тестів, помилки конфігурації виявляються на ранніх етапах розробки. Ця парадигма відображає підхід Test-Driven Development — інженери можуть написати тест безпеки для запланованої конфігурації, а потім реалізувати зміну IaC, доки тест не буде пройдено, ефективно впроваджуючи Test-Driven Infrastructure Security. Важливо, що цей підхід вирішує реальну проблему, коли хибні конфігурації хмар є основною причиною порушень безпеки. Неправильно налаштований ресурс AWS може розкрити чутливі дані в інтернеті або дозволити ескалацію привілеїв. Високопрофільні інциденти підкреслюють цей ризик: наприклад, витік даних Capital One був спричинений неправильно налаштованим веб-додатком (WAF), що дозволило зловмиснику виконати серверний запит і отримати доступ до ~100 мільйонів записів клієнтів [7]. Це сталося, незважаючи на те, що WAF є засобом безпеки — неправильна конфігурація знівелювала його захисну функцію [7]. Вбудовування перевірок конфігурації у вигляді коду може запобігти таким проблемам, зупиняючи збірку, якщо, наприклад, WAF розгортається без необхідних правил або з надто широким доступом. По суті, «Інфраструктура як протестований код» розглядає прогалини в безпеці як збої тестів, які потрібно виправити до розгортання. Цей проактивний підхід дедалі більше виправдовується галузевими даними: організації, які інтегрують перевірку безпеки IaC у конвеєри, спостерігають ~33% зменшення інцидентів у хмарі [8]. Впроваджуючи перевірки конфігурації до того, як вони досягнуть продакшну, парадигма безпосередньо зменшує одну з найбільших проблем хмарної безпеки. Для реалізації цього підходу розробники визначають очікувані безпечні стани для кожного ресурсу. Наприклад, тест може перевіряти, що «всі S3-бакети, класифіковані як чутливі, мають шифрування на стороні сервера та заблокований публічний доступ» або «жодна група безпеки не дозволяє вхідний SSH з 0.0.0.0/0» — кодування політик у вигляді коду. Інструменти, такі як Chef InSpec (який надає DSL для

тестування ресурсів AWS) або універсальні мови (Python з boto3 тощо), можуть виражати ці очікування. Наприклад, використовуючи InSpec, можна написати:

```
describe aws_s3_bucket('my-sensitive-bucket') do
 it { should have_default_encryption_enabled }
 it { should_not be_public }
end
```

Цей тест автоматично перевіряє шифрування та політику доступу до my-sensitive-bucket. Кожен такий тест аналогічний модульному тесту, зосередженому на одному аспекті інфраструктури. З часом організація може накопичити велику бібліотеку тестів безпеки (по суті, закодованих політик або найкращих практик). У данному підході інженери прагнуть, щоб кожна поширена категорія помилок конфігурації (ідентифікація та доступ, мережевий доступ, шифрування даних, логування тощо) перевірялася якимось тестом. Ця систематична перевірка контрастує з випадковими аудитами та забезпечує більшу впевненість у безпеці інфраструктури. Варто зазначити, даний підхід узгоджується з концепцією Policy-as-Code. Фреймворки, такі як HashiCorp Sentinel (для Terraform) або AWS Config Rules, дозволяють писати декларативні політики (наприклад, «усі томи EBS мають бути зашифровані»), які оцінюються щодо визначень інфраструктури. Ми використовуємо такі політики, але також йдемо далі: там, де потрібна спеціальна логіка або перевірки, специфічні для середовища, ми пишемо імперативні тести. Поєднуючи стандартні бібліотеки політик (для базової відповідності) з індивідуальними тестами (для уточнених сценаріїв), «Інфраструктура як протестований код» охоплює як загальні помилки конфігурації, так і специфічні вимоги до безпеки. Наприклад, ми можемо інтегрувати існуючі еталони (наприклад, CIS AWS Foundations) як перевірки на основі коду — багато хмарних платформ надають їх; AWS Config має ~150 керованих правил [9], які відповідають найкращим практикам і стандартам, і підтримує розробку власних правил у коді. Інтеграція цих правил у наш тестовий набір означає, що велика кількість відомих найкращих практик автоматично забезпечується (увімкнене шифрування, відсутність публічних S3, увімкнене MFA для користувачів IAM тощо), тоді як наші індивідуальні тести охоплюють архітектурно-специфічні проблеми, які загальні еталони можуть не враховувати. Ця подвійна стратегія забезпечує всебічне дотримання політик безпеки.

Безперервне тестування безпеки в CI/CD

Ключовим моментом результатів дослідження є демонстрація того, що тести безпеки для інфраструктури можуть бути безшовно інтегровані в конвеєр безперервної інтеграції/безперервного розгортання (CI/CD). У типовому робочому процесі DevSecOps, коли розробники комітять код, автоматизовані тести запускаються для виявлення помилок; ми застосовуємо те ж саме до безпеки IaC. Конкретно, щоразу, коли оновлюються шаблони Terraform (або CloudFormation), наш конвеєр автоматично виконує повний набір тестів безпеки інфраструктури. Якщо будь-який тест не проходить (вказуючи на потенційну помилку конфігурації або порушення політики), конвеєр блокує розгортання. Цей механізм ефективно зміщує безпеку вліво — виявляючи помилки на етапі перевірки коду або CI, а не після того, як ресурси запущені в експлуатацію. Щоб дозволити автоматизоване тестування хмарних конфігурацій до розгортання, ми використовуємо ізольовані хмарні середовища. Потужним інструментом у цьому відношенні є LocalStack, який емулює служби хмар AWS локально. Використовуючи LocalStack, ми можемо розгорнути весь стек IaC в локальному середовищі, яке імітує API AWS. Після розгортання наші тести взаємодіють з цим імітованим середовищем через API або SDK AWS, так само, як вони б робили це з реальним

обліковим записом AWS. Це дозволяє динамічно перевіряти конфігурації без ризику для реальних хмарних ресурсів. Проект NCC Group Aerides продемонстрував життєздатність цієї техніки: вони інтегрували LocalStack у CI, розгорнули інфраструктуру, визначену Terraform, у ньому, а потім запустили такі інструменти, як PMapper (для аналізу ризиків IAM) та власні скрипти Python unittest для перевірки властивостей безпеки. Ми побудували на подібних принципах у нашому доказі концепції. Вказуючи сканери безпеки та API-виклики на кінцеві точки LocalStack, ми фактично отримуємо переваги динамічного тестування хмари (перевірка фактичних запущених конфігурацій) на етапі перед розгортанням. Рис. 1 ілюструє цю концепцію — CI-конвеєр запускає екземпляр LocalStack, застосовує зміни IaC до нього, а потім виконує як статичні перевірки політики, так і динамічні тести проти імітованого середовища (поєднуючи методи «статичного» та «динамічного» аналізу). Як тільки всі тести проходять, зміни можуть бути безпечно розгорнуті в реальному середовищі AWS. Код інфраструктури розгортається в емуляторі AWS LocalStack, що дозволяє виконувати як перевірки політик як код, так і активні тести безпеки (наприклад, за допомогою API AWS) протягом ~30 секунд, виявляючи помилки конфігурації до потрапляння в продакшн.

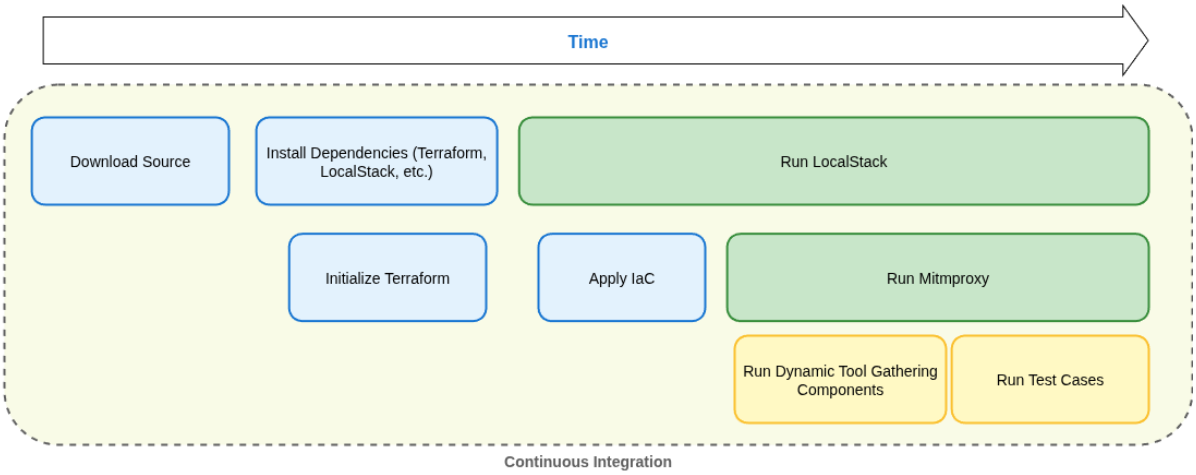

Рис. 2. Інтеграція статичних і динамічних тестів безпеки в CI/CD конвеєр за допомогою LocalStack

Перевага автоматизованого тестування до розгортання є значною. Виявляючи помилки на ранніх етапах, ми запобігаємо потраплянню ризикованих конфігурацій у живе середовище. Наприклад, якщо розробник випадково вводить надто широкий доступ для групи безпеки (відкриваючи SSH для всього світу), тест, призначений для виявлення доступу «0.0.0.0/0» на портах, зазнає невдачі в CI, і небезпечна зміна блокується. Це набагато безпечніше та дешевше, ніж розгортати, а потім виявляти проблему (або, ще гірше, зазнавати інциденту). Як зазначають Steringer та ін., «інженери можуть писати перевірки безпеки або політики як код, які виконуються щоразу, коли хтось вносить зміни», запобігаючи виникненню конфігураційних помилок високої серйозності в розгорнутій інфраструктурі. По суті, CI-конвеєр стає воротарем безпеки. Ми спостерігали в наших експериментах, що використання LocalStack робить цей процес дуже швидким: розгортання нашого тестового стеку та виконання всіх перевірок безпеки зайняло порядку десятків секунд. Це узгоджується з попередніми висновками, що повний цикл підняття LocalStack, розгортання IaC та виконання тестів може завершитися приблизно за 30 секунд для невеликого проекту. Така швидка зворотна зв’язок є благом

для розробників — це порівнянно з виконанням модульних тестів і набагато швидше, ніж розгортання в тестовий обліковий запис AWS, а потім сканування (процес, який може зайняти багато хвилин до години). Швидкий зворотний зв'язок спонукає розробників негайно виправляти проблеми, так само як вони б виправляли невдалі модульні тести, сприяючи мисленню про ітеративне поліпшення. Наша інтеграція CI/CD була реалізована за допомогою GitHub Actions у прототипі. При кожному пуші в репозиторій конвеєр виконує такі кроки (автоматично та без участі людини):

- Статичний аналіз і планування — перевірка та парсинг шаблонів Terraform на наявність синтаксичних помилок або очевидних порушень політики (за допомогою інструментів, таких як terraform validate та Checkov/TFLint для базових перевірок).
- Розгортання в пісочниці — запуск екземпляра LocalStack та застосування конфігурації Terraform до нього, створюючи імітоване середовище AWS, що представляє заплановану інфраструктуру.
- Виконання тестів безпеки — запуск набору тестів безпеки проти розгорнутого середовища LocalStack. Це включає наші власні тести (написані на Python з використанням викликів AWS SDK) та будь-які інтегровані перевірки політики як код (наприклад, оцінка політик OPA або правил Terraform Compliance). Під час цього етапу конвеєр використовує кінцеві точки API AWS, перенаправлені на LocalStack, тому виклики, такі як «описати політику бакета S3» або «перелічити відкриті правила групи безпеки», повертають інформацію з пісочниці, а не з реальної хмари.
- Результати та демонтаж — агрегування результатів тестування. Якщо всі тести проходять, конвеєр продовжує дозволяти злиття/розгортання. Якщо будь-який тест не проходить, конвеєр повідомляє, який саме контроль безпеки не пройшов (наприклад, «TestFailed: S3 bucket my-app-bucket is public»). Інфраструктура в LocalStack може бути знищена. Розробники негайно отримують повідомлення про збої через інтерфейс CI.

Цей підхід ефективно вбудовує набір регресійних тестів безпеки в конвеєр розгортання. Кожна зміна коду автоматично оцінюється за десятками перевірок безпеки. Процес повністю автоматизований і виконується з кожним запитом на злиття, що гарантує, що жодна зміна не може «проскочити» без перевірки. Важливо, що використання локального емулятора хмари означає, що ми не несемо витрат на хмару і не чекаємо на реальне постачання інфраструктури під час цих тестів. Наші результати показують, що цей метод не тільки здійсненний, але й ефективний: у нашому випадку з вивченням AWS весь набір з ~20 тестів безпеки виконався менш ніж за хвилину. Це демонструє, що безпека інфраструктури може йти в ногу з циклами DevOps, і не створювати проблему уповільнення випусків через перевірки безпеки, хоча й матиме тенденцію до накопичення часу виконання.

Реалізація доказу концепції та висновки

Щоб підтвердити вищезазначені концепції, ми реалізували доказ концепції (PoC), використовуючи репрезентативний стек AWS, і оцінили його ефективність. Сценарій використання включав багатошарову веб-додаток: екземпляр EC2, що обслуговує веб-додаток за допомогою балансувальника навантаження AWS (ELB), фаєрвол AWS WAF, що захищає фронтенд, база даних RDS PostgreSQL, а на бекенді NextJS додаток та бакет S3 для статичного контенту або резервних копій. Ця конфігурація була обрана, оскільки вона надає різноманітні типи ресурсів і потенційні помилки конфігурації. Ми навмисно

ввели перемикачі або конфігурації в шаблон IaC, які могли перемикатися між безпечними та небезпечними станами, щоб перевірити, чи виявляє наша система проблеми. Наприклад, шаблон Terraform мав параметр для увімкнення або вимкнення шифрування бакета S3; під час одного з запусків тестів ми вимкнули його, щоб перевірити, чи не пройде тест перевірки шифрування, як і очікувалося. Аналогічно, ми могли змінити правила WAF на небезпечні налаштування, щоб змоделювати помилку конфігурації, або відкрити групу безпеки для публічного доступу та перевірити, чи блокує це конвеєр. Розробка набору тестів: ми розробили набір з близько 20 тестів безпеки, що охоплюють кілька доменів безпеки інфраструктури:

- Перевірка наявності ресурсів та правильної конфігурації: ці тести перевіряють, чи присутні певні захисні ресурси та чи правильно вони налаштовані. Наприклад, один тест стверджує, що веб-ACL WAF AWS розгорнуто перед додатком і що він містить певні критичні правила (наприклад, керовані правила AWS для загроз OWASP). Якщо ресурс WAF відсутній або якщо правила не відповідають очікуваним (можливо, хтось змінив WAF, щоб дозволити весь трафік), тест не проходить. Інший тест у цій категорії перевіряє, що CloudTrail (служба ведення журналів AWS) увімкнено для облікового запису принаймні з одним слідом, що захоплює управлінські події, оскільки ведення журналів є ключовим контролем безпеки.
- Тести контролю доступу: ці тести перевіряють ролі IAM, політики бакетів S3 та інші конфігурації доступу. Для S3 ми написали тест, щоб пройтися по всіх бакетах у стеці та стверджувати, що жоден з них не є публічно доступним (ні через політику бакета, ні через ACL). Для IAM ми перевірили, що жодна роль або політика користувача IAM у проекті не надає дій wildcard * на чутливі служби AWS. Ми також включили тест, щоб забезпечити обмеженість ролі IAM, що використовується екземпляром EC2, відповідно до принципу найменших привілеїв. Ці перевірки усувають ризик надто широкого доступу, що може призвести до ескалації привілеїв або витоку даних.
- Тести мережевої безпеки: ці тести забезпечують обмеженість мережевих конфігурацій. Наприклад, група безпеки EC2 не повинна дозволяти вхідний доступ з будь-якої IP-адреси (0.0.0.0/0) на неприпустимі порти. Наш тест спеціально дозволяв веб-трафік на портах 80/443, але позначав будь-які широко відкриті порти SSH або бази даних. Ми також перевірили, що екземпляр бази даних RDS не налаштований як «Доступний публічно» (налаштування RDS); він повинен бути доступний лише зсередини VPC. Ще один мережевий тест перевіряв, чи використовує ELB або EC2 відповідну конфігурацію TLS (обмежуючи сучасними шифрами), хоча це виходить за межі найкращих практик, пов’язаних з помилками конфігурації.
- Тестування захисту даних та моніторингу: тести в цій категорії перевіряли шифрування та моніторинг. Ми стверджували, що база даних RDS має увімкнене шифрування зберігання (AWS RDS дозволяє необов’язкове шифрування даних в спокої). Аналогічно, всі томи EBS, підключені до EC2, перевірялися на наявність шифрування. Ми підтвердили, що бакет S3 має увімкнене за замовчуванням шифрування на рівні бакета. Щодо моніторингу, крім CloudTrail, ми також забезпечили, щоб Amazon GuardDuty (служба виявлення загроз) була увімкнена в обліковому записі, за допомогою тесту виклику API. Якщо будь-що з цього не відповідало дійсності, тести це позначали.

Кожен тест по суті є невеликим скриптом, який запитує розгорнуту інфраструктуру (через AWS API або SDK в середовищі LocalStack) і порівнює результат з очікуваною умовою. Тести були написані переважно на Python з використанням boto3 (AWS SDK для Python), в поєднанні з асерціями за допомогою фреймворку unittest Python. У кількох випадках ми використовували Chef InSpec за його зручний синтаксис (як ілюстровано раніше) для таких ресурсів, як S3. Тести можуть виконуватися в двох режимах: (a) Динамічний режим проти запущеного стеку (що ми і робили з LocalStack у CI), або (b) Режим статичного аналізу, де це можливо — наприклад, перевірка JSON-плану Terraform на наявність відомих шаблонів (ми робили це для швидкої перевірки правил групи безпеки в плані перед розгортанням). Динамічний режим є більш потужним і точним, тоді як статичний режим є швидшим, але обмеженим; наша структура поєднує обидва, щоб максимізувати виявлення.

Висновки: запуски доказу концепції підтвердили, що тести успішно виявляли помилки конфігурації, введені в IaC. Наприклад, коли ми переключили S3-бакет на публічний, тест доступу до S3 у конвеєрі не пройшов і вивів помилку, вказуючи, що бакет є публічним, що запобігло розгортанню. Коли ми видалили шифрування з екземпляра RDS у конфігурації Terraform, наш тест шифрування для RDS позначив це. У ще одному випробуванні ми змоделювали відсутність ресурсу WAF (ніби розробник забув включити його в шаблон) — тест на наявність WAF не пройшов, зупинивши конвеєр і запобігши сценарію розгортання без WAF. Ці результати ілюструють основну цінність: система працює як автоматизована страхова сітка, виявляючи як пропуски, так і погані конфігурації. Більше того, швидкість запусків тестів на основі LocalStack спостерігалася в межах очікуваного діапазону (приблизно 30–60 секунд для повного набору). Це означає, що такий етап тестування безпеки є практичним у реальних CI-конвеєрах; він не вводить значної затримки в розгортання. У порівнянні, розгортання в реальному тестовому середовищі AWS і виконання подібних перевірок зайняло кілька хвилин, що підкреслює ефективність нашого підходу (прискорення досягається за рахунок уникнення мережевої затримки до AWS і пропуску затримок реального постачання ресурсів). Ще одне помітне спостереження полягало в відсутності хибнопозитивних сповіщень у наших власних тестах. Традиційні статичні сканери коду для конфігурацій хмари іноді піднімають хибні тривоги — наприклад, позначаючи використання певних ресурсів як небезпечних, навіть якщо контекст виправданий, або не маючи контексту, щоб побачити, що інший контроль пом’якшує проблему. Оскільки наші тести були спеціально розроблені та обізнані про контекст (і виконувалися проти фактичного стану в LocalStack), ми мали точні знання про середовище. Ми не стикалися з жодним випадком, коли тест не пройшов би помилково через не реальну проблему; кожен збій відповідав справжній прогалині в безпеці, яку ми ввели. Це важливий результат, оскільки підкреслює перевагу підходу з налаштуванням: коли розробники довіряють набору тестів, вони з більшою ймовірністю приймуть його. Готові інструменти іноді можуть переповнити команди знахідками, включаючи багато низькосерйозних або нерелевантних, що призводить до втоми від сповіщень. На відміну від цього, наш підхід дав змогу отримати дієвий, точний зворотний зв’язок.

Оцінка ефективності та обговорення

Ми оцінюємо ефективність «Інфраструктури як протестованого коду», порівнюючи її з традиційними підходами до безпеки IaC та вимірюючи, скільки помилок конфігурації запобігається. У базовому сценарії без інтегрованого тестування помилки конфігурації виявлялися б лише після розгортання (якщо взагалі виявилися б) за допомогою періодичних сканувань або, ще гірше, зловмисниками. У нашому сценарії PoC, ймовірно, такі проблеми,

як відкрите SSH-підключення або незашифрований S3, пройшли б непоміченими без тестування, оскільки ручні перевірки коду зазвичай зосереджені на функціональності, а не на деталях безпеки. У другому сценарії, використовуючи лише інструменти статичного аналізу (наприклад, лінтери Terraform або сканери політик, такі як Checkov), деякі проблеми були б виявлені (наприклад, Checkov має правила для відкритих груп безпеки та публічних S3), але інші можуть бути пропущені. Дійсно, статичний аналіз може пропустити проблеми, які залежать від конфігурації під час виконання або взаємодії між ресурсами (наприклад, він може не виявити неправильно налаштований WAF або ситуацію, коли група безпеки прийнятна сама по собі, але небезпечна в комбінації з іншою помилкою конфігурації). Крім того, статичні інструменти можуть позначати потенційні проблеми, які при ручному перегляді не є справжніми проблемами (хибні позитивні спрацювання), або їм можуть бракувати правил для дуже нестандартних сценаріїв. Наша повна стратегія тестування, яка включає динамічну перевірку, показала найкращі результати в нашій оцінці: вона виявила всі критичні помилки конфігурації, які ми навмисно ввели, і зробила це на етапі коміту коду. Кілька прогалин у наших тестах були такими, про які ми знали, що ще не реалізовані (наприклад, проблема з довірчими відносинами IAM надзвичайно складна); визнання цих прогалин допомагає визначити майбутню роботу.

Щоб кількісно оцінити переваги, ми також посилаємося на галузеві метрики. Як зазначалося, дані Pahl, Gunduz, Sezen, Ghamgosar та El Ioini (2025) вказують на ~33% меншу кількість інцидентів у хмарах у команд, які прийняли перевірки безпеки IaC в CI [8]. Це значне зниження, що підкреслює, що багато зламів є наслідком помилок конфігурації, яких можна уникнути. Наш підхід безпосередньо націлений на ці запобіжні міри. Ми також розглядаємо конкретні історичні інциденти: чи міг би цей підхід запобігти витокам данних в Capital One [7], Twilio, Imperva [10]? Це спекулятивно, але повчально. AWS WAF Capital One був неправильно налаштований таким чином, що дозволив зловмиснику отримати облікові дані AWS (через SSRF) [7]. Якби хмарне розгортання Capital One включало автоматизований тест для перевірки конфігурації WAF та найменших привілеїв (наприклад, тест, що стверджує, що «IAM-роль WAF не може перераховувати бакети S3 або читати дані»), він би не пройшов і сповістив би команду до розгортання. Насправді, аналіз цього інциденту показав, що безперервні тести безпеки, ймовірно, змогли б пом’якшити витік, виявивши помилку конфігурації брандмауера. Наша робота реалізує цей урок: написавши специфічні тести для відомих ризикованих помилок конфігурації (як-от надто широкі ролі або ненавмисний відкритий доступ), ми створюємо можливість виправити проблеми, перш ніж їх виявлять зловмисники. Ми стверджуємо, що якби підхід «Інфраструктура як протестований код» був впроваджений, багато зламів хмар через помилки конфігурації IaC можна було б уникнути. Ця заява підтверджується як нашими практичними результатами, так і загальними тенденціями щодо інцидентів, спричинених помилками конфігурації [10].

Компроміси та практичні міркування: реалізація парадигми також виявила практичні аспекти. Одним з потенційних занепокоєнь є витрати на обслуговування тестів. У міру зміни коду інфраструктури тести можуть вимагати оновлення; це аналогічно тому, як тести програм застосування потрібно підтримувати, коли змінюються функції. Ми виявили, що залучення тих самих інженерів, які пишуть код інфраструктури, до написання тестів робить зусилля керованими — вони природно оновлюють тести під час оновлення відповідного компонента IaC. Ще одне питання полягає в кривій навчання: хмарним інженерам може знадобитися ознайомитися з фреймворками тестування та написанням асерцій (що більше схоже на розробку програмного забезпечення). Однак це можна пом’якшити, надавши стартовий пакет з загальними тестами та чіткими прикладами. З часом написання тесту для

політики бакета S3 стає таким же простим, як написання ресурсу Terraform — це просто ще одна частина робочого процесу розробки. Існує також питання інтеграції з існуючими інструментами DevOps: наш підхід був реалізований з використанням інструментів з відкритим кодом, але підприємства можуть інтегрувати подібне тестування в інструменти, такі як AWS CodePipeline або Jenkins. Це здійсненно, оскільки більшість CI-інструментів можуть виконувати скрипти та Docker-образи — можна упакувати фреймворк тестування в контейнер, який працює в будь-якому конвеєрі. Ми також підкреслюємо, що наш підхід є доповнюючим до інших заходів безпеки. Він не замінює необхідність періодичного зовнішнього тестування безпеки або моделювання загроз; скоріше, він забезпечує базову гігієну, так що очевидні помилки конфігурації не стають легкодоступними для зловмисників. Аналогічно, наш конвеєр може включати як статичний, так і динамічний аналіз: ми запускали Checkov (статичний сканер) як частину CI, щоб виявити проблеми, для яких у нього є правила, а потім запускали наші динамічні тести для глибших перевірок [11]. Ця стратегія дала нам «найкраще з обох світів», поєднуючи широту статичних наборів правил з точністю динамічного тестування. Будь-які порушення, виявлені будь-яким з методів, призводили до збою збірки. На практиці ми помітили, що статичний сканер може виявити тривіальну помилку конфігурації дещо швидше (наприклад, «ACL бакета дозволяє публічний доступ»), тоді як наш динамічний тест також виявив це, але на хвилину пізніше. Наявність перекриваючих перевірок не є шкідливою і насправді підвищує впевненість у тому, що проблема дійсно існує (два незалежні методи погоджуються). У деяких випадках статичний аналіз не може визначити намір — наприклад, він може позначити всі IAM-ролі, які не мають MFA, навіть для неінтерактивних ролей. Наші спеціально налаштовані тести мали контекст, щоб ігнорувати певні очікувані умови, зосереджуючи увагу на справжніх ризиках, тим самим зменшуючи шум.

Парадигма «Інфраструктура як протестований код» виявилася ефективною в нашому дослідженні для проактивного захисту інфраструктури AWS. Ключові результати такі:

1) Запобігання помилкам конфігурації: помилки конфігурації були автоматично виявлені та виправлені до розгортання, що, ймовірно, запобігло виникненню пов’язаних вразливостей у продакшні.
2) Швидкий зворотний зв’язок: інтеграція в CI показала, що тести безпеки можуть виконуватися за секунди до хвилини, вписуючись в агільні робочі процеси без затримок у розробці.
3) Культурна зміна: можливо, менш кількісно вимірювальний, але спостережуваний ефект полягає в тому, що ставлення до змін інфраструктури як до коду з тестами сприяє культурі відповідальності та обережності в конфігурації — розробники звикають думати «якщо я змінюю цей ресурс, чи потрібно мені оновити тест безпеки або політику?» що є саме тим мисленням про безпеку за замовчуванням, якого прагнуть організації.

Також емпіричні результати виправдовують перехід до протестованого, перевіреного коду інфраструктури як до життєздатної практики для безпеки хмари. Оскільки хмарні розгортання стають дедалі складнішими, така автоматизація, ймовірно, стане незамінною. Ми продемонстрували це на AWS, але ті ж принципи застосовні до інших хмарних постачальників з відповідними інструментами (Azure/Azure Stack для емуляції тощо). Запобігаючи помилкам конфігурації, які потрапляють у продакшн, організації можуть значно зменшити свою ризикову поверхню в хмарі. Результати реалізації доказу концепції підкріплюють аргумент, що тестування безпеки повинно еволюціонувати разом з IaC — так само, як тестування програмних застосувань еволюціонувало з агільною розробкою — щоб підтримувати хмарні середовища в безпеці за замовчуванням.

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

Результати дослідження показали, що автоматизоване тестування може бути потужним механізмом для запобігання хибним конфігураціям інфраструктури AWS шляхом виявлення та виправлення помилок до розгортання у хмарі. Реалізація proof-of-concept підтвердила цей підхід: розглядаючи визначення інфраструктури як тестований код і впроваджуючи тести, орієнтовані на безпеку, у CI/CD-конвеєр, хибні конфігурації (такі як публічно доступні сховища або надто широкі ролі IAM) були виявлені на ранніх етапах, що дозволило уникнути потенційних вразливостей у продакшні. Ця проактивна стратегія зменшує людські помилки — які, за оцінками, є причиною 82% хибних конфігурацій у хмарі — і вирішує значну причину інцидентів у хмарній безпеці (майже 23% таких інцидентів виникають через хибно налаштовані хмарні ресурси) [13]. Інтегруючи безперервні автоматизовані перевірки, підхід забезпечує дотримання найкращих практик і надає впевненість у тому, що середовища AWS залишаються узгодженими з базовими політиками безпеки. Ключові внески цієї роботи є як практичними, так і науковими. З практичної сторони, дослідження надало робочий прототип, який слугує превентивним шаром безпеки [12] для конфігурацій AWS. Цей конвеєр автоматизованих тестів діє як захисний бар'єр, зупиняючи розгортання, якщо будь-яка конфігурація порушує визначені критерії безпеки, і таким чином забезпечує надійність і відповідність конфігурацій за замовчуванням. Не менш важливими є концептуальні внески: (1) формування інфраструктури як протестованого коду, що розширює парадигму інфраструктури як коду, вимагаючи суворої перевірки специфікацій інфраструктури, подібно до модульних тестів програмного забезпечення; (2) демонстрація тест-орієнтованої безпеки у хмарному забезпеченні, де вимоги до безпеки виражаються у вигляді тестів, які повинні пройти у конвеєрі — підхід, який дозволяє виявляти помилки конфігурації на ранніх етапах і забезпечує дотримання політик безпеки. Разом ці внески просувають практики DevSecOps, об'єднуючи дисципліну тестування програмного забезпечення з інженерією хмарної безпеки, таким чином усуваючи розрив між командами розробки, експлуатації та безпеки. У майбутньому з цього дослідження виникає кілька напрямків для подальших досліджень і розробок:

- Розширення покриття хибних конфігурацій: майбутні роботи повинні розширити підхід до тестування, щоб охопити ширший спектр сервісів AWS і сценаріїв хибних конфігурацій, а також дослідити узагальнення фреймворку для інших хмарних провайдерів (наприклад, Azure, GCP). Це збільшить застосовність підходу та вирішить ризики хибних конфігурацій у мультихмарних середовищах.
- Розширення технік тестування: можна дослідити більш складні методи перевірки, щоб виявляти тонкі або глибоко вкорінені вразливості конфігурації, які базові тести можуть пропустити. Наприклад, тестування на основі властивостей, фреймворки політик як коду або навіть формальна верифікація та автоматизоване доведення можуть бути інтегровані для покращення покриття тестами безпеки та забезпечення сильніших гарантій коректності для критичних хмарних ресурсів.
- Емпірична валідація: впровадження підходу тест-орієнтованої безпеки у реальні робочі процеси DevSecOps і вивчення його впливу буде надзвичайно цінним. Такі дослідження можуть вивчати прийняття розробниками, накладні витрати на продуктивність CI/CD і вплив підходу на зменшення

кількості інцидентів безпеки. Отримані інсайти підтвердять практичну ефективність інфраструктури як протестованого коду та підкреслять області для вдосконалення (наприклад, покращення зручності використання або зменшення кількості хибнопозитивних результатів).

- Академічний прогрес: з наукової точки зору, подальші дослідження можуть формалізувати концепції, представлені тут. Це включає розробку теоретичних моделей і метрик для покриття тестами безпеки та вдосконалення парадигми тест-орієнтованої безпеки. Закріплюючи ці ідеї в академічних рамках (наприклад, пов'язуючи покриття тестами безпеки зі зменшенням ризиків або забезпеченням відповідності), майбутні роботи можуть надихнути нові методології, які ще глибше інтегрують гарантії безпеки у життєвий цикл розробки програмного забезпечення. Такі міждисциплінарні дослідження розширять базу знань про те, як найкраще поєднувати автоматизоване тестування та хмарну безпеку, зрештою сприяючи більш стійким і верифікованим розгортанням хмарної інфраструктури.

Підсумовуючи, результати підкреслюють, що превентивна, тест-орієнтована валідація хмарної інфраструктури може суттєво покращити результати безпеки. Завдяки попередньому виявленню хибних конфігурацій за допомогою автоматизації, ця робота не лише вирішує нагальну практичну проблему у хмарній безпеці, але й закладає основу для еволюції як галузевої практики, так і академічних досліджень у напрямку більш надійних, безпечних за замовчуванням хмарних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Wright, D. (2020). The state of cloud security 2020 report: Understanding misconfiguration risk. Cloud Security Alliance. https://cloudsecurityalliance.org/blog/2020/05/05/the-state-of-cloud-security-2020-report-understanding-misconfiguration-risk/
2. Thales. (2024). Cloud security in 2024: Addressing the shifting landscape. Cloud Security Alliance. https://cloudsecurityalliance.org/blog/2024/06/27/cloud-security-in-2024-addressing-the-shifting-landscape
3. Stella, J. (2019). A technical analysis of the Capital One cloud misconfiguration breach. Cloud Security Alliance. https://cloudsecurityalliance.org/blog/2019/08/09/a-technical-analysis-of-the-capital-one-cloud-misconfiguration-breach/
4. War, A., Diallo, A., Habib, A., Klein, J., & Bissyandé, T. F. (2025). Vulnerabilities in infrastructure as code: What, how many, and who? Empirical Software Engineering, 30(5), Article 120. https://doi.org/10.1007/s10664-025-10672-8
5. Verdet, A., Hamdaqa, M., Da Silva, L., & Khomh, F. (2025). Assessing the adoption of security policies by developers in Terraform across different cloud providers. Empirical Software Engineering, 30, Article 74. https://doi.org/10.1007/s10664-024-10610-0
6. OWASP (Open Web Application Security Project). (2021). Infrastructure as Code Security Cheat Sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Infrastructure_as_Code_Security_Cheat_Sheet.html
7. Madnick, S. E. (2020). A case study of the Capital One data breach (Working Paper CISL-2020-07). MIT Cybersecurity at MIT Sloan. https://web.mit.edu/smadnick/www/wp/2020-07.pdf
8. Pahl, C., Gunduz, N. G., Sezen, Ö. C., Ghamgosar, A., & El Ioini, N. (2025). Infrastructure as Code – Technology review and research challenges. In Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025) (pp. 151–158). SciTePress. https://doi.org/10.5220/0013247700003950
9. AWS Labs. (n.d.). AWS Config Rule Development Kit (RDK) [Computer software]. GitHub. https://github.com/awslabs/aws-config-rdk
10. Guffey, J. & Li, Y. (2023). Cloud Service Misconfigurations: Emerging Threats, Enterprise Data Breaches and Solutions. https://doi.org/10.1109/CCWC57344.2023.10099296

11. Yeboah-Ofori, A. et al. (2024). Fortifying Cloud DevSecOps Security Using Terraform Infrastructure as Code Analysis Tools. https://doi.org/10.1109/ICECER62944.2024.10920371
12. Rahman, A. & Williams, L. (2021). "Different Kind of Smells: Security Smells in Infrastructure as Code Scripts," IEEE Security & Privacy, 19(3), 33-41. https://doi.org/10.1109/MSEC.2021.3065190
13. SentinelOne. (2025). 50+ Cloud Security Statistics in 2025. SentinelOne. https://www.sentinelone.com/cybersecurity-101/cloud-security/cloud-security-statistics/
14. Kostiuk, Yu. V., Skladannyi, P. M., Bebeshko, B. T., Khorolska, K. V., Rzaieva, S. L., & Vorokhob, M. V. (2025). *Information and communication systems security*. [Textbook] Kyiv: Borys Grinchenko Kyiv Metropolitan University.
15. Kostiuk, Yu. V., Skladannyi, P. M., Hulak, H. M., Bebeshko, B. T., Khorolska, K. V., & Rzaieva, S. L. (2025). *Information security systems.* [Textbook] Kyiv: Borys Grinchenko Kyiv Metropolitan University.
16. Hulak, H. M., Zhyltsov, O. B., Kyrychok, R. V., Korshun, N. V., & Skladannyi, P. M. (2023). *Enterprise information and cyber security.* [Textbook] Kyiv: Borys Grinchenko Kyiv Metropolitan University.

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 1 (29), 2025
ISSN 2663 - 4023

Ivan Parkhomenko
Candidate of Technical Sciences, Associate Professor, Head of CSIP Department
Faculty of Information Technology
Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
ORCID ID: 0000-0001-6889-9284
ivan.parkhomenko@knu.ua

Mykhailo Savonik
Postgraduate at CSIP Department
Faculty of Information Technology
Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
ORCID ID: 0009-0001-7622-978X
savonikm@fit.knu.ua

TESTING-BASED PREVENTION OF MISCONFIGURATION THREATS IN AWS INFRASTRUCTURE AS CODE

Abstract. Misconfigured cloud infrastructure has emerged as a prevalent and impactful threat vector in cybersecurity. In particular, organizations deploying services on Amazon Web Services (AWS) often face significant security risks due to incorrectly configured access controls, insufficient logging, weak network segmentation, and the absence of critical protections such as Web Application Firewalls (WAF). Despite the widespread adoption of Infrastructure as Code (IaC) tools (e.g., Terraform, AWS CloudFormation) to enforce predictable, version-controlled deployments, these IaC configurations typically undergo little to no systematic security testing. Unlike application code—which routinely undergoes unit, integration, and security testing—infrastructure code is seldom tested beyond basic static analysis or post-deployment monitoring. As a result, critical security misconfigurations can remain undetected until they are exploited by attackers. To address this gap, this paper proposes a novel approach termed "Infrastructure as Tested Code". By applying proven software testing techniques—such as test assertions and continuous integration workflows—to IaC, our framework enables pre-deployment validation of an AWS environment's security posture. We develop and evaluate a proof-of-concept implementation that automates security checks for Terraform-defined AWS resources, focusing on key configurations including WAF rules, Identity and Access Management (IAM) policies, S3 bucket permissions, and security group rules. This test suite is built with open-source tools (Chef InSpec and AWS SDKs) and runs within a CI/CD pipeline using LocalStack to emulate AWS services. Through this approach, developers and DevSecOps teams can detect and remediate misconfigurations early in the development lifecycle, long before infrastructure reaches a production environment. Our experimental evaluation shows that integrating automated security tests into the DevOps pipeline significantly strengthens cloud security and mitigates misconfiguration-driven vulnerabilities. Compared to traditional static analysis tools, our approach offers greater flexibility, supports environment-specific policies, and allows developers to codify custom, testable security assertions. Even a minimal test suite proved effective in catching high-risk misconfigurations that static checks overlooked. This paradigm complements existing cloud security tools (such as AWS Config, Checkov, and other policy-as-code frameworks) and can be seamlessly integrated into DevSecOps pipelines. Finally, the paper provides a detailed implementation guide, a real-world case study, and an analysis of practical trade-offs. We conclude that just as test-driven development improved software reliability, adopting a test-driven approach to infrastructure can become a critical strategy for proactively securing cloud environments. This work lays the groundwork for future research into formalizing security testing practices for IaC, benchmarking IaC security test coverage, and developing reusable libraries of security test assertions for AWS.

Keywords: Cloud Security; Infrastructure as Code (IaC); Misconfiguration Prevention; DevSecOps; AWS Security; Automated Security Testing; Policy-as-Code; Security Configuration Validation.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Wright, D. (2020). The state of cloud security 2020 report: Understanding misconfiguration risk. Cloud Security Alliance. https://cloudsecurityalliance.org/blog/2020/05/05/the-state-of-cloud-security-2020-report-understanding-misconfiguration-risk/
2. Thales. (2024). Cloud security in 2024: Addressing the shifting landscape. Cloud Security Alliance. https://cloudsecurityalliance.org/blog/2024/06/27/cloud-security-in-2024-addressing-the-shifting-landscape
3. Stella, J. (2019). A technical analysis of the Capital One cloud misconfiguration breach. Cloud Security Alliance. https://cloudsecurityalliance.org/blog/2019/08/09/a-technical-analysis-of-the-capital-one-cloud-misconfiguration-breach/
4. War, A., Diallo, A., Habib, A., Klein, J., & Bissyandé, T. F. (2025). Vulnerabilities in infrastructure as code: What, how many, and who? Empirical Software Engineering, 30(5), Article 120. https://doi.org/10.1007/s10664-025-10672-8
5. Verdet, A., Hamdaqa, M., Da Silva, L., & Khomh, F. (2025). Assessing the adoption of security policies by developers in Terraform across different cloud providers. Empirical Software Engineering, 30, Article 74. https://doi.org/10.1007/s10664-024-10610-0
6. OWASP (Open Web Application Security Project). (2021). Infrastructure as Code Security Cheat Sheet. OWASP Cheat Sheet Series. https://cheatsheetseries.owasp.org/cheatsheets/Infrastructure_as_Code_Security_Cheat_Sheet.html
7. Madnick, S. E. (2020). A case study of the Capital One data breach (Working Paper CISL-2020-07). MIT Cybersecurity at MIT Sloan. https://web.mit.edu/smadnick/www/wp/2020-07.pdf
8. Pahl, C., Gunduz, N. G., Sezen, Ö. C., Ghamgosar, A., & El Ioini, N. (2025). Infrastructure as Code – Technology review and research challenges. In Proceedings of the 15th International Conference on Cloud Computing and Services Science (CLOSER 2025) (pp. 151–158). SciTePress. https://doi.org/10.5220/0013247700003950
9. AWS Labs. (n.d.). AWS Config Rule Development Kit (RDK) [Computer software]. GitHub. https://github.com/awslabs/aws-config-rdk
10. Guffey, J. & Li, Y. (2023). Cloud Service Misconfigurations: Emerging Threats, Enterprise Data Breaches and Solutions. https://doi.org/10.1109/CCWC57344.2023.10099296
11. Yeboah-Ofori, A. et al. (2024). Fortifying Cloud DevSecOps Security Using Terraform Infrastructure as Code Analysis Tools (accepted version). https://doi.org/10.1109/ICECER62944.2024.10920371
12. Rahman, A. & Williams, L. (2021). "Different Kind of Smells: Security Smells in Infrastructure as Code Scripts," IEEE Security & Privacy, 19(3), 33-41. https://doi.org/10.1109/MSEC.2021.3065190
13. SentinelOne. (2025). 50+ Cloud Security Statistics in 2025. SentinelOne. https://www.sentinelone.com/cybersecurity-101/cloud-security/cloud-security-statistics/
14. Kostiuk, Yu. V., Skladannyi, P. M., Bebeshko, B. T., Khorolska, K. V., Rzaieva, S. L., & Vorokhob, M. V. (2025). *Information and communication systems security*. [Textbook] Kyiv: Borys Grinchenko Kyiv Metropolitan University.
15. Kostiuk, Yu. V., Skladannyi, P. M., Hulak, H. M., Bebeshko, B. T., Khorolska, K. V., & Rzaieva, S. L. (2025). *Information security systems.* [Textbook] Kyiv: Borys Grinchenko Kyiv Metropolitan University.
16. Hulak, H. M., Zhyltsov, O. B., Kyrychok, R. V., Korshun, N. V., & Skladannyi, P. M. (2023). *Enterprise information and cyber security.* [Textbook] Kyiv: Borys Grinchenko Kyiv Metropolitan University.