Indo Asian Journal of Management and Entrepreneurship (IAJOME)
‖ ISSN: 2319-2992, Impact Factor 5.007 ‖
‖ Peer-Reviewed | Open Access | International Journal ‖
‖ Volume: 17 | Issue: 07 | July 2026 | www.iajome.com ‖

Mitigating BOLA Vulnerabilities through Secure-by-Design Database-Scoped Row-Level Security

Underthe guidanceof

Dr.B.Anuja Beatrice,MCA.,M.Phil.,Ph.D.

Associate Professor & Head, Department of Computer Applications, Sri Krishna Arts and Science College

Aiswarya V R
Dept.ofComputerApplications
24BCA002
IIIBCAA

JohanFinny
Dept.ofComputerApplication
24BCA026
IIIBCAA

Abstract*—The rapid proliferation of microservices architectures across enterprise software ecosystems has fundamentally restructured howapplications expose and govern access to data resources. In these distributed environments, Broken Object Level Authorization(BOLA) has emerged asthe most pervasive and consequential vulnerability class catalogued by OWASP, accounting for a disproportionate share of real-world data breachincidents. Conventional defenses rely on application-layermiddleware filtersapplied by individualdevelopersacrosshundredsof discrete APIroutes—aparadigm that is structurally susceptible to human omission, documentation drift, and static analysis blind spots. This paper conducts a rigorousarchitectural analysisof BOLA'smechanicsand dissectswhycontemporary automated scannersfail to detect it reliably. Wethen proposeand evaluateasecure-by-design remedy: migrating access control enforcement from application code into the database engine itself using native Row-Level Security(RLS) policies. The proposed architecture guarantees that authorization is evaluated atomically at the storage boundary for every query, irrespective ofthe developer's vigilance, eliminating the entire class of BOLA vulnerabilities through structural design rather than procedural discipline.

***Keywords*—BrokenObjectLevelAuthorization(BOLA),APISecurity,MicroservicesSprawl,Row-LevelSecurity(RLS),Secure-by-Design.**

I. Introduction

The dominant architectural paradigm of the last decade has been the decomposition of large, monolithic software applications into discrete, independently deployable microservices. Where a legacy banking application once operated as a single unified binary servingall functions from a shared codebase, modernequivalentsconsistof dozensor even hundreds of specialized services—account management, transaction processing, fraud detection, notification dispatch—each operating as an autonomous process communicating exclusively through well-defined API contracts.

This architectural shift delivered transformative benefits: development teams can iterate independently on individual services, infrastructure can scale granularly in response to load, and technology diversity becomes operationally feasible. However, the transition introduced a structural side effect of profound security consequence: an aggressive multiplication of externally accessible API endpoints. Research by enterprise API management platforms indicates that large financial institutions and healthcare networks routinely expose in excess of 600 distinct API endpoints across their microservices landscapes. Each endpoint represents a discrete surface area that must individually enforce correct access control logic.

The critical architectural constraint governing all modern API ecosystems is that clients—whether mobile applications, single-page web applications, or partner integrations—are categorically prohibited from establishing direct database connections. The relational database or document store sits behind multiple network perimeters, accessible only to service processes operating within trusted internal networks.This means every data retrieval and mutation operation must traverse at leastoneAPIendpoint.TheAPIlayeristhereforenotmerelyaconveniencelayer;itistheexclusive,non-negotiableoperational gateway to all data assets.

Within this context, the identity verification performed at the API boundary assumes supreme importance. When an API endpointreceivesarequest,itmustanswertwodistinctquestions:Isthisrequesterwhotheyclaimtobe?(authentication) and Does this specific requester have the right to access this specific data record? (object-level authorization). It is the second question—object-level authorization—that BOLA targets. The distinction between valid identity and valid data ownership is subtle but foundational, and it is precisely this subtlety that makes BOLA systematically difficult to enforce correctly across hundreds of independent service endpoints.

ThispaperisstructuredasfollowsSectionIIdissectstheprecisemechanicsofBOLAanditsattacktaxonomy.SectionIII surveys how prevalent development frameworks approach object-level authorization and catalogues their inherent failure modes. Section IV evaluates contemporary automated scanning tools and reveals their fundamental architectural blind spots withrespecttoBOLA.SectionVpresentsanoperationalhealthcarecasestudythatexposeshowthesevulnerabilitiesmanifest inproduction-gradesystems.SectionVIintroducestheproposedremedy—database-scopedRow-LevelSecurity—withprecise technical implementation details. Section VII synthesizes comparative outcomes and draws conclusions.

Indo Asian Journal of Management and Entrepreneurship (IAJOME)
‖ ISSN: 2319-2992, Impact Factor 5.007 ‖
‖ Peer-Reviewed | Open Access | International Journal ‖
‖ Volume: 17 | Issue: 07 | July 2026 | www.iajome.com ‖

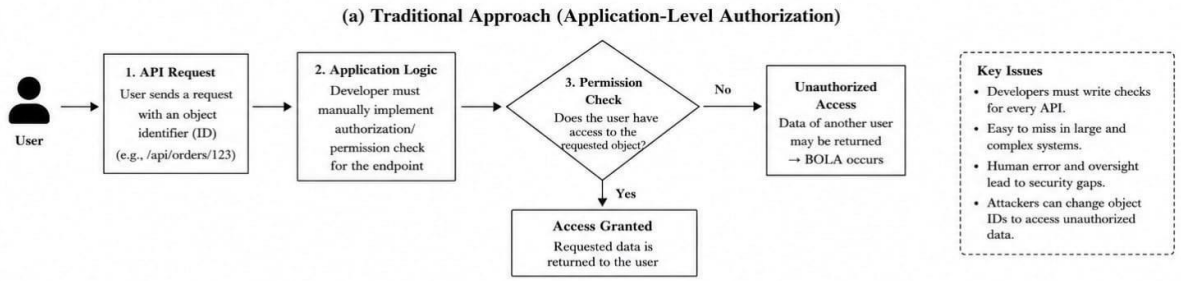

Fig.1.Traditionalapplication-levelauthorizationapproach,showinghowmanuallyimplementedpermissionchecksattheendpointcreatea singlepointofhuman-error failure that leads to BOLA.

II. TheMechanicsof BOLA:Anatomyof anAccessControlVoid

Broken ObjectLevelAuthorization arises when an APIendpointaccepts a client-supplied identifier—an accountnumber, record ID, order reference, or any other object key—and retrieves or manipulates the corresponding data record without verifyingthattheauthenticatedrequesteristhelegitimateownerof,orhasbeenexplicitlygrantedaccessto,thatspecificrecord. Thevulnerabilityisstructurallydistinctfromauthenticationfailures:theuser'sidentitytokenmaybeperfectlyvalid,thesession may be active, and the authentication middleware may report success. The failure occurs exclusively at the object ownership verification step.

ThreedistinctescalationpatternsdefinetheBOLAattacksurface.HorizontalPrivilegeEscalationrepresentsthecanonical form: a standard user accesses another standard user's data of equal privilege level simply by substituting a different object identifier. Vertical Privilege Escalation occurs when alower-privilege useraccesses records ownedby or restricted to higher-privilege roles, such as a patient record accessible only to attending physicians. Tenant Isolation BOLA is the multi-tenancy variant, in which a user of one organizational tenant accesses records belonging to a wholly separate tenant by manipulating identifiers that encode tenant boundaries.

A. PracticalAttackScenario

Consider a healthcare portal API that exposes patient lab result records. A legitimate physician authenticates successfully andreceivesaJSONWebTokenencodingtheiridentity.TheyinvoketheendpointGET/api/v1/labs/results?account_id=84920 to retrieve their assigned patient's data. The application middleware validates the JWT, confirms the session is active, and proceeds to execute the database query. However, an adversarial actor in possession of a valid token—perhaps a patient who registereda consumeraccount—modifies the queryparameterfromaccount_id=84920toaccount_id=84921andtransmits an otherwise identical request.

Theserver,havingvalidatedthetokencorrectly,proceedstoexecuteSELECT*FROMlab_resultsWHEREaccount_id= 84921withoutverifyingwhethertherequestingidentityhasanyassignedrelationshiptopatient84921.Thedatabasedutifully returns the record, and the server transmits a JSON payload of the following form: { "patient_id": 84921, "patient_name": "Priya Sharma", "diagnosis": "Type-2 Diabetes", "test_date": "2024-11-14", "physician_assigned": "Dr. Ramesh Kumar" }. Sensitive protected health information has leaked entirely due to a single missing ownership verification clause. The authentication layer performed flawlessly; the object-level authorization layer was simply absent.

III. TheDefensiveLandscape:Application-FrameworkMatrix

Contemporary web development frameworks each approach the object-level authorization problem through distinct mechanisms,andeachintroducesitsowncharacteristichuman-errorfailuremode.Understandingthesepatternsisprerequisite to appreciating why structural rather than procedural remedies are necessary.

A. UnopinionatedMiddleware(Node.js/Express.js)

Express.js, the dominant Node.js HTTP framework, is deliberately unopinionated regarding security architecture. Authorization logic must be manually authored and inserted as middleware functions into each route definition. In a microservice exposing 600 distinct routes, each route must independently invoke ownership verification logic. There is no inheritance mechanism, no framework-enforced policy template, and no compile-time guarantee that a verification function has been attached.A developerworking underdeadline pressure who adds a new route and neglects to copythe authorization middleware block creates a zero-day vulnerability the moment the code is deployed. Static analysis tools cannot reliably distinguish intentional public endpoints from accidentally unprotected ones.

B. Object-RelationalMappingLayers(Python/DjangoRESTFramework)

Django REST Framework introduces the concept of queryset filtering, wherein the get_queryset() method on a ViewSet class is overridden to append ownership filters to all queries for that resource. This is architecturally superior to per-route middleware: filters defined on the ViewSet class apply to all actions unless explicitly overridden. However, the guarantee is conditional. A developer who creates a new ViewSet for a related resource type must independently remember to override get_queryset()andapplythecorrectownershippredicate.Newresourcetypesintroducedunderfeaturepressureroutinelyship with the default unfiltered queryset, exposing all records to any authenticated requester.

C. EnterpriseDeclarativeAnnotations(Java/SpringBootSecurity)

SpringBootSecurityenablesmethod-levelauthorizationthroughdeclarativeannotationssuchas@PreAuthorize("#userId == authentication.principal.id"), applied directly to service layer methods. This approach offers the syntactic benefit of co-locatingauthorizationlogicwiththebusinessmethod,improvingcodereadability.However,themechanismexhibitsacritical limitation under dynamic permission models. The annotation expression is evaluated at method invocation time against the security context, but relational permission states that change between authentication events—such as a physician being reassignedtoadifferentwardmid-shift—arenotautomaticallyreflected.Theannotationencodesapoint-in-timeauthorization rule that becomes semantically incorrect as underlying data relationships evolve.

TABLEI.FRAMEWORKAUTHORIZATIONAPPROACHESANDHUMANCONFIGURATIONRISKS

FrameworkEnvironment	AuthorizationMethod	PrimaryHumanConfig.Risk
Node.js/Express.js	Manualmiddlewarechecksper route	Copy-pasteomissionacross600+routes
Python/DjangoREST	Customquerysetfilteroverrides	Filtersaccidentallyomittedonnewendpoints
Java/ SpringBoot	@PreAuthorizeannotations	Failswithdynamicruntimepermission changes

IV. Evaluationof AutomatedScannersandTheirSystemicBlindSpots

The security industry has invested substantially in automated tooling to detect BOLA-class vulnerabilities. However, the fundamental architectural properties of BOLA—that it manifests in the interaction between a valid identity and a valid data record at runtime—render it exceptionally resistant to static and specification-driven analysis.

A. Spec-DrivenScanners (Links2CPN)

Specification-driven scanners parse formal API documentation—typically OpenAPI or Swagger definitions—and synthesize test cases that probe documented endpoints for authorization boundary violations. This approach functions well when API documentation is synchronised with deployed code. In high-velocity microservices development environments, however, documentation drift is endemic. Development teams regularly ship new endpoint versions, modify parameter semantics, and deprecate routes without updating specification files. A scanner operating against a stale specification has no visibility into the undocumented endpoints. Modified routes that differ from their documented counterparts simply do not appearinthescanner'stestsurface.Theattacksurfaceinvisibletodocumentationispreciselythesurfacemostlikelyto harbour authorization oversights.

B. StaticCodeAnalyzers(BolaRay)

Static analysis tools traverse application source code—typically constructing Abstract Syntax Trees—seeking patterns indicative of missing authorization checks. A fundamental trade-off in static analysis is between coverage and performance. Tools that attempt complete data-flow analysis across large codebases incur prohibitive computational costs. To maintain scanning speeds compatible with CI/CD pipeline integration, tools such as BolaRay apply conservative heuristics: they prioritizewriteoperations(INSERT,UPDATE,DELETE)andintentionallydeprioritizeorentirelybypassread-onlySELECT queries. This optimization creates a catastrophic blind spot. BOLA's most damaging manifestation is data leakage through unauthorized reads. An endpoint that returns another user's medical records, financial transactions, or personal messages by executing a SELECT query is precisely the scenario that the scanner has elected to skip.

C. CatalogandNetworkTrafficMonitors(Beacon)

Traffic monitoring tools observe the network layer, cataloguing API calls and attempting to correlate response data with requester identity to detect anomalous cross-user data access. The fundamental architectural obstacle is database connection pooling. Modern application servers maintain persistent connection pools to their databases, reusing connections across multiple client requests to avoid the overhead of establishing new connections per request. At the database transaction layer, every query arrives from the application server's pool identity, not from the originating client's identity. The database has no nativevisibilityintowhichclientrequestinitiatedagivenquery.Beacon-classtoolsattemptingtotracedataaccessbackto the

Indo Asian Journal of Management and Entrepreneurship (IAJOME)
ISSN: 2319-2992, Impact Factor 5.007
Peer-Reviewed | Open Access | International Journal
Volume: 17 | Issue: 07 | July 2026 | www.iajome.com

originating user identity encounter a fundamental opacity at the connection pool boundary that cannot be resolved throughnetwork observation alone.

TABLEII.AUTOMATEDSCANNERCLASSESANDTHEIRARCHITECTURALBLINDSPOTS

ScannerClass	TargetMetric/Input	CoreArchitecturalBlind Spot
Spec-Driven(Links2CPN)	OpenAPI/Swaggerdocumentation	Documentationdriftfromrapidcodeupdates
StaticAnalyzer(BolaRay)	SourcecodeASTtraversal	IntentionallyskipsSELECT queries
TrafficMonitor(Beacon)	Networkrequest/responselogs	Connectionpoolinghidesclientidentity

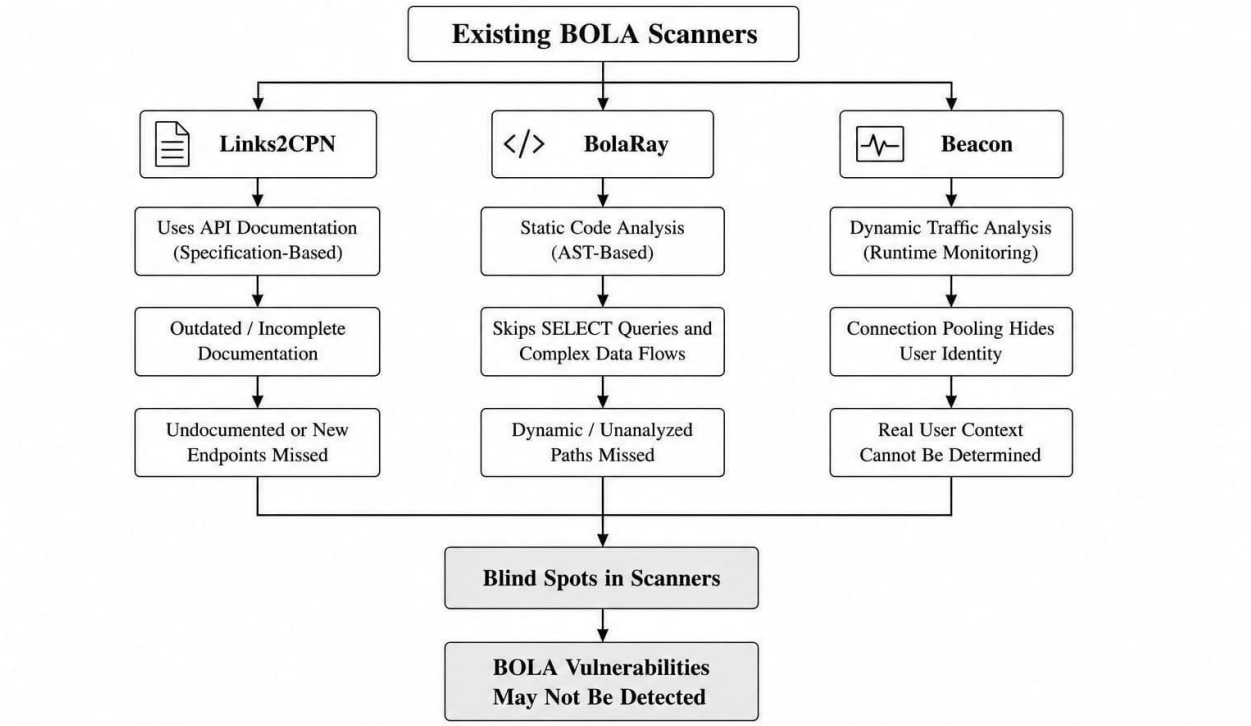

Fig.2.LimitationsofexistingBOLA scannersandtheirinherentblindspots.

V. AppliedCaseStudy:High-VelocityHealthcareAPI Architectures

Thehealthcaredomainprovidesaninstructivecasestudy forBOLAanalysisbecauseitcombinesextremedatasensitivity, complexandrapidlychangingrelationalpermissionstructures,highregulatorystakes,andintensedevelopmentvelocitydriven by digital health transformation mandates.

ConsideraregionalhospitalnetworkoperatingamodernElectronicHealthRecord(EHR)platform.Theplatformexposes clinical data through a RESTful microservices API consumed by the hospital's mobile physician application, nursing station terminals, and administrative dashboards. One operational endpoint is: GET /api/v1/labs/export?id=9021. This endpoint retrieves a complete laboratory results report for a patient identified by the id parameter.

Theapplicationmiddlewarecorrectlyvalidatestherequestingphysician'sJWT,confirminganactivesession.However,the endpoint handler proceeds to execute the database query using only the supplied patient ID, without evaluating whether the requesting physician has an active care assignment to that patient. The oversight is understandable in isolation: the endpoint was built by a developer focused on the data retrieval logic, who assumed that the patient ID would be communicated to the physician only through the application's own assignment workflow. The assumption fails the moment any actor—whether an external adversary or an over-curious internal user—directly constructs the URL with a different ID.

Indo Asian Journal of ement and Entrepreneurship (IAJOME)
ISSN: 2319-2992, Impact Factor 5.007
Peer-Reviewed | Open Access | International Journal
Volume: 17 | Issue: 07 | July 2026 | www.iajome.com

ThestaleauthorizationwindowproblembecomesacuteinEmergencyRoomworkflows.WhenapatientarrivesintheER, their record is temporarily accessible to the entire on-duty ER team pending primary physician assignment. Once assigned, access should narrow to the assigned physician and their care team. If the patient is subsequently discharged or transferred, access should be revoked immediately. Application-layer filters that encode the authorization state at the moment of session initiation introduce a window of incorrect access: a physician whose session was established before the patient's discharge retainsaccesstothatpatient'srecordsforthedurationoftheirsession,becausetheapplication'sin-memoryauthorizationcache hasnotbeeninvalidated.Inenvironmentswheresessionspersistforaneight-hourclinicalshift,thisstaleauthorizationwindow can remain open for hours after the underlying permission relationship has terminated.

VI. TheProposedSolution:ASecure-by-DesignDatabase-ScopedArchitecture

Database-scopedRow-LevelSecurityrepresentsastructuralinversionoftheauthorizationenforcementmodel.Ratherthan placing the burden of authorization logic on every application developer writing every API endpoint, RLS encodes access control rules as database-native policies that are evaluated automatically by the storage engine for every query that touches a protected table, without exception and without developer intervention.

The technical implementation proceeds in two steps. First, the API server—having authenticated the requesting user—passes the verified user identity to the database session through a localized session variable before executing any business query:

```
--Step 1:Establishidentitycontext
SETLOCALapp.current_user_id='verified_user_id';

--Step2:Executebusinessquery(RLSenforcesownership) SELECT *
FROM lab_results WHERE patient_id = $1;
```

The SET LOCAL statement confines the identity binding to the current transaction, ensuring that connection pool reuse doesnotpermitidentityleakageacrossclientsessions.Thesubsequentbusinessqueryiswrittenwithoutanyexplicitownership predicate—the developer's query is simple and readable.

Second,thedatabaseadministratorprovisionsaRow-LevelSecuritypolicyontheprotectedtable:

```
--EnableRLSontheprotectedtable
ALTERTABLElab_resultsENABLEROWLEVELSECURITY;

--Createaccesspolicy
CREATEPOLICYphysician_access_policy ON
  lab_results
  USING(
    patient_idIN(
      SELECTpatient_idFROMcare_assignments WHERE
      physician_id =
        current_setting('app.current_user_id')
      AND assignment_status = 'Active'
    )
  );
```

Once this policyis inplace,the storage engine intercepts everyqueryagainstlab_results andappends the policypredicate asamandatoryWHEREclausebeforeexecution.ThebusinessquerySELECT *FROMlab_resultsWHEREpatient_id = $1 executesinternallyasSELECT*FROMlab_resultsWHEREpatient_id=$1ANDpatient_idIN(SELECTpatient_idFROM care_assignments WHERE physician_id = current_setting('app.current_user_id') AND assignment_status = 'Active').

The security implications are profound and comprehensive. If an adversary modifies the patient ID parameter in their requestURL,theapplicationserver'sbusinessquerywillincludetheattacker-suppliedIDintheWHEREclause.However,the RLS policy will simultaneously evaluate whether that patient ID belongs to the care assignments of the authenticated user. If it does not, the policy filter eliminates all candidate rows before the result set is formed, and the query returns exactly zero rows.Nodatais leaked,noerroris exposedthatwouldconfirmthe recordexists,andthe applicationcode thatformulatedthe query required no modification.

Critically, because the policy definition references the care_assignments table with a live status predicate (assignment_status = 'Active'), the authorization evaluation is performed fresh against current data state at query execution time. A patientwho was discharged fiveminutesago will havetheircare_assignmentsrecord updated toassignment_status = 'Discharged',andallsubsequentqueries—regardlessofanyphysician'ssessionage—willimmediatelyreturnzerorowsforthat patient's data. The stale authorization window problem is eliminated structurally.

The scanner blind spot is simultaneously resolved. A static analyzer that previously ignored SELECT queries now encountersqueriesthatreturnzerorowsforunauthorizedidentifiers.Thedifferenceinresultsetsizesbetweenauthorized and unauthorizedqueriesisthedetectionsignal.Furthermore,becausetheSELECTquery itselfcorrectlyreturnsnodatawhenthe RLS policy is active, the system produces the semantically correct outcome—data not returned to unauthorized requesters—independent of whether any scanner detects the pattern.

Indo Asian Journal of Management and Entrepreneurship (IAJOME)
ISSN: 2319-2992, Impact Factor 5.007
Peer-Reviewed | Open Access | International Journal
Volume: 17 | Issue: 07 | July 2026 | www.iajome.com

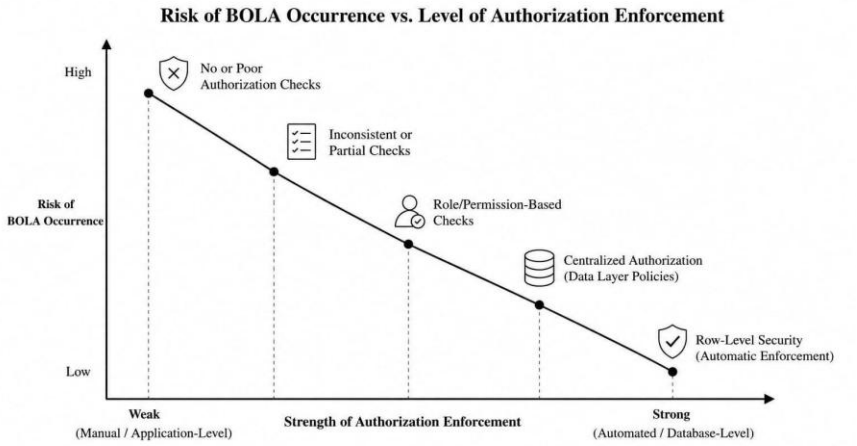

Fig.3.Conceptualrelationshipbetweenthestrengthofauthorizationenforcementandtheriskof BrokenObjectLevelAuthorization (BOLA).

VII. OutcomeandConclusion

This paperhas tracedthe architectural evolution from monolithic applicationsto microservices anddemonstratedhow the resulting endpoint proliferation creates a structural environment in which object-level authorization failures—BOLA vulnerabilities—are both systematically easy to introduce and systematically difficult to detect. The analysis of leading development frameworks revealed that each approach encodes a specific human-error failure mode: Express.js routes fail throughcopy-pasteomission,DjangoViewSetsfailthroughuncustomizedquerysets,andSpringBootannotationsfailthrough stale runtime permission states.

The evaluation of automatedscanning tools demonstrated thateachscannerclass carriesan architectural blind spot that is notincidentalbutstructural:specification-driventoolsaredefeatedbydocumentationdrift,staticanalyzersdeliberatelybypass theread-onlyqueryclassmostexploitedbyBOLA,andtrafficmonitorscannotresolveclientidentitythroughconnectionpool opacity. The healthcare case study concretised these failure modes in a high-stakes production context, showing how stale authorization windows arising from application-layer state management create real-world exposure windows measured in hours.

The proposed database-scoped Row-Level Security architecture addresses each of these failure modes through a single structural intervention. By relocating authorization enforcement from application code to the database storage engine, the system ensures that access control is evaluated with identical correctness for every query, through every endpoint, regardless of the developer's discipline, the scanner's coverage, or the age of the application session. The developer experience is simplified—business queries require no authorization predicates—while the security guarantee is strengthened from probabilistic to structural.

TABLEIII.COMPARATIVEANALYSIS:TRADITIONALAPP-LAYERVS.DATABASERLSARCHITECTURE

ComparisonAxis	TraditionalApp-LayerScanning	ProposedDatabaseRLSSolution
DevelopmentVelocity	Slowswitheachnewendpoint added	Noadditionalcoderequiredperendpoint
HumanConfig.Risk	High—developermustaddchecksmanually	Zero—policyenforcedatstorageengine
ScannerOmissionVuln.	SELECTqueriessilentlybypassed	SELECT returns0rows;nodataexposed
LifecycleStateAdapt.	Stalefiltersafterstatechange	Policyre-evaluatedoneveryquery

The secure-by-design principle embodied by database-scopedRLS represents a maturation of the API security discipline: movingfromamodelthatdependsondeveloperconsistentlymakingcorrectsecuritydecisionsacrosshundredsofendpoints to a model that makes the secure outcome the automatic structural consequence of any query execution. Future research directions include extending RLS policy frameworks to support attribute-based access control models, integrating policy auditing pipelines into CI/CD workflows, and evaluating RLS performance characteristics under high-concurrency microservices workloads.

References

[1] OWASPFoundation,"OWASPAPISecurityTop102023:API1:2023BrokenObjectLevelAuthorization,"OWASP,2023.[Online].Available:https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/
[2] E.SheridanandJ.Leeson,"ExploitingObject-LevelAuthorizationFlawsinRESTfulAPIs,"inProc.IEEEInt.Conf.onCybersecurityandCloudComputing (CSCloud), pp. 112–119, 2022.
[3] PostgreSQLGlobalDevelopmentGroup,"RowSecurityPolicies,"PostgreSQL16Documentation,2024.[Online].Available:https://www.postgresql.org/docs/current/ddl-rowsecurity.html
[4] A.Barua,M. A.R. Bhuiyan, andK.Koscher,"Links2CPN:Automated BOLADetectionviaOpenAPI SpecificationAnalysis," IEEE Trans.DependableSecure Comput., vol. 21, no. 2, pp. 583–596, 2024.
[5] R.FieldingandJ.Reschke,"HypertextTransferProtocol(HTTP/1.1):SemanticsandContent,"RFC7231,IETF, 2014.
[6] N.Jain,S.Arora,andP.Kulkarni,"BolaRay:StaticAnalysisTechniquesforAPIAuthorizationVulnerabilityDetection," J.Inf.Secur.Appl.,vol.78,pp.103–118,2023.
[7] H. Hu, G. Ahn, and K. Bhargava, "Dynamic Access Control in Multi-Tenant Cloud Environments Using Database-Native Policy Engines," ACM Trans.Inf. Syst. Secur., vol. 26, no. 1, pp. 1–28, 2023.
[8] M.Zalewski,"HealthcareAPISecurity:AddressingIdentityFragmentationinFederatedEHRMicroservices,"inProc.ACMSymp.Appl.Comput.(SAC), pp. 1047–1055, 2023.
[9] P.DeRyck,"APISecurityBestPractices:PreventingBrokenObjectLevelAuthorization,"APISecurityResearch,2023.[Online].Available:https://pragmaticwebsecurity.com
[10] Microsoft, "Secure API Development and Authorization Guidance," Microsoft Learn, 2024. [Online]. Available:https://learn.microsoft.com/aspnet/core/security
[11] NIST,"SecurityandPrivacyControlsforInformationSystemsandOrganizations,"NISTSpecialPublication800-53Rev.5,NationalInstituteofStandards and Technology, 2020.
[12] Google Cloud,"BestPracticesforAPISecurity,"GoogleCloudDocumentation,2024.[Online].Available:https://cloud.google.com/apis/docs/security-best-practices
[13] OpenWorldwideApplicationSecurityProject(OWASP),"AuthorizationCheatSheet,"OWASPFoundation,2024.[Online].Available:https://cheatsheetseries.owasp.org
[14] C.Richardson,"MicroservicesSecurityPatterns:AuthorizationandAccessControl,"Microservices.io,2023.[Online].Available:https://microservices.io
[15] S.Jajodia,P.Samarati,andV.S.Subrahmanian,"ALogicalLanguage forExpressingAuthorizations,"inProc.IEEESymposiumonSecurityandPrivacy, pp. 31–42, 1997.
[16] NIST,"ZeroTrustArchitecture,"NISTSpecialPublication800-207,NationalInstituteofStandardsandTechnology,Aug. 2020.
[17] D.Ferraiolo,D.Kuhn,andR.Chandramouli,Role-BasedAccessControl,2nded.Norwood,MA,USA:ArtechHouse, 2007.
[18] AmazonWebServices(AWS),"SecurityBestPracticesforAmazonAPIGateway,"AWSDocumentation,2024.[Online].Available:https://docs.aws.amazon.com/apigateway
[19] OracleCorporation,"OracleDatabaseSecurityGuide,"OracleDatabase23aiDocumentation,2024.[Online].Available:https://docs.oracle.com
[20] M.Colesky, J. Hoepman, and C. Hillen, "ACritical Analysis of Privacy Design Strategies," in Proc.IEEE Security andPrivacy Workshops, pp. 33–40,2016.
[21] J.Saltzer andM.Schroeder,"TheProtectionof InformationinComputerSystems,"Proc.IEEE,vol.63,no.9,pp.1278–1308,Sept. 1975.
[22] PostgreSQLGlobalDevelopmentGroup,"PostgreSQL17Documentation:RowSecurityPolicies,"PostgreSQLDocumentation,2025.[Online].Available:https://www.postgresql.org/docs/current/ddl-rowsecurity.html