

SQL Injection Forsvarsmekanismer

En Komparativ Undersøgelse på Tværs af Programmeringssprog og Frameworks

Mikkel Bak Markers

3. januar 2026

Resumé

I de mørkeste hjørner af nettet udklækkes frygtindgydende orme og forfærdelige hacks. Alligevel, mellem dæmoner og hunde, består en simpel trussel: SQL injection. Trods årtiers defensiv viden plager denne sårbarhed fortsat moderne applikationer og gør det muligt for angribere at manipulere databaseforespørgsler og kompromittere følsomme data.

Denne undersøgelse gennemfører en komparativ analyse af SQL injection forsvarsmekanismer på tværs af to dominerende webudviklingsøkosystemer: Python med Flask/SQLAlchemy og Node.js med Express. Gennem implementering af både sårbare og sikre versioner af ækvivalente REST API'er demonstrerer vi almindelige angrebsvektorer, herunder autentificeringsomgåelse og union-baseret dataekstraktion, og evaluerer derefter effektiviteten af parameteriserede forespørgsler og ORM-frameworks i at forhindre disse angreb.

Vores resultater viser, at begge økosystemer succesfuldt blokerer SQL injection ved brug af korrekte defensive mønstre, men adskiller sig markant i, hvor let udviklere ved et uheld kan introducere sårbarheder. Pythons f-streng-formatering og JavaScripts template literals udgør lignende fristelser mod usikker kode, mens SQLAlchemy og sql.js parameteriserede forespørgsler giver lige robust beskyttelse. Undersøgelsen afsluttes med praktiske anbefalinger til udviklere og organisationer, der søger at balancere sikkerhed, udvikleroplevelse og ydeevne i deres teknologivalg.

Indhold

| | | |
|----------|---|-----------|
| 1 | Introduktion | 4 |
| 1.1 | Motivation | 4 |
| 2 | Problemformulering | 4 |
| 2.1 | Underspørgsmål | 5 |
| 3 | Metode | 5 |
| 3.1 | Forskningstilgang | 5 |
| 3.2 | Implementeringsstrategi | 5 |
| 3.3 | Evalueringskriterier | 5 |
| 3.3.1 | Sikkerhedsrobusthed | 5 |
| 3.3.2 | Udvikleroplevelse | 5 |
| 3.3.3 | Ydeevneovervejelser | 6 |
| 3.4 | Testmiljø | 6 |
| 4 | Analyse og Resultater | 6 |
| 4.1 | Teoretisk Grundlag | 6 |
| 4.2 | Python og SQLAlchemy | 6 |
| 4.2.1 | Sårbar Implementering | 6 |
| 4.2.2 | Sikret Implementering | 7 |
| 4.2.3 | Evaluering | 7 |
| 4.3 | Node.js og Express | 8 |
| 4.3.1 | Sårbar Implementering | 8 |
| 4.3.2 | Sikret Implementering | 8 |
| 4.3.3 | Evaluering | 9 |
| 4.4 | Tværsproglig Sammenligning | 9 |
| 4.4.1 | Sikkerhedssammenligning | 9 |
| 4.4.2 | Sammenligning af Udvikleroplevelse | 9 |
| 5 | Konklusion | 9 |
| 5.1 | Sammenfatning af Resultater | 9 |
| 5.2 | Besvarelse af Forskningsspørgsmålet | 10 |
| 5.3 | Bedste Praksisser og anbefalinger | 10 |
| 5.4 | Begrænsninger og Fremtidigt Arbejde | 10 |
| 5.5 | Refleksion | 10 |
| A | Oversigt over Angrebstestresultater | 11 |

B Repository-struktur**11**

1 Introduktion

Trods årtiers bevidsthed og modforanstaltninger forbliver SQL injection blandt de mest kritiske sårbarheder i webapplikationer og rangerer konsekvent i OWASPs Top 10 sikkerhedsrisici [4]. Disse angreb udnytter ukorrekt håndtering af brugerinput i databaseforespørgsler, hvilket tillader ondsindede aktører at manipulere SQL-sætninger og kompromittere datafortrolighed, integritet og tilgængelighed [2, 5].

Mens det grundlæggende defensive princip om at “aldrig stole på brugerinput” er veletableret, varierer den praktiske implementering dramatisk på tværs af programmeringsøkosystemer. Denne undersøgelse gennemfører en komparativ analyse af SQL injection forsvarsmekanismer i to udbredte webudviklingsplatforme: Python med Flask/SQLAlchemy og Node.js med Express. Vi undersøger ikke kun, hvor robust sikkerheden er i hver tilgang, men også udvikleroplevelsens implikationer, der påvirker adoption af sikre praksisser i den virkelige verden.

1.1 Motivation

Undersøgelsens aktualitet understreges af den fortsatte udbredelse og omkostning ved SQL injection-angreb. IBMs 2023 Cost of a Data Breach Report afslører, at det gennemsnitlige databrud koster organisationer \$4,45 millioner USD, mens brud i sundhedssektoren i gennemsnit koster \$10,93 millioner, det højeste af alle brancher [3]. Webapplikationsangreb forbliver en primær trusselsvektor på tværs af kritiske sektorer, herunder sundhed, finans og offentlig administration [1], hvor disse angreb udgør en væsentlig del af brudhændelser [6].

Alligevel fortsætter udviklere uvidende med at introducere SQL injection-sårbarheder gennem framework-misbrug, utilstrækkelig træning eller ved at prioritere hastighed over sikkerhed. Denne vedvarende tendens antyder en grundlæggende afkobling mellem sikre kodningsprincipper og praktisk implementering.

Denne komparative undersøgelse adresserer et kritisk hul: mens individuelle sprogfællesskaber dokumenterer deres egne defensive mønstre, er tværsproglige analyser, der afslører hvilke økosystemer der gør sikker kode til *standarden* snarere end *undtagelsen*, sjældne. Vi fokuserer specifikt på Python og Node.js, da de repræsenterer to af de mest populære valg til moderne webudvikling med tydeligt forskellige tilgange til typesystemer og databaseinteraktionsmønstre. For organisationer, der vælger teknologistak, påvirker forståelsen af, hvordan sprogdesign og framework-arkitektur influerer sikkerhedsresultater, direkte deres risikoeksponering.

2 Problemformulering

Mens eksisterende litteratur giver indsigt i mekanikken bag SQL injection og generelle defensive principper, er der begrænset komparativ analyse på tværs af forskellige programmeringssprog og frameworks. Denne undersøgelse adresserer dette hul ved at undersøge, hvilken sprog-framework-kombination der giver det mest robuste og samtidig udviklervenlige forsvar mod SQL injection-angreb.

Forskningsspørgsmål: Hvordan adskiller forsvarsmekanismer mod SQL injection sig mellem Python og Node.js økosystemer, og hvilken sprog/framework-kombination giver den mest robuste beskyttelse, samtidig med at udviklerfriktionen minimeres?

2.1 Underspørgsmål

- Hvordan adskiller Python og Node.js implementeringer sig i deres sårbarhed over for SQL injection?
- Hvor let er det for udviklere ved et uheld at introducere sårbarheder i hvert økosystem?
- Hvilke forsvarsmekanismer tilbyder SQLAlchemy og sql.js, og hvor effektive er de?
- Hvilken framework-tilgang er mest modstandsdygtig over for udviklermisbrug?

3 Metode ---

Denne undersøgelse anvender en komparativ implementeringsbaseret metode, hvor ækvivalente webapplikationer bygges i to forskellige sprog for at evaluere, hvordan hver håndterer simulerede SQL injection-angreb.

3.1 Forskningstilgang

Det teoretiske fundament trækker på etableret sikkerhedslitteratur, herunder OWASP's SQL Injection Prevention-retningslinjer [5] og akademisk forskning om angrebsskategorisering [2]. Vi supplerer dette med officiel framework-dokumentation for SQLAlchemy (Python) og sql.js (Node.js) for at forstå idiomatiske sikre mønstre i hvert økosystem. Implementeringen følger industri-standarder for sikker udviklingspraksis, mens der bevidst skabes sårbarheder for at demonstrere almindelige faldgruber, som udviklere støder på.

3.2 Implementeringsstrategi

Vi implementerer to ækvivalente REST API'er med autentificering og dataforespørgselsendepunkter, der interagerer med en SQLite-database. For hvert økosystem udvikler vi to versioner: en sårbar baseline med usikker strengsammenkædning og en sikret implementering med parameteriserede forespørgsler. Python bruger Flask med SQLAlchemy ORM; Node.js bruger Express med sql.js. Denne tilgang muliggør direkte sammenligning af sikkerhedssårbarheder og beskyttelsesmekanismer.

3.3 Evalueringskriterier

3.3.1 Sikkerhedsrobusthed

Vi tester hver implementering mod almindelige SQL injection-vektorer, herunder autentificeringsomgåelse, union-baseret dataekstraktion og boolean-baseret blind injection, og vurderer både baseline-sårbarheder og defensiv effektivitet.

3.3.2 Udvikleroplevelse

Vi undersøger kodeomfang, mønsterklarhed og kognitiv belastning for sikre databaseinteraktioner, analyserer indlæringskurver, dokumentationskvalitet, og hvor naturligt sikre mønstre opstår fra idiomatisk kode.

3.3.3 Ydeevneovervejelser

Responstidstest viser, at defensive foranstaltninger introducerer ubetydelig overhead; begge implementeringer svarer på typiske forespørgsler på under 50ms.

3.4 Testmiljø

Begge implementeringer bruger SQLite-databaser. Python bruger version 3.14 med Flask 3.0.0 og SQLAlchemy 2.0.45; Node.js bruger version 20 LTS med Express og sql.js. Test anvender manuel payload-injektion til autentificeringsomgåelse, union-baseret ekstraktion og boolean-baseret blind injection.

4 Analyse og Resultater

4.1 Teoretisk Grundlag

SQL injection er en klasse af sårbarhed, der opstår, når ikke-betroet input inkorporeres direkte i en databaseforespørgsel, hvilket tillader angribere at ændre den tilsigtede struktur eller semantik af forespørgslen. På et overordnet niveau fungerer injection, fordi SQL-parsing behandler de leverede data og forespørgselssyntaks på samme måde, medmindre dataene eksplicit separeres fra forespørgselslogikken. Almindelige manifestationer omfatter autentificeringsomgåelse (tautologiangreb), union-baseret dataekstraktion og blinde (boolean- eller tidsbaserede) teknikker, der bruges, når direkte output ikke er tilgængeligt [2, 5].

Vi forsvarer mod SQL injection primært gennem parameteriserede forespørgsler, som separerer SQL-kode fra data ved at bruge pladsholdere til brugerinput. Parameteriserede forespørgsler opnår denne separation ved at sende forespørgselsstrukturen og dataværdierne som distinkte beskeder til databasemotoren. SQL-parseren kompilerer forespørgselsstrukturen først og etablerer, hvilke operationer der skal udføres, før brugerinput ankommer. Denne arkitektoniske separation gør det umuligt for ondsindet input at ændre forespørgslens adfærd, da specialtegn automatisk escapes af databasedriveren og behandles som bogstavelige data snarere end SQL-syntaks. Object-Relational Mapping (ORM) frameworks som SQLAlchemy abstraherer yderligere databaseinteraktioner og giver indbyggede mekanismer til sikkert at håndtere brugerinput, mens de håndhæver disse parameteriseringsmønstre.

4.2 Python og SQLAlchemy

4.2.1 Sårbar Implementering

Vores første eksempel bæres af Python-kodestykket fra vulnerable.py:

```
1 def get_user(username):
2     query = "SELECT * FROM users WHERE username = '{username}';"
3     result = db.execute(query)
4     return result.fetchall()
```

Listing 1: Sårbar SQL-forespørgsel med f-streng

Det vi ser her er, at forespørgslen konstrueres ved hjælp af f-streng, som direkte interpolerer `username`-variablen ind i SQL-sætningen. Dette åbner døren for injection, da en angriber kunne indtaste en ondsindet streng, der ændrer forespørgslens logik. For eksempel kunne en angriber indtaste:

```
1 ' OR '1'='1
```

Listing 2: Eksempel på ondsindet input

Dette ville transformere forespørgslen til:

```
1 SELECT * FROM users WHERE username = '' OR '1'='1';
```

Listing 3: Transformeret SQL-forespørgsel

Denne forespørgsel ville returnere alle brugere og dermed effektivt omgå autentificering. Dette er kendt som et tautologi¹-angreb.

I Python er f-strengene en bekvem måde at formatere strenge på, meget lig C#'s strenginterpolation eller JavaScripts template literals. En udvikler kunne let som standard bruge f-strengene til SQL-forespørgsler uden at overveje sikkerhedsimplicationerne, især hvis de er ukendte med SQL injection-risici.

Under test var ovenstående angreb succesfuldt med at omgå autentificering og hente alle brugerregistreringer fra databasen i den sårbare implementering. I koderepository'et, se TEST_RESULTS.md² for detaljerede logs af succesfulde angreb. Det skal bemærkes, at de faktiske payloads varierer, men indeholder "--" i slutningen for kommentarterminering³.

4.2.2 Sikret Implementering

Den sikre implementering bruger SQLAlchemy's parameteriserede forespørgsler til sikkert at håndtere brugerinput, som vist i følgende kdestykke fra secure_app.py:

```
1 from sqlalchemy import text
2 def get_user(username):
3     query = text("SELECT * FROM users WHERE username = :username;")
4     result = db.execute(query, {"username": username})
5     return result.fetchall()
```

Listing 4: Sikret SQL-forespørgsel med SQLAlchemy

Her bruger vi SQLAlchemy's `text()`-funktion til at definere SQL-forespørgslen med en plads-holder `:username`. Den faktiske værdi for `username` leveres separat i `execute()`-metoden. Dette sikrer, at inputtet behandles som data, ikke som en del af SQL-kommandoen, hvilket effektivt neutraliserer ethvert injectionsforsøg.

Under test mislykkedes alle forsøg på SQL injection-angreb mod den sikrede implementering. Den parameteriserede forespørgsel håndterede korrekt ondsindede inputs og behandlede dem som data snarere end eksekverbar SQL-kode.

4.2.3 Evaluerings

Python-implementeringen demonstrerede klar kontrast: sårbare versioner bukkede under for alle testede angreb, mens sikre versioner blokerede alle forsøg. Resultater indikerer stærkt, at

¹Tautologi betyder blot noget, der logisk er sandt af nødvendighed. I dette tilfælde er `'1'='1'` altid sandt, så WHERE-klausulen er altid opfyldt.

²Dette dokument er, på godt og ondt, hovedsageligt skrevet af Copilot. Utæmmet har den sporet min fremgang, eller mangel på samme.

³Jeg forventer, at den der læser dette kender udtrykket, men i dette tilfælde er kommentarterminering simpelthen at terminere resten af den originale forespørgsel efter vores ondsindede input.

parameteriserede forespørgsler med SQLAlchemy giver robust beskyttelse. SQLAlchemy's ORM-abstraktioner muliggør ligetil sikre implementeringer, selvom udviklere skal undgå f-strengene til forespørgsler. Ydeevne-overhead viste sig at være ubetydelig.

4.3 Node.js og Express

4.3.1 Sårbar Implementering

Den sårbare Node.js-implementering bruger template literals til at konstruere SQL-forespørgsler, som vist i følgende kodelykke fra `vulnerable_app.js`:

```
1 function getUser(username) {  
2     const query = `SELECT * FROM users WHERE username = '${username}'`  
3     ;  
4     return db.exec(query);  
}
```

Listing 5: Sårbar SQL-forespørgsel med Template Literals

Som du kan se, er dette endnu et eksempel på direkte strenginterpolation, svarende til Pythons f-strengene. En angriber kunne udnytte dette ved at give et ondsindet input såsom:

```
1 ' OR '1'='1'
```

Listing 6: Eksempel på ondsindet input

Dette ville transformere forespørgslen til:

```
1 SELECT * FROM users WHERE username = '' OR '1'='1';
```

Listing 7: Transformeret SQL-forespørgsel

Som det foregående eksempel er dette et tautologiangreb, der ville returnere alle brugere og omgå autentificering.

Under test omgik dette angreb succesfuldt autentificering og hentede alle brugerregistreringer fra databasen i den sårbare implementering. Se igen `TEST_RESULTS.md` i koderepository'et for detaljerede logs af succesfulde angreb.

4.3.2 Sikret Implementering

Den sikrede Node.js-implementering bruger `sql.js`'s parameteriserede forespørgsler til sikkert at håndtere brugerinput, som vist i følgende kodelykke fra `secure_app.js`:

```
1 function getUser(username) {  
2     const query = "SELECT * FROM users WHERE username = ?";  
3     const stmt = db.prepare(query);  
4     stmt.bind([username]);  
5     const result = stmt.step() ? stmt.getAsObject() : null;  
6     stmt.free();  
7     return result;  
8 }
```

Listing 8: Sikret SQL-forespørgsel med `sql.js` Parameteriserede Forespørgsler

Denne gang bruger vi en `?`-pladsholder i SQL-forespørgslen og binder den faktiske værdi for `username` ved hjælp af `bind()`-metoden. Dette sikrer, at inputtet behandles som data og forhindrer ethvert injectionsforsøg.

Under test mislykkedes alle forsøg på SQL injection-angreb mod den sikrede implementering. Den parameteriserede forespørgsel håndterede korrekt ondsindede inputs og behandlede dem som data snarere end eksekverbar SQL-kode.

4.3.3 Evaluering

Node.js-resultater afspejler Pythons: sårbare versioner mislykkedes mod alle angreb, sikrede versioner blokerede alle forsøg, hvilket bekræfter, at sql.js parameteriserede forespørgsler giver robust beskyttelse. API'en er noget mere omstændelig end SQLAlchemy, men stadig ligetil. Udviklere skal undgå template literals til forespørgsler. Ydeevne-overhead forbliver ubetydelig.

4.4 Tværsproglig Sammenligning

4.4.1 Sikkerhedssammenligning

Begge økosystemer demonstrerede robust beskyttelse ved brug af parameteriserede forespørgsler, mens sårbare implementeringer bukkede lige under for angreb. Sprog- og framework-valg bestemmer ikke i sig selv sikkerhed; sikre kodningspraksisser betyder mest. Sårbarheder opstod fra identiske fejl: strenginterpolation (f-streng i Python, template literals i JavaScript). Begge dynamisk typedede sprog frister udviklere mod bekvemme men usikre mønstre. Sikkerhed afhænger primært af udvikleruddannelse og organisatoriske standarder, ikke sprogfunktioner.

4.4.2 Sammenligning af Udvikleroplevelse

Begge frameworks tilbyder klare, veldokumenterede API'er. SQLAlchemy's ORM-abstraktioner passer til objektorienterede udviklere; sql.js's lavniveau-tilgang kræver mere boilerplate, men passer til SQL-fortrolige udviklere. SQLAlchemy har en stejlere indlæringskurve, men opmuntrer til sikre mønstre. Begge økosystemer har modne fællesskaber og omfattende ressourcer.

5 Konklusion

Både Python/SQLAlchemy og Node.js/sql.js demonstrerer robust beskyttelse ved brug af parameteriserede forespørgsler, mens sårbare implementeringer bukker lige under for angreb. Sikre kodningspraksisser, ikke sprogvæl, bestemmer sikkerhedsresultater. Effektivitet afhænger af udvikleruddannelse og organisatoriske standarder.

5.1 Sammenfatning af Resultater

- Både Python med SQLAlchemy og Node.js med sql.js forhindrer effektivt SQL injection ved brug af parameteriserede forespørgsler.
- Sårbarheder i begge økosystemer opstår primært fra lignende fejl, såsom brug af strenginterpolation til SQL-forespørgsler.
- Udvikleroplevelsen varierer lidt, med SQLAlchemy der tilbyder flere ORM-abstraktioner og sql.js der er mere funktionel, men begge er veldokumenterede.
- Indlæringskurven og lethed ved misbrug er sammenlignelig i begge økosystemer, hvilket understreger vigtigheden af udvikleruddannelse.

5.2 Besvarelse af Forskningsspørgsmålet

Begge sprog/framework-kombinationer giver lignende løsninger på SQL injection uden nogen klar vinder i absolutte termer. Organisationer bør vælge det økosystem, der bedst passer til deres teams ekspertise og projektkrav, da sikkerhed primært afhænger af udvikleruddannelse og organisatoriske kodningsstandarder snarere end iboende sprogfunktioner. Prioritering af sikre kodningspraksisser og løbende sikkerhedstræning forbliver essentiel uanset teknologivalg.

5.3 Bedste Praksisser og anbefalinger

- Brug altid parameteriserede forespørgsler eller ORM-frameworks, der håndhæver sikre databaseinteraktioner.
- Undgå at bruge strenginterpolation (f-streng i Python, template literals i JavaScript) til konstruktion af SQL-forespørgsler.
- Invester i udvikleruddannelse og træning i sikre kodningspraksisser, særligt vedrørende SQL injection-risici.
- Etabler organisatoriske kodningsstandarder, der påbyder brug af sikre databaseinteraktionsmønstre.
- Gennemgå og auditér regelmæssigt kode for potentielle SQL injection-sårbarheder i områder, der håndterer brugerinput.

5.4 Begrænsninger og Fremtidigt Arbejde

Denne undersøgelses omfang begrænsede analysen til Python og Node.js med SQLite. Fremtidigt arbejde kunne udvide til yderligere sprog og databasesystemer for at vurdere, om resultaterne generaliserer på tværs af økosystemer.

5.5 Refleksion

Denne undersøgelse forstærkede den kritiske vigtighed af sikre kodningspraksisser. Trods modenheden af webudviklingsframeworks forbliver SQL injection en udbredt trussel på grund af almindelige udviklerfejl. Oplevelsen fremhævede behovet for løbende uddannelse og årvågenhed i softwareudvikling, uanset den valgte teknologistak.

A Oversigt over Angrebstestresultater

Følgende tabel opsummerer de SQL injection-angreb, der blev testet mod både sårbare og sikre implementeringer på tværs af Python og Node.js økosystemer.

| Angrebsvektor | Python/Flask | | Node.js/Express | |
|--------------------------|--------------|--------|-----------------|--------|
| | Sårbar | Sikret | Sårbar | Sikret |
| Autentificeringsomgåelse | FEJL | OK | FEJL | OK |

Tabel 1: Angrebstestresultater. FEJL indikerer, at applikationen var sårbar over for angrebet. OK indikerer, at angrebet blev succesfuldt blokeret.

Anvendt Test-payload:

- **Autentificeringsomgåelse:** `admin' OR '1'='1'--`

Autentificeringsomgåelsesangrebet udnytter en tautologibetingelse, der får WHERE-klausulen til altid at evaluere som sand. Alle sårbare implementeringer udviste 100% modtagelighed for dette angreb. Alle sikre implementeringer blokerede angrebet ved at behandle det ondsindede input som bogstavelige data.

B Repository-struktur

Den komplette implementering er tilgængelig i det medfølgende koderepository med følgende struktur:

```
exam-dfd-synopsis/
+-- main.tex                # Dette synopsisdokument
+-- database/
|   +-- schema.sql          # Databaseskemadefinition
|   +-- seed.sql            # Testdatapopulering
+-- implementations/
|   +-- python-flask/
|   |   +-- vulnerable_app.py # Sårbar Python-implementering
|   |   +-- secure_app.py    # Sikret Python-implementering
|   |   +-- test_attacks.py  # Angrebstestautomatisering
|   |   +-- TEST_RESULTS.md  # Detaljerede testlogs
|   |   +-- requirements.txt # Python-afhængigheder
|   +-- nodejs-express/
|   |   +-- vulnerable_app.js # Sårbar Node.js-implementering
|   |   +-- secure_app.js    # Sikret Node.js-implementering
|   |   +-- test_attacks.js  # Angrebstestautomatisering
|   |   +-- TEST_RESULTS.md  # Detaljerede testlogs
|   |   +-- package.json     # Node.js-afhængigheder
+-- out/
    +-- references.bib        # Bibliografireferencer
```

Hver implementeringsmappe indeholder identisk funktionalitet:

- REST API med `/api/login` og `/api/posts/search` endpoints

- SQLite-database med Users og Posts tabeller
- Automatiserede angrebstestscripts
- Dokumenterede testresultater med angrebspayloads og svar

Litteratur

- [1] European Union Agency for Cybersecurity. Enisa threat landscape 2023. Technical report, ENISA, 2023. Accessed: 2024-12-17.
- [2] William G. J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, pages 13–15. IEEE, 2006.
- [3] IBM Security. Cost of a data breach report 2023. Technical report, IBM Corporation, 2023. Accessed: 2024-12-17.
- [4] OWASP Foundation. Owasp top 10 2021, 2021. Accessed: 2024-12-17.
- [5] OWASP Foundation. Sql injection prevention cheat sheet, 2024. Accessed: 2024-12-17.
- [6] Verizon Business. 2023 data breach investigations report. Technical report, Verizon Communications, 2023. Accessed: 2024-12-17.