Kauno technologijos universitetas

Informatikos fakultetas

PHP kodo statinės analizės tobulinimas

Baigiamasis magistro projektas

Mindaugas Vedegys

Projekto autorius

doc. Jonas Čeponis

Vadovas

Kaunas, 2019

Kauno technologijos universitetas

Informatikos fakultetas

PHP kodo statinės analizės tobulinimas

Baigiamasis magistro projektas

Informacijos ir informacinių technologijų sauga (6211BX008)

Mindaugas Vedegys

Projekto autorius

doc. Jonas Čeponis

Vadovas

dr. Ignas Martišius

Recenzentas

Kaunas, 2019

Kauno technologijos universitetas

Informatikos fakultetas

Mindaugas Vedegys

PHP kodo statinės analizės tobulinimas

Akademinio sąžiningumo deklaracija

Patvirtinu, kad mano, Mindaugo Vedegio, baigiamasis projektas tema „PHP kodo statinės analizės tobulinimas“ yra parašytas visiškai savarankiškai ir visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

(vardą ir pavardę įrašyti ranka)	(parašas)

Vedegys, Mindaugas. PHP kodo statinės analizės tobulinimas. Magistro baigiamasis projektas / vadovas doc. Jonas Čeponis; Kauno technologijos universitetas, Informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Informatikos inžinerija.

Reikšminiai žodžiai: sauga, programinės įrangos sauga, statinė analizė, kodo analizė.

Kaunas, 2019. 62 p.

Santrauka

Programinė įranga, informacinės sistemos bei mobiliosios programėlės yra vis plačiau naudojamos įvairiose srityse, nuo laisvalaikio iki organizacijų veiklos. Priklausomai nuo panaudojimo atvejų, programinės įrangos kokybės, patikimumo bei saugumo užtikrinimo svarba gali būti itin reikšminga. Šios savybės yra visuomet aktualios ir nuolat tobulinamos sritys. Siekiant efektyviau įvertinti šias savybes bei rizikas, naudojami automatizuoti įrankiai, pagal įvairias metodikas analizuojantys programinį kodą ir jo veikimą.

Vienas tokių įrankių tipų yra statinės kodo analizės įrankiai, analizuojantys programinės įrangos kodą jo nevykdant. Tai sąlyginai greitas analizės būdas, turintis privalumų ir trūkumų. Statinės kodo analizės įrankiai yra gana plačiai naudojami įvairių programinės įrangos kodo savybių įvertinimui bei saugumo spragų analizei, tačiau aptinkama ir šiems įrankiams būdingų trūkumų, todėl atsiranda poreikis įvertinti jų kokybę ir tikslumą, išspręsti problemas bei patobulinti įrankius.

Darbe analizuojami programinio kodo analizės metodai ir įrankiai, apžvelgiami jų privalumai ir trūkumai. Analizės dalyje daroma išvada, kad pagrindiniai trūkumai šiuo metu yra „false positive“ problema ir įrankių netikslumas – problemų neaptikimas. Pasirinkta spręsti įrankių netikslumo problemą kuriant statinės kodo analizės taisykles.

Apžvelgus statinės kodo analizės įrankius, įvertinus juos pagal pasirinktus kriterijus, nuspręsta analizės taisykles realizuoti kaip taisyklių rinkinius, integruojamus į jau egzistuojantį „PHPStan“ įrankį. Sukurti du analizės taisyklių rinkiniai, atlikti eksperimentai, skirti realizuoto sprendimo efektyvumui pademonstruoti.

Padaryta išvada, kad atlikti eksperimentai pagrindžia sukurtų statinės kodo analizės taisyklių veikimą, leidžia teigti, kad jos papildo ir išplečia esamų statinės analizės įrankių funkcionalumą ir yra pritaikomos praktinėje aplinkoje.

Vedegys, Mindaugas. Improvement of Static Code Analysis in PHP. Master's Final Degree / supervisor assoc. prof. Jonas Čeponis; The Faculty of Informatics, Kaunas University of Technology.

Study field and area (study field group): Informatics Engineering.

Keywords: security, software security, static analysis, code analysis.

Kaunas, 2019. 62p.

Summary

Software, information systems and mobile applications are widely used in various spheres from leisure activities to organisation processes. Depending on use cases, ensuring software quality, reliability and security can be very important. These characteristics are always relevant and constantly improved. In order to measure these characteristics more effectively, automated tools are used which use various methods to analyse the code and how it behaves.

One type of such tools is static code analysis tools which analyse program code without running it. Static code analysis is a fast method, but it has advantages and disadvantages. These tools are widely used to measure various code characteristics and detect security vulnerabilities, but they also have some common problems and issues so there is a need to evaluate their quality and precision, to solve these problems and improve the tools.

In this work code analysis methods and tools are analysed, their advantages and disadvantages are reviewed. In analysis chapter a conclusion is made that currently most important problems of code analysis are “false positive” situations and inaccuracy - failure to detect code problems. It is decided to solve the problem of inaccuracy by creating static analysis rules.

After reviewing existing static code analysis tools and evaluating them by chosen criteria it was decided to implement analysis rules as rule sets for existing “PHPStan” tool. Two rule sets were created and experiments were conducted to demonstrate the efficiency of the solution.

A conclusion is made that conducted experiments validate effectiveness of created solution and allow stating that created analysis rules complement and expand functionality of existing tools and are applicable in practical environment.

Turinys

Lentelių sąrašas

Paveikslų sąrašas

Terminų ir santrumpų žodynas

1. „False positive“ – situacija, kuomet automatizuoti statinės analizės įrankiai nurodo aptiktą saugumo spragą ar klaidą vietoje, kurioje iš tiesų problemos nėra.

2. Derinimas (angl. „*debugging*“) – procesas, kurio metu randamos tikslios programinės įrangos klaidų, pažeidžiamumų vietos, dažniausiai apima ir jų ištaisymą.

3. Profiliavimas (angl. „*profiling*“) – procesas, kuomet programinės įrangos vykdymo metu yra matuojamas naudojamas atminties kiekis, vykdymo laikas, siekiant įvertinti ar pagerinti programos efektyvumą.

4. Kompiuterinis mokymasis (angl. „*machine learning*“) – Metodai, mokantys kompiuterius „mąstyti“, kai sukurta sistema prisitaiko prie duomenų („apsimoko“) naudodamasi statistika.

5. Abstraktus sintaksės medis (angl. „*abstract syntax tree*“) – medžio struktūros forma perteikiama programinio kodo struktūra, vykdymo keliai. Grafinio atvaizdavimo pavyzdys pateikiamas 7.1 priede.

6. „AST“ – termino „abstraktus sintaksės medis“ (angl. „*abstract syntax tree*“) trumpinys.

Įvadas

Darbas priklauso „Informacijos ir informacinių technologijų sauga“ studijų programai.

Programinė įranga, internetinės informacinės sistemos bei mobiliosios programėlės yra vis plačiau naudojamos įvairiose srityse, nuo laisvalaikio iki organizacijų veiklos. Priklausomai nuo panaudojimo atvejų, programinės įrangos kokybės, patikimumo bei saugumo užtikrinimo svarba gali būti itin reikšminga. Šios savybės yra visuomet aktualios ir nuolat tobulinamos sritys. Siekiant efektyviau įvertinti šias savybes bei rizikas, naudojami automatizuoti įrankiai, pagal įvairias metodikas analizuojantys programinį kodą ir jo veikimą.

Vienas tokių įrankių tipų yra statinės kodo analizės įrankiai, analizuojantys programinės įrangos kodą jo nevykdant. Tai sąlyginai greitas analizės būdas, turintis privalumų ir trūkumų.

Darbo problematika ir aktualumas

Statinės kodo analizės įrankiai yra gana plačiai naudojami įvairių programinės įrangos kodo savybių įvertinimui bei saugumo spragų analizei. Tačiau pačių statinės analizės įrankių veikimo principai ir tikslumas nėra plačiai išnagrinėti, aptinkama šiems įrankiams būdingų trūkumų, todėl kyla poreikis įvertinti jų kokybę ir tikslumą, išspręsti problemas bei patobulinti įrankius.

Darbo tikslas ir uždaviniai

Darbo tikslas – patobulinti „PHP“ programavimo kalbos statinės kodo analizės įrankius ir su saugumu susijusių problemų aptikimą.

Kad tikslas būtų pasiektas, reikia įgyvendinti šiuos uždavinius:

- išanalizuoti statinės kodo analizės metodus ir principus;
- išskirti bei išanalizuoti esminius statinės kodo analizės metodų ir įrankių trūkumus;
- išanalizuoti metodus ir įrankius atsižvelgiant į išskirtus esminius trūkumus;
- pateikti sprendimus, padedančius pašalinti trūkumus ar sumažinti jų neigiamą poveikį;
- atlikti bandymus ir eksperimentus, rodančius sprendimų efektyvumą.

Darbo rezultatai ir jų svarba

Darbo rezultatai yra aktualūs programinės įrangos saugumo, pažeidžiamumų ir klaidų aptikimo sričiai. Darbo metu sukurti sprendimai padidina statinės kodo analizės įrankių efektyvumą, suteikia galimybę aptikti daugiau su programinės įrangos saugumu susijusių kodo problemų.

Darbo struktūra

Darbą sudaro penki pagrindiniai skyriai:

- programinio kodo analizės metodų ir įrankių tyrimas - atliekama probleminės srities analizė, apžvelgiami statinės kodo analizės principai ir metodai, įvardijamos pagrindinės statinės analizės problemos. Taip pat įvardijami pagrindiniai statinės kodo analizės įrankių naudotojai, apžvelgiami esami problemos sprendimai;
- statinės analizės įrankio taisyklių rinkinių koncepcija - remiantis analizės informacija, įvardijamos ir aprašomos sprendžiamos problemos, pateikiami pavyzdžiai. Taip pat glaustai aprašomas siekiamas sprendimas ir jam keliami reikalavimai;
- statinės analizės įrankio taisyklių rinkinių aprašymas - kuriamiems taisyklių rinkiniams aprašomi konkretūs reikalavimai, priskiriami pavadinimai. Aprašomas taisyklių rinkinių veikimas, pateikiamos veikimą iliustruojančios diagramos, taip pat taisyklių kūrimui ir testavimui naudoti duomenys;
- sukurtų statinės analizės taisyklių bandymai, eksperimentai - atliekami bandymai ir eksperimentai, skirti sprendimo efektyvumui pademonstruoti. Kuriamo sprendimo rezultatai lyginami su kitų įrankių ir taisyklių analizės rezultatais. Taip pat pateikiamas pavyzdys, demonstruojantis kaip sukurto sprendimo rezultatai leidžia išnaudoti pažeidžiamumą realiame programavimo karkase, taip pagrindžiant sprendimo efektyvumą;
- išvados - apibendrinami darbo rezultatai ir pateikiamos išvados.

1. PROGRAMINIO KODO ANALIZĖS METODŲ IR ĮRANKIŲ TYRIMAS

Šiame skyriuje nusakoma tyrimo sritis, tyrimo objektas, atliekama tyrimo objekto naudotojų analizė, taip pat įvardijami darbo tikslai ir uždaviniai.

1.1. Analizės tikslas

Analizės tikslas yra išanalizuoti statinės kodo analizės metodų ir įrankių veikimo principus, sudaryti statinės kodo analizės metodų ir įrankių vertinimo kriterijų sąrašą, taip pat išskirti bei išanalizuoti esminius statinės kodo analizės įrankių privalumus ir trūkumus.

1.2. Tyrimo objektas, sritis ir problema

Statinė kodo analizė, jos metodai bei įrankiai.

1.3. Tyrimo objekto analizė

Šiame skyriuje aprašoma kodo analizė, jos tipai, paskirtis bei privalumai ir trūkumai.

1.3.1. Kodo analizė

Programinės įrangos produktų kiekiui, jų apimtims bei sudėtingumui augant, tampa vis sudėtingiau išvengti klaidų ir užtikrinti programinio kodo kokybę. Programinės įrangos kokybė, remiantis straipsniu [1], susideda iš išorinės ir vidinės kokybės – išorinė kokybė nusako ar programinė įranga atitinka reikalavimus ir suteikia reikiamą funkcionalumą, o vidinė kokybė nusako nefunkcines savybes. Vidinės kokybės užtikrinimas svarbus siekiant išvengti klaidų ir padidinti programinės įrangos vystymo ir palaikymo efektyvumą.

Vidinė kokybė matuojama skirtingų tipų - dinaminės ir statinės kodo analizės įrankiais. Šių tipų įrankiai detaliau analizuojami tolimesniuose skyriuose, o remiantis straipsnyje [1] pateikiamu palyginimu pagal teigiamas ir neigiamas naudotojų patirtis, dinaminės analizės įrankiai turi daugiau problemų identifikuojant klaidas, yra sunkiau konfigūruojami ir brangesni, tačiau yra atsparūs „false positive“ situacijoms. Tuo tarpu statinės kodo analizės įrankiai yra lengvai konfigūruojami, sėkmingai identifikuoja problemas, yra pigesni, bet mažiau tikslūs – „false positive“ situacija yra pakankamai dažna.

Ričardas E. Fairley‘us straipsnyje [2] pateikia statinės ir dinaminės kodo analizės apibūdinimą ir palyginimą: statinė analizė aptinka struktūrines ir semantines klaidas, o dinaminė analizė užtikrina tinkamą programinės įrangos veikimą.

1.3.2. Dinaminė kodo analizė

Dinaminė analizė vykdoma programinės įrangos vykdymo metu, apima derinimą (angl. „debug“) ir profiliavimą (angl. „profiling“), taip pat testavimą. Remiantis šaltiniu [3], derinimu vadinamas procesas, kurio metu randami ir ištaisomi programinės įrangos defektai ir klaidos, o tam pasiekti naudojami derinimo įrankiai (angl. „debugger“), leidžiantys veiksmus atlikti iš eilės vieną po kito ir nuolat stebėti kintamųjų reikšmes. Tuo tarpu profiliavimas, kaip teigiama straipsnyje [4], yra laiko ir atminties sunaudojimo matavimas kiekvienoje kodo eilutėje ir funkcijoje programos vykdymo metu. Profiliavimas vykdomas siekiant įvertinti ar pagerinti programos efektyvumą.

Remiantis straipsnyje [1] pateikiamu palyginimu pagal teigiamas ir neigiamas naudotojų patirtis, dinaminės kodo analizės įrankių privalumai ir trūkumai yra:

Privalumai:

- atsparumas „false positive“ situacijoms;
- rezultatų pateikimo aiškumas.

Trūkumai:

- kaina;
- konfigūravimo sudėtingumas;
- probleminės kodo vietos identifikavimo sudėtingumas.

Tuo tarpu šaltinyje [5] pateikiami dinaminės kodo analizės privalumai ir trūkumai yra šie:

Privalumai:

- pažeidžiamumų ieškoma vykdymo aplinkoje;
- galima analizuoti programą prie kurios tikrojo programinio kodo prieiga nėra galima;
- suteikia galimybę patikrinti statinės kodo analizės rezultatus, įsitikinti jų tikslumu;
- gali būti naudojama daugelyje programų.

Trūkumai:

- suteikia netikrą saugumo jausmą, tarsi viskas būtų pilnai ištestuota;
- negali užtikrinti visiško kodo apėmimo (angl. „coverage”);
- gali pateikti klaidinančius rezultatus, nėra 100 proc. tikslūs;
- šių įrankių tikslumas priklauso nuo analizės taisyklių teisingumo;
- sudėtinga nustatyti tikslą kodo vietą, kuri sąlygoja pažeidžiamumą ar kelia kitas problemas.

Taigi, dinaminė kodo analizė turi įvairių privalumų ir trūkumų, kurie skirtingų naudotojų gali būti vertinami skirtingai. Vienas pagrindinių privalumų yra tai, jog programinis kodas yra analizuojamas vykdymo metu, taip siekiant atkartoti būsimą tikrąją kodo veikimo aplinką. Tačiau tokie įrankiai nėra iki galo tikslūs ir kartais gali pateikti klaidingus rezultatus.

1.3.3. Statinė kodo analizė

Statinė kodo analizė, kaip apibrėžia Melina K. ir Dzenana D. [6], yra programinės įrangos kodo, galimų vykdymo kelių bei kintamųjų reikšmių analizė, nevykdant pačios programos.

Statinės kodo analizės procesas yra paremtas automatizuota programinio kodo analize, kurios rezultatai, priklausomai nuo situacijos, atskleidžia konkrečią arba galimą probleminę vietą ir nurodo programinės įrangos inžinieriams programinio kodo vietas, kurios turėtų būti peržiūrėtos ir sutvarkytos. Kaip teigiama straipsnyje [7], šios problemos gali būti programinio kodo standartų (stiliaus, konstrukcijų) nesilaikymas, įvairios vykdymo metu pasireiškiančios kritinės klaidos, taip pat klaidos, susijusios su duomenų apdorojimu, o pastaruoju metu statinė kodo analizė vis dažniau naudojama siekiant aptikti saugumo pažeidžiamumus.

Informacinėmis technologijomis, programine įranga realizuojant vis daugiau funkcionalumo, tampa vis svarbiau užtikrinti tinkamą programinės įrangos veikimą ir sumažinti pažeidžiamumų kiekį, tai ypač aktualu sistemose, kurių neteisingas veikimas gali sukelti tiesioginį pavojų žmonėms. Pavyzdžiui, vokiečių publikacijoje [8] aprašomas statinės kodo analizės taikymas analizuoti programinę įrangą, skirtą naudojimui kosmoso programoje. Tokios programinės įrangos efektyvus tinkamo veikimo užtikrinimas yra itin svarbus, pateikiamos kelios priežastys:

- milijonus ar milijardus kainuojančios kosminės programos dėl klaidų gali nepavykti ir lemti didelius nuostolius;
- klaidos rizika – netinkamai veikianti programinė įranga gali sukelti pavojų personalui ir aplinkiniams žmonėms;
- kiekvienas programinės įrangos atnaujinimas kosmose esančiam palydovui dėl ribotų ryšio sudarymo galimybių gali trukti net kelias dienas.

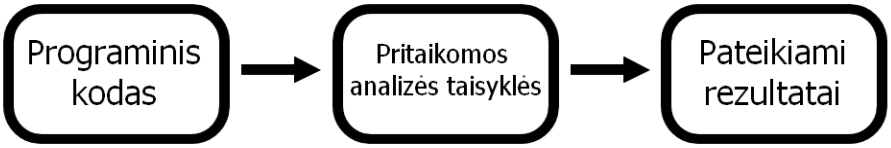
Taigi, kad būtų išvengta išvardintų problemų, šioje programoje naudojamoje programinėje įrangoje buvo pradėta naudoti statinės kodo analizės metodika, padedanti aptikti klaidas programinės įrangos kūrimo metu.

Statinė kodo analizė yra nuo seno aktuali metodika - dar 1978 metais publikuotame straipsnyje [9] statinė analizė apibūdinama kaip efektyvus metodas, papildantis dinaminę analizę, taip pat rekomenduojama naudoti statinę analizę daugumoje kritinės programinės įrangos. Tuo tarpu 1997 metais publikuotame straipsnyje [10] aprašomi pagrindiniai aspektai, kuriais tuo metu programinis kodas buvo vertinamas naudojant statinę kodo analizę:

- programos struktūros analizė, vykdymo kelių atvaizdavimas;
- kodo stiliaus gairių ir standartų laikymosi užtikrinimas;
- metrikų matavimai. pavyzdžiui ciklinio sudėtingumo (angl. „cyclomatic complexity“) matavimas.

Remiantis šia informacija galima teigti, jog statinė analizė nuo seno naudojama analizuoti programinio kodo struktūrą, užtikrinti stiliaus gairių, standartų laikymąsi bei skaičiuoti metrikas, nusakančias programinio kodo kokybę.

Remiantis Gabriel Diaz‘o ir Juan Ramon Bermej‘aus straipsniu [11], abstraktus statinės kodo analizės principas yra:



1.1 pav. Statinės kodo analizės principas

Tačiau visos trys analizės dalys yra kintančios - skiriasi programinės įrangos programavimo kalbos, realizacija, taip pat analizei gali būti naudojamos įvairios analizės taisyklės, skirtos skirtingoms savybėms analizuoti. Rezultatų pateikimas taip pat nėra standartizuotas ir gali skirtis, tačiau tai nėra esminis statinės analizės aspektas.

Remiantis straipsnyje [1] pateikiamu palyginimu pagal teigiamas ir neigiamas naudotojų patirtis, statinės kodo analizės įrankių privalumai ir trūkumai yra:

Privalumai:

- rezultatų pateikimo aiškumas;
- kaina;
- nesudėtingas probleminės kodo vietos identifikavimo procesas.

Trūkumai:

- dažnos „false positive“ situacijos;
- pateikiami nepakankamai išsamūs rezultatai.

Taip pat, straipsnyje [5] pateikiami kitokie statinės kodo analizės privalumai ir trūkumai:

Privalumai:

- gali aptikti konkrečią kodo eilutę, kurioje yra problema;
- galimybę aptikti konkrečią problemos vietą leidžia užtikrinti trumpesnį problemos šalinimo laiką;
- leidžia aptikti problemas ankstyvame programos kūrimo etape;
- gali aptikti problemas, kurių dinaminė kodo analizė aptikti negali arba tai padaryti labai sudėtinga:
    - Nepasiekiamas kodas;
    - Nedeklaruoti, nenaudojami kintamieji;
    - Nenaudojamos funkcijos;
    - Ribinių reikšmių pažeidimas (angl. „boundary values“).

Trūkumai:

- vykdant rankiniu būdu, tai yra ilgas procesas;
- automatiniai įrankiai nėra visiškai tikslūs, gali pateikti klaidingus rezultatus;
- suteikia netikrą saugumo jausmą, tarsi viskas būtų pilnai ištestuota;
- šių įrankių tikslumas priklauso nuo analizės taisyklių teisingumo;
- negali aptikti problemų, kurios pasireiškia tik vykdymo metu.

Pagrindiniai statinės kodo analizės įrankių privalumai yra galimybė nustatyti konkrečią kodo eilutę, kurioje egzistuoja tam tikra problema. Tai leidžia šiems įrankiais pateikti aiškius problemų paaiškinimus, aprašymus. Tačiau statinės, kaip ir dinaminės, analizės įrankiai nėra visiškai tikslūs ir gali pateikti klaidingus rezultatus.

Statinės kodo analizės taisyklių rinkinys priklauso nuo to, kokias programinės įrangos savybes siekiama išanalizuoti. Zeineb Zhioua‘as, Stuart Short‘as ir Yves Roudier‘as [12] apibrėžia pagrindinius analizės metodus skirtingoms programinės įrangos savybėms:

- modelių tikrinimas – tikrinama, ar modelių būsenos, priklausančios nuo tokių sąlygų kaip laikas, atitinka logikos reikalavimus;
- vykdymo sekos (angl. *„control flow“*) analizė – analizuojama, kaip procedūros kviečia viena kitą, kurios funkcijos kviečiamos efektyviai;
- duomenų tėkmės (angl. *„data flow“*) analizė – programinės įrangos priklausomybė nuo įvedamų duomenų, duomenų įtaka jos veikimui, visos įmanomos duomenų aibės apibrėžimas;
- simbolių (angl. *„symbolic“*) analizė – kintamųjų ir programos vykdymo kelių analizė, siekiant aptikti neefektyvius ciklus ir panašias efektyvumo problemas;
- informacijos perdavimo (angl. „*information flow*“) analizė – dažniausiai vykdoma naudojant dinaminę analizę, tačiau gali būti atliekama ir statinės analizės įrankiais, siekiant įvertinti kaip duomenys keliauja tarp kintamųjų, funkcijų bei išanalizuoti, ar šiame procese jie išlieka saugūs.

Statinės kodo analizės įrankiuose šie metodai gali būti naudojami įvairioms taisyklėms kurti, tam tikrais atvejais metodai gali būti ir sujungiami tose pačiose taisyklėse. Įvairių metodų naudojimas ir skirtingų kodo savybių analizė leidžia aptikti įvairias problemas.

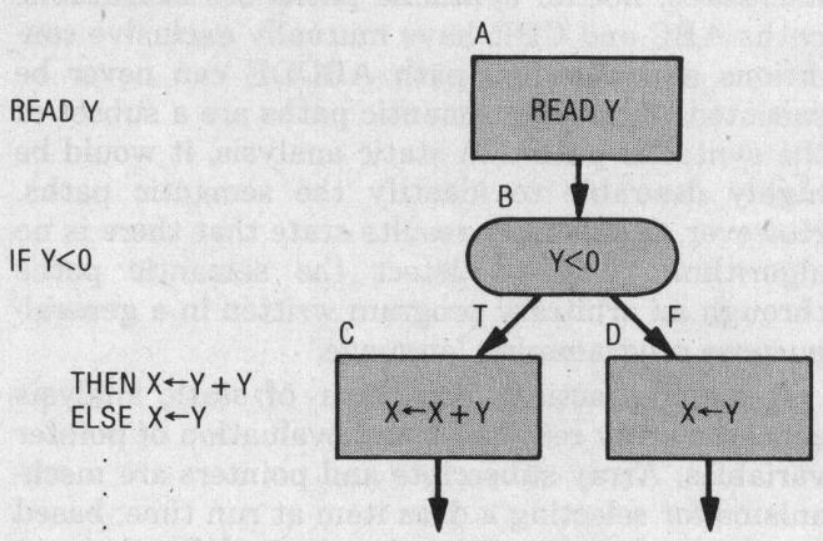
1.3.4. Statinės kodo analizės aktualumas praeityje

Statinė kodo analizė yra nuo seno naudojama, tiriama ir analizuojama metodika, todėl šiame skyriuje apžvelgiami įvairių laikotarpių darbai ir straipsniai šia tema, siekiant atskleisti skirtingais laikotarpiais naudotą informaciją apie statinę kodo analizę ir palyginti temos aktualumą su dabartimi.

1.3.4.1. Pirmasis laikotarpis

Pirmasis straipsnis – 1978 metais Richard E. Fairley'aus parašytas darbas [2] pavadinimu „Static Analysis and Dynamic Testing of Computer Software". Iliustracinis šio darbo teiginys yra „Programose klaidų ieškoti galima statiškai ir dinamiškai. Statinė analizė ieško struktūrinių ir semantinių klaidų. Dinaminė analizė užtikrina tinkamą veikimą ir padeda aptikti klaidas.". Taip pat pabrėžiama, jog statinė kodo analizė atliekama nevykdant analizuojamo programinio kodo. Pateikiamas statinės ir dinaminės analizės rezultatų palyginimas: statinė analizė gali aptikti jog kintamasis yra neinicijuotas visuose kodo vykdymo keliuose, tuo tarpu dinaminė analizė aptinka visas kintamajam priskirtas reikšmes viso vykdymo metu, tačiau tik konkrečiame vykdymo kelyje, kuris yra analizuojamas.

Straipsnyje pateikiamas ir programinio kodo vykdymo kelių diagramos pavyzdys:



1.2 pav. Programinio kodo vykdymo kelių diagramos pavyzdys

Taigi, nors nuo 1978 metų praėjo ne vienas dešimtmetis, technologijų ir programinės įrangos sudėtingumas smarkiai išaugo, tačiau bendri apibrėžimai, statinės ir dinaminės analizės skirtumų suvokimas bei programinio kodo vykdymo kelio apibrėžimas ir atvaizdavimas išliko labai panašūs.

1.3.4.2. Antrasis laikotarpis

Autorių grupės – B.A. Wichmann'o, A.A. Canning'o, D.L. Clutterbuck'o, L.A. Winsborrow'o, N.J. Ward'o ir D.W.R. Marsh'o 1995 metais parašytame darbe [9] analizuojamos statinės kodo analizės pritaikymo galimybės, lyginamos techninės galimybės analizuoti sistemas paprastais aspektais bei nuodugnios analizės (angl. „deep") efektyvumas. Galima daryti išvadą, jog statinė kodo analizė jau buvo paplitusi metodika, tačiau visiškai efektyviam jos išnaudojimui dar trūko techninės įrangos resursų. Pateikiama konkrečių statinės kodo analizės pritaikymo įmonėse pavyzdžių, tai rodo jog tuo metu statinė kodo analizė jau buvo paplitusi įmonėse, organizacijose.

Darbo išvadose pateikiami teiginiai:

- statinė analizė yra efektyvus metodas ir papildo dinaminę analizę;
- rekomenduojama naudoti statinę analizę daugumoje kritinės programinės įrangos;
- nėra statinę analizę apibrėžiančių standartų, todėl sudėtinga įvertinti analizės efektyvumą;
- tam, kad įvertinti analizės rezultatų validumą, reikia apibrėžti analizuojamos programinės įrangos sudėtingumo lygį.

Šiame straipsnyje analizuojami klausimai atskleidžia, jog 1995 metais statinė kodo analizė buvo jau pakankamai paplitusi įvairiose organizacijose, buvo keliami klausimai dėl jos pritaikymo ir efektyvumo, ieškoma būdų įvertinti analizės rezultatų patikimumą ir tikslumą.

1.3.4.3. Trečiasis laikotarpis

2004 metais išleistame straipsnyje [13] autoriai B. Chess‘as ir G. McGrou analizuoja statinės kodo analizės pritaikymą aptinkant saugumo spragas. Statinės analizės apibrėžimai ir sąvokos išlieka tie patys, akcentuojama kad rankiniu būdu tikrinti kodą reikia daug laiko resursų ir žinių apie galimus pažeidžiamumus, jų pasireiškimo priežastis. Taip pat pažymima, kad nei vienas analizės įrankis negali užtikrinti visiško saugumo, aptikti visų saugumo spragų, o analizės rezultatai ir jų aktualumo patikrinimas reikalauja žmogaus įsikišimo. Įvardijama, jog to priežastis yra kontekstas – analizės įrankis negali numatyti kokio tipo programa yra kuriama, kokie aspektai yra svarbiausi ir kurios kodo vietos gali būti labiausiai pažeidžiamos.

Norint pabrėžti statinės kodo analizės svarbą, tekste teigiama: statinė analizė vardan saugumo turėtų būti reguliariai naudojama kiekviename modernių programų kūrimo procese.

Taigi, statinės analizės naudojimas tampa vis įprastesne metodika, naudojama ir saugumo spragoms aptikti, taip pat svarstomas įrankių tikslumas ir galimybės visiškai apsaugoti nuo pažeidžiamumų. Teigiama, jog statinės analizės naudojimas turi būti standartinio programinės įrangos proceso dalis.

Apžvelgus praeityje parašytus darbus, matoma, jog lyginant dabartinę situaciją su praeitimi, statinės kodo analizės idėja ir esminiai principai pasikeitė nedaug, tačiau padidėjo šios metodikos panaudojamumas – statinė analizė naudojama įvairiose srityse, ne tik klaidoms, bet ir saugumo spragoms aptikti, o pasak kai kurių autorių, statinė analizė turėtų būti standartinė programinės įrangos kūrimo proceso dalis. Taip pat pastebima, kad techninė įrangos spartos apribojimai praeityje ribojo statinės analizės galimybes, tačiau techninei įrangai tobulėjant šis klausimas išnyko – sprendimui ar naudoti statinę analizę, techninės įrangos sparta įtakos dažniausiai nebeturi.

Tiesa, nors statinės analizės metodika ir taisyklės yra nuolat tobulinamos, tačiau vis dar neįmanoma garantuoti rezultatų tikslumo ir visiškos apsaugos nuo saugumo pažeidžiamumų.

1.3.5. Statinės kodo analizės problemos

Daromos prielaidos ir faktinės tiesos santykiai gali būti įvardijami žodžių junginiais, nusakančiais kokia prielaida buvo daroma ir kokia iš tiesų yra faktinė tiesa - išvada. Remiantis straipsniu [14], šios reikšmės yra:

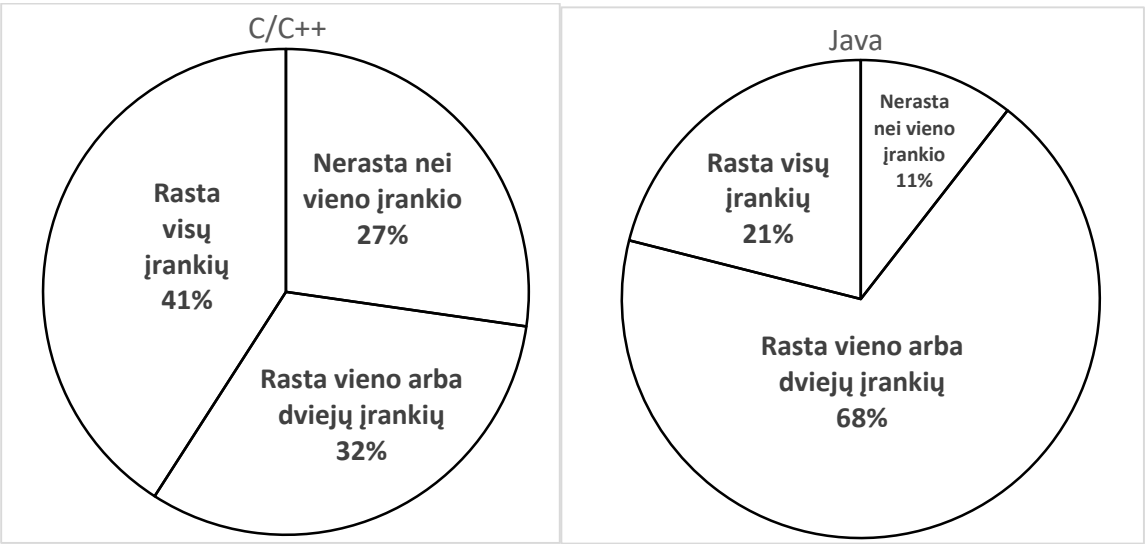
1.1 lentelė Prielaidos ir išvados santykiai

		Prielaida	
		Tiesa	**Netiesa**
Išvada	**Tiesa**	"True positive"	"False negative"
	Netiesa	"False positive"	"True negative"

Viena didžiausių statinės kodo analizės įrankių problemų yra „false positive“ situacija. Pagal Indianos ir Merilendo universitetų darbuotojų [15] apibrėžimą, tai situacija kai statinės kodo analizės įrankiai generuoja perspėjimo pranešimus kodo vietoms, kuriose problemos iš tiesų neegzistuoja.

„False positive“ problema gali įvykti dėl įvairių priežasčių, kaip analizės įrankio klaidos ar naudojamos programavimo kalbos versijos. Tai ženkliai sumažina pasitikėjimą analizės įrankiais ir dažnai neleidžia jų rezultatų naudoti automatizuotuose procesuose – reikalingas žmogaus įsiterpimas ir įvertinimas, ar problema iš tiesų egzistuoja, ar tai „false positive“ situacija. Taip pat, nors analizė yra automatizuota, tokiu atveju žmogus turi papildomai išanalizuoti vietą, kurioje nurodoma problema, todėl bereikalingai sunaudojami laiko resursai.

Klaidingi perspėjimai reikalauja papildomo laiko analizei, bet tai nesumažina analizės įrankio užtikrinamo saugumo. Atlikus trijų statinės kodo analizės įrankių galimybių aptikti saugumo pažeidžiamumus tyrimą, straipsnyje [16] pateikiamos išvados, jog programiniame kode aptiktų pažeidžiamumų statistika yra:



1.3 pav. Statinės analizės įrankių pažeidžiamumų aptikimo eksperimento rezultatai

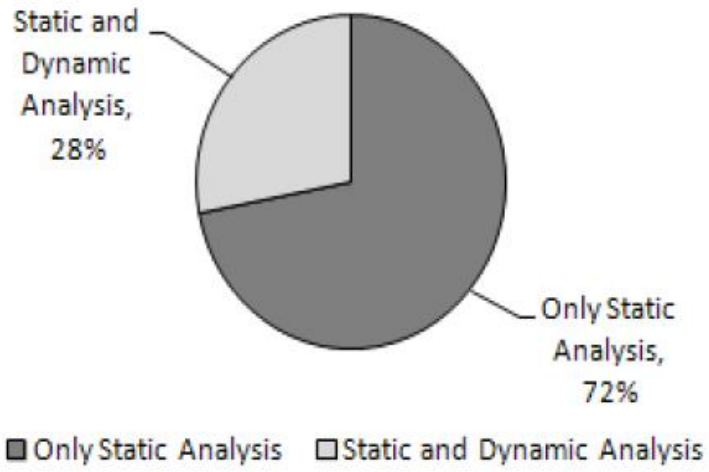
Taigi, šio eksperimento rezultatai rodo, jog tik 21 - 41% pažeidžiamumų buvo aptikti visų įrankių, 32 - 68% dalies įrankių, ir 11 – 27% pažeidžiamumų buvo neaptikti. Tokia statistika atskleidžia, jog įrankių tikslumas vis dar yra tobulintinas, o renkantis įrankį reikia atlikti kelių įrankių analizę ir bandymus, kad būtų galima įvertinti tikslumą ir tinkamumą.

1.4. Statinės kodo analizės naudotojų analizė

Statinė kodo analizė gali būti naudojama bet kokiam programiniam kodui - įvairioms veiklos sritims ar tikslams skirtai programinei įrangai analizuoti.

Dažniausiai statinės kodo analizės naudotojai yra programinės įrangos inžinieriai, naudojantys statinę analizę kodo savybių, rodančių kodo kokybę, saugumo spragų kiekį ir panašias metrikas, analizei. Tai leidžia užkirsti kelią saugumo spragų atsiradimui, užtikrinti kodo standartų laikymąsi ir panašias programinės įrangos savybes.

Remiantis programinės įrangos inžinierių apklausa [1], statinės ir dinaminės kodo analizės įrankių naudojimo statistika yra:



1.4 pav. Statinės ir dinaminės analizės įrankių naudojimo statistika

Taigi, apibendrinus:

- 100% respondentų naudojo statinės analizės įrankius;
- 28% naudojo ir statinės, ir dinaminės analizės įrankius;
- 72% naudojo tik statinės analizės įrankius;
- nei vienas respondentas nenaudojo tik dinaminės analizės įrankių.

1.5. Esamų problemos sprendimų analizė

Šiame skyriuje pateikiama informacija apie įrankių vertinimą, kriterijus kuriais remiantis yra vertinamas įrankių tinkamumas naudojimui, taip pat aprašoma metodika, skirta įvertinti įrankių tikslumą ir atsparumą „false positive“ situacijoms.

1.5.1. „Web application security consortium" įrankių vertinimo kriterijai

„Web application security consortium" pateikiamame statinės kodo analizės įrankių vertinimo kriterijų straipsnyje [17] atsižvelgiant į galimus įrankių naudotojus bei teikiamą funkcionalumą, nurodomi reikiamo funkcionalumo įvertinimo kriterijai, taip pat netiesioginiai, naudojimo aplinkos (organizacijos specifikos, rezultatų pateikimo reikalavimų) kriterijai, leidžiantys pasirinkti tinkamiausią įrankį konkrečiam atvejui.

Pateikiamas naudojimo aplinkos kriterijų sąrašas, į kurį reikėtų atsižvelgti prieš analizuojant įrankių funkcionalumą:

- kas vykdys statinę kodo analizę;
- įrankio licencija;
- kaip analizės įrankis bus integruojamas į darbo procesus;
- kokiu būdu bus pateikiami analizės rezultatai;
- ar skiriama lėšų įrankiui įsigyti;
- ar kodas gali būti analizuojamas už organizacijos ribų (potencialiai atskleistas trečiosioms šalims).

Tai netiesioginiai kriterijai, tačiau gana svarbūs – nors įrankis ir atitiktų visus keliamus funkcinius reikalavimus, tačiau jei jis yra mokamas, o lėšų įsigijimui nėra skirta, įrankio analizė tampa beprasmė. Įrankio nenaudoti gali tekti dėl netinkamu būdu ar formatu pateikiamų rezultatų, taip pat aktualu ar programinis kodas gali būti prieinamas trečiosioms šalims, jei analizė vykdoma nutolusiame serveryje. Apibendrinant „Web application security consortium" pateikiamus funkcinius vertinimo kriterijus, jų sąrašas yra:

- platformos palaikymas – diegimo, konfigūravimo ir naudojimo instrukcijų pateikimas ir kokybė, įrankio trūkumų (pavyzdžiui neanalizuojamų spragų sąrašo) pateikimas, lygiagretaus analizės proceso vykdymo palaikymas;
- technologijų palaikymas – palaikomos programavimo kalbos, ar atsižvelgiama į programavimo karkasų specifikas, konfigūracijos galimybės ir palaikomi konfigūracijos failų formatai;
- vykdymas ir valdymas – ar galima analizės vykdymą integruoti į programuotojams skirtą programinę įrangą (angl. „IDE"), ar galima redaguoti esamas analizės taisykles, pridėti savo taisykles. taip pat, ar yra galimybė sudaryti analizės tvarkaraštį („scheduled scan"), ar rezultatai gali būti pateikiami realiu laiku;
- pažeidžiamumų testavimo galimybės – kiek ir kokių pažeidžiamumų įrankis gali aptikti;
- analizės taisyklių, pažeidžiamumų sąrašo atnaujinimas – kaip dažnai įrankis ir jo taisyklės yra atnaujinamos, ar naudotojams yra galimybė pranešti apie klaidas, pageidaujamus atnaujinimus ar patobulinimus;
- rezultatų pateikimo lankstumas – skirtingais pjūviais pateikiamų rezultatų, atskiriant aukšto lygmens duomenis, skirtus organizacijos vadybai ir detalizuotus duomenis

skirtus programinės įrangos inžinieriams, palaikymas. Dviejų analizių rezultatų palyginimo ir panašios galimybės;

- meta duomenų pateikimas ir valdymas – aptikto pažeidžiamumo aprašymo, grėsmės lygmens, konkrečios vietos, kodo eilutės pateikimas. ar galima rastas problemas pažymėti kaip „false positive“ atvejus, ar įrankis turi galimybę tokius atvejus ateityje ignoruoti;
- organizacijos lygmens (angl. „enterprise“) programinės įrangos poreikiai – integracijos į kitas sistemas, duomenų gavybos (angl. „data mining“) siekiant analizuoti saugumo ir klaidų atsiradimo tendencijas, palaikymas.

Ilgas statinės analizės įrankių vertinimo kriterijų sąrašas, apimantis įvairiausius aspektus, rodo, jog tinkamiausio įrankio pasirinkimas gali būti sudėtingas procesas, reikalaujantis programinės įrangos inžinierių bei vadybos, teisinės srities žmonių įvertinimo skirtingais aspektais, pavyzdžiui įrankio konfigūravimo, licencijos ir aukšto lygmens ataskaitų generavimo galimybių. Daugybę kriterijų visiškai atitinkantį įrankį surasti daugeliu atvejų gali būti neįmanoma, todėl reikia nuspręsti kurie privalumai ir trūkumai yra svarbūs, o kurie nereikšmingi – tai taip pat reikalauja papildomų resursų. Taigi statinės analizės įrankių vertinimas ir pasirinkimas, pritaikymas konkretiems atvejams yra gana sudėtingas procesas, kuris gali būti analizuojamas ir tobulinamas.

1.5.2. „False positive“ problemos sprendimai

Siekiant nustatyti statinės kodo analizės įrankio pateikiamų „False positive“ įspėjimų pasikartojimo dažnumą, šią reikšmę, remiantis Aleksandro Ramos‘o Knudsen‘o darbu [18], galima apskaičiuoti pagal formulę:

$$\frac{\text{"False positive" kiekis}}{\text{"False positive" kiekis} + \text{"True negative" kiekis}}$$

Tuo tarpu, remiantis tuo pačiu šaltiniu, statinės analizės įrankio tikslumas apskaičiuojamas pagal formulę:

$$\frac{\text{"True positive" kiekis}}{\text{"True positive" kiekis} + \text{"False positive" kiekis}}$$

Šios reikšmės leistų nustatyti, kaip dažnai įrankis pateikia klaidingus pranešimus, tačiau kad būtų galima atlikti skaičiavimus, reikia žinoti „false positive“ pranešimų kiekį, metodus jų identifikavimui.

Siekiant nustatyti „false positive“ pranešimų priežastis buvo atliktas tyrimas [19], kurio metu buvo išanalizuotos programinio kodo vietos, sukėlusios šią situaciją, sudarytas labiausiai pažeidžiamų kodo struktūrų sąrašas. Pritaikius kompiuterinio mokymosi (angl. „machine learning“) „Naive Bayes“ ir „Long Short-Term Memory“ metodus, sukurti algoritmai, aptinkantys „false positive“ pranešimus.

Šių algoritmų tikslumai pateikiami lentelėje:

1.2 lentelė „False positive“ aptikimas naudojantis kompiuterinio mokymosi algoritmais

Algoritmas	Tikslumas
Naive Bayes	67.5%
LSTM	87.3%

Taigi, bendrasis tikslumas gana didelis, tačiau abu metodai į „false positive“ sąrašą įtraukė ir tas programinio kodo vietas, kuriose problemos iš tiesų egzistavo („true positive“). Naudojant „Long Short-Term Memory“, vidutiniškai aptikta 97.15% „false positive“ pranešimų, tačiau į sąrašą įtraukta 20.25% vietų su egzistuojančiomis kodo problemomis.

1.6. Darbo tikslas, uždaviniai, planas ir siekiami privalumai

Darbo tikslas - patobulinti „PHP“ programavimo kalbos statinės kodo analizės įrankius ir su saugumu susijusių problemų aptikimą.

Kad tikslas būtų pasiektas, reikia įgyvendinti šiuos uždavinius:

- išanalizuoti statinės kodo analizės metodus ir principus;
- išskirti bei išanalizuoti esminius statinės kodo analizės metodų ir įrankių trūkumus;
- išanalizuoti metodus ir įrankius atsižvelgiant į išskirtus esminius trūkumus;
- pateikti sprendimus, padedančius pašalinti trūkumus ar sumažinti jų neigiamą poveikį;
- atlikti bandymus ir eksperimentus, rodančius sprendimų efektyvumą.

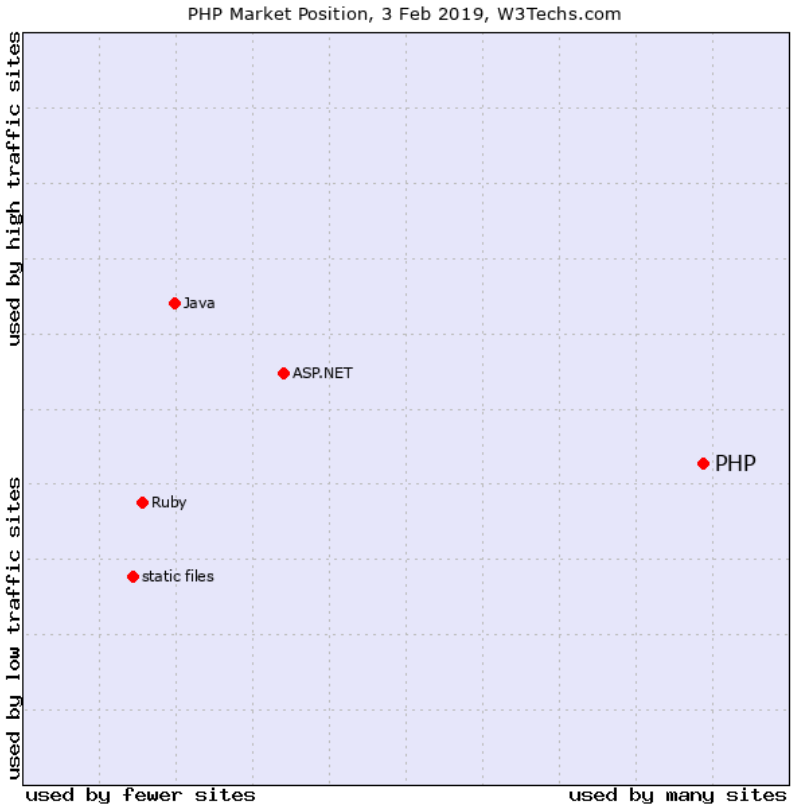
Išanalizavus statinės kodo analizės sritį, aktualiausias problemas, dažniausiai naudojamus vertinimo kriterijus, išskyrus esminius įrankių trūkumus, planuojama pateikti pasiūlymus ar sprendimus, padedančius pašalinti įrankių trūkumus ar sumažinti neigiamą pasirinktų trūkumų poveikį, taip pat atlikti bandymus ir eksperimentus, rodančius pateiktų sprendimų efektyvumą. Tai turėtų padidinti statinės kodo analizės įrankių efektyvumą ir patikimumą.

1.6.1. „PHP“ programavimo kalba

Darbe analizuojami „PHP“ („Hypertext Preprocessor“) programavimo kalbai skirti statinės kodo analizės įrankiai. Ši kalba sukurta 1994 metais ir nuo to laiko nuolat atnaujinama. Remiantis „PHP“ dokumentacija [20], tai yra bendro panaudojimo programavimo kalba, labiausiai tinkanti internetinių informacinių sistemų kūrimui. Taip pat pabrėžiama, jog ši kalba gali būti ne tik plačiai panaudojama, tačiau ją išmokti yra paprasta, todėl naudoti gali ir pradedantieji, ir patyrę programinės įrangos inžinieriai.

Šaltinio [21] autoriai apibūdina PHP programavimo kalbą kaip dinamiškai tipizuotą kalbą, palaikančią šimtus bibliotekų, modulių bei funkcijų. Taip pat pabrėžiama, jog PHP nėra kompiliuojama kalba, todėl programinės įrangos kūrimo procesas vyksta sparčiau nei naudojant kompiliuojamas kalbas.

Remiantis apklausas vykdančios organizacijos duomenimis [22], „PHP" kalba yra naudojama apklaustų 78,9% sistemų. Pateikiama diagrama, atspindinti 2019 m. naudojamų programavimo kalbų pasiskirstymą pagal naudojančių sistemų kiekį ir sistemos naudotojų srauto kriterijus:



1.5 pav. - Kalbų pasiskirstymas pagal naudojančių sistemų kiekį ir naudotojų srautą

Remiantis šią diagrama galima teikti, jog „PHP" kalba yra dažniausiai naudojama vidutinio srauto sistemose, o didelio srauto, dideles apkrovas turinčiose sistemose dažniau naudojamos „Java" ar „ASP.NET" technologijos.

Tačiau, nors „PHP" dažniausiai naudojama „vidutinėse" sistemose, pateikiamas sąrašas populiariausių sistemų, kurios naudoja šią kalbą – pirmose sąrašo vietose yra „Facebook" ir „Wikipedia". Taigi, „PHP" kalba yra naudojama ir įprastose, nesudėtingose, ir didelės apimties, korporacijų sistemose, todėl įvairaus sudėtingumo realizacijų bei sistemas kuriančių asmenų kompetencijos, kyla grėsmė atsirasti įvairaus tipo saugumo pažeidžiamumams.

Taip pat kaip pažeidžiamumų atsiradimo priežastį savo darbe [23] Johannes'as Dahs'as įvardija tai, jog „PHP" iš pradžių buvo tiesiog bibliotekų rinkinys skirtas asmeniniams tinklalapiams kurti, o vėliau kaip kalba „PHP" buvo ne kartą keista, atnaujinta, tobulinta – dėl šių priežasčių neišvengta projektavimo ir realizacijos klaidų, leidžiančių atsirasti pažeidžiamumams.

1.6.2. Sistemų ir karkasų pažeidžiamumų pavyzdžiai

„RIPSTech“ straipsnyje [24] pabrėžiama, jog nors perduoti naudotojų įvedamus duomenis „unserialize“ funkcijai griežtai nerekomenduojama, tai daroma itin dažnai. Teiginiui pagrįsti pateikiami realių sistemų ir karkasų objektų įterpimo pažeidžiamumų pavyzdžiai:

1.6.2.1. „Pydio“ sistemos pažeidžiamumas

„Pydio“ yra failų dalinimosi sprendimas, prieinamas viso pasaulio naudotojams. Aprašomas pažeidžiamumas yra ištaisytas, tačiau iki 8.2.2 versijos jis leido įsilaužėliams perimti failų sistemos valdymą, perskaityti privačius duomenis.

Naudotojams suteikiama galimybė turėti personalizuotus nustatymus, tokius kaip grafinės sąsajos parametrai, kartais šie parametrai saugomi kaip reikšmių masyvai. Kadangi sistemoje naudojama „MySQL“ duomenų bazė neturi duomenų lauko tipo, leidžiančio saugoti reikšmių masyvus, jie serializuojami ir duomenų bazėje saugomi kaip tekstinės reikšmės. Kai kurie parametrai saugomi kaip tekstinės ar skaitinės reikšmės, kiti kaip serializuoti masyvai, tad tam jog atskirtų kurios reikšmės turi būti deserializuojamos, prie serializuotų reikšmių priekyje pridedamas tekstas „*$phpserial$*“. Kuomet parametras yra nuskaitomas iš duomenų bazės, tikrinama ar jo reikšmė prasideda šiuo tekstu – jeigu taip, reikšmė yra deserializuojama.

Taigi, saugumo spraga yra tai, jog naudotojas gali sąsajos kalbos parametro „lang“ reikšmės pridėti tekstą „*$phpserial$*“ ir taip pats inicijuoti įvesto teksto deserializavimą. Tai leidžia inicijuoti norimus objektus su pasirinktomis reikšmėmis siekiant gauti prieigą prie apsaugotų duomenų.

Pažeidžiamumą nusakančios programinio kodo iškarpos:

/plugins/conf.sql/SqlUser.php

```
165  public function setPref($prefName, $prefValue)
166      {
167          // If the preference value (the user input) is not a string, serialize it
168          if (!is_string($prefValue)) {
169              $prefValue = '$phpserial$'.serialize($prefValue);
170          }
171
172          ⋮
173          // Insert the preference into the database.
174          dibi::query('INSERT INTO [ajxp_user_prefs] ([login],[name],[val]) VALUES (%s,
175          ⋮
176  }
```

1.6 pav. „Pydio“ pažeidžiamumas, parametro saugojimas

/plugins/conf.sql/SqlUser.php

```
208     public function getPref($prefName)
209     {
210         $p = parent::getPref($prefName);
211             if (strpos($p, '$phpserial$') === 0) {
212                 $p = substr($p, strlen('$phpserial$'));
213                 return unserialize($p);
214             }
215             ⋮
216         return $p;
217     }
```

1.7 pav. „Pydio“ pažeidžiamumas, parametro nuskaitymas

plugins/action.share/src/Http/MinisiteAuthMiddleware.php

```
110         if (isSet($_GET["lang"])) {
111             if ($ctx->hasUser()) {
112                 $ctx->getUser()->setPref("lang", $_GET["lang"]);
113             }
114         }
```

1.8 pav. „Pydio“ pažeidžiamumas, išnaudojimas

Taigi, serializacijos funkcionalumo naudojimas nerekomenduojamu būdu ir tam tikri duomenų saugojimo būdo sprendimai sukelia pažeidžiamumą, kurį išnaudoti yra gana paprasta – pakanka pateikti reikiamą nustatymų parametro reikšmę, kad gauti prieigą prie privačių duomenų.

1.6.2.2. „PrestaShop“ pažeidžiamumas

„PrestaShop“ yra atviro kodo elektroninės komercijos sprendimas, remiantis turinio valdymo sistemų naudojimo statistika [25], patenkantis tarp dešimties dažniausiai naudojamų. Kadangi „PrestaShop“ naudojančiose sistemose dažniausiai vykdoma prekyba, atliekami mokėjimo pavedimai, šių sistemų saugumo užtikrinimas yra svarbus uždavinys.

Tačiau ir čia aptiktas pažeidžiamumas, susijęs su serializacijos procesu ir pasireiškiantis versijose iki 1.7.2.4. Pažeidžiamumas aptiktas užsakymų valdymo posistemėje, kad būtų galima išnaudoti, reikalinga sąlyga yra, kad naudotojui būtų priskirta viena iš rolių, turinčių prieigą prie užsakymų posistemės – „Salesman“, „Logistician“, „Admin“.

Programiniame kode aptiktas „unserialize“ funkcijos naudojimas su naudotojo pateiktais duomenimis. Kodo autoriams ši situacija žinoma, ir, kad sistema būtų apsaugota nuo objekto įterpimo pažeidžiamumo, pridėta uždengianti (angl. „wrapper“) funkcija „unSerialize“, kurioje prieš perduodant duomenis į originalią „unserialize“ funkciją, duomenys yra tikrinami reguliariosios išraiškos šablonu, siekiant nustatyti ar nėra bandoma perduoti serializuotą objektą.

Ši funkcija:

```php
public static function unSerialize($serialized, $object = false)
{
    if (is_string($serialized) && (strpos($serialized, 'O:') === false
        || !preg_match('/(^|;|{|})O:[0-9]+:"/', $serialized)) && !$object
        || $object) {
        return @unserialize($serialized);
    }

    return false;
}
```

1.9 pav. „PrestaShop“ uždengiančio metodo apsauga nuo serializacijos pažeidžiamumų

Taigi, tokia apsauga turėtų apsaugoti nuo perdavimo tokių duomenų, kaip:

```
O:8:"stdClass":0:{}
```

1.10 pav. „PrestaShop“ pažeidžiamumo išnaudojimui netinkami duomenys

Tai tik elementarus pavyzdys, iliustruojantis primityvaus objekto be papildomų reikšmių serializaciją, tačiau pastebėta, kad yra būdų tą pačią reikšmę modifikuoti taip, kad „unserialize“ rezultatas būtų tas pats, tačiau apsauginė reguliarioji išraiška neaptiktų objekto įterpimo, pavyzdžiui:

```
O:+8:"stdClass":0:{}
```

1.11 pav. „PrestaShop“ pažeidžiamumo išnaudojimui tinkami duomenys

Pavyzdyje prie klasės pavadinimo ilgį nusakančio skaičiaus prijungiamas pliuso ženklas, kuris nekeičia serializacijos rezultato, tačiau nėra aptinkamas apsauginės „unSerialize“ funkcijos. Tai leidžia inicijuoti pasirinktus objektus, keisti jų klasės kintamuosius, taip išnaudoti įvairius kitus pažeidžiamumus.

Šie pažeidžiamumų pavyzdžiai, pasireiškiantys tikrose sistemose, karkasuose, naudojamuose daugelio žmonių, įmonių įvairiais tikslais ir poreikiais, leidžia teigti jog apibūdintos problemos yra aktualios ir jų sprendimo būdai būtų naudingi kuriant naujas ir tobulinant esamas sistemas ir programavimo karkasus.

1.6.3. „PHP“ programavimo kalbos statinės analizės įrankiai

Statinei programinio kodo analizei vykdyti yra sukurta įvairių komercinių ar atviro kodo įrankių skirtų įvairiems tikslams – analizuoti programinio kodo stilių, sintaksę, saugumo spragas ir panašias savybes. Remiantis Morico Bellerio, Radino Bolanato, Šeino Makintošo ir Endžio Zaidmano straipsnyje [26] pateikiamais apklausos rezultatais, apie 30 – 36% atvirojo kodo projektų naudoja automatinius statinės kodo analizės įrankius. Tai rodo, jog nors ir nėra itin paplitę atviro kodo projektuose, statinės analizės įrankiai yra reikalingi, todėl prasminga analizei pasirinkti atviro kodo statinės analizės įrankį. Siekiant išrinkti įrankį, kuris bus analizuojamas ir tobulinamas šiame darbe, parinkta trijų atviro kodo įrankių aibė, įrankių aprašymai pateikiami šiame skyriuje.

1.6.3.1. „Phan“ įrankis

Šis įrankis skirtas analizuoti bendrinius kodo aspektus – nenaudojamų funkcijų aptikimas, kodo suderinamumas su serveryje įdiegta „PHP“ versija, dvigubo klasių ar funkcijų deklaravimo aptikimas ir kitų panašių aspektų analizė. Įrankis yra naudojamas, kartais atnaujinamas bendruomenės. Iš populiariausios „PHP“ paketų saugyklos „Packagist“ šis įrankis buvo įdiegtas virš 259 tūkst. kartų.

Įrankio „Phan“ rezultatų pateikimas:

```
src/Vulnerability/RuntimeConfigurationDisplayErrorsDirective/example.php:11 PhanTypeMismatchArgument Argument 1 (year) is '999' but \MBD\Vulnerability\RuntimeConfigur
ationDisplayErrorsDirective\Car::__construct() takes int defined at src/Vulnerability/RuntimeConfigurationDisplayErrorsDirective/Car.php:11
```

1.12 pav. „Phan“ rezultatų atvaizdavimas

Rezultatai spausdinami panašiu formatu į programinių klaidų (angl. „exceptions“) formatą, nurodant failą, eilutę kurioje egzistuoja problema ir problemos aprašymo tekstą.

1.6.3.2. „PHPSA“ įrankis

Įrankis taip pat skirtas analizuoti įvairius kodo aspektus - nuo dalybos iš nulio, tuščių funkcijų deklaravimo iki serializacijos pažeidžiamumų aptikimo. Šis įrankis yra atviro kodo, palaikomas bendruomenės, kuri taiso klaidas, atnaujina ir kuria naujas analizės taisykles. Iš paketų saugyklos „Packagist“ šis įrankis buvo įdiegtas virš 12 tūkst. kartų.

„PHPSA“ dokumentacijoje pateikiamas analizės rezultatų atvaizdavimo pavyzdys:

```
$ ./bin/phpsa check fixtures/

Syntax error:  Syntax error, unexpected T_RETURN on line 11 in fixtures/simple/syntax/Error2.php

    $b = $a + 1; 123123

Notice:  Constant BBBB does not exist in self scope in fixtures/simple/undefined/Const.php on 29 [undefined-const]

    return self::BBBB;

Notice:  You are trying to cast 'string' to 'string' in fixtures/simple/code-smell/StandardFunctionCall.php on 16 [stu

    return (string) json_encode(array(
```

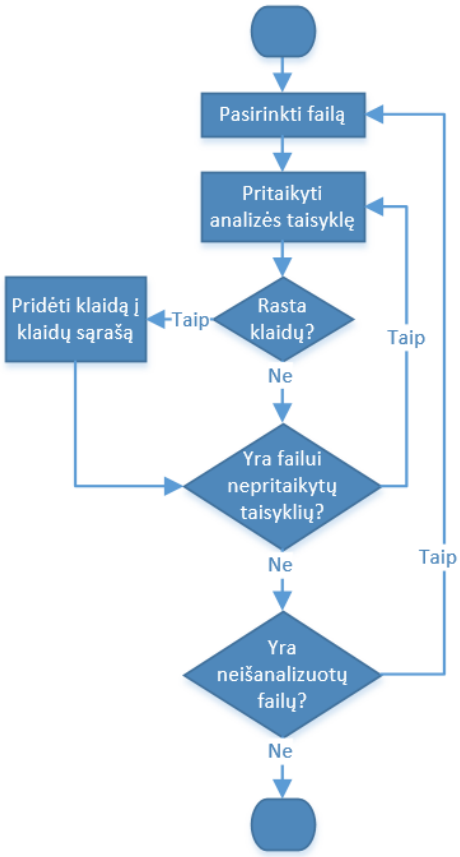
1.13 pav. „PHPSA“ analizės rezultatų atvaizdavimas

Rezultatai atvaizduojami pateikiant klaidos pranešimą ir probleminės kodo eilutės numerį.

1.6.3.3. „PHPStan“ įrankis

„PHPStan“ yra aktyviai naudojamas (Iš paketų saugyklos „Packagist“ įrankis įdiegtas virš 2,8 mln. kartų), palaikomas ir tobulinamas atviro kodo statinės kodo analizės įrankis, skirtas „PHP“ programavimo kalbai. „PHPStan“ gali aptikti įvairias nerekomenduojamas kodo konstrukcijas, saugumo spragas, kodo klaidas (angl. „bugs“), šiam įrankiui galima kurti savo plėtinius (angl. „Extension“), todėl įrankį nesunku pritaikyti savo reikmėms. „PHPStan“ turi plėtinių, specializuotų „Doctrine“, „PHPUnit“, „Symfony“ karkaso analizei. Įrankio veikimas pagrįstas kodo analizės taisyklių pritaikymu programinio kodo failų rinkiniui.

Analizės procesą iliustruojanti diagrama:



1.14 pav. „PHPStan“ įrankio analizės procesas

Analizės procesas vykdomas kiekvienam analizuojamam failui pritaikant analizės taisykles ir aptiktas klaidas analizės pabaigoje pateikiant sąrašu. Analizės rezultatų pateikimo pavyzdys:

```
18/18 [============================] 100%

------ ------------------------------------------------------------------------------------------------------------------------------------------------------
 Line   ObjectInjectionVulnerableFunctionsInDestruct\VulnerableExec.php
------ ------------------------------------------------------------------------------------------------------------------------------------------------------
 15     Function 'exec' with class property 'cacheDirectory' as parameter used inside method '__destruct' - vulnerable to object injection
------ ------------------------------------------------------------------------------------------------------------------------------------------------------

------ ------------------------------------------------------------------------------------------------------------------------------------------------------
 Line   ObjectInjectionVulnerableFunctionsInDestruct\VulnerablePassthru.php
------ ------------------------------------------------------------------------------------------------------------------------------------------------------
 15     Function 'passthru' with class property 'cacheDirectory' as parameter used inside method '__destruct' - vulnerable to object injection
------ ------------------------------------------------------------------------------------------------------------------------------------------------------
```

1.15 pav. „PHPStan“ analizės rezultatų atvaizdavimas

Rezultatai pateikiami grupuojant pagal failą, nurodant eilutės numerį, kurioje problema aptikta.

1.6.4. Pasirinktas statinės kodo analizės įrankis

Įrankio pasirinkimui sudaryta lentelė su pasirinkimo kriterijais ir parametrais, leidžiančiais palyginti įrankio populiarumą, išplečiamumą, universalumą:

1.3 lentelė Statinės kodo analizės įrankių parametrai

Įrankis	Įdiegtas kartų	Galimybė integruoti savo taisykles	Analizuojami problemų tipai
„Phan“	259 000	Nenurodyta dokumentacijoje	Universalus, įvairios problemos
„PHPSA“	12 000	Taip	Universalus, įvairios problemos
„PHPStan“	2 800 000	Taip	Universalus, įvairios problemos

Remiantis pateiktais parametrais, analizei pasirinktas atviro kodo įrankis „PHPStan“. Sprendimas priimtas dėl dviejų priežasčių. Pirma priežastis yra įrankio išplečiamumas - įrankio programinio kodo saugykloje [27] teigiama, jog prie įrankio vystymo galima prisidėti kuriant naujas ar tobulinant esamas statinės analizės taisykles, taip pat kuriant savo taisyklių rinkinius – įrankio plėtinius. Antroji priežastis - „PHPStan“ iš paketų saugyklos įdiegtas virš 10 kartų daugiau nei antroje vietoje pagal šį kriterijų esantis įrankis, taip pat jau yra sukurta plėtinių, specializuotų įvairioms „PHP“ bibliotekoms ir karkasams, tai rodo jog įrankis yra plačiai naudojamas ir pritaikomas įvairiems tikslams.

1.7. Išvados

Analizės metu atlikta probleminės srities analizė, apžvelgti statinės kodo analizės principai ir metodai, įvardintos pagrindinės problemos. Taip pat įvardinti pagrindiniai statinės kodo analizės įrankių naudotojai, apžvelgti esami problemos sprendimai.

Esamų problemos sprendimų analizės metu išsiaiškinta, jog įrankio pasirinkimui naudojamų funkcinių ir nefunkcinių kriterijų yra daugybė, o pats pasirinkimo procesas, jei siekiama rasti patį tinkamiausią įrankį, gali pareikalauti nemažai organizacijos išteklių.

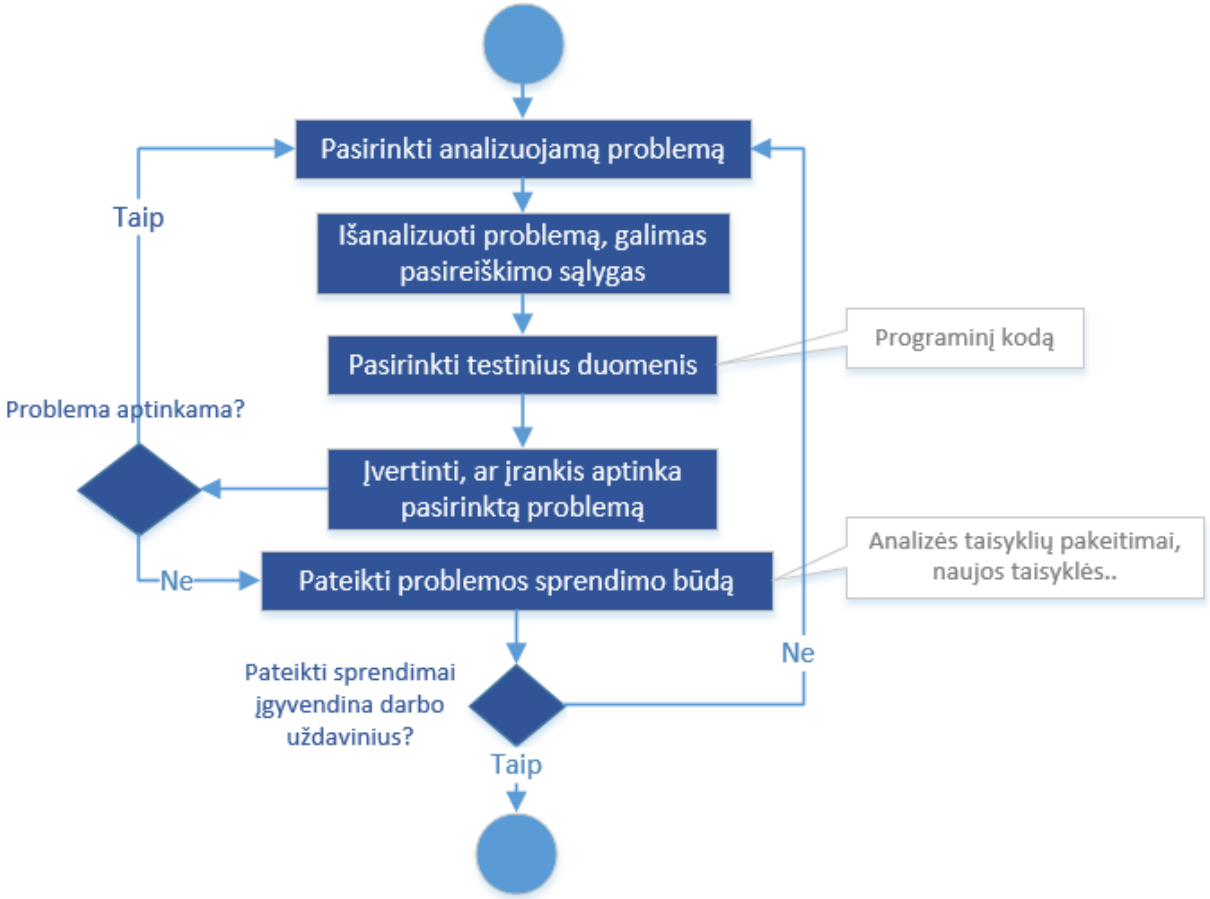
Išanalizavus statinės kodo analizės įrankių problemas, nustatyta, jog pagrindiniai trūkumai šiuo metu yra „false positive“ problema, kuomet statinės analizės įrankiai nurodo aptiktą saugumo spragą ar klaidą vietoje, kurioje iš tiesų problemos nėra, ir įrankių netikslumas – egzistuojančių pažeidžiamumų neaptikimas. „False positive“ situacijos pasikartojimo dažnumui apskaičiuoti naudojamos formulės yra žinomos, taip pat sukurti ir kompiuterio mokymosi algoritmai, skirti identifikuoti pažeidžiamiausias programinio kodo vietas, tačiau šie algoritmai vis dar nėra itin tikslūs.

Statinės kodo analizės įrankių, skirtų PHP programavimo kalbai tobulinimui, pasirinktos trys klaidų ir pažeidžiamumų atsiradimo priežasčių grupės - objektų serializacija, konfigūracijos pažeidžiamumai, nereikalinga išvestis. Pagal nustatytus kriterijus, iš statinės kodo analizės įrankių pasirinktas šiame darbe tobulintinas įrankis – „PhpStan“.

2. STATINĖS ANALIZĖS ĮRANKIO TAISYKLIŲ RINKINIŲ KONCEPCIJA

Siekiamas sprendimas – išanalizavus statinės kodo analizės įrankio veikimo principus, privalumus ir trūkumus, pakoreguoti įrankyje naudojamas analizės taisykles taip, kad įrankis aptiktų daugiau programinio kodo klaidų, saugumo spragų, arba jau analizuojamas problemas aptiktų tiksliau.

Siekiamas sprendimas turėtų patobulinti statinės analizės įrankį – papildyti ar pakoreguoti naudojamas statinės kodo analizės taisykles. Remiantis anksčiau atlikta analize ir aprašytu įrankių veikimu, sprendimui įgyvendinti pasirinktas darbo procesas, pateikiamas diagramoje:



2.1 pav. Siekiamo sprendimo įgyvendinimo procesas

Pasirinkus programavimo kalboje pasireiškiančias problemas ir galimas saugumo spragas, išanalizuoti kiekvieną jų, įvertinti galimas pasireiškimo sąlygas, kokiais atvejais problemos gali kilti. Pagal šią informaciją pasirinkti testinius duomenis (programinio kodo rinkinį), kurie leistų problemą atkartoti, tuomet įvertinti ar įrankis aptinka pasirinktą problemą. Jei problema neaptinkama, pateikiamas problemos sprendimo būdas modifikuojant esamas analizės taisyklės arba sukuriant naujas, jei problema aptinkama – tobulinimui poreikio nėra, pereinama prie kitos problemos analizės.

2.1. Sprendžiamos PHP kalbos problemos

Išanalizavus ir palyginus kelių statinės kodo analizės įrankių veikimą, nuspręsta išanalizuoti ir pasiūlyti sprendimus problemoms, kurias galima suskirstyti į tris grupes:

- objektų serializacija;
- konfigūracijos pažeidžiamumai;
- nereikalinga išvestis.

2.1.1. Objektų serializacija – Objektų įterpimas

Remiantis straipsniu [28], PHP kalboje serializacija leidžia reikšmes, objektus paversti į tekstinę reikšmę, o vėliau tekstinę reikšmę paversti atgal į originalią. Serializacija PHP kalboje vykdoma naudojant „serialize“ funkciją, o originali reikšmė gaunama panaudojus priešingą – „unserialize“ funkciją.

Objekto serializacijos proceso pavyzdys pateikiamas „RIPS Tech“ straipsnyje [24]:

- objektas serializuojamas naudojant funkciją „serialize“:

```php
<?php
$object = new stdClass();
$object->data = "Some data!";
$cached = serialize($object);
```

2.2 pav. Objekto serializavimo pavyzdys

- gaunama tekstinė reikšmė, apibrėžianti originalų objektą:

```
O:8:"stdClass":1:{s:4:"data";s:10:"Some data!";}
```

2.3 pav. Serializuoto objekto tekstinė reikšmė

- tekstinė reikšmė paverčiama atgal į objektą naudojant funkciją „unserialize“:

```php
<?php
$object = unserialize( str: 'O:8:"stdClass":1:{s:4:"data";s:10:"Some data!";}');
echo $object->data;
```

2.4 pav. Tekstinės reikšmės konvertavimas į originalų objektą

Taigi įprastą objekto serializacijos procesą sudaro trys žingsniai – objekto pavertimas į tekstinę reikšmę („serialize“), tekstinės reikšmės panaudojimas (saugojimas duomenų bazėje, perdavimas tinklu ar pan.), tekstinės reikšmės pavertimas atgal į objektą („unserialize“).

Naudotis šiuo funkcionalumu gana paprasta, tačiau jis turi ir saugumo spragų, viena jų yra objektų įterpimo (angl. „object injection“). Remiantis Johannes‘o Dahs‘o darbu [23], tai yra sudėtingesnis ir mažiau žinomas pažeidžiamumas, lyginant su populiariais pažeidžiamumais, kaip

„SQL Injection“, dėl šios priežasties skiriama mažiau pastangų apsisaugoti nuo jo išnaudojimo, o atakuotojai jį dažnai sėkmingai išnaudoja. Sąvoka „objekto įterpimas“ gali apimti keletą būdų išnaudoti serializacijos pažeidžiamumus, šiame darbe analizuojami du būdai – paremti „wakeup“ ir „destruct“ metodų naudojimu.

2.1.1.1. Objektų įterpimas „wakeup“ metode

Šios spragos išnaudojimas yra paremtas tuo, jog kviečiant „unserialize“ funkciją kai reikšmė yra objektas, objekto klasėje ieškoma metodo „wakeup“ ir jis yra automatiškai įvykdomas. Nors „__wakeup“ metodo realizacijos pakeisti serializacijos metu neįmanoma, tačiau jei jis naudoja klasės kintamuosius (angl. „property“), šias reikšmes galima pakeisti serializacijos metu ir jos bus panaudotos „unserialize“ vykdymo metu. Pažeidžiamumo pavyzdys „OWASP“ sistemoje [29]:

Originalus kodas

```
class Example2
{
   private $hook;

   function __construct()
   {
      // some PHP code...
   }

   function __wakeup()
   {
      if (isset($this->hook)) eval($this->hook);
   }
}
```

Įterpiamas kodas

```
class Example2
{
   private $hook = "phpinfo();";
}
```

2.5 pav. Objekto įterpimo pažeidžiamumo pavyzdys nr. 1

Pavyzdyje pateikto įterpiamo kodo klasės serializacijos ir panaudojimo pasekmė bus „unserialize“ vykdymo metu pakeista klasės kintamojo „hook“ reikšmė ir šios reikšmės, tai yra PHP funkcijos „phpinfo“ įvykdymas.

Išsamesnis objektų įterpimo pažeidžiamumo pavyzdys su komentarais:

```php
<?php

class SomeClass
{
    public $command = 'pwd';
    public $directory;

    // Metodas vykdomas "Unserialize" vykdymo metu
    // Kintamajam $directory priskiriamas komandos "pwd" rezultatas (esama direktorija)
    function __wakeup()
    {
        $this->directory = exec($this->command);
    }
}

$object = new SomeClass();
$serialized = serialize($object); // O:9:"SomeClass":2:{s:7:"command";s:3:"pwd";s:9:"directory";N;}

// Tarkime, kad reikšmė perduodama tinklu, tačiau yra pakeičiama:
$exploitString = 'O:9:"SomeClass":2:{s:7:"command";s:20:"rm importantFile.txt";s:9:"directory";N;}';

unserialize($exploitString); // Vykdoma komanda "rm importantFile.txt" - ištrinamas failas
```

2.6 pav. Objekto įterpimo pažeidžiamumo pavyzdys nr. 2

Šiame pavyzdyje demonstruojama situacija, kuomet normaliu vykdymo atveju vykdant „unserialize“ funkciją „directory“ kintamajam yra priskiriama esamos direktorijos reikšmė panaudojant „pwd“ komandą.

Tačiau esant galimybei pakeisti serializacijos rezultatą, „pwd“ komandą galima pakeisti kita norima, pavyzdžiui „rm importantFile.txt“, kurios paskirtis yra ištrinti failą.

2.1.1.2. Objektų įterpimas „destruct“ metode

Klasės destruktorius PHP kalboje aprašomas realizuojant metodą „__destruct“ – remiantis kalbos dokumentacija [20], šis metodas yra vykdomas kai nebelieka nuorodų į objektą arba tiesiog kodo vykdymo pabaigoje. Kaip ir „__wakeup“ atveju, šio metodo realizacijos pakeisti serializacijos metu neįmanoma, tačiau jei jis naudoja klasės kintamuosius (angl. „property“), šias reikšmes galima pakeisti serializacijos metu ir jos bus panaudotos objekto sunaikinimo metu.

„OWASP“ specifikacijoje [29] pateikiamas šio pažeidžiamumo pavyzdys:

```
class Example1
{
   public $cache_file;

   function __construct()
   {
      // some PHP code...
   }

   function __destruct()
   {
      $file = "/var/www/cache/tmp/{$this->cache_file}";
      if (file_exists($file)) @unlink($file);
   }
}
```

2.7 pav. Objekto įterpimo „destruct“ metode pažeidžiamumą turinčios klasės pavyzdys

Šios klasės destruktoriaus tikslas yra objekto sunaikinimo metu ištrinti podėlio (angl. „cache“) failą, kurio pavadinimas saugomas klasės kintamajame “cache_file”. Tačiau serializacijos metu, pakeitus tekstinę serializacijos rezultato reikšmę, galima šio kintamojo reikšmę pakeisti, taip siekiant ištrinti kitą failą. Pakeistų duomenų pavyzdys:

```
O:8:"Example1":1:{s:10:"cache_file";s:15:"../../index.php";}
```

2.8 pav. Objekto įterpimo „destruct“ metode pažeidžiamumą išnaudojančių duomenų pavyzdys

Taigi, pakeitus „cache_file“ kintamojo reikšmę į „../../index.php“, „destruct“ metodo vykdymo rezultatas bus ištrintas failas „var/www/index.php“.

2.1.2. Objektų serializacija – Nefiltruojama serializacija

Dar viena objektų serializacijos problema – galimybė „unserialize“ funkcijai perduoti suklastotą ar pakeistą reikšmę ir tokiu būdu inicijuoti bet kokios sistemoje esančios klasės objektą. Pasinaudojus šia galimybe, galima nesunkiai išnaudoti objektų įterpimo pažeidžiamumą, nes tikslas yra atrasti nors vieną „unserialize“ panaudojimo atvejį, kuriam galima perduoti norimą reikšmę – nereikia žinoti kokioje sistemos dalyje kokios klasės objektai inicijuojami ir prie to prisitaikyti. Šios problemos pavyzdys:

```php
<?php

class VulnerableClass
{
    // Tarkime ši klasė turi "Object Injection" pažeidžiamumą.
    // Klasė gali priklausyti ir trečiosios šalies bibliotekai, kuri naudojama projekte.
}

$x = [];
$serialized = serialize($x);

// Tarkime, kad $serialized reikšmė perduodama tinklu, tačiau yra pakeičiama:
$serialized = 'O:15:"VulnerableClass":0:{}';

$result = unserialize($serialized); // Reikšmė yra "VulnerableClass" objektas
```

2.9 pav. Nefiltruojamos serializacijos situacijos pavyzdys

Pavyzdyje matoma, jog nesvarbu kokia reikšmė yra serializuojama, ir kokio rezultato tikimasi – pakanka sugebėti „unserialize“ funkcijai perduoti norimą reikšmę ir tokiu būdu inicijuojamas objektas, taip pat automatiškai gali būti išnaudojamas ir objektų įterpimo pažeidžiamumas.

Šios problemos padeda išvengti papildomas „unserialize“ funkcijos parametras „allowed_classes“, skirtas sugriežtinti palaikomų klasių sąrašą. Galimos parametro reikšmės yra:

- visas klases (numatytoji reikšmė);
- jokių klasių;
- tik išvardintas klases (angl. „whitelist“).

Taigi, geroji praktika turėtų būti arba neleisti jokių klasių, arba leisti tik išvardintas klases.

2.1.3. Nereikalinga išvestis

Programinės įrangos kūrimo procese reikia nuolat stebėti programinio kodo atitikimą reikalavimams, tam naudojami automatiniai testai, taip pat kintamųjų reikšmių tikrinimui naudojami „debugger“ įrankiai. Tačiau remiantis straipsniu „How Developers Debug“ [30], kintamųjų analizė tiesiog spausdinant reikšmes ekrane vis dar itin dažnai naudojama programinės įrangos kūrimo procese. Šie veiksmai gali būti atliekami ir jau veikiančioje sistemoje, siekiant aptikti ir ištaisyti kodo klaidą. Tai turėtų būti atliekama tik darbo aplinkoje, tačiau naudojant reikšmių išvedimo į ekraną funkcijas, kartais pamirštama jas pašalinti ir šis kodas patenka į tikrąją sistemą. Tai neapdairumo pasekmė, savaime nesukelianti pažeidžiamumų, tačiau galinti atskleisti labai svarbios informacijos įsilaužėliams – direktorijų struktūrą, naudojamas bibliotekas, klasių, metodų pavadinimus ir pan.

Tačiau programa gali išvedinėti informaciją ir iš tiesų reikiamose vietose, siekiant pateikti informaciją naudotojui, pavyzdžiui vykdant kodą per komandinę eilutę. Tam reikia atskirti informacijos išvedimo funkcijas į pagalbines, dažniausiai naudojamas reikšmių tikrinimui ir tas, kurios iš tiesų pateikia informaciją naudotojui. Pavyzdžiui, funkcijos „var_dump“ ir „var_export“ pateikia išsamią informaciją apie kintamąjį, jo reikšmę ir tipą, todėl priklauso pagalbinėms funkcijoms, kurios neturėtų būti naudojamos programoje.

Taip pat informacija apie sistemą gali būti atskleista ir dėl konfigūracijos. Naudojant standartinę konfigūraciją, PHP kalbos klaidos atvaizduojamos ekrane, naršyklėje, pateikiant informaciją kurioje kodo vietoje klaida įvyko. Konfigūracijos nustatymai leidžia apriboti išvedamas klaidas ir pranešimus pagal jų svarbos lygį („notice“, „warning“, „error“ ir t.t.), taip pat visiškai išjungti arba įjungti šių klaidų išvedimą.). Reikia išanalizuoti ir atvejus, kuomet konfigūracijos parametrai valdomi programiniame kode proceso vykdymo metu, naudojant funkciją „ini_set“ (tik tam procesui kuriame panaudota ši funkcija, galima pakeisti ne visus parametrus).

Taigi, kalbos elementai, kurių taikymą reikėtų analizuoti:

- „var_dump“ funkcija – kintamųjų išvedimas;
- „var_export“ funkcija – kintamųjų išvedimas;
- „print_r“ funkcija – kintamųjų išvedimas;
- „debug_zval_dump“ funkcija – kintamųjų išvedimas.

2.1.4. Konfigūracijos pažeidžiamumai

PHP kalbos konfigūracija gali būti valdoma statiniais failais, skaitomais PHP procesui startuojant (bendroji konfigūracija). Konfigūracijos failo pavyzdys pateikiamas PHP dokumentacijoje [31]:

```
; any text on a line after an unquoted semicolon (;) is ignored
[php] ; section markers (text within square brackets) are also ignored
; Boolean values can be set to either:
;    true, on, yes
; or false, off, no, none
register_globals = off
track_errors = yes

; you can enclose strings in double-quotes
include_path = ".:/usr/local/lib/php"

; backslashes are treated the same as any other character
include_path = ".;c:\php\lib"
```

2.10 pav. „PHP“ konfigūracijos failo pavyzdys

Taip pat konfigūracija gali būti keičiama ir proceso vykdymo metu, naudojant funkciją „ini_set“ (tik tam procesui kuriame panaudota ši funkcija, galima pakeisti ne visus parametrus). Todėl saugumui užtikrinti nepakanka teisingai aprašyti konfigūracijos failą, taip pat reikia stebėti bei analizuoti programiniame kode nustatomas konfigūracijos reikšmes, galinčias lemti klaidas ar saugumo spragas.

OWASP specifikacijoje [32] pateikiamas sąrašas konfigūracijos parametrų, galinčių turėti įtaką sistemos saugumui. Keletas šių parametrų:

- disable_functions – galimybė išjungti pasirinktas PHP funkcijas. Rekomenduojama išjungti funkcijas, skirtas vykdyti komandinės eilutės komandas, keisti failų ir direktorijų prieigos teises, trinti direktorijas. Konkretūs nustatymai priklauso nuo programos specifikos;
- allow_url_include – leidžiama arba draudžiama įtraukti („include“, „require“) vykdymui nutolusius resursus (failus pasiekiamus tinklu);
- open_basedir – nustatoma „bazinė“ PHP direktorija. Leidžiama atidaryti ir vykdyti failus, esančius gilesnėse nei nurodyta direktorijose. Neteisinga reikšmė gali leisti vykdyti failus esančius už programos ribų.

Konfigūracijos failams analizuoti reikalinga speciali prieiga, kadangi šie failai yra laikomi „PHP“ diegimo kataloguose, dažnai apsaugotuose griežtomis prieigos teisių taisyklėmis, todėl tikėtina kad

konfigūracijos failo analizė būtų vykdoma laikinai perkeliant failą į pasiekiamą direktoriją ir analizę vykdant failo kopijai. Taigi, įrankyje reikėtų realizuoti galimybę nurodyti kelią iki konfigūracijos failo, kurį norima analizuoti.

2.2. Siekiami problemų sprendimo būdai

Šioje dalyje pateikiami siekiamų sprendimų apibrėžimai (problema - sprendimas):

- objektų įterpimas – įrankyje „phpstan“ sukurti statinės kodo analizės taisykles, aptinkančias pažeidžiamumą leidžiančių išnaudoti funkcijų („exec“, „system“, „shell_exec“, „passthru“, „unlink“) naudojimą metoduose „__wakeup“, „__destruct“;
- nefiltruojama serializacija – įrankyje „phpstan“ sukurti statinės kodo analizės taisykles, aptinkančias funkcijos „unserialize“ naudojimą su parametro „allowed_classes“ reikšme „leisti visas klases“ (parametras nenurodytas);
- nereikalinga išvestis – įrankyje „phpstan“ sukurti statinės kodo analizės taisykles, aptinkančias pagalbinių išvesties funkcijų („var_dump”, „var_export”, „print_r”, „debug_zval_dump”) naudojimą;
- nereikalinga išvestis – įrankyje „phpstan“ sukurti statinės kodo analizės taisykles, aptinkančias konfigūracijos parametrų, valdančių klaidų matomumą ir atvaizduojamų klaidų lygius („display_errors”, „error_reporting“, „display_startup_errors“), keitimą programiniame kode proceso vykdymo metu;
- konfigūracijos pažeidžiamumai – sukurti analizės įrankį ir taisykles, aptinkančias pažeidžiamumus galinčių sąlygoti parametrų („allowed_classes“, „open_basedir“, „allow_url_fopen“..) naudojimą su numatytosiomis arba netinkamomis reikšmėmis.

Siekiamas sprendimas susideda iš dviejų dalių – taisyklių rinkinio „PHPStan“ įrankiui, skirto konfigūracijos failų analizei, ir taisyklių rinkinio kitoms išvardintoms problemoms aptikti.

2.3. Išvados

Šioje dalyje, remiantis analizės rezultatais, apibrėžtos grupės, kurioms priklausančias problemas siekiama spręsti, tai yra objektų serializacija, konfigūracijos pažeidžiamumai bei nereikalinga išvestis. Aprašius visus pažeidžiamumus, pateikus jų pavyzdžius, programinio kodo iškarpas bei įvardijus konkrečias siekiamas spręsti problemas, pastebėta jog taisykles galima skirstyti į du taisyklių rinkinius, todėl nuspręsta realizuoti du statinės kodo analizės taisyklių rinkinius – vieną konfigūracijos analizei, kitą programinio kodo problemoms aptikti.

3. STATINĖS ANALIZĖS ĮRANKIO TAISYKLIŲ RINKINIŲ APRAŠYMAS

Skyriuje aprašomi sprendimui keliami reikalavimai – problemų, kurias sukurtos taisyklės turėtų aptikti, sąrašas. Taip pat aprašomas pats sprendimas, jo veikimo principas.

3.1. Reikalavimai sprendimui

Šiame skyriuje pateikiami reikalavimai siekiamam sprendimui. Remiantis analizės dalimi, sprendimas susideda iš dviejų pagrindinių taisyklių rinkinių „PHPStan“ įrankiui:

- objektų serializacijos ir nereikalingos išvesties problemų analizės taisyklių rinkinio;
- „PHP“ konfigūracijos failų analizės taisyklių rinkinio.

3.1.1. Reikalavimai objektų serializacijos ir nereikalingos išvesties analizės taisyklių rinkiniui

„PHPStan“ analizės taisyklių rinkiniui keliami reikalavimai (problema - sprendimas):

- objektų įterpimas – aptikti pažeidžiamumą leidžiančių išnaudoti funkcijų („exec“, „system“, „shell_exec“, „passthru“, „unlink“) naudojimą metoduose „__wakeup“, „__destruct“;
- nefiltruojama serializacija – aptikti funkcijos „unserialize“ naudojimą su parametro „allowed_classes“ reikšme „leisti visas klases“ (parametras nenurodytas);
- nereikalinga išvestis – aptikti pagalbinių išvesties funkcijų („var_dump”, „var_export”, „print_r”, „debug_zval_dump”) naudojimą;
- nereikalinga išvestis – aptikti konfigūracijos parametrų, valdančių klaidų matomumą ir atvaizduojamų klaidų lygius („display_errors”, „error_reporting“, „display_startup_errors“), keitimą programiniame kode proceso vykdymo metu.

3.1.2. Reikalavimai „PHP” konfigūracijos failų analizės taisyklių rinkiniui

Konfigūracijos failų analizės taisyklių rinkiniui keliami reikalavimai aptikti šių konfigūracijos parametrų reikšmes (parametras - reikšmė):

- allow_url_fopen – įjungta;
- allow_url_fopen – nenurodyta („undefined“ / null);
- allow_url_include – įjungta;
- display_errors – įjungta;
- display_errors - nenurodyta („undefined“ / null);
- error_reporting – E_ALL;
- open_basedir - nenurodyta („undefined“ / null).

3.2. Realizacijos aprašymas

Analizės taisyklių rinkiniams parinkti paskirtį atspindintys pavadinimai:

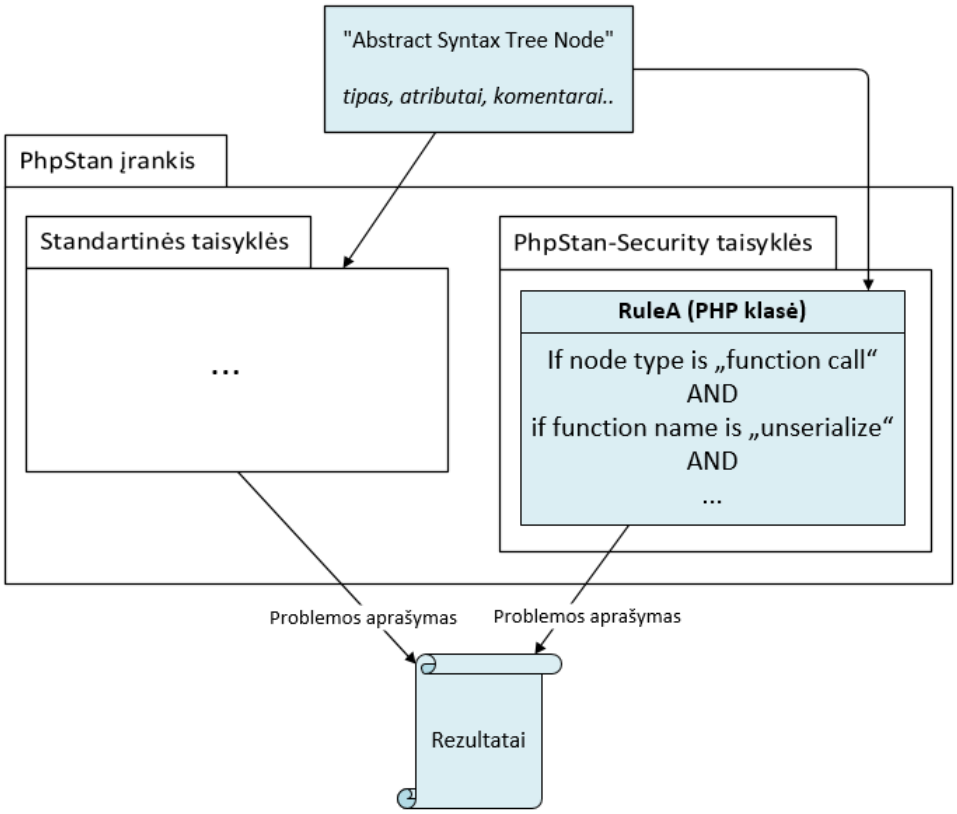
- objektų serializacijos ir nereikalingos išvesties taisyklių rinkiniui – „phpstan-security“;
- „PHP“ konfigūracijos failų analizės taisyklių rinkiniui - „phpstan-config-security“.

3.2.1. Analizės taisyklių rinkinys „phpstan-security“

Taisyklių rinkinys kuriamas kaip įskiepis, saugomas šiam įskiepiui skirtoje kodo saugykloje (angl. „repository“), o į „PHPStan“ įrankį integruojamas redaguojant „PHPStan“ konfigūracijos failą ir naudojant „include“ komandą – nurodant kelią iki „phpstan-security“ taisyklių rinkinio konfigūracijos failo.

Šiame įrankyje analizės taisyklė yra „PHP“ kalbos klasė, kurios priimami duomenys yra abstraktaus sintaksės medžio („AST“) mazgas (angl. „node“) – mazgas „PHP“ kalba yra analizuojamas remiantis mazgo tipo, atributų ir kita informacija, siekiant aptikti pažeidžiamumus ir problemas.

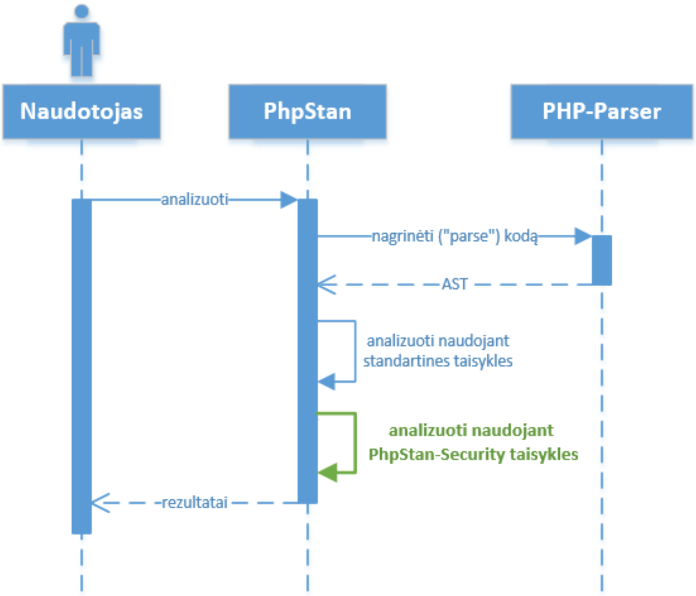
Principinė „phpstan-security“ taisyklių rinkinio veikimo schema vaizduojama diagramoje:



3.1 pav. „phpstan-security“ veikimo principas

„AST“ mazgas perduodamas į standartines „PhpStan“ įrankio analizės taisykles ir į „phpstan-security“ analizės taisykles – visų šių taisyklių išvestis, problemų aprašymai yra sujungiami ir atvaizduojami kaip galutinis analizės rezultatas.

Pateikiama ir principinė vykdymo proceso, sekos diagrama:



3.2 pav. „phpstan-security“ veikimo sekos diagrama

Pirmiausiai programinis kodas perduodamas „PHP-Parser“ bibliotekai, kuri analizuoja kodą ir grąžina „AST“ informaciją „PhpStan“ įrankiui. Remiantis „AST“ informacija pritaikomos standartinės analizės taisyklės, tada „phpstan-security“ bei kitų papildomų taisyklių rinkinių taisyklės. Rinkinys integruojamas į „PHPStan“ įrankį, todėl vykdymas ir rezultatų pateikimas taip pat yra standartinis – problemų pranešimai pateikiami grupuojant juos pagal failus. Analizės vykdymo pavyzdys:

```
[redacted]$ ./vendor/bin/phpstan analyse src/Vulnerability -c phpstan.neon --level max
10/10 [████████████████████████████] 100%

 ------ -------------------------------------------------------------------
  Line   DisplayErrorsDirectiveRuntimeConfiguration/exploit.php
 ------ -------------------------------------------------------------------
  5      Runtime configuration to enable 'display_errors' directive - possibly unintended output.
 ------ -------------------------------------------------------------------

 ------ -------------------------------------------------------------------
  Line   DisplayStartupErrorsDirectiveRuntimeConfiguration/exploit.php
 ------ -------------------------------------------------------------------
  5      Runtime configuration to enable 'display_startup_errors' directive - possibly unintended output.
 ------ -------------------------------------------------------------------

 ------ -------------------------------------------------------------------
  Line   ErrorReportingUsingConfigDirective/exploit.php
 ------ -------------------------------------------------------------------
  5      Config directive 'error_reporting' is set to 'E_ALL' - possibly unintended output.
  6      Config directive 'error_reporting' is set to 'E_ALL' - possibly unintended output.
  7      Config directive 'error_reporting' is set to 'E_ALL' - possibly unintended output.
 ------ -------------------------------------------------------------------

 ------ -------------------------------------------------------------------
  Line   EvalObjectInjectionInWakeup/VulnerableClass.php
 ------ -------------------------------------------------------------------
  15     Eval with class property 'command' as parameter used inside method '__wakeup' - vulnerable to object injection
 ------ -------------------------------------------------------------------

 ------ -------------------------------------------------------------------
  Line   UnintendedOutputFunctions/exploit.php
 ------ -------------------------------------------------------------------
  5      Variable output function 'var_dump' used - possibly unintended output.
  6      Variable output function 'var_export' used - possibly unintended output.
  7      Variable output function 'print_r' used - possibly unintended output.
  8      Variable output function 'debug_zval_dump' used - possibly unintended output.
 ------ -------------------------------------------------------------------

 ------ -------------------------------------------------------------------
  Line   UnserializeAllowedClasses/exploit.php
 ------ -------------------------------------------------------------------
  13     allowed_classes option undefined for unserialize() - no options provided.
 ------ -------------------------------------------------------------------

 ------ -------------------------------------------------------------------
  Line   VulnerableFunctionsObjectInjectionInWakeup/VulnerableClass.php
 ------ -------------------------------------------------------------------
  15     Function 'exec' with class property 'command' as parameter used inside method '__wakeup' - vulnerable to object injection
  16     Function 'system' with class property 'command' as parameter used inside method '__wakeup' - vulnerable to object injection
  17     Function 'shell_exec' with class property 'command' as parameter used inside method '__wakeup' - vulnerable to object injection
 ------ -------------------------------------------------------------------

 [ERROR] Found 14 errors
```

3.3 pav. „phpstan-security“ taisyklių rinkinio vykdymo pavyzdys

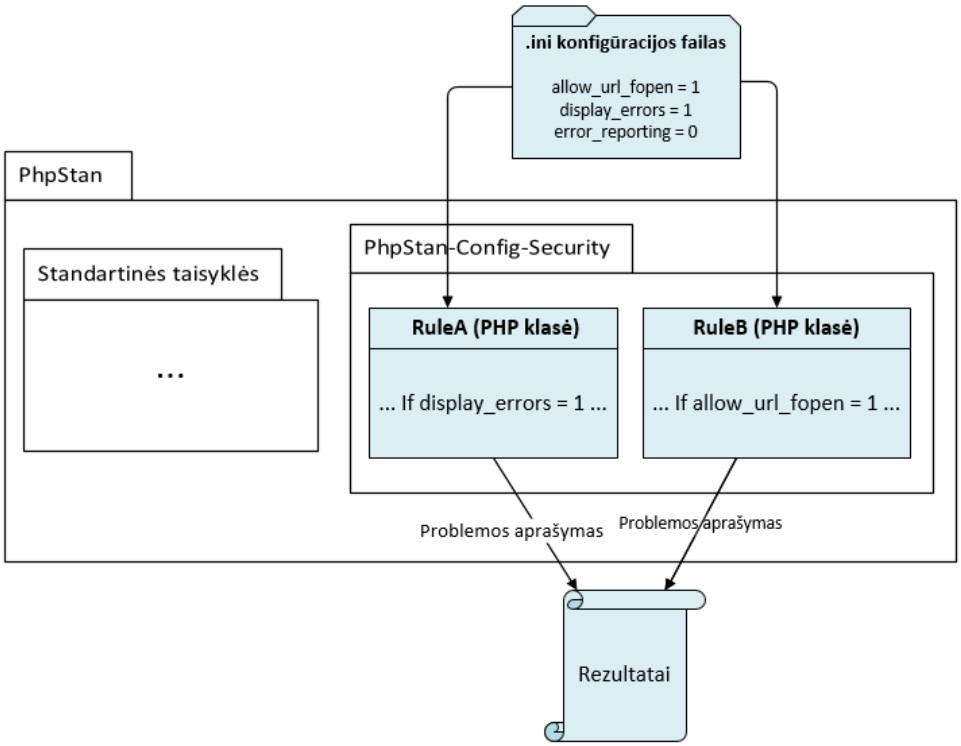
Analizės rezultatai grupuojami pagal analizuojamų failų pavadinimus, nurodoma kurioje failo eilutėje aptikta problema ir taip pat pateikiamas problemos aprašymas, klaidos žinutė.

3.2.2. Konfigūracijos failų analizės taisyklių rinkinys „phpstan-config-security“

Konfigūracijos failų analizės taisyklių rinkinys realizuotas panašiu principu kaip „phpstan-security“ rinkinys. Rinkinys „phpstan-config-security“ į „PHPStan“ įrankį taip pat integruojamas redaguojant „PHPStan“ konfigūracijos failą ir naudojant „include“ komandą – nurodant kelią iki „phpstan-config-security“ taisyklių rinkinio konfigūracijos failo.

Analizės taisyklė taip pat yra „PHP“ kalbos klasė, šiai klasei paduodami konfigūracijos duomenys – „rakto – reikšmės“ principu sugrupuotos konfigūracijos faile nurodytos reikšmės.

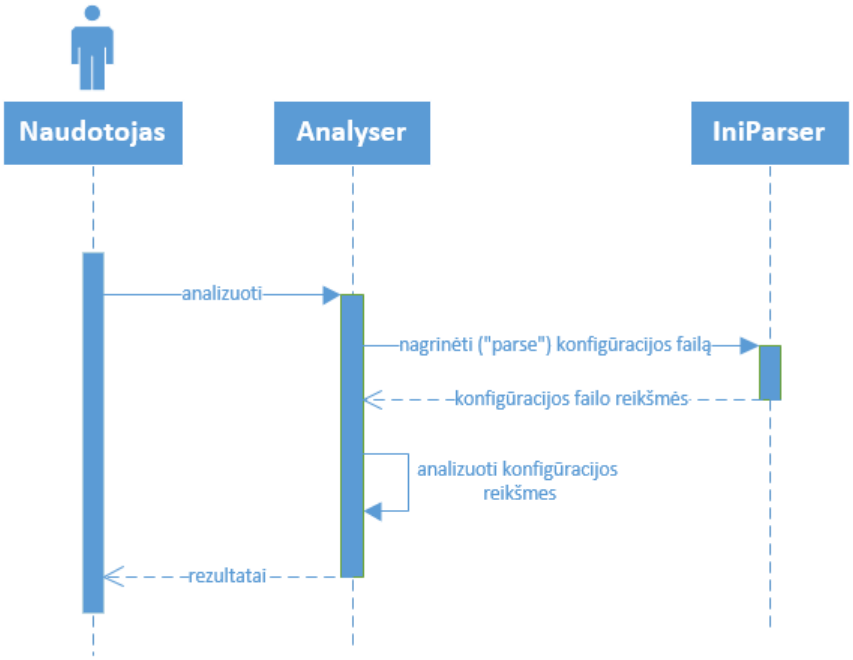
Pateikiama principinė „phpstan-config-security“ taisyklių rinkinio veikimo diagrama:



3.4 pav. „phpstan-config-security“ veikimo principas

Ankstesniuose darbo skyriuose apibrėžta, jog „PHP“ kalbos konfigūracinis „ini“ failas yra rakto – reikšmės (angl. „key - value“) formatu aprašomas failas. Šiame rinkinyje „ini“ failo reikšmės yra nuskaitomos ir konvertuojamos į masyvą, kuris perduodamas analizės taisyklėms. Taisyklė yra „PHP“ klasė, kurioje aprašoma nustatymų ir jų reikšmių analizės logika. Šių taisyklių išvestis – aptiktų problemų ir jų aprašymų rinkinys, yra galutinis analizės rezultatas.

Pateikiama apibendrinta „phpstan-config-security“ veikimo proceso, sekos diagrama:



3.5 pav.– „phpstan-config-security“ veikimo sekos diagrama

Inicijavus analizę konfigūracijos failo reikšmės yra nagrinėjamos ir konvertuojamos į reikšmių masyvą rakto – reikšmės formatu. Šios reikšmės yra analizuojamos įrankio analizės taisyklėmis, kurių logika aprašoma „PHP“ klasėmis.

Kadangi veikiančioje sistemoje prieiga prie konfigūracijos failų turi būti apribota, taip pat analizės įrankių diegti tikrajame sistemos serveryje nėra patartina, norint vykdyti analizę, reikia nurodyti kelią iki konfigūracijos failo.

Analizės vykdymo ir rezultatų pateikimo pavyzdys:

```
>.\vendor\bin\phpstan analyse src\Vulnerability -c phpstan.neon --memory-limit=2G
17/17 [============================] 100%

------ ------------------------------------------------------------------------------------------------------------------------
 Line   php.ini
------ ------------------------------------------------------------------------------------------------------------------------
       Undefined setting 'open_basedir': Undefined open_basedir value results in default value - allow all files to be opened.
       allow_url_fopen (1): Allowing fopen wrappers to access URL objects like files may lead to code or command injections.
       allow_url_include (1): Including file contents from remote resources leads to Remote File Inclusion vulnerability.
       display_errors (1): Displaying error information exposes implementation details.
       error_reporting (E_ALL): Error reporting level E_ALL should not be used in production.
------ ------------------------------------------------------------------------------------------------------------------------

[ERROR] Found 5 errors
```

3.6 pav.- „phpstan-config-security“ analizės vykdymo pavyzdys

Rezultatai pateikiami sąrašu, kurį sudaro laisva forma sudaromos žinutės, tačiau numatytasis žinutės formatas yra pateikti konfigūracijos direktyvos pavadinimą, esamą reikšmę (jei tokia yra) ir trumpą problemos aprašymą.

3.2.3. „PhpStan-Security“ taisyklėms skirti programinio kodo pavyzdžiai

Šiame skyriuje pateikiami programinio kodo pavyzdžiai, kuriais remiantis buvo kuriamos analizės taisyklės. Esminis pavyzdžių kriterijus yra paprastumas – minimalus programinio kodo kiekis, iliustruojantis pažeidžiamumą ir perteikiantis pažeidžiamumo išnaudojimo būdus ir pasekmes. Pateikiami trumpi taisyklių veikimo paaiškinimai – ko ieškoma, ką sudėtingiausia aptikti ir panaši informacija.

3.2.3.1. Objekto įterpimas „destruct“ metode

Šio pažeidžiamumo esmė yra klasės kintamųjų reikšmių naudojimas objekto sunaikinimo metu. Kadangi metodas „destruct“ yra visada įvykdomas kai objektas sunaikinamas, jei jis naudoja klasės kintamuosius, o objektas yra serializuojamas, yra galimybė pakeisti šio objekto klasės kintamųjų reikšmes ir taip įvykdyti norimą komandą. Daugiausiai problemų gali sukelti funkcijų „exec“, „system“, „shell_exec“, „passthru“ naudojimas, nes jos skirtos sisteminėms komandoms vykdyti. Taip pat svarbūs ir „unlink“ (failų trynimo funkcija) naudojimo atvejai, nes išnaudojus pažeidžiamumą yra galimybė ištrinti failus kurie neturėtų būti pasiekiami. Pažeidžiamo kodo pavyzdys, funkcijos „exec“ naudojimas metode „destruct“:

```php
class VulnerableExec
{
    protected $cacheDirectory = '/tmp/random';

    // Some code that uses cache files

    function __destruct()
    {
        exec( command: "rm -r {$this->cacheDirectory}");
    }
}
```

3.7 pav. Pažeidžiamas kodas - objekto įterpimas „destruct“ metode

Pateikto kodo pirminis tikslas yra objekto sunaikinimo metu išvalyti tam tikrą podėlio direktoriją, tačiau serializacijos metu pakeitus klasės kintamojo „$cacheDirectory” reikšmę, galima ištrinti pasirinktą direktoriją.

Tokią situaciją lemiantis kodas:

```php
use MBD\Vulnerability\ObjectInjectionVulnerableFunctionsInDestruct\VulnerableExec;

// "O:77:\"MBD\Vulnerability\ObjectInjectionVulnerableFunctionsInDestruct\VulnerableExec\":1:{s:17:\"\0*\0cacheDirectory\";s:11:\"/tmp/random\";}";}"
$serialized = serialize(new VulnerableExec());

// cache directory changed from /tmp/random to /var/www/important/files
$serialized = "O:77:\"MBD\Vulnerability\ObjectInjectionVulnerableFunctionsInDestruct\VulnerableExec\":1:{s:17:\"\0*\0cacheDirectory\";s:24:\"/var/www/important/files\";}";

$unserializedObject = unserialize($serialized);

// "rm -r /var/www/important/files" executed when $unserializedObject is destroyed
```

3.8 pav. Pažeidžiamumo išnaudojimas - objekto įterpimas „destruct“ metode

Pateiktame kode simuliuojama situacija, kuomet serializuota objekto reikšmė tam tikri būdu yra pakeičiama taip, kad klasės kintamojo „cacheDirectory“ reikšmė būtų pakeista į norimą. Šio kodo pasekmė yra įvykdyta komanda „*rm –r /var/www/important/files*“, ištrinanti failus nurodytoje direktorijoje. Tokiu pat būdu gali būti pažeidžiamas ir kodas, naudojantis funkcijas „system“, „shell_exec“, „passthru“, „unlink“. Funkcijos „unlink“ atveju skirtumas yra tai, jog šios funkcijos tiesioginė paskirtis yra trinti failus, todėl jei perduodamo parametro reikšmė yra kelias iki direktorijos, failo, o ne visa trynimo komanda.

Remiantis šiuo pavyzdžiu, sukurta taisyklė, aptinkanti dvi pažeidžiamumą sukeliančias kodo konstrukcijas:

- kuomet klasės kintamasis yra tiesioginis funkcijos parametras (pavyzdžiui „*exec($this->command)*“);
- kuomet klasės kintamasis yra sudėtinė tekstinės reikšmės dalis (pavyzdžiui „*exec(„rm –r {$this->cachedirectory}“)*“).

Taigi, šio pažeidžiamumo aptikimo sudėtingumas yra tai, jog klasės kintamojo reikšmė į funkciją gali būti perduodama netiesiogiai, įterpiant ją į tekstinę reikšmę keliais būdais.

3.2.3.2. Objekto įterpimas „wakeup“ metode

Pažeidžiamumo esmė labai panaši į objekto įterpimo „destruct“ metode, pasireiškia naudojant funkcijas „exec“, „system“, „shell_exec“, „passthru“. Pagrindinis skirtumas yra tai, jog serializacijos metu pakeista klasės kintamojo reikšmė yra naudojama kai serializuotą reikšmę bandoma konvertuoti atgal į objektą naudojant funkciją „unserialize“. Pažeidžiamo kodo pavyzdys:

```php
class VulnerableExec
{
    protected $command = 'pwd';
    protected $directory;

    function __wakeup()
    {
        $this->directory = exec($this->command);
    }
}
```

3.9 pav. Pažeidžiamas kodas - objekto įterpimas „wakeup“ metode

Šios klasės numatytasis veikimas yra „unserialize“ funkcijos vykdymo metu nuskaityti esamą direktoriją naudojant komandą „pwd“. Tačiau dėl klasės kintamojo naudojimo, yra galimybė serializuotą reikšmę pakeisti taip, kad kintamasis „$command“ įgautų norimą reikšmę. Tokios situacijos programinis kodas:

```
use MBD\Vulnerability\ObjectInjectionVulnerableFunctionsInWakeup\VulnerableExec;

// "O:75:\"MBD\Vulnerability\ObjectInjectionVulnerableFunctionsInWakeup\VulnerableExec\":2:{s:10:\"\0*\0command\";s:3:\"pwd\";s:12:\"\0*\0directory\";N;}"
$serialized = serialize(new VulnerableExec());

// value gets changed
$serialized = "O:75:\"MBD\Vulnerability\ObjectInjectionVulnerableFunctionsInWakeup\VulnerableExec\":2:{s:10:\"\0*\0command\";s:16:\"rm important.pdf\";s:12:\"\0*\0directory\";N;}";

// "rm important.pdf" executed
$unserializedObject = unserialize($serialized);
```

3.10 pav. Pažeidžiamumo išnaudojimas - objekto įterpimas „wakeup“ metode

Pažeidžiamumas išnaudojamas klasės kintamojo „command” reikšmę iš „pwd” pakeičiant į „*rm important.pdf*“ – komanda, ištrinanti failą „*important.pdf*“. Kai serializuota reikšmė perduodama funkcijai „unserialize“, ši komanda yra iškart įvykdoma.

Remiantis šiuo pavyzdžiu, sukurta taisyklė, aptinkanti dvi pažeidžiamumą sukeliančias kodo konstrukcijas:

- kuomet klasės kintamasis yra tiesioginis funkcijos parametras (pavyzdžiui „*exec($this->command)*“);
- kuomet klasės kintamasis yra sudėtinė tekstinės reikšmės dalis (pavyzdžiui „*exec(„rm –r {$this->cachedirectory}“)*“).

Šio pažeidžiamumo aptikimo sudėtingumas yra tai, jog klasės kintamojo reikšmė į funkciją gali būti perduodama netiesiogiai, įterpiant ją į tekstinę reikšmę keliais būdais.

3.2.3.3. Konfigūracijos parametrų keitimas vykdymo metu

Konfigūracijos parametrų keitimas vykdymo metu leidžia pakeisti reikšmes „lokaliai“, tai yra tam procesui, kurio metu yra vykdomas reikšmes keičiantis kodas, taip perrašant numatytąsias, konfigūracijos faile nurodytas reikšmes. Tai atliekama naudojant funkciją “ini_set”, jai perduodant konfigūracijos parametro pavadinimą ir reikšmę. Dažnas šio funkcionalumo panaudojimo atvejis yra leidžiamos atminties kiekio padidinimas kodui, kuris reikalauja daug resursų – skaičiavimai, konvertavimai ir panašiai. Tačiau sistemos kūrimo ir priežiūros etapuose nereta situacija yra kuomet keičiamos klaidų išvedimą ir atvaizdavimą reguliuojantys parametrai („display_errors”, „error_reporting“, „display_startup_errors“). Tai leidžia greitai peržiūrėti vykdymo metu įvykusias klaidas ir lokalioje, darbo aplinkoje nesukelia jokių pažeidžiamumų, tačiau palikus klaidų atvaizdavimą įjungtą, kyla rizika tikrojoje sistemoje naudotojams atskleisti sistemos failų sistemos struktūrą ir panašias detales.

Pavyzdžiui, esant klasei, kurios metodas priima skaitinės reikšmės parametrą „year”:

```php
class Car
{
    private $year;

    public function __construct(int $year)
    {
        $this->year = $year;
    }

    public function getYear(): int
    {
        return $this->year;
    }
}
```

3.11 pav. Klasės programinio kodo pavyzdys

Perdavus tekstinę reikšmę:

```php
ini_set( varname: 'display_errors',  newvalue: '1');

$car = new Car( year: '999');
```

3.12 pav. Klaidos atvaizdavimą lemiantis programinis kodas

Kadangi parametro „display_errors“ reikšmė vykdymo metu pakeičiama į „1”, atvaizduojama klaida:

```
Fatal error: Uncaught TypeError: Argument 1 passed to MBD\Vulnerability\RuntimeConfigurationDisplayErrorsDirective\Car::__construct() must be of the type integer
, string given, called in C:\Users\MV\PhpstormProjects\mbd-phpstan-security\src\Vulnerability\RuntimeConfigurationDisplayErrorsDirective\example.php on line 11 a
```

3.13 pav. Atvaizduojamos klaidos pavyzdys

Šios klaidos išvedimas atskleidžia direktorijų struktūrą. Tai nėra saugumo pažeidžiamumas, tačiau naudojantis šia informacija galima bandyti gauti prieigą prie failų, pavyzdžiui prie naudotojų įkeltų failų, sistemos sugeneruotų failų. Parametrų keitimui vykdymo metu aptikti sukurtos analizės taisyklės, aptinkančios:

- parametro „display_errors“ reikšmės nustatymą į „1” (įjungta);
- parametro „display_startup_errors“ reikšmės nustatymą į „1” (įjungta);
- parametro „error_reporting“ reikšmės nustatymą į „32767“ (atvaizduoti visas klaidas). Ši reikšmė saugoma konstantoje „e_all“, todėl taisyklė pritaikyta aptikti keletą reikšmės perdavimo būdų – kai naudojama konstanta „e_all“, kai naudojama tekstinė reikšmė „32767“, ir kai skaitinė reikšmė konvertuojama į tekstinę „(string) 32767“.

3.2.3.4. Nereikalinga išvestis

„PHP“ kalbos funkcijų, skirtų reikšmių atvaizdavimui, naudojimas programinio kodo kūrimo metu yra įprasta praktika, leidžianti kintamųjų reikšmes peržiūrėti ekrane, nenaudojant derinimo (angl. „debug“) įrankių – šiomis funkcijomis reikšmės spausdinamos ekrane, o vėliau funkcijų kvietimas pašalinamas. Tačiau paprasta klaida – nepašalintas šių funkcijų naudojimas gali lemti įvairios privačios informacijos atskleidimą – priklausomai nuo kintamojo, tai gali būti bet kokia sistemoje esanti informacija.

Tarp šių funkcijų yra „var_dump“, „debug_zval_dump“. Taip pat „var_export“, „print_r“, tačiau šių funkcijų ypatybė yra antrasis parametras, kuris suteikia galimybę reikšmę ne spausdinti ekrane, o grąžinti.

Šių funkcijų naudojimo pavyzdžiai:

```
var_dump(['hello world']);
debug_zval_dump(['hello world']);
```

3.14 pav. Išvesties funkcijų naudojimas

```
var_export(['hello world'], return: false);
print_r(['hello world'], return: false);
var_export(['hello world']);
print_r(['hello world']);
```

3.15 pav. Išvesties funkcijų su reikšmės grąžinimo galimybe naudojimas

Konkretus pavyzdys, spausdinant ankstesniuose skyriuose naudotos klasės objektą naudojant funkciją „var_dump“:

```
object(MBD\Vulnerability\ObjectInjectionVulnerableFunctionsInDestruct\VulnerableExec)#2 (1) {
  ["cacheDirectory":protected]=>
  string(11) "/tmp/random"
}
```

3.16 pav. „var_dump“ funkcijos išvesties pavyzdys

Suprantama, spausdinama reikšmė gali būti kur kas didesnis objektas su daugiau reikšmių, galinčių atskleisti svarbesnę informaciją nei podėlio direktorija, tai gali būti ir duomenų bazės prisijungimo valdymui skirtas objektas ar kiti jautrią informaciją saugantys kintamieji.

Pavyzdžiuose pateiktų funkcijų naudojimui aptikti sukurtos analizės taisyklės, aptinkančios:

- funkcijų „var_dump“, „debug_zval_dump“ naudojimą;
- funkcijų „var_export“, „print_r“ naudojimą su nenurodyta antrojo parametro reikšme (numatytoji reikšmė – spausdinti ekrane) arba reikšme „false“ – spausdinti ekrane.

3.2.3.5. Nefiltruojama serializacija

Serializuotų tekstinių reikšmių konvertavimui į originalias reikšmes, objektus naudojama funkcija „unserialize“. Šią funkciją galima naudoti su įvairiomis serializuotomis tekstinėmis, skaitinėmis reikšmėmis, objektais. Įprastu atveju, funkcijai „unserialize“ perdavus serializuotą objekto reikšmę, grąžinamo objekto klasė nustatoma pagal serializuotoje reikšmėje nurodytą klasę. Tačiau iš to kyla problema, jog esant galimybei modifikuoti serializuotą reikšmę, galima inicijuoti bet kokios sistemoje esančios klasės objektą, tai gali būti ir klasė turinti serializacijos pažeidžiamumų (pavyzdžiui, objektų įterpimo).

Taigi, šios funkcijos naudojimas su nekontroliuojamais duomenimis nesukelia tiesioginio pažeidžiamumo, tačiau suteikia galimybę išnaudoti kitus sistemoje esančius pažeidžiamumus. Tam, kad nebūtų galima inicijuoti bet kokios klasės objekto, funkcijoje „unserialize“ pridėtas parametras „allowed_classes“, kurio paskirtis yra apriboti leidžiamų inicijuoti klasių sąrašą.

Programinio kodo su nefiltruojama serializacija, pavyzdys:

```php
// Original value
$serialized = serialize([]);

// Modified by someone..
$serialized = 'O:59:"MBD\Vulnerability\UnserializeAllowedClasses\VulnerableClass":0:{}';

$result = unserialize($serialized);

// Dangerous code in VulnerableClass wakeup and destruct methods was executed.
```

3.17 pav. Pažeidžiamas kodas – nefiltruojama serializacija

Nors originali serializuota reikšmė yra tuščias masyvas, tačiau šią reikšmę tam tikru būdu (pavyzdžiui, jei reikšmė perduodama tinklu, ar ją įmanoma įvesti sistemos formose ir pan.) modifikavus ir pakeitus į serializuoto objekto reikšmę, jis yra sėkmingai inicijuojamas, taigi vykdomas šios klasės „wakeup“ metodas, o objekto sunaikinimo metu ir „destruct“ metodas. Kaip apibrėžta ankstesniuose skyriuose, tai leidžia išnaudoti objektų įterpimo pažeidžiamumus.

Šiai problemai spręsti sukurtos analizės taisyklės, aptinkančios:

- funkcijos „unserialize“ naudojimą be parametrų;
- funkcijos „unserialize“ naudojimą su kitais parametrais, tačiau be „allowed_classes“ parametro.

3.2.4. „PhpStan-Config-Security“ taisyklėms skirti konfigūracijos failo pavyzdžiai

Šiame skyriuje pateikiami konfigūracijos pavyzdžiai, kuriais remiantis buvo kuriamos analizės taisyklės. „PHP“ konfigūracijos failas gali būti sudarytas iš daugybės eilučių ir parametrų, tačiau vardan paprastumo žemiau pateikiami pavieniai nustatymai, kuriuos analizuoja sukurtos taisyklės.

```
[PHP]

display_errors = On
error_reporting = E_ALL
allow_url_fopen = On
allow_url_include = On
```

3.18 pav. „PHP“ konfigūracijos failo pavyzdys

Remiantis šiame pavyzdyje nurodytomis reikšmėmis, taip pat analizuojant tuščią failą, taip imituojant nenurodytų konfigūracijos reikšmių aptikimą, sukurtos konfigūracijos failo analizės taisyklės, aptinkančios:

- „allow_url_fopen“ su reikšme „įjungta“;
- nenurodytas parametras „allow_url_fopen“ (numatytoji reikšmė – „įjungta“);
- „allow_url_include“ su reikšme „įjungta“;
- „display_errors“ su reikšme „įjungta“;
- nenurodytas parametras „display_errors“ (numatytoji reikšmė – „įjungta“);
- „error_reporting“ su reikšme „E_ALL“ (rodyti viską);
- nenurodytas parametras „open_basedir“ (numatytoji reikšmė – leisti atidaryti visus failus).

3.3. Išvados

Šiame skyriuje iškelti reikalavimai kuriamiems taisyklių rinkiniams, sukurti rinkinių pavadinimai – „PhpStan-Security“ ir „PhpStan-Config-Security“. Taip pat aprašytas kuriamų taisyklių rinkinių veikimas, pateiktos principinės veikimo diagramos, sekos diagramos, rezultatų atvaizdavimo pavyzdžiai, pateikti visų taisyklių kūrimui ir testavimui naudoti duomenys ir paaiškinimai kokios problemos juose egzistuoja. Remiantis ankstesniuose ir šiame skyriuje pateikta informacija, realizuotos statinės kodo analizės taisyklės, jų veikimo eksperimentai pateikiami kitame skyriuje.

4. SUKURTŲ STATINĖS ANALIZĖS TAISYKLIŲ BANDYMAI, EKSPERIMENTAI

Šiame skyriuje pateikiama informacija apie atliktus eksperimentus, bandymus, skirtus įvertinti sukurtų statinės analizės taisyklių veikimo efektyvumą, pateikiama informacija apie naudotus duomenis bei eksperimento rezultatus.

4.1. Eksperimentas naudojant taisyklių rinkiniams kurti naudotus testinius duomenis

Naudojant realizacijos aprašymo skyriuje pateiktus programinio kodo ir konfigūracijos failų pavyzdžius sukurti testiniai duomenys automatiniams testams realizuoti, su šiais duomenimis atlikti eksperimentai, siekiant nustatyti ar įrankiai aptinka pateiktas problemas. Eksperimentuose naudoti šio darbo pagrindu sukurti analizės taisyklių rinkiniai „PhpStan-Security“ ir „PhpStan-Config-Security“.

Pirmasis eksperimentas atliktas su programinio kodo problemų pavyzdžiais, naudojant „PhpStan-Security“ analizės taisyklių rinkinį. Pateikiamas analizuotų problemų sąrašas:

A) Objekto įterpimas – “exec” naudojimas metode “destruct” su klasės kintamaisiais;
B) Objekto įterpimas – “passthru” naudojimas metode “destruct” su klasės kintamaisiais;
C) Objekto įterpimas – “shell_exec” naudojimas metode “destruct” su klasės kintamaisiais;
D) Objekto įterpimas – “system” naudojimas metode “destruct” su klasės kintamaisiais;
E) Objekto įterpimas – “system” naudojimas metode “unlink” su klasės kintamaisiais;
F) Objekto įterpimas – “exec” naudojimas metode “construct” su klasės kintamaisiais;
G) Objekto įterpimas – “passthru” naudojimas metode “construct” su klasės kintamaisiais;
H) Objekto įterpimas – “shell_exec” naudojimas metode “construct” su klasės kintamaisiais;
I) Objekto įterpimas – “system” naudojimas metode “construct” su klasės kintamaisiais;
J) Nefiltruojama serializacija – nenurodytas nei vienas parametras;
K) Nefiltruojama serializacija – nenurodytas “allowed_classes” parametras (gali būti nurodyti kiti).

Eksperimento rezultatai - taisyklių rinkinys „PhpStan-Security“ aptiko visas sąraše pateiktas problemas. Tai rodo, jog automatiniuose testuose aprašytus reikalavimus šis taisyklių rinkinys išpildo.

Antrasis eksperimentas – konfigūracijos failų analizės eksperimentas, šio bandymo problemų sąrašas ir rezultatai:

A) „allow_url_fopen“ su reikšme „įjungta“;
B) Nenurodytas parametras „allow_url_fopen“ (numatytoji reikšmė – „įjungta“);
C) „allow_url_include“ su reikšme „įjungta“;
D) „display_errors“ su reikšme „įjungta“;
E) Nenurodytas parametras „display_errors“ (numatytoji reikšmė – „įjungta“);
F) „error_reporting“ su reikšme „E_ALL“ (rodyti viską);.
G) Nenurodytas parametras „open_basedir“ (numatytoji reikšmė – leisti atidaryti visus failus).

Taisyklių rinkinys „PhpStan-Config-Security“ aptiko visas problemas, automatiniai testai sėkmingai įvykdyti. Rezultatai rodo, jog taisyklių rinkinio „PhpStan-Config-Security“ integravimas į esamą analizės įrankį leidžia išplėsti šio įrankio funkcionalumą, taip pridedamas naujas analizės tipas – konfigūracijos failų analizė.

4.2. Eksperimentas analizuojant konfigūracijos failus su numatytosiomis reikšmėmis

Naudojant „PHP“ kodo saugykloje [33] pateikiamus konfigūracijos failus su numatytosiomis konfigūracijos reikšmėmis, atliktas eksperimentas siekiant įvertinti kokias problemas šiuose failuose aptinka „PhpStan-Config-Security“ taisyklių rinkinio taisyklės. Naudoti du failai:

- php.ini-development – darbo aplinkos (angl. „development)“ konfigūracija, skirta naudoti sistemos kūrimo metu;
- php.ini-production – realios sistemos aplinkos (angl. „production“) konfigūracija, skirta naudoti realioje sistemoje, su daugiau apribojimų.

Šių eksperimentų metu aptiktų problemų kiekiui įvertinti eksperimentų aprašymuose pateikiamos lentelės, rodančios kelių rūšių problemas taisyklių rinkinys gali aptikti, ir kiek jų buvo aptikta. Primenamas sąrašas problemų, kurias taisyklių rinkinys gali aptikti:

A) „allow_url_fopen“ su reikšme „įjungta“;
B) Nenurodytas parametras „allow_url_fopen“ (numatytoji reikšmė – „įjungta“);
C) „allow_url_include“ su reikšme „įjungta“;
D) „display_errors“ su reikšme „įjungta“;
E) Nenurodytas parametras „display_errors“ (numatytoji reikšmė – „įjungta“);
F) „error_reporting“ su reikšme „E_ALL“ (rodyti viską);
G) Nenurodytas parametras „open_basedir“ (numatytoji reikšmė – leisti atidaryti visus failus).

4.2.1. „Development“ konfigūracijos failo analizė

Naudojant „PhpStan-Config-Security“ analizės taisyklės išanalizavus darbo aplinkos konfigūracijos failą su numatytosiomis reikšmėmis, gauti rezultatai:

```
>vendor\bin\phpstan analyse src\Vulnerability -c phpstan.neon --memory-limit=2G
 1/1 [============================] 100%

------ ------------------------------------------------------------------------------------------------------------------------
 Line   php.ini-development.ini
------ ------------------------------------------------------------------------------------------------------------------------
 10     Undefined setting 'open_basedir': Undefined open_basedir value results in default value - allow all files to be opened.
 10     allow_url_fopen (1): Allowing fopen wrappers to access URL objects like files may lead to code or command injections.
 10     display_errors (1): Displaying error information exposes implementation details.
 10     error_reporting (E_ALL): Error reporting level E_ALL should not be used in production.
------ ------------------------------------------------------------------------------------------------------------------------
```

4.1 pav. „Development“ konfigūracijos failo analizės rezultatai

Aptiktos keturios problemos, susijusios su nustatymais, nusakančiais direktorijų pasiekiamumą, nutolusių failų atidarymo galimybę, klaidų atvaizdavimą. Šių problemų aprašymai pateikti ankstesniuose skyriuose. Aptiktų problemų tipų sąrašas pateikiamas lentelėje:

4.1 lentelė „Development“ konfigūracijos failo analizės metu aptiktų problemų tipai

Įrankis / Problema	A	B	C	D	E	F	G
PhpStan-Config-Security	✓	×	×	✓	×	✓	✓

Rezultatai rodo, jog konfigūracijos failas turi keturis probleminius nustatymus iš aptinkamų septynių. Tačiau šių probleminių nustatymų buvimą galima paaiškinti tuo, jog sistemos kūrimo metu aktualu žinoti visą informaciją apie klaidas, taip pat leisti atidaryti visus reikiamus failus, nes sistema nėra prieinama naudotojams.

4.2.2. „Production“ konfigūracijos failo analizė

Naudojant „PhpStan-Config-Security“ analizės taisyklės išanalizavus realios vykdymo aplinkos konfigūracijos failą su numatytosiomis reikšmėmis, gauti rezultatai:

```
>vendor\bin\phpstan analyse src\Vulnerability -c phpstan.neon --memory-limit=2G
1/1 [============================] 100%

------ ------------------------------------------------------------------------------------------------------------------------
 Line   php.ini-production.ini
------ ------------------------------------------------------------------------------------------------------------------------
 10     Undefined setting 'open_basedir': Undefined open_basedir value results in default value - allow all files to be opened.
 10     allow_url_fopen (1): Allowing fopen wrappers to access URL objects like files may lead to code or command injections.
------ ------------------------------------------------------------------------------------------------------------------------
```

4.2 pav. „Production“ konfigūracijos failo analizės rezultatai

Aptiktos dvi problemos, susijusios su nustatymais, nusakančiais direktorijų pasiekiamumą ir nutolusių failų atidarymo galimybę. Šių problemų aprašymai pateikti ankstesniuose skyriuose. Aptiktų problemų tipų sąrašas pateikiamas lentelėje:

4.2 lentelė „Production“ konfigūracijos failo analizės metu aptiktų problemų tipai

Įrankis / Problema	A	B	C	D	E	F	G
PhpStan-Config-Security	✓	×	×	×	×	×	✓

Rezultatai rodo, jog konfigūracijos failas turi du probleminius nustatymus iš aptinkamų septynių.

4.3. Eksperimentas analizuojant „Symfony“ programavimo karkasą

„Symfony“ yra karkasas, apjungiantis įvairius „Symfony“ komponentus, skirtus internetinėms sistemos kurti. Komponentai gali būti naudojami kartu kaip „Symfony“ sudedamosios dalys, arba atskirai, kaip pavieniai komponentai kitose sistemose, todėl jų panaudojimas yra labai platus. Dokumentacijoje [34] teigiama, jog „Symfony“ naudojamas daugiau nei šešių šimtų tūkstančių programinės įrangos inžinierių ir per mėnesį yra parsisiunčiamas daugiau nei 48 mln. kartų.

Naudojant šio karkaso programinį kodą atliktas eksperimentas, skirtas palyginti įrankių aptinkamas klaidas ir problemas. Kodas analizuotas šiame darbe sukurtais analizės taisyklių rinkiniais „PhpStan-Security“ ir „PhpStan-Config-Security“, taip pat kitais įrankiais – „PHPSA“ ir „Phan“.

Pirmiausia pateikiamas aptinkamų problemų sąrašas su trumpais aprašymais, kiekvienai problemai priskiriama sąrašo raidė, susiejanti problemą sąraše su stulpeliu lentelėje:

A) Nefiltruojama serializacija – nenurodytas nei vienas parametras;
B) Objekto įterpimas – funkcijos „unlink” naudojimas metode “destruct” su klasės kintamaisiais;
C) „Callable“ parametro reikšmė yra masyvas su vienu, o ne dvejais elementais. „PHP“ kalboje „callable“ ar kitaip „callback“ parametrai gali būti funkcijos arba masyvai su dviem elementais, nurodančiais klasę ir jos metodą;
D) Klasė, pati kviečianti savo „construct“ metodą – konstruktorių;
E) Statiniu būdu kviečiami metodai, kurie nėra apibrėžti kaip statiniai. Tokie metodai turėtų būti kviečiami naudojant inicijuotą objektą („non-static“);
F) Nepasiekiamų („private“) klasės kintamųjų ir konstantų naudojimas. Privatūs klasės kintamieji ir konstantos nėra pasiekiami už klasės ribų, tačiau yra bandoma juos panaudoti kitose kodo vietose.

Eksperimento rezultatai, išvardintų problemų aptikimas naudojant skirtingas taisykles ir įrankius, pateikiami lentelėje:

4.3 lentelė „Symfony“ karkaso analizės eksperimento – palyginimo rezultatai

Įrankis / Problema	A	B	C	D	E	F
PhpStan-Security	✓	✓	×	×	×	×
PhpStan-Config-Security	×	×	×	×	×	×
Phan	×	×	✓	✓	✓	✓
PHPSA	✓	×	×	×	×	×

Pagal lentelėje pateiktus rezultatus galima teigti, kad „Symfony“ programiniame kode daugiausiai problemų aptiko įrankis „Phan“. Tačiau šio įrankio aptinkamos problemos yra bendro pobūdžio, tokios kaip netinkami metodų kvietimai ar bandymas naudoti privačius kintamuosius.

Tuo tarpu su serializacija susijusias problemas aptiko taisyklių rinkinys „PhpStan-Security“ ir įrankis „PHPSA“. Abu įrankiai aptiko nefiltruojamą serializaciją (sąraše A), taip pat „PhpStan-Security“ taisyklės aptiko klasės kintamojo naudojimą metode „destruct“ (sąraše B).

4.3.1. Aptiktas pažeidžiamumas – objekto įterpimas

Eksperimento metu atliekant „Symfony“ karkaso kodo analizę naudojant „PhpStan-Security“ taisyklių rinkinį, viename iš karkaso komponentų aptiktas objekto įterpimo pažeidžiamumas, kuris buvo sėkmingai išnaudotas. Pažeidžiamumui pagrįsti sukurta pavyzdinė programa (angl. „proof of concept“), demonstruojanti pažeidžiamumo, kuris suteikia galimybę ištrinti norimus failus, išnaudojimą. Ši informacija perduota „Symfony“ autoriams per „Intigriti“ platformą, kuri dalyvauja EU-FOSSA vykdomoje pažeidžiamumų paieškos (angl. „bug bounty“) programoje. Pažeidžiamumas patvirtintas ir ištaisytas.

„Intigriti“ [35] – etiško įsilaužimo, pažeidžiamumų pranešimo platforma, suteikianti įvairių karkasų ir sistemų pažeidžiamumų pranešimo funkcionalumą, turi galimybę pateikti pažeidžiamumo aprašymą, programinio kodo pavyzdžius, nuorodas į šaltinius, taip pat įvertinti pažeidžiamumo svarbą ir išnaudojimo sudėtingumą.

EU-FOSSA („EU-Free and Open Source Software Auditing“) yra Europos Parlamento programos, skirtos stebėti ir užtikrinti atviro kodo programinės įrangos kokybę, dalis. Kaip teigiama aprašyme [36], EU-FOSSA vykdo sistemingą veiklą kartu su Europos Sąjungos institucijomis, kad užtikrintų plačiai naudojamos, kritinės programinės įrangos patikimumą.

Objekto įterpimo pažeidžiamumas aptiktas „Symfony/Cache“ komponente. Kad pažeidžiamumas būtų sėkmingai išnaudotas, turi būti tenkinamos šios sąlygos:

- Įdiegtas „Symfony/Cache“ komponentas (gali būti tiesiogiai nenaudojamas, svarbu, kad komponento klasės būtų pasiekiamos naudojimui);
- Bent vienoje sistemos vietoje turi būti panaudota funkcija „unserialize“ be leidžiamų klasių sąrašo (be parametro „allowed_classes“, arba parametro reikšmė „true“);
- Šiai „unserialize“ funkcijai yra galimybė perduoti naudotojo pateiktus duomenis.

Taigi, pasireiškus šioms sąlygoms, išnaudotas objekto įterpimo pažeidžiamumas, kurio pasekmė yra ištrintas pagrindinis sistemos vykdymo failas („index.php“) – sistema tampa nebepasiekiama.

Dėl EU-FOSSA programos techninių detalių viešinimo apribojimų, taip pat dėl to, kad „Symfony/Cache“ komponentas yra naudojamas daugelyje sistemų ir detalios pažeidžiamumo informacijos atskleidimas gali sukelti pavojų jų saugumui, pateikiamas apibendrintas pažeidžiamumo techninis aprašymas.

Vienos komponento klasių metode „destruct“ (šis metodas vykdomas objekto sunaikinimo metu, tai yra įvykdomas visada – arba automatinio sunaikinimo metu proceso pabaigoje, arba naikinant

objektą rankiniu būdu) yra naudojama failų trynimo funkcija „unlink“, kurios parametro, nurodančio kelią iki trinamo failo, reikšmė yra pateikiama iš klasės kintamojo (angl. „property“). Kadangi serializacijos metu klasės kintamųjų reikšmės taip pat paverčiamos į tekstinę reikšmę, pakoregavus šią tekstinę reikšmę galima pakeisti ir klasės kintamąjį. Taigi, pakeitus kintamojo reikšmę ir panaudojus funkciją „unserialize“ kuri tekstinę reikšmę paverčia atgal į objektą, gaunamas objektas, kurio klasės kintamojo reikšmė rodo kelią į atakuotojo nurodytą failą, ši reikšmė objekto sunaikinimo metu yra perduodama į trynimo funkciją „unlink“ ir taip yra ištrinamas atakuotojo pasirinktas failas.

Informaciją apie paveiktas „Symfony“ versijas ir kaip pažeidžiamumas ištaisytas, galima rasti karkaso puslapio naujienų įraše [37].

4.4. Išvados

Naudojant skirtingus duomenis atlikti eksperimentai skirti realizuoto sprendimo efektyvumui pademonstruoti. Pirmiausiai analizės taisyklėms sukurti automatiniai testai, skirti įvertinti ar taisyklės atitinka reikalavimus ir aptinka reikiamas problemas. Atliktas konfigūracijos failų analizės eksperimentas analizuojant konfigūracijos failus su numatytosiomis reikšmėmis, išanalizuoti du failai, skirti „development“ ir „production“ aplinkoms. Taip pat išanalizuotas „Symfony“ programavimo karkaso ir jo komponentų kodas, aprašytos aptiktos problemos.

Vienas iš karkase aptiktų pažeidžiamumų buvo ištirtas ir išnaudotas, sukurta pavyzdinė programa, demonstruojanti pažeidžiamumo išnaudojimą, to pasekmė – ištrintas pagrindinis sistemos vykdomasis failas, sistema tampa neveikianti ir nepasiekiama. Pažeidžiamumo informacija perduota karkaso autoriams per etiško įsilaužimo platformą, dalyvaujančią Europos Parlamento vykdomoje programoje, skirtoje stebėti ir užtikrinti atviro kodo programinės įrangos kokybę. Pažeidžiamumas patvirtintas ir ištaisytas.

Atlikti eksperimentai pagrindžia sukurtų statinės kodo analizės taisyklių veikimą, leidžia teigti, kad tai papildo ir išplečia esamų statinės analizės įrankių funkcionalumą. Taip pat, populiariame programavimo karkase aptiktas ir ištaisytas pažeidžiamumas leidžia teigti, kad realizuotas sprendimas yra efektyvus ir pritaikomas praktinėje aplinkoje.

5. IŠVADOS

1. Išanalizavus programinio kodo analizės metodus, apžvelgus metodų privalumus ir trūkumus nustatyta, kad dinaminės kodo analizės privalumai yra vykdymas tikroje aplinkoje, atkartojant tikrąjį programos veikimą ir tikslumas, o trūkumai - kaina ir diegimo bei konfigūravimo sudėtingumas. Statinės kodo analizės privalumai yra nedidelė kaina, nesudėtingas konfigūravimas, aptinkama konkreti programinio kodo eilutė kurioje egzistuoja problema, taip pat ankstyvas problemų identifikavimas. Statinės analizės trūkumai yra netikslumas ir problemų, pasireiškiančių kodo vykdymo metu, neaptikimas.
2. Išanalizavus statinės kodo analizės įrankių problemas nustatyta, kad pagrindiniai trūkumai šiuo metu yra „false positive“ problema (įrankis nurodo problemą ten, kur jos iš tiesų nėra) ir įrankių netikslumas – problemų neaptikimas. Pasirinkta spręsti įrankių netikslumo problemą kuriant statinės kodo analizės taisykles, leidžiančias aptikti daugiau problemų ir pažeidžiamumų.
3. PHP programavimo kalbos statinės kodo analizės tobulinimui pasirinktos trys klaidų ir problemų grupės - objektų serializacija, konfigūracijos pažeidžiamumai, nereikalinga išvestis.
4. Apžvelgus statinės kodo analizės įrankius, įvertinus jų populiarumą, naudojimo paplitimą, naujų taisyklių integravimo galimybes bei aptinkamų problemų tipus, nuspręsta analizės taisykles realizuoti kaip taisyklių rinkinius, integruojamus į „PHPStan“ įrankį.
5. Sukurti du analizės taisyklių rinkiniai: rinkinys „PhpStan-Security“ skirtas aptikti objektų serializacijos ir nereikalingos išvesties problemas, taip pat rinkinys „PhpStan-Config-Security“, skirtas aptikti konfigūracijos problemas.
6. Naudojant skirtingus duomenis atlikti eksperimentai, skirti realizuoto sprendimo efektyvumui pademonstruoti. Eksperimentai atlikti su šiame darbe aprašytais programinio kodo ir konfigūracijos failų pavyzdžiais, taip pat su tikromis konfigūracijos reikšmėmis ir programavimo karkasais. Remiantis „Symfony“ karkaso analizės metu aptikta informacija, vienas iš karkase aptiktų pažeidžiamumų buvo sėkmingai išnaudotas, pažeidžiamumo informacija perduota karkaso autoriams per etiško įsilaužimo platformą, pažeidžiamumas patvirtintas ir ištaisytas.
7. Atlikti eksperimentai pagrindžia sukurtų statinės kodo analizės taisyklių veikimą, leidžia teigti, kad tai papildo ir išplečia esamų statinės analizės įrankių funkcionalumą. Taip pat, populiariame programavimo karkase aptiktas ir ištaisytas pažeidžiamumas leidžia teigti, kad realizuotas sprendimas yra efektyvus ir pritaikomas praktinėje aplinkoje.

6. LITERATŪRA

[1] RADHIKA D. Venkatasubramanyam, SOWMYA G. R. *Why Is Dynamic Analysis Not Used as Extensively as Static Analysis: An Industrial Study*, 2014. ISBN 978-1-4503-2859-3.

[2] FAIRLEY Richard E. *Static Analysis and Dynamic Testing of Computer Software*, 1978. ISSN 1558-0814.

[3] BARUFI VERAS Gabriel. *Supporting Swarm Debugging in interpreted programming languages*, 2018.

[4] HUANG Ruizhu, XU Weijia, LIVERANI Silvia, HILTBRAND Dave, STAPLETON Ann E. *A Case Study of R Performance Analysis and Optimization*, 2018. ISBN 978-1-4503-6446-1.

[5] GHAHRAI Amir. *Static Analysis vs Dynamic Analysis in Software Testing*. 2018 [Žiūrėta 2019-01-14]. Prieiga per: https://www.testingexcellence.com/static-analysis-vs-dynamic-analysis-software-testing/

[6] KULENOVIC Melina, DONKO Dzenana. *A survey of static code analysis methods for security vulnerabilities detection*, 2014. ISBN 978-953-233-077-9.

[7] KASTNER Daniel, MAUBORGNE Laurent, FERDINAND Christian. *Detecting Safety-and Security-Relevant Programming Defectsby Sound Static Analysis*, 2017. ISBN 978-1-61208-605-7.

[8] PRAUSE Christian R., BIBUS Markus, DIETRICH Carsten, JOBI Wolfgang. *Software product assurance at the German space agency*, 2016.

[9] WICHMANN B.A., CANNING A.A., CLUTTERBUCK D.L., WINSBORROW L.A., WARD N.J., MARSH D.W.R. *Industrial perspective on static analysis*, 1995. ISBN 0268-6961.

[10] LICHTER Horst, RIEDINGER Gerhard. *Improving Software Quality by Static Program Analysis*, 1997.

[11] DÍAZ Gabriel, BERMEJO Juan Ramón. *Static analysis of source code security: Assessment of tools against SAMATE tests*, 2013.

[12] ZHIOUA Zeineb, SHORT Stuart, ROUDIER Yves. *Static Code Analysis for Software Security Verification: Problems and Approaches*, 2014. ISBN 978-1-4799-3578-9.

[13] CHESS B., MCGRAW G. *Static analysis for security*, 2004. ISSN 1558-4046.

[14] MEDEIROS Ibéria, NEVES Nuno, CORREIA Miguel. *Detecting and Removing Web Application Vulnerabilities with Static Analysis and Data Mining*, 2016. ISSN 1558-1721.

[15] REYNOLDS Zachary P., JAYANTH Abhinandan B., KOC Ugur, PORTER Adam A., RAJE Rajeev R., HILL James H. *Identifying and Documenting False Positive Patterns Generated by Static Code Analysis Tools*, 2017. ISBN 978-1-5386-2797-6.

[16] GOSEVA-POPSTOJANOVAA Katerina, PERHINSCHI Andrei. *On the capability of static code analysis to detect security vulnerabilities*, 2015.

[17] „Web application security consortium“. *Static analysis tools evaluation criteria*, 2013.

[18] KNUDSEN Alexander Ramos. *Evaluating the ability of static code analysis tools to detect injection vulnerabilities*, 2016.

[19] KOC Ugur, SAADATPANAH Parsa, FOSTER Jeffrey S., PORTER Adam A. *Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools*, 2017. ISBN 978-1-4503-5071-6.

[20] PHP dokumentacija [Žiūrėta 2019-01-14]. Prieiga per: http://php.net

[21] ESHKEVARI Laleh, DOS SANTOS Fabien, CORDY James R., ANTONIOL Giuliano. *Are PHP Applications Ready for Hack?*, 2015. ISBN 978-1-4799-8469-5.

[22] „w3techs“. *Usage statistics and market share of PHP for websites* [Žiūrėta 2019-03-03]. Prieiga per: https://w3techs.com/technologies/details/pl-php/all/all

[23] DAHSE Johannes. *Static Detection of Complex Vulnerabilities in Modern PHP Applications*, 2016.

[24] SCANNELL Simon. *What is PHP Object Injection*. 2018 [Žiūrėta 2019-02-17]. Prieiga per: https://blog.ripstech.com/2018/php-object-injection

[25] „w3techs“. *Usage of content management systems* [Žiūrėta 2019-02-17]. Prieiga per: https://w3techs.com/technologies/overview/content_management/all

[26] BELLER Moritz, BHOLANATH Radjino, MCINTOSH Shane, ZAIDMAN Andy. *Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software*, 2016. ISBN 978-1-5090-1855-0.

[27] „PhpStan“ dokumentacija [Žiūrėta 2019-04-02]. Prieiga per: https://github.com/phpstan/phpstan

[28] DAHSE Johannes, KREIN Nikolai, HOLZ Thorsten. *Code Reuse Attacks in PHP: Automated POP Chain Generation*, 2014. ISBN 978-1-4503-2957-6.

[29] „OWASP“ specifikacija [Žiūrėta 2019-03-03]. Prieiga per: https://www.owasp.org/index.php/PHP_Object_Injection

[30] „PeerJ Preprints“. *How developers debug*, 2017.

[31] PHP dokumentacija [Žiūrėta 2019-04-02]. Prieiga per: http://php.net/manual/en/configuration.file.php

[32] „OWASP“ specifikacija [Žiūrėta 2019-04-02]. Prieiga per: https://www.owasp.org/index.php/PHP_Configuration_Cheat_Sheet

[33] PHP dokumentacija [Žiūrėta 2019-03-03]. Prieiga per: https://github.com/php/php-src

[34] „Symfony“ tinklalapis [Žiūrėta 2019-04-19]. Prieiga per: https://symfony.com

[35] „Intigriti“ tinklalapis [Žiūrėta 2019-04-19]. Prieiga per: https://www.intigriti.com

[36] EU-FOSSA aprašymas [Žiūrėta 2019-04-19]. Prieiga per: https://joinup.ec.europa.eu/collection/eu-fossa-2/about

[37] „Symfony“ tinklaraštis [Žiūrėta 2019-04-19]. Prieiga per: https://symfony.com/blog/cve-2019-10912-prevent-destructors-with-side-effects-from-being-unserialized

7. PRIEDAI

7.1. Priedas – „AST“ grafinio atvaizdavimo pavyzdys