ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΕΠΕΞΕΡΓΑΣΙΑΣ ΠΛΗΡΟΦΟΡΙΑΣ ΚΑΙ ΥΠΟΛΟΓΙΣΜΩΝ

Εργαλείο στατικής ανάλυσης κώδικα Javascript με στόχο την ασφάλεια εφαρμογών λογισμικού

Διπλωματική Εργασία

Ιωάννης Κ. Γκουζιώκας

ΑΕΜ: 8127

Επιβλέποντες

Ανδρέας Λ. Συμεωνίδης, Αναπληρωτής Καθηγητής Α.Π.Θ

Κυριάκος Χατζηδημητρίου, Μεταδιδακτορικός Ερευνητής Α.Π.Θ

Θεσσαλονίκη, Οκτώβριος 2019

Ευχαριστίες

Η περάτωση της παρούσας διπλωματικής σηματοδοτεί και το τέλος των σπουδών μου στο τμήμα των Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Αριστοτέλειου Πανεπιστήμιου Θεσσαλονίκης.

Αρχικά θα ήθελα να ευχαριστήσω τον κύριο Ανδρέα Συμεωνίδη για την εμπιστοσύνη που μου έδειξε αναθέτοντας τη συγκεκριμένη διπλωματική. Επίσης θα ήθελα να ευχαριστήσω τον μεταδιδακτορικό ερευνητή κύριο Κυριάκο Χατζηδημητρίου για τη καθοδήγηση και τις συμβουλές οι οποίες οδήγησαν στην επιτυχή ολοκλήρωση της εργασίας.

Ολοκληρώνοντας, θα ήθελα να ευχαριστήσω τους γονείς μου και την αδερφή μου για τη συνεχή στήριξη και σε όλη την διάρκεια των σπουδών μου. Τέλος οφείλω ένα μεγάλο ευχαριστώ στους φίλους μου, οι οποίοι μου στάθηκαν όλα αυτά τα χρόνια και μου χάρισαν όμορφες στιγμές.

Περίληψη

Τα τελευταία χρόνια ο αριθμός διαδικτυακών εφαρμογών και υπηρεσιών αυξάνεται εκθετικά. Όπως είναι αναμενόμενο, η ανάπτυξη λογισμικού αυτού του τύπου έχει οδηγήσει σε αύξηση των απαιτήσεων και της πολυπλοκότητας όσον αφορά την ασφάλεια. Οι διαδικτυακές επιθέσεις καθημερινά αυξάνονται στοχεύοντας σε αδυναμίες για τις οποίες οι τεχνολογικοί τρόποι αντιμετώπισης και πρόβλεψης δεν είναι επαρκείς.

Η έρευνα που πραγματοποιείται για την εύρεση μεθόδων και τεχνικών που αντιμετωπίζουν και κυρίως προλαμβάνουν τις διαδικτυακές επιθέσεις είναι συνεχής. Η στατική ανάλυση κώδικα αποτελεί μία διαδεδομένη τεχνική που μπορεί να δώσει χρήσιμες λύσεις όσον αφορά την πρόληψη των επιθέσεων αλλά και την ποιοτικότερη εποπτεία του κώδικα της εφαρμογής. Οι linters, είναι ένα εργαλείο στατικής ανάλυσης κώδικα και θεωρούνται απαραίτητοι σε γλώσσες προγραμματισμού όπως η JavaScript, που δεν περιλαμβάνουν τη διαδικασία της συμβολομετάφρασης. Πολλοί ερευνητές ασχολούνται τα τελευταία χρόνια στόχο τη βελτίωση της ασφάλειας λογισμικού γραμμένου σε JavaScript με τη χρήση των εργαλείων linting. Ο δημοφιλέστερος linter για κώδικα JavaScript θεωρείται ο ESLint. Αποτελεί ένα εργαλείο ανοιχτού λογισμικού, το οποίο είναι πλήρως διαμορφώσιμο και δίνει τη δυνατότητα χρήσης του σε διαφορετικά πεδία σχετικά με την ποιότητα κώδικα.

Στόχος αυτής της διπλωματικής είναι η κατασκευή ενός συνόλου κανόνων που βασίζεται σε αδυναμίες πάνω στις διαδικτυακές εφαρμογές σε JavaScript και κυρίως στο περιβάλλον Node.js, το οποίο με στατική ανάλυση μέσω του linter ESLint έχει ως στόχο την ανίχνευση πιθανών αδυναμιών του κώδικα όσον αφορά την ασφάλεια. Η υλοποίηση των κανόνων πραγματοποιήθηκε στην πλατφόρμας AST Explorer για την ανάλυση και τη διάσχιση του πηγαίου κώδικα JavaScript μέσω συντακτικών δέντρων. Ο έλεγχος της εγκυρότητας των κανόνων έγινε μέσω του εργαλείου Rule Tester που παρέχει ο ESLint. Τέλος, το σύνολο των κανόνων ομαδοποιήθηκε σε ένα npm πακέτο για κοινή χρήση, ενώ ταυτόχρονα ελέγχθηκε η λειτουργία του σε δημοφιλή αποθετήρια της JavaScript από το GitHub.

Ιωάννης Κ. Γκουζιώκας

Ioannis.gkouziokas@gmail.com

Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

Οκτώβριος 2019

ARISTOTLE UNIVERSITY OF THESSALONIKI

FACULTY OF ENGINEERING

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

DEPARTMENT OF COMPUTERS & ELECTRONICS

DIPLOMA THESIS

JAVASCRIPT STATIC CODE ANALYSIS TOOL FOR SECURITY IN SOFTWARE APPLICATIONS

Supervisor:
Associate Professor Andreas L. Symeonidis

Ioannis K. Gkouziokas (AEM 8127)

Co-supervisor:
Postdoc Researcher
Kyriakos Chatzidimitriou

Abstract

In recent years the number of web applications and services has been exponentially increased. As might be expected, the development of this type of software has led to the rise of security requirements and complexity. Web attacks are increasing day by day, aiming to exploit the apps' vulnerabilities, for which the means of forecast and defense are not (yet) enough.

There is an ongoing research for the development of new methods and techniques in order to deal and mainly prevent the web attacks. Static code analysis is a widespread technique that can provide useful solutions in terms of attack prevention as well as better supervision of applications code. Linters are a static code analysis tool and are considered essential in programming languages, such as Javascript, which lack the part of interpretation. Many researches, in the last years, are working on the development of linting tools for the improvement of software security in JavaScript by using linting tools. The most famous linter, for a JavaScript code, is ESLint. It is an open source software tool, that is fully configurable and can be used in different areas of code quality.

This thesis aims to collect and build a set of rules based on the vulnerabilities and attacks on web applications, written in Javascript and more specifically in the Node.js environment. This set of rules, using static analysis via ESLint, aims to locate and prevent possible security vulnerabilities of the code. For the implementation of this project, the AST Explorer platform was used for the analysis and the traverse of the JavaScript source code, through abstract syntax trees, while the validity check of the rules was done via Rule Tester tool that comes with ESLint. Finally, the set of rules was grouped into a npm package for common/mutual use and its functionality was tested in famous JavaScript repositories provided by GitHub.

email: ioannis.gkouziokas@gmail.com

Περιεχόμενα

Λίστα Εικόνων

Λίστα Πινάκων

Λίστα Διαγραμμάτων

Λεξιλόγιο Όρων

- **Npm**: Node package manager
- **AST**: Abstract Syntax Tree
- **VM**: Virtual Machine
- **SAST**: Static Application Security Testing
- **ROI** : Return of Investment
- **OS**: Operating System
- **XSS**: Cross Site Scripting
- **DoS**: Denial of Service
- **CSRF**: Cross Site Request Forgery
- **CSP**: Content Security Policy
- **SID**: Session Identifier
- **MITM**: Man, In The Middle
- **ReDoS**: Regular expression Denial of Service
- **crlf**: carriage return line feed
- **FP**: False Positive
- **FN**: False Negative
- **I/O**: Input/Output
- **API:** Application Program Interface
- **SAT:** Static Analysis Tool

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

Στις 6 Αυγούστου του 1991, ο Tim Berners-Lee λάνσαρε την πρώτη ιστοσελίδα (info.sern.cz) [1], η οποία ήταν μία απλή στατική σελίδα χωρίς καθόλου αλληλεπίδραση με τον χρήστη και δυναμικά δεδομένα. Έτσι δεν υπήρχε σχεδόν τίποτα που να μας απασχολεί από την πλευρά της ασφάλειας και αυτό γιατί 'χακάρωντας' τέτοιου είδους ιστοσελίδες μπορούσες να αποκτήσεις πρόσβαση στα στατικά δεδομένα, διαγράφοντας κάποιες σελίδες ή αλλάζοντας το περιεχόμενο τους.

Από τότε έχουν περάσει σχεδόν τριάντα χρόνια και έχουν αλλάξει πάρα πολλά στο χώρο των διαδικτυακών εφαρμογών. Πλέον οι εφαρμογές χαρακτηρίζονται από πολύπλοκα συστήματα και διεπαφές, αποθηκευμένα δεδομένα και υψηλού επιπέδου λειτουργικότητα. Από τις εφαρμογές με ελάχιστη πληροφορία, περάσαμε στα τεράστια διαδικτυακά καταστήματα, στα μέσα κοινωνικής δικτύωσης, στις τραπεζικές υπηρεσίες και στις mailing πλατφόρμες. Σήμερα οι χρήστες μπορούν να αλληλοεπιδράσουν με οποιαδήποτε μεταδιδόμενη πληροφορία εντός της εφαρμογής, να υποβάλλουν παραμέτρους, να χρησιμοποιήσουν εργαλεία για πρόσβαση στην εφαρμογή και να στείλουν διάφορα αιτήματα.

Η ραγδαία όμως ανάπτυξη των διαδικτυακών εφαρμογών οδήγησε ταυτόχρονα και στην γιγάντωση του πεδίου δράσης των επίδοξων 'χάκερ'. Περισσότερη πολυπλοκότητα της εφαρμογής σημαίνει αυτόματα και δυσκολότερος έλεγχος αυτής. Αυτό έχει ως αποτέλεσμα η ασφάλεια να βρίσκεται στην κορυφή της λίστας των προγραμματιστών και των χρηστών, καθώς από τη στιγμή που οι εφαρμογές αποτελούν κομμάτι της ζωής και της καθημερινότητας όλων, είμαστε υπεύθυνοι για τις δράσεις και τις ενέργειες που πραγματοποιούμε καθημερινά χρησιμοποιώντας τις εφαρμογές, και πολύ περισσότερο όταν τις κατασκευάζουμε.

1.1 Περιγραφή του προβλήματος

Στις μέρες μας ο κώδικας σε ένα πολύ μεγάλο βαθμό έχει γίνει open source. Πλέον έχοντας πρόσβαση σε διάφορα τμήματα κώδικα μέσα σε λίγα δευτερόλεπτα, ο καθένας μπορεί να συμπεριλάβει κομμάτια κώδικα και πακέτα μέσω τρίτων, ώστε να χτίσει ή να βελτιώσει την εφαρμογή του. Έτσι μπορεί να επιτεύχθηκε ο στόχος της βελτίωσης παροχής υπηρεσιών και δυνατοτήτων των εφαρμογών αλλά η ασφάλεια τέθηκε σε δεύτερη μοίρα, καθώς ο πλήρης έλεγχος της εφαρμογής έγινε ακόμη δυσκολότερος.

Για πολλά χρόνια η ασφάλεια θεωρούνταν 'ταμπού' στην κοινότητα των προγραμματιστών. Έτσι οι διάφορες τεχνικές για την προστασία των εφαρμογών έφτασαν να θεωρούνται ξεπερασμένες. Ιδιαίτερα σε περιβάλλοντα όπως το Node.js, τα οποία έγιναν πολύ δημοφιλή σε πολύ μικρό χρονικό διάστημα, η έρευνα για την ασφάλεια ήταν μηδαμινή. Έτσι οι 'χάκερ' εκμεταλλευόμενοι την άγνοια των προγραμματιστών κατάφεραν να εκδηλώσουν πολλές επιθέσεις στο πέρασμα των χρόνων. Προσωπικά δεδομένα, τραπεζικοί λογαριασμοί κ.α., μπορούν να μπουν στο στόχαστρο μέσω τέτοιων επιθέσεων.

Τα τελευταία χρόνια γίνονται πολλές έρευνες στη βελτίωση των υπάρχοντών τεχνικών ασφαλείας, στην εξεύρεση νέων αλλά και στην ευαισθητοποίηση των προγραμματιστών ώστε να θέτουν την ασφάλεια στην κορυφή της λίστας με τις προτεραιότητες. Η στατική ανάλυση κώδικα είναι μία από τις μεθόδους που έχουν μελετηθεί ώστε να προβλέψουν

διάφορες τεχνικές 'bad coding' και να βρουν πιθανά σημεία στον κωδικό που ένας επίδοξος χάκερ θα μπορούσε να εκμεταλλευτεί. Υπάρχουν διάφορα εργαλεία στατικής ανάλυσης τα οποία αναλύουν τον κώδικα πριν εκτελεστεί και δίνουν μία καλύτερη εικόνα της δομής του, και ένα από αυτά είναι και οι Linters. Παρότι οι Linters είναι σχετικά παλιά εργαλεία, τα τελευταία χρόνια γίνονται έρευνες στο πως μπορούν να βελτιώσουν την ασφάλεια μέσω της στατικής ανάλυσης.

1.2 Στόχος και μεθοδολογία

Στόχος της παρούσας διπλωματικής είναι η κατασκευή ενός αυτοματοποιημένου εργαλείου μέσω του linter ESLint, που αποσκοπεί στην καλύτερη εποπτεία του κώδικα και την βελτίωση της ασφάλειας των διαδικτυακών εφαρμογών. Για το σκοπό αυτό υλοποιήθηκε ένα σύνολο κανόνων το οποίο συλλέχθηκε με βάση τις πιο γνωστές αδυναμίες-επιθέσεις στον τομέα των διαδικτυακών εφαρμογών και πιο συγκεκριμένα στις εφαρμογές που είναι βασισμένες στην JavaScript και κυρίως στο περιβάλλον Node.js. Σκοπός του είναι να προβλέψει συγκεκριμένες αδυναμίες από τα πρώτα στάδια κατασκευής του κώδικα και να ενημερώσει τον προγραμματιστή για πιθανά 'τρωτά σημεία'. Ταυτόχρονα αποτελεί ένα εφόδιο για καλύτερη εποπτεία της εφαρμογής από τη σκοπιά της ασφάλειας από τα πρώτα στάδια υλοποίησης της. Το σύνολο των κανόνων υλοποιήθηκε μέσω του AST explorer το οποίο αποτελεί ένα εργαλείο για την ανάλυση και τη διάσχιση του πηγαίου κώδικα μέσω συντακτικών δέντρων. Οι κανόνες ομαδοποιήθηκαν σε ένα plugin το οποίο ακολουθεί τη βασική διαμόρφωση του linter ESLint, ενώ μετατράπηκε και σε πακέτο npm για για κοινή χρήση. Τέλος πραγματοποιήθηκε έλεγχος του πακέτου npm σε δημοφιλή αποθετήρια της JavaScript από την πλατφόρμα του GitHub.

1.3 Διάρθρωση

Η δομή του κειμένου είναι η εξής:

- Στο Κεφάλαιο 1 παρουσιάστηκε μία γενική ιδέα σχετικά με την παρούσα διπλωματική καθώς και ο στόχος της
- Στο Κεφάλαιο 2 περιγράφονται το θεωρητικό καθώς και το πρακτικό υπόβαθρο που χρειάζεται να γνωρίζει ο χρήστης, ώστε να εξοικειωθεί και να κατανοήσει καλύτερα το πρόβλημα και τη διαδικασία που ακολουθήθηκε για να επιτευχθεί το τελικό αποτέλεσμα.
- Στο Κεφάλαιο 3 αναλύονται κάποια από τα πιο διάσημα είδη επιθέσεων και παρουσιάζονται παραδείγματα για το πως ένας επιτιθέμενος μπορεί να προκαλέσει ζημία στην εφαρμογή με βάση αυτές τις επιθέσεις.
- Στο Κεφάλαιο 4 γίνεται ανάλυση των ερευνών και της βιβλιογραφίας που έχουν γίνει στο χώρο της στατικής ανάλυσης με στόχο την ασφάλεια αλλά και του linting. Ο στόχος αυτού του κεφαλαίου είναι να αναδειχθεί η έρευνα και οι προσπάθειες που γίνονται πάνω στο αντικείμενο, καθώς και το πόσο μείζον και σύγχρονο θέμα αποτελεί με τα περιθώρια μελέτης που μπορούν να γίνουν.
- Στο Κεφάλαιο 5 αναλύεται η διαδικασία και η μεθοδολογία που ακολουθήθηκε ώστε να επιτευχθούν οι στόχοι της διπλωματικής.
- Στο Κεφάλαιο 6 παρουσιάζονται αναλυτικά τα επιμέρους αποτελέσματα που προέκυψαν από την παρούσα διπλωματική

- Στο Κεφάλαιο 7 εξηγούνται αναλυτικά τα συμπεράσματα που προέκυψαν από την έρευνα στο υπάρχον θέμα και προτείνονται μελλοντικές λύσεις και επεκτάσεις.

ΚΕΦΑΛΑΙΟ 2

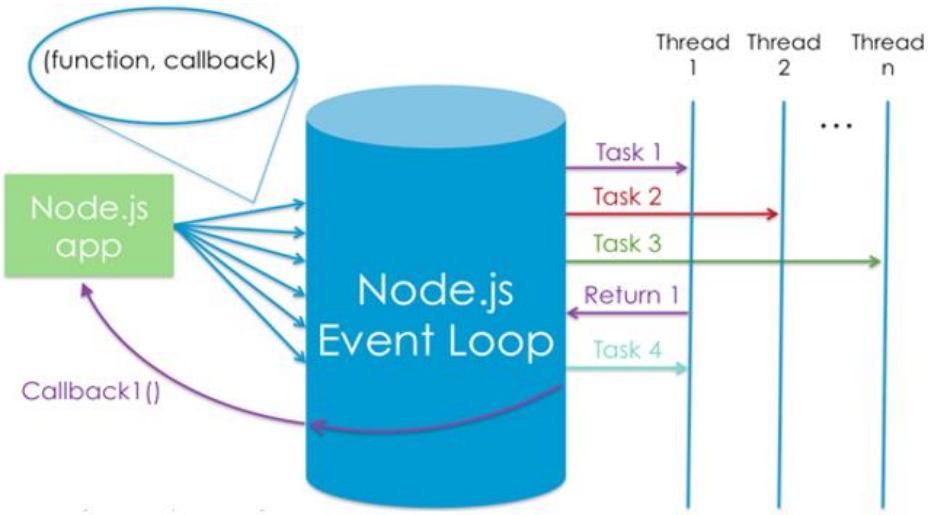
ΘΕΩΡΗΤΙΚΟ ΥΠΟΒΑΘΡΟ

Στο παρόν κεφάλαιο παρουσιάζονται οι βασικές έννοιες που συνδέονται άμεσα με το θέμα της διπλωματικής. Αρχικά αναλύονται οι εμπλεκόμενες τεχνολογίες όπως το περιβάλλον Node.js, η έννοια και τα αυτοματοποιημένα εργαλεία στατικής ανάλυσης, καθώς και η χρήση τους για τη βελτίωση της ασφάλειας των εφαρμογών. Επίσης γίνεται μία επισκόπηση των εργαλείων Linting και ειδικότερα του ESLint. Τέλος γίνεται αναφορά στην μετατροπή του πηγαίου κώδικα σε συντακτικά δέντρα και στην πλατφόρμα AST explorer.

2.1 Node.js

Το *Node.js* είναι ένα περιβάλλον εκτέλεσης ανοιχτού λογισμικού (open-source runtime environment), το οποίο είναι κτισμένο πάνω στη Chrome's V8 JavaScript μηχανή. Παρέχει ένα event-driven, non-blocking (ασύγχρονο) I/O (εισόδου-εξόδου) περιβάλλον εκτέλεσης για την υλοποίηση εφαρμογών υψηλής απόδοσης από την πλευρά του διακομιστή (server-side), χρησιμοποιώντας Javascript [2].

Το Node.js χρησιμοποιεί μονό νήμα (thread). Μερικές από τις πιο διάσημες τεχνολογίες όπως, PHP, ASP.NET, Ruby & Java Servers χρησιμοποιούν πολλαπλά νήματα όπου κάθε αίτημα (request) από την πλευρά του πελάτη (client-side), δημιουργεί ένα καινούργιο νήμα ή ακόμα και μία καινούργια διαδικασία. Τα αιτήματα (requests) στο Node.js τρέχουν στο ίδιο νήμα με κοινόχρηστους πόρους. Αυτή για την ακρίβεια είναι η μισή αλήθεια, καθώς το Node.js στην πραγματικότητα είναι ένα περιβάλλον εκτέλεσης μονού νήματος (single-thread) που διαθέτει εργάτες στο υπόβαθρο (background workers). Ο κύριος βρόχος συμβάντων (event-loop) είναι μονού νήματος αλλά οι περισσότερες εργασίες εισόδου-εξόδου (I/O) τρέχουν σε διαφορετικά νήματα. Αυτό συμβαίνει επειδή τα προγράμματα εισόδου-εξόδου (I/O APIs) του Node.js είναι ασύγχρονα (non-blocking) λόγω σχεδιασμού, προκειμένου να φιλοξενηθεί το event-loop. [3]. Στην παρακάτω εικόνα παρουσιάζεται ο τρόπος λειτουργίας του Node.js



Εικόνα 1 Λειτουργία του Node.js

1. Οι εφαρμογές Node περνάνε ασύγχρονες εργασίες στον κύριο βρόχο συμβάντων (event loop) μαζί με μία επανάκληση (callback).
2. Ο κύριος βρόχος συμβάντων διαχειρίζεται αποτελεσματικά ένα thread pool (πολλαπλά νήματα τα οποία περιμένουν να λάβουν και να εκτελέσουν 'επίπονες' εργασίες).
3. Όταν μία εργασία ολοκληρώνεται επιστρέφει μέσω του κύριου βρόχου συμβάντων (event loop) στην εφαρμογή

Τα πλεονεκτήματα του Node.js είναι τα εξής [4]:

- Χρησιμοποιεί Javascript.
- Είναι πολύ ελαφρύ και γρήγορο και αυτό γιατί είναι κτισμένο στη μηχανή Chrome's V8. Το V8 έχει τη δυνατότητα να μεταγλωττίζει και να εκτελεί JavaScript σε υψηλή ταχύτητα, κυρίως επειδή μεταγλωττίζει την JavaScript σε κώδικα μητρικής μηχανής. Επιπλέον το Node.js διαθέτει ένα πολύ γρήγορο κύριο βρόχο συμβάντων, το οποίο είναι ένα νήμα που εκτελεί όλες τις λειτουργίες εισόδου-εξόδου (I/O) με ασύγχρονο τρόπο.
- Έχει πολύ μεγάλη απόδοση.
- Χρησιμοποιεί την ίδια γλώσσα για όλα τα επίπεδα (layer).
- Είναι εύκολο στην επεξεργασία και στο συντήρηση (maintenance)

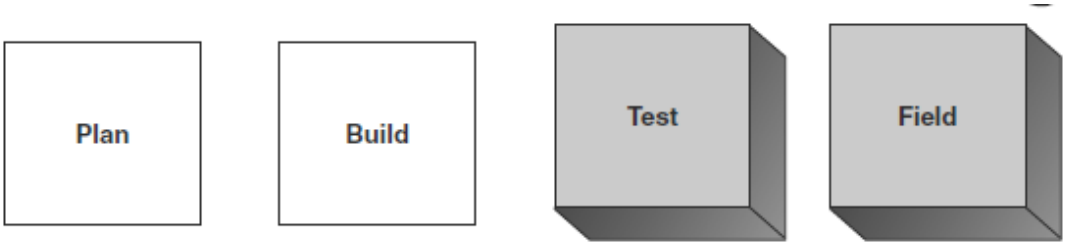
2.2 Στατική Ανάλυση

Η στατική ανάλυση που καλείται συχνά και σαν στατική ανάλυση κώδικα, είναι μία μέθοδος αποσφαλμάτωσης (debugging) για υπολογιστικά προγράμματα, η οποία εξετάζει τον κώδικα χωρίς όμως να τον εκτελεί. Η όλη διαδικασία παρέχει την κατανόηση της δομής του κώδικα και μπορεί να βοηθήσει ώστε να επιβεβαιωθεί ή όχι αν ο κώδικας συμμορφώνεται με κάποια συγκεκριμένα πρότυπα. Πιο συγκεκριμένα οι Chess και West αναφέρουν για την στατική ανάλυση: "Κάθε εργαλείο που αναλύει κώδικα χωρίς να απαιτείται η εκτέλεση του, εκτελεί στατική ανάλυση" [5]. Τα αυτοματοποιημένα εργαλεία μπορούν να βοηθήσουν τους προγραμματιστές στο να χρησιμοποιήσουν τη στατική ανάλυση. Η διαδικασία ελέγχου του κώδικα μόνο οπτικά, χωρίς τη βοήθεια αυτοματοποιημένων εργαλείων, καλείται μερικές φορές ως κατανόηση του προγράμματος.
Το μεγάλο πλεονέκτημα της στατικής ανάλυσης είναι ότι μπορεί να φανερώσει ένα λάθος το οποίο δεν θα φανερωθεί ποτέ μέχρι να προκαλέσει μία πολύ μεγάλη καταστροφή, βδομάδες, μήνες ή και χρόνια μετά την κυκλοφορία του προγράμματος. Στο παρακάτω υποκεφάλαιο αναλύεται η σύνδεση της στατικής ανάλυσης με την ασφάλεια και τα πλεονεκτήματα που προσφέρουν τα εργαλεία στατικής ανάλυσης σε αυτήν.

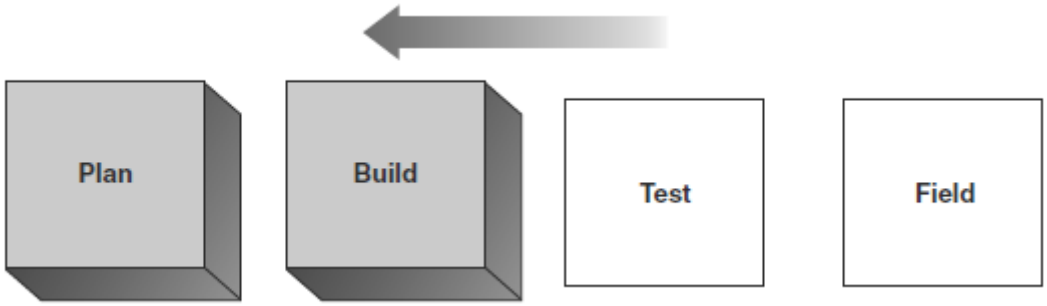
2.2.1 Εργαλεία στατικής ανάλυσης και ασφάλεια λογισμικού

Για πολλά χρόνια το θέμα της ασφάλειας των εφαρμογών αντιμετωπίζονταν με ένα τελείως διαφορετικό τρόπο. Firewalls, και διάφορες εφαρμογές σχετικές με αυτά, συστήματα εντοπισμού, δοκιμές διείσδυσης (penetration tests) κ.α. είναι μόνο μερικές από τις μεθόδους που προσπάθησαν να λύσουν το πρόβλημα της ασφάλειας. Η εικόνα 3 απεικονίζει αυτή την όχι και τόσο πετυχημένη προσέγγιση. Το πρόβλημα είναι αρκετά απλό σε αυτές τις μεθόδους, δεν προσφέρουν λύσεις. Για την ακρίβεια δημιουργούν έναν ατέρμων βρόχο από επαναλαμβανόμενα προβλήματα. Η σωστή απάντηση στο ερώτημα, ποια είναι η λύση, φαίνεται στην εικόνα 4, όπου πρέπει να γίνονται περισσότερες προσπάθειες στο σκοπό των περισσότερων προβλημάτων ασφάλειας λογισμικού, δηλαδή στον τρόπο που κτίζεται το

λογισμικό. Η ασφάλεια πρέπει να αποτελεί αναπόσπαστο κομμάτι στον τρόπο με τον οποίο το λογισμικό σχεδιάζεται [6].



Εικόνα 2 ασφάλεια που παρέχει ένα firewall



Εικόνα 3 static source code analysis

Διάφορες τεχνικές όπως ο αμυντικός προγραμματισμός, και οι έλεγχοι διείσδυσης που στοχεύουν στη δημιουργία αξιόπιστων λογισμικών, δεν προσφέρουν επαρκείς λύσεις στο θέμα της ασφάλειας. Τεράστιο ρόλο στην παραπάνω διαπίστωση παίζουν τα εργαλεία στατικής ανάλυσης κώδικα. Αυτά τα εργαλεία βοηθούν στον εντοπισμό προβλημάτων τα οποία είναι ορατά στον κώδικα. Τα πλεονεκτήματα αυτών των εργαλείων είναι τα παρακάτω:

- Τα εργαλεία στατικής ανάλυσης εφαρμόζουν τους ελέγχους διεξοδικά και με συνέπεια, χωρίς τις προκαταλήψεις που μπορεί να έχει ένας προγραμματιστής σχετικά με τα κομμάτια του κώδικα τα οποία είναι “ενδιαφέρον” από πλευράς ασφάλειας ή τα κομμάτια που μπορούν να εξασκηθούν μέσω δυναμικών δοκιμών. Ουσιαστικά εφαρμόζουν ένα “αμερόληπτο” έλεγχο πάνω στα θέματα ασφάλειας και ξεφεύγουν από την μεροληπτική κρίση του προγραμματιστή.
- Εξετάζοντας τον κώδικα, τα εργαλεία στατικής ανάλυσης συχνά στοχεύουν στη ρίζα του προβλήματος ασφάλειας, και όχι απλά σε ένα από τα συμπτώματα του. Αυτό είναι ιδιαίτερα σημαντικό έτσι ώστε να επιβεβαιωθεί ότι οι αδυναμίες του κώδικα (vulnerabilities) διορθώνονται κατάλληλα.
- Η στατική ανάλυση μπορεί να βρει λάθη νωρίς, κατά τη διάρκεια της ανάπτυξης (development), ακόμα και πριν το πρόγραμμα ξεκινήσει να τρέχει. Βρίσκοντας ένα λάθος στην αρχή της διαδικασίας ανάπτυξης, όχι μόνο μειώνει το κόστος της επιδιόρθωσης του λάθους, αλλά μπορεί να δοθεί και σωστή ανατροφοδότηση (feedback) ώστε να βελτιωθεί η δουλειά του προγραμματιστή ουσιαστικά. Ο προγραμματιστής έχει την ευκαιρία να διορθώσει λάθη και παραλείψεις τις οποίες μέχρι πριν αγνοούσε.
- Όταν ένας ερευνητής σχετικός με την ασφάλεια (security researcher) ανακαλύπτει μία καινούργια μορφή επίθεσης, τα εργαλεία στατικής ανάλυσης είναι πολύ εύκολο

να ελέγξουν το μεγαλύτερο κομμάτι του κώδικα ώστε να διαπιστώσουν που μπορεί να εφαρμοστεί αυτό το καινούργιο είδος της επίθεσης.

Το πιο συνηθισμένο "παράπονο" κατά των εργαλείων στατικής ανάλυσης είναι ότι παράγουν πάρα πολύ θόρυβο. Πιο συγκεκριμένα παράγουν πολλές περιπτώσεις ψευδών συναγερμών (false-positives**)**. Στο πλαίσιο αυτό, σαν ψευδής συναγερμός δηλώνεται η αναφορά ενός προβλήματος σε ένα πρόγραμμα, ενώ αυτό ουσιαστικά δεν υφίσταται. Γίνεται εύκολα αντιληπτό ότι ένας μεγάλος αριθμός από ψευδείς συναγερμούς μπορεί να προκαλέσει αρκετές δυσκολίες, όχι μόνο όσον αφορά το χρόνο που ενδεχομένως να σπαταλήσει ο προγραμματιστής ώστε να αναλύσει αυτές τις περιπτώσεις αλλά και τις τυχόν παραλείψεις που με βεβαιότητα θα κάνει σε άλλα κομμάτια του κώδικα που απαιτούν πολύ μεγαλύτερη προσοχή. Συνεπώς είναι βέβαιο ότι οι ψευδείς συναγερμοί (false-positives) είναι σίγουρα ανεπιθύμητοι, αλλά από την σκοπιά της ασφάλειας τα false-negatives είναι σίγουρα πολύ χειρότερα. Με τον όρο false-negative αναφέρεται η περίπτωση όπου ένα λάθος υπάρχει στον κώδικα αλλά το εργαλείο αδυνατεί να το εντοπίσει. Οι συνέπειες στην περίπτωση των false-positives είναι ο χρόνος που απαιτείται από τον προγραμματιστή ώστε να αναγνωρίσει αυτές τις περιπτώσεις. Οι συνέπειες στην περίπτωση των false-negative είναι ότι οι διάφορες αδυναμίες παραμένουν στον κώδικα, καθώς δεν εντοπίστηκαν από το εργαλείο.

Όλα τα εργαλεία στατικής ανάλυσης εγγυημένα παράγουν μερικά false-positives ή μερικά false-negatives. Τα περισσότερα παράγουν και τα δύο. Η ισορροπία μεταξύ των false-positives και των false-negatives συχνά καθορίζεται από τον σκοπό του εργαλείου. Η ισορροπία είναι τελείως διαφορετική για στατικά εργαλεία τα οποία σχεδιάζονται για την εύρεση διάφορων προβλημάτων (bugs) και για εργαλεία τα οποία στοχεύουν αποκλειστικά σε θέματα που αφορούν την ασφάλεια. Το κόστος του να χαθεί ένα πρόβλημα (bug) μέσα στην εφαρμογή δεν είναι και τόσο μεγάλο, με την έννοια ότι υπάρχουν πολλές διαδικασίες και τεχνικές που διασφαλίζουν ότι τα σημαντικότερα bugs έχουν επιλυθεί. Για αυτό το λόγο εργαλεία που στοχεύουν στην ποιότητα του κώδικα συνηθίζουν να παράγουν μικρό αριθμό από false-positives και δέχονται πολύ περισσότερο τα false-negatives. Η ασφάλεια όμως είναι μία τελείως διαφορετική ιστορία. Οι συνέπειες της παράβλεψης ενός σφάλματος που αφορά την ασφάλεια μπορεί να είναι τεράστιες, οπότε τα εργαλεία ασφάλειας συνήθως παράγουν περισσότερα false-positives έτσι ώστε να μειώσουν τα false-negatives.

2.3 Linting

Το Linting είναι η διαδικασία ελέγχου του πηγαίου κώδικα (source code) για προγραμματιστικά και στιλιστικά σφάλματα. Ο όρος πρωτοεμφανίστηκε σε ένα εργαλείο για UNIX το οποίο εξέταζε κώδικα γραμμένο σε C. Συγκεκριμένα, το 1978, ο Stephen C. Johnson στη Bell Labs δημιούργησε το πρώτο εργαλείο linting, ενώ προσπαθούσε να κάνει debug κώδικα C. Ο όρος "lint" προέρχεται από τα ανεπιθύμητα τμήματα κλωστής ή άλλων υλικών που βρίσκονται στο μαλλί των προβάτων.

2.3.1 Linters

Ένας linter είναι ένα αυτοματοποιημένο εργαλείο που τρέχει σε στατικό κώδικα για να βρει διάφορες μορφοποιήσεις, μη τήρηση προτύπων και συμβάσεων κώδικα (coding standards), και λογικά λάθη στον κώδικα. Τρέχοντας ένα linter (static code analyzer) στον πηγαίο κώδικα βελτιώνεται αυτόματα η ποιότητα του κώδικα, και επίσης βοηθάει στο να επιβεβαιωθεί ότι ο κώδικας είναι ευανάγνωστος και εύκολος στη συντήρηση (maintenance).

Οι Linters είναι επίσης πολύ χρήσιμοι για την διαμόρφωση της μορφής (format) του κώδικα και τη συμμόρφωση με τις βέλτιστες πρακτικές που σχετίζονται με την εκάστοτε γλώσσα. Πολλοί μεταγλωττιστές (compilers) χρησιμοποιούν τεχνικές linting για καλύτερη απόδοση και ταχύτερη δημιουργία του κώδικα. Τα τελευταία χρόνια οι linters είναι ιδιαίτερα χρήσιμοι και έχουν μεγάλη απήχηση σε γλώσσες προγραμματισμού που δεν χρησιμοποιούν μεταγλωττιστή όπως η Javascript και η Python. Με αυτό τον τρόπο μπορούν να εντοπίσουν λάθη που υπάρχουν στον κώδικα πριν την εκτέλεση. Ο εντοπισμός των σφαλμάτων στον κώδικα γίνεται από τους linters χρησιμοποιώντας μεθόδους στατικής ανάλυσης του κώδικα. [7]

2.3.2 Linters στην Javascript

Οι δημοφιλέστεροι Linters στην JavaScript είναι οι παρακάτω:

- **JSLint:** Ο JSLint αναλύει κώδικα Javascript και διασφαλίζει ότι ακολουθούνται ορισμένες συμβάσεις όσον αφορά τον κώδικα. Αποτελεί έναν ελεγκτή (checker) της Javascript όσον αφορά τη σύνταξη καθώς επίσης και έναν επικυρωτή (validator). Ο JSLint σαρώνει τον πηγαίο κώδικα JavaScript για να βρει ένα πρόβλημα, και επιστρέφει ένα μήνυμα που περιγράφει το πρόβλημα και μια κατά προσέγγιση θέση (αριθμός γραμμής) εντός του κώδικα. Αναφέρει επίσης ένα σφάλμα σύνταξης, ορισμένα ζητήματα συμβάσεων στυλ και άλλα διαρθρωτικά προβλήματα κώδικα. Οι οδηγίες του επιτρέπουν στους προγραμματιστές να ορίζουν μεταβλητές και να παρέχουν άλλες επιλογές στον JSLint απευθείας από τον πηγαίο κώδικα. Αυτό απαλλάσσει τους προγραμματιστές από την επανειλημμένη ρύθμιση των επιλογών JSLint GUI.
- **JSHint:** Το JSHint είναι ένα εκτελέσιμο στη γραμμή εντολών (command line). Το JSHint σαν εργαλείο βοηθάει το χρήστη να γράψει πιο αξιόπιστο και συνεπή κώδικα JavaScript. Έχει υποστήριξη για πολλές βιβλιοθήκες, όπως jQuery, Qunit, Node.js, Mocha κ.ο.κ. Επίσης χρησιμοποιείται για την επιβολή συμβάσεων όσον αφορά τον τρόπο γραφής κώδικα και στιλιστικούς οδηγούς (style guides). Το JSHint είναι ρυθμιζόμενο και διατίθεται μέσω του node package manager (npm). Οι προγραμματιστές μπορούν να ελέγξουν τη συμπεριφορά του JSHint μέσω του καθορισμού των επιλογών του linter.
- **ESLint:** Το ESLint, είναι ένα εργαλείο που αφορά την ποιότητα του κώδικα. Επίσης εντοπίζει και αναφέρει τα πρότυπα που βρέθηκαν στον κώδικα JavaScript. Το ESLint χρησιμοποιεί τον Espree για την ανάλυση του κώδικα JavaScript και επίσης ένα αφηρημένο συντακτικό δέντρο (abstract syntax tree) για να αξιολογήσει τα πρότυπα στον κώδικα. Το ESLint είναι πλήρως διαμορφώσιμο. Κάθε κανόνας λειτουργεί μόνο με τη βασική επικύρωση συντακτικού (basic syntax validation) ή αναμιγνύεται και ταιριάζει με τους δεσμευμένους και προσαρμοσμένους κανόνες για να καταστήσει το ESLint ένα τέλειο εργαλείο. Ένας προγραμματιστής μπορεί να τροποποιήσει τους κανόνες που χρησιμοποιεί το έργο (project) του, χρησιμοποιώντας ακόμα και ένα αρχείο ρυθμίσεων.

2.3.3 ESLint, η μεγάλη εικόνα

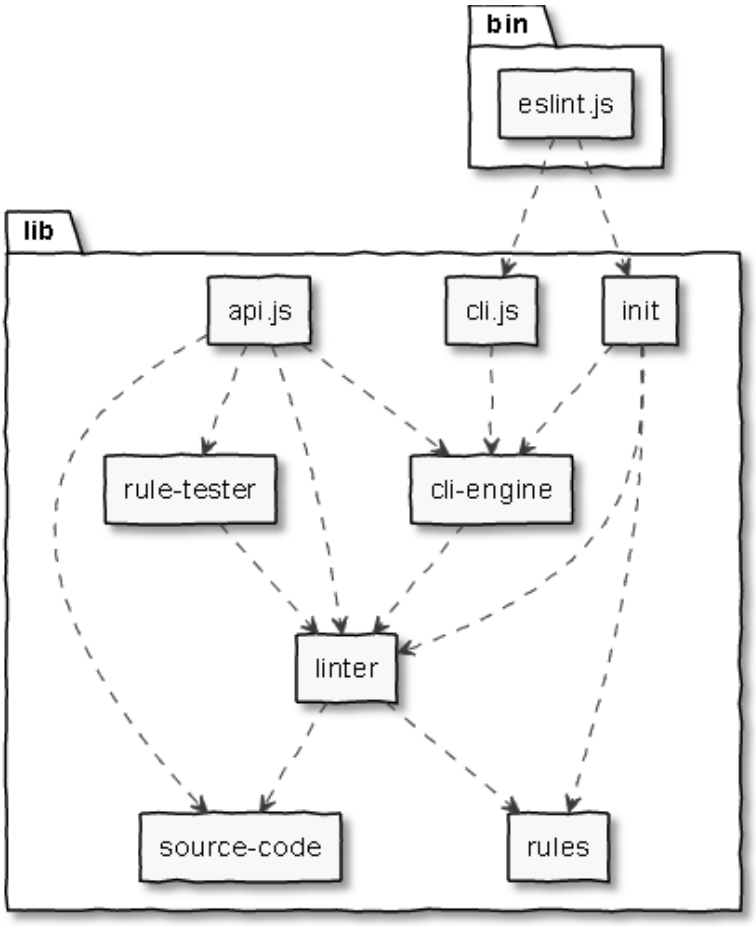
Σ ’αυτό το υποκεφάλαιο θα γίνει προσπάθεια να εξηγηθεί σε βάθος η λειτουργικότητα του ESLint όσον αφορά τη διαμόρφωση που μπορεί να γίνει, τους κανόνες και τον τρόπο που δημιουργούνται, καθώς και η λειτουργικότητα που προσφέρουν αυτοί και τα plugin.

ΣΧΕΤΙΚΑ

Το ESLint είναι ένα linting πρόγραμμα ανοιχτού κώδικα JavaScript, το οποίο δημιουργήθηκε αρχικά από τον Nicholas C. Zakas τον Ιούνιο του 2013. Το Code linting είναι ένας τύπος στατικής ανάλυσης που χρησιμοποιείται συχνά για την εύρεση προβληματικών μοτίβων ή κώδικα που δεν συμμορφώνεται με συγκεκριμένες στιλιστικές οδηγίες (code styles). Υπάρχουν code linters για τις περισσότερες γλώσσες προγραμματισμού και ακόμα και οι μεταγλωττιστές (compilers) μερικές φορές ενσωματώνουν linting στη διαδικασία σύνταξης (compilation process). Η JavaScript, που είναι μια δυναμική γλώσσα, η οποία είναι ιδιαίτερα επιρρεπής στο σφάλμα του προγραμματιστή. Χωρίς το πλεονέκτημα μιας διαδικασίας σύνταξης, ο κώδικας JavaScript τυπικά εκτελείται για να βρει συντακτικά ή άλλα σφάλματα. Τα εργαλεία linting όπως το ESLint επιτρέπουν στους προγραμματιστές να ανακαλύψουν προβλήματα στον κώδικα JavaScript χωρίς να το εκτελέσουν.

Ο πρωταρχικός λόγος για τον οποίο δημιουργήθηκε το ESLint ήταν για να επιτρέψει στους προγραμματιστές να δημιουργήσουν τους δικούς τους κανόνες linting. Το ESLint έχει σχεδιαστεί έτσι ώστε να έχει όλους τους κανόνες πλήρως συνδεδεμένους (pluggable). Οι προεπιλεγμένοι (default) κανόνες γράφονται όπως ακριβώς θα γραφόταν και οι κανόνες ενός plugin. Μπορούν όλοι να ακολουθήσουν το ίδιο μοτίβο ή πρότυπο, τόσο για τους ίδιους τους κανόνες όσο και για τα test. Ενώ το ESLint κυκλοφορεί με ορισμένους ενσωματωμένους κανόνες για να προσφέρει μία λειτουργικότητα εξαρχής, μπορεί ο καθένας να φορτώσει δυναμικά κανόνες σε οποιαδήποτε χρονική στιγμή. Το ESLint έχει υλοποιηθεί χρησιμοποιώντας το Node.js για να παρέχει ένα περιβάλλον γρήγορης εκτέλεσης καθώς και εύκολη εγκατάσταση μέσω του npm. [8]

ΑΡΧΙΤΕΚΤΟΝΙΚΗ



Εικόνα 4 Αρχιτεκτονική του ESLint

Σε ανώτερο επίπεδο (high level) υπάρχουν μερικά βασικά τμήματα του ESLint [9]:

- **bin/eslint.js:** Είναι το αρχείο που εκτελείται στην πραγματικότητα με τη βοήθεια της γραμμής εντολών (command line). Ουσιαστικά αποτελεί μόνο ένα περιτύλιγμα, το οποίο απλώς περνάει τα ορίσματα στο **cli**. Είναι σκοπίμως μικρό καθώς δεν απαιτεί μεγάλο έλεγχο (testing).
- **lib/api.js:** Αυτό το αρχείο αποτελεί το σημείο εισόδου του require('eslint'). Ουσιαστικά εκθέτει ένα αντικείμενο (object) το οποίο περιέχει τις δημόσιες κλάσεις (public classes) Linter, CLIEngine, RuleTester και SourceCode
- **lib/cli.js:** Αυτό το αρχείο αποτελεί την καρδιά του ESLint CLI. Ουσιαστικά λαμβάνει έναν πίνακα από ορίσματα και στη συνέχεια χρησιμοποιεί το eslint για να εκτελέσει τις εντολές. Διατηρώντας αυτό ως ξεχωριστό βοηθητικό πρόγραμμα, επιτρέπει σε άλλους να καλούν αποτελεσματικά το ESLint από ένα άλλο πρόγραμμα Node.js σαν να έγινε στη γραμμή εντολών. Η κλήση της main γίνεται μέσω της μεθόδου cli.execute().
- **lib/init/:** Αυτό το module περιέχει την εντολή –init, η οποίο δημιουργεί ένα αρχείο διαμόρφωσης (configuration file) για τους χρήστες.
- **lib/cli-engine/:** Αυτό το module είναι η CLIEngine κλάση η οποία βρίσκει τα αρχεία πηγαίου κώδικα (source code) και τα αρχεία διαμόρφωσης (configuration) και στη συνέχεια κάνει επικύρωση του κώδικα (code verifying) μέσω της κλάσης Linter. Αυτό

περιλαμβάνει τη λογική φόρτωση των αρχείων διαμόρφωσης (configuration files), τους αναλυτές (parsers), τα plugins καθώς και τους μορφοποιητές (formatters).

- **lib/linter:** Αυτό το module είναι η κλάση Linter, το οποίο κάνει επαλήθευση του κώδικα (code verifying) βασισμένη στη διαμόρφωση που έχει γίνει στις επιλογές διαμόρφωσης (configuration options). Αυτός ο φάκελος ουσιαστικά δεν αλληλοεπιδρά καθόλου με την κονσόλα.
- **lib/rule-tester:** Αυτό το module είναι η κλάση RuleTester το οποίο χρησιμοποιεί Mocha, έτσι ώστε οι κανόνες να μπορούν να ελέγχονται μεμονωμένα.
- **lib/rules:** Ουσιαστικά εδώ εμπεριέχονται οι κανόνες οι οποίοι κάνουν επικύρωση (verify) του πηγαίου κώδικα.

ΔΙΑΜΟΡΦΩΣΗ

Ο ESLint έχει σχεδιαστεί για να είναι πλήρως διαμορφώσιμος, δηλαδή ο χρήστης μπορεί να απενεργοποιήσει κάθε κανόνα και να τον τρέξει μόνο με τη βασική επαλήθευση, ή να συνδυάσει τους κανόνες και να δημιουργήσει δικούς του ώστε να κάνει τον ESLint κατάλληλο για το έργο του. Υπάρχουν 2 βασικοί τρόποι διαμόρφωσης του ESLint [10]:

- **Διαμόρφωση μέσω σχολίων:** Με χρήση σχολίων JavaScript μπορεί να ενσωματωθεί η διαμόρφωση απευθείας σε ένα αρχείο.
- **Αρχεία διαμόρφωσης:** Με χρήση ενός Javascript, JSON ή YAML αρχείου καθορίζεται η διαμόρφωση για ένα ολόκληρο φάκελο και όλους τους υποφακέλους. Αυτό μπορεί να είναι στην μορφή ενός .eslintrc* αρχείου ή ένα πεδίο eslintConfig στο αρχείο package.json. Ο ESLint θα αναζητήσει και θα διαβάσει και τα δύο.

Αν υπάρχει ένα αρχείο διαμόρφωσης στο βασικό ευρετήριο (home directory), ο ESLint θα το χρησιμοποιήσει μόνο αν δεν μπορεί να βρει άλλο αρχείο διαμόρφωσης. Άλλα πράγματα που μπορούν να διαμορφωθούν από το χρήστη είναι:

- **Το περιβάλλον** (environment)**:** Δηλαδή τα περιβάλλοντα στα οποία τρέχει ο κώδικας. Κάθε περιβάλλον έρχεται με ένα σύνολο προκαθορισμένων global μεταβλητών.
- **Globals:** Οι επιπλέον global μεταβλητές που χρησιμοποιεί ο κώδικας κατά την εκτέλεση.
- **Parser**: Το αντίστοιχο εργαλείο που χρησιμοποιείται για να γίνει ανάλυση του κώδικα (parse).
- **Κανόνες**: Ορίζεται ποιοι κανόνες είναι ενεργοποιημένοι και σε ποιο επίπεδο σφάλματος.

ΔΙΑΜΟΡΦΩΣΗ ΚΑΝΟΝΩΝ ΚΑΙ PLUGIN

Plugins

Ο ESLint υποστηρίζει τη χρήση εξωτερικών plugin. Πριν γίνει χρήση ενός plugin πρέπει πρώτα να εγκατασταθεί μέσω npm. Ο καθορισμός των plugin που χρησιμοποιούνται γίνεται μέσω του αρχείου διαμόρφωσης χρησιμοποιώντας την ιδιότητα plugins. Παρακάτω φαίνεται ένας πίνακας που εμπεριέχει τα ονόματα των plugins. Το όνομα μπορεί να είναι σκέτο ή με το πρόθεμα eslint-plugin- όπως παρουσιάζεται παρακάτω.

```
{
    "plugins": [
        "plugin1",
        "eslint-plugin-plugin2"
    ]
}
```

Εικόνα 5 Διαμόρφωση plugin

Και στο YAML:

```
---
  plugins:
    - plugin1
    - eslint-plugin-plugin2
```

Εικόνα 6 Διαμόρφωση plugin στο YAML

Rules

Ένα άλλο πεδίο που μπορεί να διαμορφωθεί στο ESLint είναι οι κανόνες. Το ποιους κανόνες θα χρησιμοποιήσει το έργο του εκάστοτε χρήστη ελέγχεται από τα σχόλια διαμόρφωσης (configuration comments) καθώς και τα αρχεία διαμόρφωσης (configuration files). Κάθε κανόνες διαθέτει ένα ID που είναι μοναδικό για τον καθένα και κάποιες τιμές οι οποίες όταν μεταβάλλονται αλλάζει και η χρήση του κανόνα. Οι τιμές που έχει ένας κανόνας είναι οι παρακάτω:

- **0** ή **off** : Για την απενεργοποίηση ενός κανόνα.
- **warn** ή **1**: Για ενεργοποίηση του κανόνα ως προειδοποίηση (warning).
- **error** ή **2:** Για την ενεργοποίηση του κανόνα ως σφάλμα.

Η διαμόρφωση των κανόνων μπορεί να γίνει και μέσω σχολίων. Συγκεκριμένα παρουσιάζονται κάποια παραδείγματα που φανερώνουν τον τρόπο με τον οποίο μπορεί να γίνει η διαμόρφωση κανόνων μέσω αυτών.

```
/* eslint eqeqeq:0, curly: 2 */
```

Εικόνα 7 Διαμόρφωση κανόνων μέσω σχολίων

Η διαμόρφωση μέσω αρχείων ακολουθεί διαφορετική λογική, η οποία φαίνεται παρακάτω. Η διαμόρφωση γίνεται με χρήση του πεδίου “rules” με κλειδιά τα rule IDs και σαν ιδιότητες τις αντίστοιχες επιθυμητές τιμές. Αντίθετα για να διαμορφωθεί ένας κανόνας που βρίσκεται μέσα σε ένα plugin πρέπει πριν το όνομα του κανόνα να μπει το όνομα του plugin όπως φαίνεται στην παρακάτω εικόνα.

```
{
    "plugins": [
        "plugin1"
    ],
    "rules": {
        "eqeqeq": "off",
        "curly": "error",
        "quotes": ["error", "double"],
        "plugin1/rule1": "error"
    }
}
```

Εικόνα 8 Διαμόρφωση κανόνων μέσω Configuration files

Η απενεργοποίηση των κανόνων μπορεί επιπλέον να γίνει τοπικά για μερικές γραμμές μέσα σε ένα αρχείο, με χρήση σχολίου που ξεκινά με τη φράση eslint-disable. Στην παρακάτω εικόνα φαίνεται ένα παράδειγμα απενεργοποίησης όλων των κανόνων τοπικά.

```
/* eslint-disable */
  alert('foo');
/* eslint-enable */
```

Εικόνα 9 Απενεργοποίηση όλων των κανόνων τοπικά

Η απενεργοποίηση συγκεκριμένων κανόνων γίνεται:

```
/* eslint-disable no-alert, no-console */

alert('foo');
console.log('bar');

/* eslint-enable no-alert, no-console */
```

Εικόνα 10 Απενεργοποίηση συγκεκριμένων κανόνων

Η απενεργοποίηση όλων των κανόνων για μία συγκεκριμένη γραμμή γίνεται:

```
alert('foo'); // eslint-disable-line

// eslint-disable-next-line
alert('foo');

/* eslint-disable-next-line */
alert('foo');

alert('foo'); /* eslint-disable-line */
```

Εικόνα 11 Απενεργοποίηση όλων των κανόνων σε συγκεκριμένη γραμμή

Η απενεργοποίηση συγκεκριμένου κανόνα σε συγκεκριμένη γραμμή γίνεται:

```
alert('foo'); // eslint-disable-line no-alert

// eslint-disable-next-line no-alert
alert('foo');

alert('foo'); /* eslint-disable-line no-alert */

/* eslint-disable-next-line no-alert */
alert('foo');
```

Εικόνα 12 Απενεργοποίηση συγκεκριμένου κανόνα σε συγκεκριμένη γραμμή

ΚΑΝΟΝΕΣ, Η ΜΕΓΑΛΗ ΕΙΚΟΝΑ

Προηγουμένως αναλύθηκε η χρησιμότητα η αρχιτεκτονική και ο τρόπος με τον οποίο μπορεί να διαμορφωθεί το ESLint. Παρακάτω παρουσιάζεται η δομή των κανόνων. Κάθε κανόνας στον ESLint έχει τρία αρχεία με το αναγνωριστικό του:

- Στο **lib/rules** directory: Έναν πηγαίο φάκελο (π.χ no-extra-semi.js).
- Στο **test/lib/rules** directory: Έναν φάκελο ελέγχου (π.χ no-extra-semi.js).
- Στο **docs/rules** directory: Ένα Markdown φάκελο που περιέχει πληροφορίες για τον κανόνα (π.χ. no-extra-semi).

Παρακάτω παρουσιάζεται η βασική μορφή στον πηγαίο φάκελο ενός κανόνα [11].

```
/**
 * @fileoverview Rule to disallow unnecessary semicolons
 * @author Nicholas C. Zakas
 */

"use strict";

//------------------------------------------------------------------------------
// Rule Definition
//------------------------------------------------------------------------------

module.exports = {
    meta: {
        type: "suggestion",

        docs: {
            description: "disallow unnecessary semicolons",
            category: "Possible Errors",
            recommended: true,
            url: "https://eslint.org/docs/rules/no-extra-semi"
        },
        fixable: "code",
        schema: [] // no options
    },
    create: function(context) {
        return {
            // callback functions
        };
    }
};
```

Εικόνα 13 βασικό format ενός κανόνα

Βασικά στοιχεία των κανόνων:

Ο πηγαίος φάκελος του κανόνα, δηλαδή εκεί που γίνεται η υλοποίηση του κανόνα, εξάγει (export) ένα αντικείμενο (object) με τις ακόλουθες ιδιότητες:

meta: {object}, αντικείμενο το οποίο περιέχει τα μετα-δεδομένα για τον κανόνα:

- **type:** Δηλώνει τον τύπο του κανόνα. Μπορεί να πάρει 3 τιμές, “suggestion”, “problem” ή “layout”.

  “suggestion”: Σημαίνει ότι ο κανόνας αναγνώρισε κάτι το οποίο θα μπορούσε να είχε γίνει με καλύτερο τρόπο, αλλά ακόμα και αν δεν αλλάξει η υλοποίηση, δεν θα εμφανιστούν λάθη (errors) στον κώδικα.

  “problem”: Σημαίνει ότι ο κανόνας αναγνώρισε σημείο στον κώδικα, το οποίο είτε θα προκαλέσει σφάλμα, είτε κάποια μη συνηθισμένη συμπεριφορά. Οι προγραμματιστές θα πρέπει να θεωρούν το συγκεκριμένο ως υψηλή προτεραιότητα.

  “layout”: Σημαίνει ότι ο κανόνας επικεντρώνεται στο πως φαίνεται ο κώδικας και όχι στο πως τρέχει, δηλαδή κενά, παρενθέσεις, ερωτηματικά. Τέτοιου τύπου κανόνες δουλεύουν σε κομμάτια του κώδικα που δεν είναι ορισμένα στα αόριστα συντακτικά δέντρα (abstract syntax trees).

- **docs:** Απαιτείται για τους βασικούς κανόνες του ESLint.

  description: {string}, παρέχει μία σύντομη περιγραφή του κανόνα

  category: {string}, ορίζει την κεφαλίδα κάτω από την οποία θα ενσωματωθεί ο κανόνας.

recommended: {boolean}, είναι ενεργό όταν θέλουμε να ενεργοποιήσουμε τον κανόνα μέσω του "extends:" eslint: recommended" στο φάκελο διαμόρφωσης.
url: {string}, δηλώνει το url στο οποίο βρίσκονται οι οδηγίες για τον κανόνα (documentation file).

- **fixable:** {string}, μπορεί να είναι είτε "code" είτε "whitespace", και σημαίνει ότι με την εντολή -- fix, στη γραμμή εντολών (command line), το πρόβλημα διορθώνεται αυτόματα από τον κανόνα.
- **deprecated:** {boolean}, δηλώνει εάν ο κανόνας έχει αφαιρεθεί.
- **replaceBy:** {array}, στην περίπτωση ενός κανόνα που έχει αφαιρεθεί, δηλώνει τον κανόνα που τον έχει αντικαταστήσει.
- **create:** {function}, συνάρτηση η οποία επιστρέφει ένα αντικείμενο με μεθόδους τις οποίες ο ESLint καλεί για να "επισκεφθούν" τα nodes ενώ διασχίζει το αφηρημένο συντακτικό δέντρο (abstract syntax tree) της Javascript.

Σημείωση:

- Ένας κανόνας μπορεί να χρησιμοποιήσει το τρέχον node και το "δέντρο" που το περιβάλει, για να δηλώσει ένα πρόβλημα.

ΤΟ ΑΝΤΙΚΕΙΜΕΝΟ CONTEXT

Το αντικείμενο (object) context περιέχει πρόσθετη λειτουργικότητα η οποία είναι χρήσιμη για τους κανόνες ώστε να εκτελέσουν το έργο τους. Όπως υποδηλώνεται και από το όνομα, το αντικείμενο context περιέχει πληροφορία η οποία είναι σχετική με το περιεχόμενο του κανόνα.

context.report()

Η κύρια μέθοδος που χρησιμοποιείται στο αντικείμενο context είναι η context.report(), η οποία δημοσιεύει μία προειδοποίηση (warning) ή ένα σφάλμα (error), ανάλογα με τη διαμόρφωση που έχει προηγηθεί, όπως αναφέρθηκε προηγουμένως. Αυτή η μέθοδος δέχεται ένα όρισμα, το οποίο είναι ένα αντικείμενο που περιέχει τις παρακάτω ιδιότητες:

- **message:** Το μήνυμα του προβλήματος που εντόπισε ο κανόνας.
- **node:** (προαιρετικό), το AST node το οποίο σχετίζεται με το πρόβλημα.
- **loc:** (προαιρετικό), ένα αντικείμενο το οποίο δηλώνει την τοποθεσία του προβλήματος.
- **data:** (προαιρετικό), είναι ένα placeholder για τα δεδομένα του μηνύματος
- **fix:** (προαιρετικό), μία συνάρτηση η οποία εφαρμόζει μία επιδιόρθωση ώστε να επιλύσει το πρόβλημα.

Ένα απλό παράδειγμα χρήσης του context.report() για την δήλωση ενός προβλήματος (μέσω μηνύματος) που εντοπίστηκε από τον κανόνα είναι το παρακάτω:

```
context.report({
    node: node,
    message: "Unexpected identifier"
});
```

Εικόνα 14 context.report

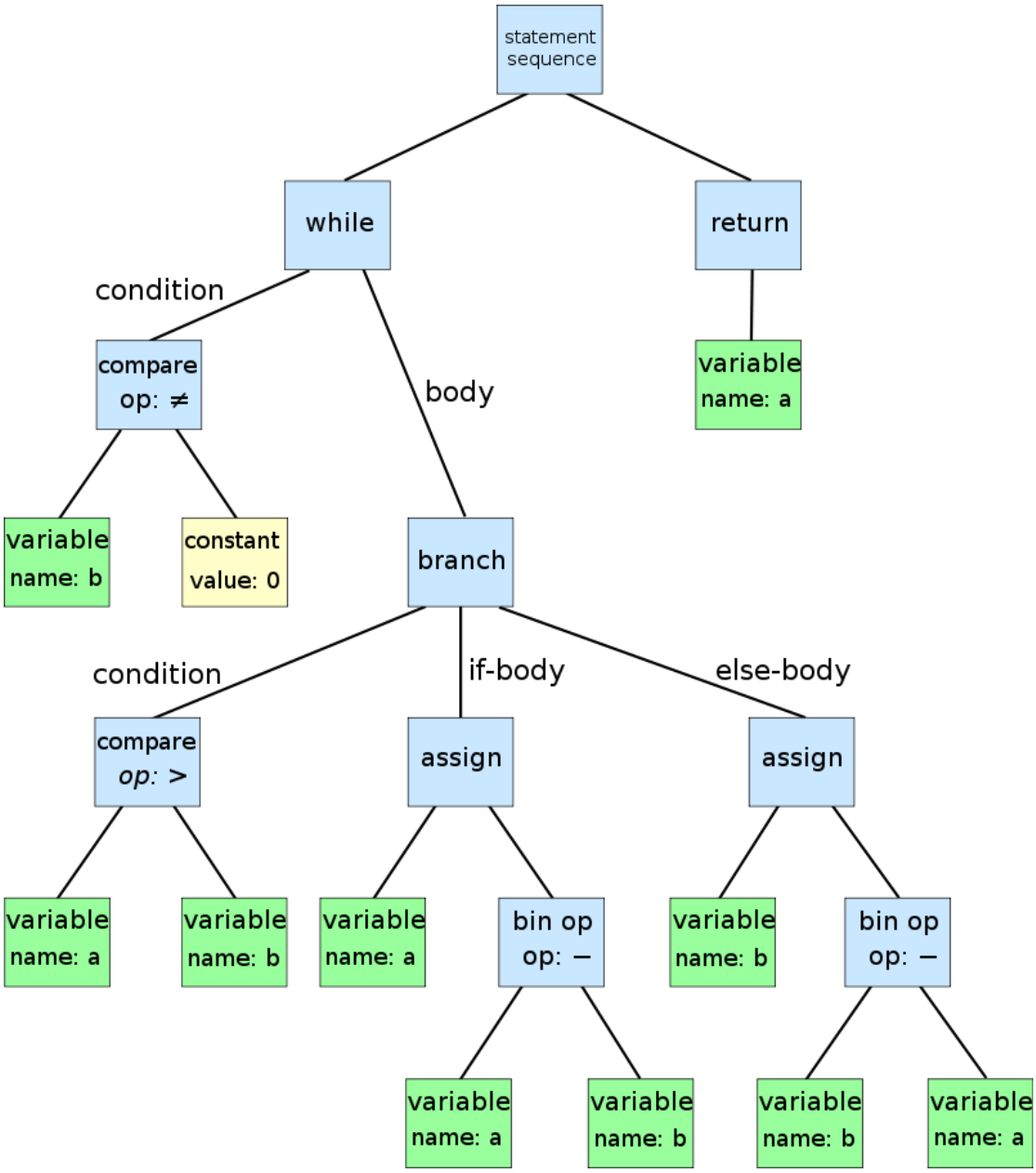
2.4 Abstract Syntax Trees

Με τον όρο αφηρημένο συντακτικό δέντρο (Abstract Syntax Tree-AST) ή απλώς συντακτικό δέντρο [12] στην επιστήμη των υπολογιστών ορίζεται η αναπαράσταση ενός δέντρου της αφηρημένης συντακτικής δομής (abstract syntactic structure) του πηγαίου κώδικα που είναι γραμμένος σε μία γλώσσα προγραμματισμού. Κάθε (node) κόμβος του δέντρου υποδηλώνει μία κατασκευή που εμφανίζεται στον πηγαίο κώδικα. Με τον όρο "αφηρημένη", εννοούμε ότι η δομή του δέντρου δεν αντιπροσωπεύει κάθε λεπτομέρεια που εμφανίζεται στην πραγματική σύνταξη, αλλά αναπαριστά περισσότερο μία δομή σχετική με τις λεπτομέρειες του περιεχομένου. Για παράδειγμα λεπτομέρειες που αφορούν τη σύνταξη όπως παρενθέσεις, ερωτηματικά, δεν αναπαράγονται σε διαφορετικούς κόμβους σε ένα αφηρημένο συντακτικό δέντρο. Ομοίως μία συντακτική δομή όπως μία if-condition μπορεί να ορίζεται από έναν κόμβο με τρεις κλάδους. Αυτό είναι που διαχωρίζει τα συγκεκριμένα συντακτικά δέντρα (concrete syntax trees), που συχνά αναφέρονται και σαν δέντρα ανάλυσης (parse trees) από τα αφηρημένα συντακτικά δέντρα (ASTs). Τα parse trees τυπικά κτίζονται από έναν αναλυτή (parser) κατά τη διάρκεια της συμβολομετάφρασης του κώδικα.

Τα αφηρημένα συντακτικά δέντρα είναι δομές δεδομένων (data structures) τα οποία χρησιμοποιούνται στους μεταγλωττιστές (compilers) για να αναπαραστήσουν τη δομή του προγράμματος του κώδικα. Ένα AST (abstract syntax tree) είναι συχνά το αποτέλεσμα της συντακτικής ανάλυσης του μεταγλωττιστή. Συχνά χρησιμεύει ως μία ενδιάμεση αναπαράσταση του προγράμματος μέσω διάφορων σταδίων που απαιτεί ο μεταγλωττιστής και έχει ισχυρό αντίκτυπο στην τελική έξοδο του. Ένα AST έχει αρκετές ιδιότητες που βοηθούν να πάει ένα βήμα παρακάτω η διαδικασία της μεταγλώττισης. (compilation process).

- Ένα AST μπορεί να ενισχυθεί με πληροφορίες όπως οι ιδιότητες για κάθε στοιχείο που περιέχει. Τέτοιου είδους επεξεργασία είναι αδύνατο με τον πηγαίο κώδικα, καθώς θα σήμαινε αλλαγή του.
- Σε σύγκριση με τον πηγαίο κώδικα, το AST δεν περιλαμβάνει μη ουσιώδη σημεία στίξης όπως (αγκύλες, ερωτηματικά και παρενθέσεις).
- Ένα AST περιλαμβάνει συνήθως πληροφορίες για το πρόγραμμα, λόγω των διαδοχικών σταδίων ανάλυσης από τον μεταγλωττιστή. Για παράδειγμα, μπορεί να αποθηκεύσει τη θέση κάθε στοιχείου (element) στον πηγαίο κώδικα, επιτρέποντας στον μεταγλωττιστή (compiler) να εκτυπώνει χρήσιμα μηνύματα σφάλματος.

Παρακάτω παρουσιάζεται ένα παράδειγμα AST που ακολουθεί τον κώδικα του ευκλείδειου αλγόριθμου [13].

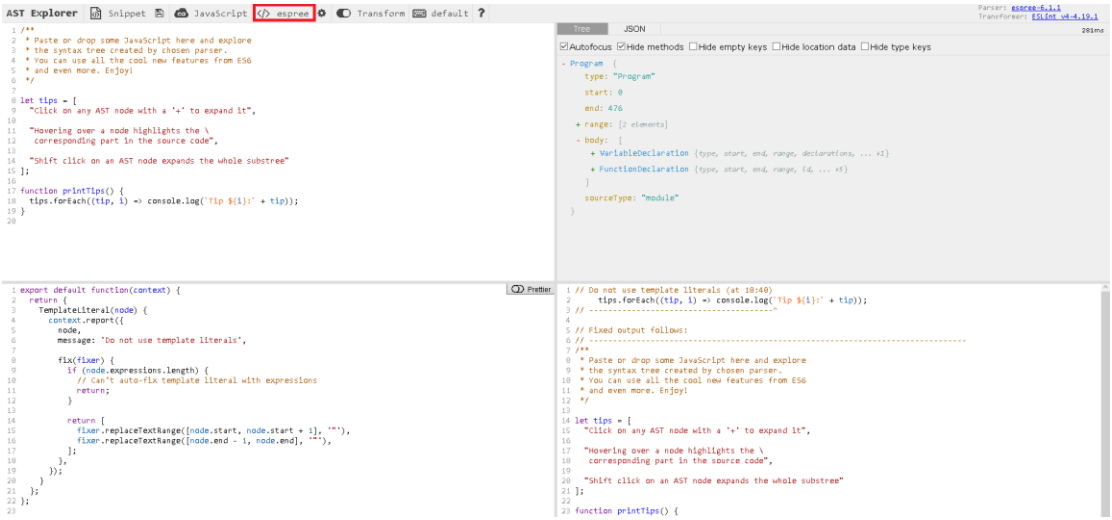



```
while b ≠ 0
  if a > b
    a := a - b
  else
    b := b - a
return a
```

Εικόνα 15 AST:Αλγόριθμος του Ευκλείδη

2.4.1 AST Explorer

Παρακάτω αναλύεται η χρήση του AST Explorer το οποίο αποτελεί ένα διαδικτυακό εργαλείο για την ανάλυση και την επεξεργασία των ASTs που δημιουργούνται από τον παραγόμενο κώδικα σε διάφορες γλώσσες προγραμματισμού, δίνοντας την επιλογή ανάμεσα σε 19 διαφορετικούς parsers, όπου ανάμεσα σε αυτούς βρίσκεται και ο **espree** που χρησιμοποιεί ο ESLint. Το AST explorer αποτελεί και το βασικό εργαλείο που χρησιμοποιήθηκε για την συγγραφή των κανόνων του plugin της παρούσας διπλωματικής.



Εικόνα 16 αναπαράσταση του AST

- Σε κόκκινο πλαίσιο φαίνεται ο espree ανάμεσα στους διαφορετικούς parser που μπορεί ο χρήστης να χρησιμοποιήσει.
- Στο διπλανό πλαίσιο φαίνεται η γλώσσα προγραμματισμού που έχει επιλεγεί και είναι η JavaScript.
- Παρατηρώντας το εργαλείο, φαίνεται ότι είναι χωρισμένο σε 4 διαφορετικά μέρη.
- Το πάνω αριστερά πλαίσιο, απεικονίζει τον πηγαίο κώδικα που θέλει ο προγραμματιστής να ελέγξει με έναν κανόνα.
- Το πάνω δεξιά πλαίσιο, απεικονίζει το AST, όπως αυτό δημιουργήθηκε ύστερα από την επιλογή του parser (στην παρούσα περίπτωση του espree). Ουσιαστικά έγινε η ανάλυση του κώδικα που αναφέρθηκε σε ένα συντακτικό δέντρο (abstract syntax). tree.
- Στο κάτω αριστερά πλαίσιο απεικονίζεται η υλοποίηση του κανόνα από τον προγραμματιστή, έτσι ώστε να εκτελεί την ενέργεια για την οποία σχεδιάστηκε.
- Στο κάτω δεξιά πλαίσιο απεικονίζεται η έξοδος που παράγει ο κανόνας, (warning ή error) μέσω ενός μηνύματος προς τον χρήστη. Επίσης σε περίπτωση λανθασμένης σχεδίασης, ενημερώνει τον χρήστη ότι ο κανόνας δεν ενεργοποιήθηκε, ύστερα από την ανάλυση που έγινε στον πηγαίο κώδικα.

ΚΕΦΑΛΑΙΟ 3

ΒΑΣΙΚΕΣ ΚΑΤΗΓΟΡΙΕΣ ΕΠΙΘΕΣΕΩΝ

Στο κεφάλαιο αυτό παρουσιάζονται αναλυτικά οι βασικές κατηγορίες επιθέσεων όσον αφορά τις διαδικτυακές εφαρμογές. Αναλύονται οι διάφορες κατηγορίες επιθέσεων, η "ζημία" που μπορεί να επιφέρουν καθώς και παραδείγματα που φανερώνουν τις διάφορες τεχνικές με τις οποίες ένας επίδοξος χάκερ μπορεί να εκμεταλλευτεί τις αδυναμίες μίας εφαρμογής βασιζόμενος σε αυτούς τους τύπους επιθέσεων. Τέλος προτείνονται διάφορες λύσεις για να αποτραπούν οι επιθέσεις.

3.1 Injection Attacks

Οι επιθέσεις διάχυσης (injection attacks) θεωρούνται οι πιο συχνές και επικίνδυνες μορφές επιθέσεων στις διαδικτυακές εφαρμογές. Τυπικά μία επίθεση τέτοιας μορφής συμβαίνει όταν μία εφαρμογή στέλνει μία επικίνδυνη είσοδο χρήστη (untrusted user input) σε έναν διερμηνέα ως μέρος μίας εντολής ή ενός query. Σε αυτή την κατηγορία εντάσσονται επιθέσεις όπως επιθέσεις διάχυσης εντολής (command Injections attacks), και επιθέσεις διάχυσης βάσης (database injection attacks) που περιλαμβάνει τις SQL και NoSQL επιθέσεις.

3.1.1 Command Injection Attacks

Χρησιμοποιώντας αυτή τη μορφή επίθεσης, ένας επιτιθέμενος μπορεί να εκτελέσει αυθαίρετες εντολές στο λειτουργικό σύστημα (host operating system) μίας ευάλωτης εφαρμογής. Αυτό δίνει τεράστιες ευκαιρίες και δυνατότητες στον επιτιθέμενο, όπως το να αποκτήσει πρόσβαση σε απαγορευμένα αρχεία και να εγκαταστήσει κακόβουλο λογισμικό με το οποίο μπορεί να αποκτήσει τον πλήρη έλεγχο του διακομιστή και του host network. Στο παρακάτω παράδειγμα παρουσιάζεται ο τρόπος με τον οποίο ένας επιτιθέμενος μπορεί να προκαλέσει ζημία μέσω αυτής της επίθεσης.

child processes:

Μηχανισμός Επίθεσης

Η child_process αποτελεί βασική ενότητα του Node.js (core module) και δίνει τη δυνατότητα στους προγραμματιστές του Node.js να επικαλεστούν OS εντολές (εντολές του λειτουργικού συστήματος) από τον κώδικα της εφαρμογής. Εξαιτίας του ονόματος της αλλά και της απλότητας της χρήσης της, η μέθοδος child_process.exec() συχνά χρησιμοποιείται για να εκτελέσει εντολές συστήματος. Η μέθοδος exec λαμβάνει 3 ορίσματα, μία εντολή σε μορφή string, ένα προαιρετικό αντικείμενο options, και μία συνάρτηση επανάκλησης (callback) όπως φαίνεται στην παρακάτω εικόνα

```
child_process.exec(command[, options][, callback])
```

Εικόνα 17 child_process.exec() method

Παρ' ότι η μέθοδος exec εκτελεί εντολές OS με ασύγχρονο τρόπο (nonblocking). συμβαδίζοντας τέλεια με τον ασύγχρονο προγραμματισμό του Node.js, η ευελιξία της να περνάει την εντολή ως string συχνά δημιουργεί κενά ασφάλειας. Αυτή είναι ακριβώς η

περίπτωση όπου μία είσοδος από έναν χρήστη χρησιμοποιείται για να κατασκευαστεί η εντολή. Παρακάτω φαίνεται η χρήση της child_process.exec η οποία επικαλείται την εντολή gzip, και ενώνει ένα δυναμικό file path προερχόμενο από το χρήστη, έτσι ώστε να κτίσει την gzip εντολή.

```
child_process.exec(
 'gzip ' + req.body.file_path,
 function (err, data) {
  console.log(data);
 });
```

Εικόνα 18 Η μέθοδος child_prcess.exec() με συνδυασμό string (string concatenation)

Για να εκμεταλλευτεί την αδυναμία διάχυσης (injection vulnerability) στον παραπάνω κώδικα, ένας επιτιθέμενος μπορεί να πληκτρολογήσει στην είσοδο (user-supplied input) το παρακάτω: **; rm -rf /**. Αυτό επιτρέπει στον επιτιθέμενο ουσιαστικά να βγει εκτός του ορίου της εντολής gzip και να εκτελέσει μία κακόβουλη εντολή η οποία διαγράφει όλα τα αρχεία στον διακομιστή. Σαν μέρος της εισόδου από τον χρήστη ο επιτιθέμενος μπορεί να συνδέσει πολλαπλές εντολές χρησιμοποιώντας χαρακτήρες όπως **;**,**&**,**&&**,**||**,**$()**,**<**,**>**, και **>>**. Ουσιαστικά η επίθεση συμβαίνει γιατί η μέθοδος exec δημιουργεί μία νέα bin/sh διαδικασία (process) και περνάει την εντολή από το όρισμα για εκτέλεση στο σύστημα shell. Το παραπάνω ισοδυναμεί με το να ανοίξει ο επιτιθέμενος ένα διαμεσολαβητή bash και να εκτελέσει ότι εντολές αυτός επιθυμεί, έχοντας μάλιστα τα ίδια ακριβώς δικαιώματα με την ευάλωτη εφαρμογή.

Αντιμετώπιση της Επίθεσης

Όταν είναι δυνατό, πρέπει να χρησιμοποιείται από το core module η child_process.execFile() μέθοδος ή η child_process.spawn() αντί της exec. Αντίθετα με την exec οι παραπάνω μέθοδοι δίνουν τη δυνατότητα στον προγραμματιστή να διαχωρίσει την εντολή και τα ορίσματα της. Στο παρακάτω παράδειγμα απεικονίζεται η εκτέλεση της gzip εντολής αυτή τη φορά με την execFile μέθοδο.

```
// Extract user input from request
var file_path = req.body.file_path;


// Execute gzip command
child_process.execFile(
 'gzip',
 [file_path],
 function (err, data) {
  console.log(data);
 });
```

Εικόνα 19 gzip με την execFile μέθοδο

Στην περίπτωση αυτή κάθε κακόβουλη εντολή η οποία συνδέεται με το file_path το οποίο αποτελεί είσοδο του χρήστη καταλήγει στο δεύτερο όρισμα της εντολής τύπου πίνακα. Κάθε κακόβουλη εντολή που προέρχεται από είσοδο του χρήστη απλώς αγνοείται ή προκαλεί συντακτικό λάθος και έτσι αποτυγχάνουν οι προσπάθειες για επίθεση εντολής διάχυσης. Παρότι οι μέθοδοι execFile και spawn είναι πιο ασφαλείς μέθοδοι από την exec, δεν μπορούν να αποτρέψουν πλήρως τις επιθέσεις διάχυσης. Ο τρόπος με τον οποίο μπορεί να πραγματοποιηθεί επίθεση σε αυτές τις μεθόδους είναι όταν οι προγραμματιστές επικαλούνται custom scripts τα οποία επεξεργάζονται τις εισόδους του χρήστη, ή εκτελούν συγκεκριμένες OS εντολές όπως **find**, **awk** και **sed** οι οποίες δίνουν τη δυνατότητα για την εισαγωγή κάποιων επιλογών (options) που ενεργοποιούν το file read/write.

Ο καλύτερος τρόπος προστασίας από αυτής της μορφής τις επιθέσεις είναι ο προγραμματιστής να επαληθεύει τα ορίσματα αυτών των εντολών καθώς και τις εισόδους του χρήστη χρησιμοποιώντας λίστες (whitelists) ή διάφορα πακέτα (modules) που παρέχουν αυτή τη δυνατότητα. Επίσης στην περίπτωση που μία τέτοια επίθεση γίνει με επιτυχία, αν ο επιτιθέμενος έχει δικαιώματα επιπέδου OS μπορεί πρακτικά να κάνει ότι αυτός επιθυμεί. Για αυτό μία καλή τακτική είναι να ακολουθείται από τον προγραμματιστή η μέθοδος των λιγότερων δικαιωμάτων (principle of least privilege) έτσι ώστε σε κάθε χρήστη να αποδίδονται όσα ακριβώς δικαιώματα χρειάζεται.

eval:

Η eval (το όνομα της προέρχεται από το evaluate) είναι μία συνάρτηση η οποία δέχεται σαν όρισμα ένα string και το “εκτελεί” σαν να ήταν κανονικός κώδικας γραμμένος σε JavaScript. Αυτό αυτόματα την καθιστά εύκολο στόχο για τους επιτιθέμενους. Γενικά έχει επικρατήσει η άποψη ότι η eval() σαν συνάρτηση πρέπει να αποφεύγεται (“eval is evil”) και ότι η παρουσία της αυτόματα σημαίνει κακή σχεδίαση από πλευράς προγραμματιστή.

Μηχανισμός Επίθεσης

Παρακάτω παρουσιάζεται ο τρόπος με τον οποίο ένας επιτιθέμενος μπορεί να εκματελλευτεί τη χρήση της eval και να εκτελέσει κακόβουλες εντολές ώστε να βλάψει την εφαρμογή. Όπως ήδη αναφέρθηκε η eval() λαμβάνει ένα όρισμα σε μορφή string και το εκτελεί σαν να ήταν κανονικός κώδικας JavaScript. Η περίπτωση που η eval μπορεί να γίνει καταστροφική για την εφαρμογή είναι όταν το όρισμα της προέρχεται από είσοδο του χρήστη. Εάν η είσοδος του χρήστη περνάει σαν όρισμα στην eval(user_input) χωρίς να ακολουθεί κάποια ενδιάμεση διαδικασία (όπως το να γίνεται έλεγχος της εισόδου και του περιεχομένου της) τότε ο επιτιθέμενος μπορεί να προκαλέσει τεράστια ζημία ανάλογα τη φαντασία του. Παρακάτω είναι μόνο μερικές από τις εισόδους που μπορεί να δώσει ένας επιτιθέμενος σε ένα πεδίο εισόδου (input field) και να προκαλέσει ανυπολόγιστη ζημία στην εφαρμογή. [14]

- **while(1):** Λαμβάνοντας σαν όρισμα την παραπάνω εντολή η eval θα εκτελέσει την εντολή. Αυτό θα έχει ως αποτέλεσμα μία επιτυχημένη επίθεση άρνησης εξυπηρέτησης (DOS-Denial Of Service, η οποία θα αναλυθεί παρακάτω) καθιστώντας την εφαρμογή ανήμπορη να λειτουργήσει. Με δεδομένο ότι οι εφαρμογές σε Node.js είναι μονού νήματος τότε τα αποτελέσματα είναι μοιραία.
- **process.exit():** Η παραπάνω εντολή όταν δοθεί σαν όρισμα στην eval θα έχει σαν αποτέλεσμα να τερματιστεί η εφαρμογή.

3.1.2 Database Injection Attacks

Με μία επιτυχή επίθεση διάχυσης βάσης (database injection attack) ένας επιτιθέμενος μπορεί να εκτελέσει κακόβουλες εντολές στην βάση για να κλέψει ευαίσθητα δεδομένα, να αλλοιώσει τα αποθηκευμένα δεδομένα, να εκτελέσει λειτουργίες στη βάση σαν διαχειριστής, και να αποκτήσει πρόσβαση σε περιεχόμενα φακέλων που βρίσκονται στο σύστημα της βάσης.

SQL injection: Μηχανισμός επίθεσης

Τα δυναμικά queries στην βάση τα οποία περιλαμβάνουν είσοδο που προέρχεται από χρήστη είναι ο πρώτος στόχος όσον αφορά τις SQL injection επιθέσεις. Όταν κακόβουλα δεδομένα συνδυάζονται σε ένα SQL query, ο SQL διερμηνέας αποτυγχάνει να διακρίνει τη διαφορά μεταξύ των εντολών και των δεδομένων εισόδου, με αποτέλεσμα την εκτέλεση των κακόβουλων δεδομένων ως εντολές SQL. Στο παρακάτω παράδειγμα απεικονίζεται ένα ευάλωτο SQL query, το οποίο πιστοποιεί ένα χρήστη.

```
connection.query(
 'SELECT * FROM accounts WHERE username ="'
 + req.body.username + '" AND password = "' + passwordHash + '"',
 function(err, rows, fields) {
  console.log("Result = " + JSON.stringify(rows));
});
```

Εικόνα 20 SQL vulnerable query

Το παραπάνω παράδειγμα απεικονίζει κώδικα ο οποίος δημιουργεί δυναμικά ένα SQL query συνδυάζοντας την είσοδο από το χρήστη (username). Για να επωφεληθεί από αυτή την αδυναμία ένας επιτιθέμενος μπορεί να πληκτρολογήσει στην είσοδο **admin'--**, το οποίο θα οδηγήσει στην εκτέλεση του query όπως φαίνεται παρακάτω.

```
SELECT * FROM accounts WHERE username = 'admin'
```

Εικόνα 21 Resultant query with username as admin'--

Η κακόβουλη είσοδος από τον επιτιθέμενο, του δίνει τη δυνατότητα να μην υποβάλει τον σωστό κωδικό επειδή το κομμάτι μετά από το **- -** σε ένα SQL query ερμηνεύεται ως σχόλια με αποτέλεσμα να παρακάμπτεται έτσι η σύγκριση των κωδικών πρόσβασης. Ένα ακόμη SQL injection, μάλιστα ακόμη πιο καταστροφικό είναι όταν η βάση υποστηρίζει την εκτέλεση πολλαπλών δηλώσεων όταν αυτές οι δηλώσεις ακολουθούνται από ερωτηματικό. Στο παρακάτω παράδειγμα παρουσιάζεται η περίπτωση όπου ένας επιτιθέμενος τοποθετεί σαν είσοδο το string **admin'; DELETE FROM accounts; --** σαν όνομα χρήστη, με αποτέλεσμα το query να είναι ισοδύναμο με δύο δηλώσεις, όπως φαίνεται παρακάτω.

```
SELECT * FROM accounts WHERE username = 'admin';
DELETE FROM accounts;
```

Εικόνα 22 query που προκύπτει από τον συνδυασμό πολλαπλών δηλώσεων

Το παραπάνω query έχει σαν αποτέλεσμα να αφαιρεθούν όλοι οι λογαριασμοί των χρηστών από τη βάση.

Αντιμετώπιση της επίθεσης

Το SQL injection είναι εύκολα αντιμετωπίσιμο. Η πιο συχνή λύση θεωρούνται τα παραμετροποιημένα queries (parameterized queries), τα οποία είναι πολύ αποτελεσματικά στην αντιμετώπιση της επίθεσης. Αυτός ο τύπος των queries εμποδίζει τον επιτιθέμενο να ξεγελάσει των διερμηνέα της SQL. Το παρακάτω παράδειγμα απεικονίζει αυτή τη μέθοδο

```
var mysql = require('mysql2');
var bcrypt = require('bcrypt-nodejs');


// Prepare query parameters
var username = req.body.username;
var passwordHash = bcrypt.hashSync(req.body.password,
  bcrypt.genSaltSync());


// Make connection to the MySQL database
var connection = mysql.createConnection({
 host : 'localhost',
 user : 'db_user',
 password : 'secret',
 database : 'node_app_db'
});
connection.connect();


// Execute prepared statement with parameterized user inputs
var query = 'SELECT * FROM accounts WHERE username=? AND password=?';
connection.query(query, [username, passwordHash],
  function (err, rows, fields) {
 console.log("Results = " + JSON.stringify(rows));
});

connection.end();
```

Εικόνα 23 Παραμετροποιημένα queries για την αποτροπή του SQL injection

Στο παραπάνω παράδειγμα εάν ο επιτιθέμενος επιχειρήσει να θέσει σαν είσοδο το **admin' --**, θα έχει ως αποτέλεσμα το query να αναζητήσει ένα όνομα χρήστη με ακριβώς αυτό το όνομα και όχι με το admin όπως είδαμε στο προηγούμενο παράδειγμα. Επίσης μία άλλη αποτελεσματική μέθοδος είναι η χρήση των Whitelists, δηλαδή των έλεγχο όλων των εισόδων μέσω λιστών για το τι επιτρέπεται να περάσει και τι όχι.

3.2 Cross-Site Scripting

Το XSS (Cross-site Scripting) [15] θεωρείται ίσως η πιο διαδεδομένη αδυναμία στις διαδικτυακές εφαρμογές. Οι επιτιθέμενοι χρησιμοποιούν το XSS με διαφορετικούς τρόπους, όπως η κατάχρηση του session ενός χρήστη, η μεταφορά του χρήστη σε μία κακόβουλη σελίδα, η καταγραφή των πληκτρολογίων (logging keystrokes), ο χειρισμός των περιεχομένων της σελίδας, η μεταφορά και η δημοσίευση δεδομένων σε κάποιον απομακρυσμένο διακομιστή ή η εγκατάσταση κάποιου κακόβουλου λογισμικού στο μηχάνημα ενός χρήστη.

Μηχανισμοί επίθεσης

Ο μηχανισμός κλειδί πίσω από τις επιθέσεις XSS, είναι η διάχυση περιεχομένου (content injection), όπου ένας επιτιθέμενος εισάγει κακόβουλο περιεχόμενο Javascript σε μία σελίδα. Οι επιτιθέμενοι εισάγουν συνήθως μόνο ένα μικρό κομμάτι του αρχικού script, το οποίο όταν εκτελείται κατεβάζει και πρόσθετα απαιτούμενα scripts. Ανάλογα με το αν η εφαρμογή-στόχος αποθηκεύει το εχθρικό κακόβουλο λογισμικό υπάρχουν 2 κύριοι μηχανισμοί επίθεσης XSS, οι ανακλώμενοι (reflected) και αυτοί που αποθηκεύονται (stored). Παρακάτω εξετάζονται λεπτομερώς αυτοί οι μηχανισμοί.

Reflected XSS

Σε αυτή την περίπτωση ο επιτιθέμενος παρέχει έναν σύνδεσμο στον χρήστη-θύμα με κακόβουλο περιεχόμενο ενσωματωμένο στις παραμέτρους αιτήματος (request parameters). Όταν ένας χρήστης-θύμα κάνει κλικ σε αυτόν τον σύνδεσμο, ένα αίτημα πηγαίνει στο διακομιστή της εφαρμογής (server), όπου ένα ελάττωμα στον κώδικα της εφαρμογής έχει ως αποτέλεσμα να συμπεριλαμβάνεται το κακόβουλο περιεχόμενο στο αίτημα (request), σαν να είναι στο σώμα απάντησης (response body). Όταν το περιεχόμενο αυτής της απόκρισης φορτώνεται στο πρόγραμμα περιήγησης του χρήστη, ο κακόβουλος κώδικας εκτελείται προκαλώντας επιτυχή επίθεση XSS. Για εφαρμογές που δημιουργούν περιεχόμενο σελίδας (page content) στην πλευρά του πελάτη (client-side), το XSS μπορεί να εκδηλωθεί αν ο μη ασφαλής κώδικας JavaScript μεταφέρει το κακόβουλο περιεχόμενο από το URL που εισέρχεται από τον εισβολέα και το εισάγει στη διαδικτυακή σελίδα.Το Reflected XSS είναι συνήθως μια επίθεση ένα προς ένα, επειδή ένας εισβολέας πρέπει να στείλει τον κακόβουλο σύνδεσμο σε κάθε χρήστη. Επομένως, επηρεάζει μόνο εκείνους τους χρήστες που το κάνουν κλικ. Έτσι, έχει σχετικά λιγότερες πιθανότητες επιτυχίας σε σύγκριση με την αποθηκευμένη επίθεση XSS η οποία εξετάζεται στη συνέχεια.

Stored XSS

Σε μία επίθεση stored XSS, το κακόβουλο περιεχόμενο που εισέρχεται από έναν επιτιθέμενο αποθηκεύεται στην εφαρμογή. Μετά από αυτό, κάθε χρήστης που επισκέπτεται την σελίδα της εφαρμογής πέφτει ουσιαστικά θύμα της επίθεσης XSS. Για παράδειγμα, ας σκεφτούμε τα σχόλια σε ένα blog post. Εάν τα σχόλια με κακόβουλο περιεχόμενο που θα ποσταριστούν από τον επιτιθέμενο δεν έχουν ελεγχθεί (validate and sanitized) πριν αποθηκευτούν στη βάση δεδομένων, κάθε φορά που το ποστ διαβάζεται από ένα χρήστη, ο κακόβουλος κώδικας που έχει γίνει inject στα σχόλια εκτελείται. Αυτό έχει ως αποτέλεσμα η stored XSS να θεωρείται ένας τύπος επίθεσης που μπορεί να προκαλέσει σοβαρή ζημία.

Τρόποι Αντιμετώπισης του XSS

Η περίπτωση XSS εμφανίζεται όταν ο κώδικας της εφαρμογής λαμβάνει μη έμπιστα δεδομένα τα οποία προέρχονται από τον χρήστη και τα καθιστά στο πρόγραμμα περιήγησης (browser) χωρίς να γίνεται η σωστή επικύρωση (validation) ή το σωστό escaping. Σε αυτή τη περίπτωση το πρόγραμμα περιήγησης αδυνατεί να διακρίνει τα script που έχουν προστεθεί από τον προγραμματιστή με αυτά που έχουν προστεθεί από τον επιτιθέμενο.

CSP Header: Η προσθήκη της κεφαλίδας CSP (Content Security Policy Header) είναι ο πιο εύκολος και πιο εύρωστος τρόπος να περιορίσει κανείς τις επιθέσεις XSS. Η συγκεκριμένη κεφαλίδα επιτρέπει στους προγραμματιστές να ορίσουν μία λίστα (whitelist) με τους τομείς (domains) από τα οποίους το πρόγραμμα περιήγησης επιτρέπεται να λαμβάνει περιεχόμενο. Έτσι το πρόγραμμα περιήγησης μπλοκάρει κάθε script το οποίο εισέρχεται από έναν επιτιθέμενο επειδή αυτά τα script δεν προέρχονται από τους τομείς που έχουν οριστεί από την λίστα (whitelist). Εκτός από τα JavaScript αρχεία, η συγκεκριμένη κεφαλίδα μπορεί να κάνει whitelist πηγές για εικόνες, CSS fonts, ή ακόμη και βίντεο. Παρακάτω παρουσιάζεται ένα παράδειγμα, που απεικονίζει τον τρόπο να με τον οποίο μπορεί να γίνει προσθήκη CSP Header σε Node εφαρμογές μέσω του πακέτου helmet.

```
var policy = {
  defaultPolicy: {
    "default-src": ["'self'"], ❶
    "img-src": ["static.example.com"] ❷
  }
}
helmet.csp.policy(policy);
```

Εικόνα 24 CSP response header χρησιμοποιώντας Helmet

Στο παραπάνω παράδειγμα θέτοντας το **default-src** σε **self** επιτρέπουμε στο πρόγραμμα περιήγησης να κατεβάσει κώδικα JavaScript μόνο από το ίδιο domain από το οποίο φορτώθηκε η σελίδα. Επίσης στο πεδίο **img-src**, προσθέτουμε μία εξαίρεση για εικόνες από ένα επιπρόσθετο domain, το static.example.com

3.3 Broken Authentication and Session Management

Όπως δηλώνεται και από το όνομα τους, οι αδυναμίες σφάλματος ελέγχου ταυτότητας και διαχείρισης του session στοχεύουν στις ελλείψεις όσον αφορά την εξακρίβωση της ταυτότητας του χρήστη αλλά και την υλοποίηση όσον αφορά τη διαχείριση του session. Ένας επιτιθέμενος μπορεί να εκμεταλλευτεί αυτές τις αδυναμίες ώστε να παραστήσει άλλους χρήστες και να εκτελέσει κακόβουλες ή μη επιθυμητές λειτουργίες από μέρους τους.

Τα τυπικά σενάρια επίθεσης περιλαμβάνουν τους αδύναμους κωδικούς λογαριασμών, τους κωδικούς που υποκλέπτονται από παραβιάσεις της βάσης δεδομένων, session ID’s χρηστών που υποκλέπτονται από το πρόγραμμα περιήγησης ενός χρήστη ή από την διαδικτυακή επικοινωνία, ή ακόμα και bugs σε χαρακτηριστικά διαχείρισης κωδικών όπως είναι η ανάκτηση ενός κωδικού ή η αλλαγή του. Παρακάτω παρουσιάζονται κάποιοι μηχανισμοί επίθεσης που στοχεύουν στο μηχανισμό ταυτοποίησης (authentication mechanism).

Password Cracking

Η παραβίαση του κωδικού πρόσβασης είναι ισοδύναμη με έναν διαρρήκτη που παραβιάζει την κλειδαριά στην μπροστινή πόρτα ενός σπιτιού για να μπει μέσα. Η παραβίαση του κωδικού πρόσβασης δεν είναι πάντα τόσο δύσκολη όσο θα περίμενε κανείς, δεδομένου των διαφόρων εργαλείων και τεχνικών που είναι διαθέσιμα για την αυτοματοποίηση της διαδικασίας. Εγγυημένη αλλά χρονοβόρα μέθοδος για να σπάσει κάποιος κωδικός πρόσβασης είναι η επίθεση βίαιης δύναμης (brute-force attack). Λειτουργεί περνώντας μέσα από κάθε πιθανό συνδυασμό, που κυμαίνεται από ένα χαρακτήρα έως χαρακτήρες N, και προσπαθεί να συνδεθεί. Εργαλεία όπως ο John the Ripper ή ο Brutus χρησιμοποιούνται συνήθως για να το επιτύχουν. Επειδή η τεχνική brute-force είναι αργή, μια πολύ ταχύτερη έκδοση είναι η επίθεση dictionary. Σε αντίθεση με τη brute-force attack, μια επίθεση dictionary δεν απαριθμείται με κάθε δυνατό χαρακτήρα. Αντί αυτού, περνάει μέσα από λέξεις σε ένα συγκεκριμένο λεξικό ή λίστα λέξεων για να ελέγξει εάν λειτουργεί ως κωδικός πρόσβασης. Εργαλεία όπως το Hydra μπορούν όχι μόνο να αυτοματοποιήσουν αυτή τη διαδικασία, αλλά και να μορφοποιήσουν τους χαρακτήρες με λέξεις δοκιμάζοντας συνδυασμούς αυτών των λέξεων.

Securing Session Management

Επειδή το HTTP είναι ένα stateless πρωτόκολλο, για να θυμάται ένα χρήστη, ο διακομιστής διατηρεί ένα αναγνωριστικό λειτουργίας (session identifier ή SID) που μεταδίδεται μεταξύ του πελάτη και του διακομιστή. Αυτό το SID είναι ισοδύναμο με τη ταυτότητα του χρήστη για τον διακομιστή (server). Ως εκ τούτου, αυτό αποτελεί πρωταρχικό στόχο για τους επιτιθέμενους οι οποίοι προσπαθούν να μιμηθούν άλλους χρήστες. Συνεπώς η διαχείριση του session θεωρείται ένα κρίσιμο κομμάτι για την ασφάλεια της εφαρμογής. Απαιτεί από τους προγραμματιστές να φροντίζουν για την προστασία της ταυτότητας του session (session id), της ασφαλούς αποθήκευσης των διαπιστευτηρίων των χρηστών, της διάρκειας των session, και την προστασία των κρίσιμων δεδομένων του session κατά την μεταφορά.

<u>Μηχανισμοί επίθεσης</u>

Σενάριο 1: Ένας επιτιθέμενος καταφέρνει να αποκτήσει πρόσβαση στο σύστημα της βάσης δεδομένων. Οι κωδικοί των χρηστών δεν έχουν γίνει hash με τον σωστό τρόπο, με αποτέλεσμα να διαρρεύσουν τελικώς όλοι οι κωδικοί πρόσβασης των χρηστών στον επιτιθέμενο.

Σενάριο 2: Ο επιτιθέμενος λειτουργεί σαν man-in-the-middle (MITM) και αποκτά την ταυτότητα του session του χρήστη (session id) από την κίνηση στο δίκτυο (network traffic). Έπειτα χρησιμοποιεί το πιστοποιημένο session id για να συνδεθεί στην εφαρμογή χωρίς να χρειαστεί να βάλει το όνομα του χρήστη και τον κωδικό.

Παρακάτω ακολουθούν παραδείγματα για το πως θα μπορούσαν να αποφευχθούν τα παραπάνω σενάρια επίθεσης.

Προστασία διαπιστευτηρίων χρήστη

Ο κωδικός αποθηκεύεται σε απλό κείμενο στη βάση δεδομένων.

```
// Create user document
var user = {
    userName: userName,
    firstName: firstName,
    lastName: lastName,
    password: password //received from request param
};
```

Εικόνα 25 Αποθηκευμένος κωδικός χωρίς κρυπτογράφηση

Για να γίνει με πιο ασφαλή τρόπο, ο κωδικός θα πρέπει να κρυπτογραφηθεί (one way encryption using salt hashing).

```
// Generate password hash
var salt = bcrypt.genSaltSync();
var passwordHash = bcrypt.hashSync(password, salt);

// Create user document
var user = {
    userName: userName,
    firstName: firstName,
    lastName: lastName,
    password: passwordHash
};
```

Εικόνα 26 One way encryption using salt hashing

Προστασία των cookies κατά την μεταφορά

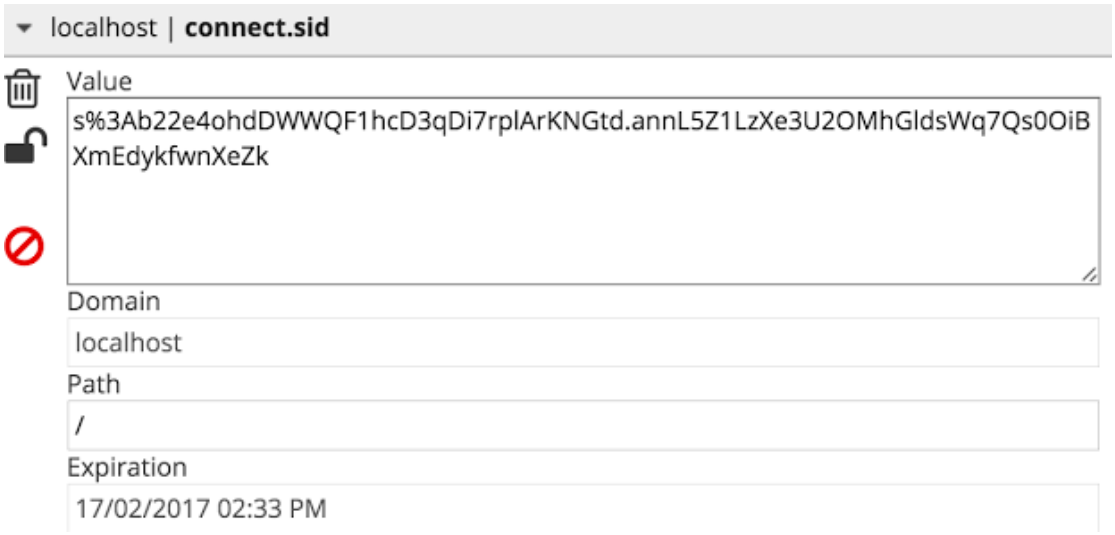
Τα cookies πρέπει να μην είναι προσβάσιμα από script τρίτων ώστε να προστατεύονται από XSS επιθέσεις. Επίσης τα cookies πρέπει να μην στέλνονται σε μη ασφαλείς HTTP συνδέσεις. Για να καλυφθούν οι παραπάνω περιπτώσεις στο πακέτο express-session του express υπάρχουν τα flags **HTTPOnly** και **secure**. Θέτοντας το πρώτο flag ως αληθές (true) αποτρέπεται η πρόσβαση των cookies από script τρίτων. Θέτοντας το δεύτερο flag ως αληθές (true) η εφαρμογή χρησιμοποιεί μόνο HTTPS ασφαλείς συνδέσεις. [16]

```
app.use(express.session({
    secret: "s3Cur3",
    cookie: {
        httpOnly: true,
        secure: true
    }
}));
```

Εικόνα 27HTTPOnly and secure flags in express-session

Ανωνυμία του sessionID που χρησιμοποιήθηκε

Οι τεχνολογίες που χρησιμοποιεί μία εφαρμογή πρέπει να μην αποκαλύπτονται. Εάν ένας επιτιθέμενος γνωρίζει τι είδους τεχνολογία χρησιμοποιεί η εφαρμογή, μειώνει κατά πολύ το εύρος των πιθανών αδύναμων σημείων στην εφαρμογή. Ένας τρόπος να αποκαλυφθούν εσωτερικές διαδικασίες που υλοποιήθηκαν στην εφαρμογή είναι το όνομα του session cookie. Παρακάτω φαίνεται ένα παράδειγμα με το όνομα ενός session cookie.



Εικόνα 28 Default name of express session cookie

Στην παραπάνω εικόνα το όνομα του session cookie είναι το **connect.sid**. Αυτό είναι το εξορισμού όνομα που θέτει το express framework στην εφαρμογή. Κάποιος που μπορεί να δει αυτό το cookie μπορεί αμέσως να αναγνωρίσει ότι αυτή η εφαρμογή είναι σε Node.js, εκτός εάν ο προγραμματιστής της εφαρμογής έχει μεταμφιέσει το όνομα του cookie. Συνεπώς για να μη γίνεται γνωστή η τεχνολογία της εφαρμογής ο προγραμματιστής πρέπει να αλλάζει το εξορισμού όνομα του session-cookie μέσω τις ιδιότητας **name** στο express-session.

```
var express = require('express');
var session = require('express-session');
var app = express();

app.use(session({
    name: 'SESS_ID',
    secret: '^#$5sX(Hf6KUo!#65^',
    resave: false,
    saveUninitialized: true,
    cookie: {
        secure: true,
        httpOnly: true
    }
}));
```

Εικόνα 29 property name in express-session cookie

Στην παραπάνω εικόνα φαίνεται η αλλαγή του εξορισμού ονόματος του session-cookie μέσω της ιδιότητας name.

3.4 Unvalidated Redirects and Forwards

Παρότι η συγκεκριμένη κατηγορία επιθέσεων δεν ανήκει στις πιο δημοφιλείς (σύμφωνα με το OWASP top 10 risks), αποτελεί την αγαπημένη τακτική για το phising. Με τον όρο phising δηλώνεται η ενέργεια εξαπάτησης χρηστών του διαδικτύου, κατά την οποία ο επιτιθέμενος υποδύεται μία αξιόπιστη οντότητα, καταχρώμενος την άγνοια του χρήστη, με σκοπό την αθέμιτη απόκτηση ευαίσθητων δεδομένων [17]. Σύμφωνα με την έρευνα που διεξάχθηκε από την Verizon [18] πάνω στο phishing, το 30% των παραληπτών ανοίγει τα μηνύματα ηλεκτρονικού ψαρέματος και το 12% κλικ άρει τους κακόβουλους συνδέσμους. Αυτό αποκαλύπτει τον βαθμό επιτυχίας των επιθέσεων ψαρέματος (phishing).

Μηχανισμοί Επίθεσης

Redirects: Συνήθως μία ανακατεύθυνση (redirect) περιλαμβάνει ένα διακομιστή που στέλνει μία απάντηση HTTP 302 στο πρόγραμμα περιήγησης με διαφορετική διεύθυνση URL προορισμού. Κατά τη λήψη του το πρόγραμμα περιήγησης κάνει ένα δεύτερο αίτημα στην παρεχόμενη διεύθυνση URL προορισμού.

Forwards: Οι προωθήσεις δεν φαίνονται στο πρόγραμμα περιήγησης. Ο διακομιστής μεταφέρει εσωτερικά ένα αίτημα σε ένα διαφορετικό πόρο (resource) του διακομιστή.

Και τα δύο μπορεί να γίνουν πολύ επικίνδυνα εάν χρησιμοποιείται είσοδος από το χρήση σαν παράμετρος για να αποφασιστεί ο προορισμός.

Σενάριο 1: Ανακατεύθυνση σε μία εξωτερική ιστοσελίδα.
Είναι πολύ συχνή η περίπτωση να ανακατευθύνονται οι χρήστες σε διαφορετικές ιστοσελίδες μετά το πάτημα ενός κουμπιού. Για παράδειγμα στον παρακάτω σύνδεσμο.

```
http://trustedbank.com/offers?redirect=
http://affiliatesite.com
```

Εικόνα 30 Παράδειγμα URL ανακατεύθυνσης

Όταν ο χρήστης κλικ άρει στο σύνδεσμο και το αίτημα (request) φτάσει στον διακομιστή όπως φαίνεται στο παρακάτω παράδειγμα, ο κώδικας της εφαρμογής απλώς παίρνει το προβλεπόμενο URL από την παράμετρο και εκτελεί μία ανακατεύθυνση στέλνοντας μία απάντηση HTTP 302 στο πρόγραμμα περιήγησης.

```
var app = express();
app.get("/offers", function(req, res, next) {
 // Some logic to track user activity here ...
 // Once done, take the user to the destination site
 return res.redirect(req.query.redirect);
});
```

Εικόνα 31 Μη ασφαλής υλοποίηση ανακατεύθυνσης URL

Τέτοιες ανακατευθύνσεις συχνά αναφέρονται και ως ανοιχτές ανακατευθύνσεις, και συχνά παρέχουν ένα τρόπο στους επιτιθέμενους να εφαρμόσουν phishing επιθέσεις. Ένας

επιτιθέμενος μπορεί να δημιουργήσει ένα σύνδεσμο (link), βάζοντας την παράμετρο ανακατεύθυνσης να δείχνει σε μία κακόβουλη εξωτερική ιστοσελίδα, όπως το παρακάτω URL.

```
http://trustedbanksite.com/offers?redirect=http://evilsite.com
```

Εικόνα 32 Παράδειγμα URL ανακατεύθυνσης σε κακόβουλη ιστοσελίδα

Μάλιστα για να το κάνει ακόμα πιο σύνθετο ο επιτιθέμενος μπορεί να κωδικοποιήσει τη τιμή του παρόντος URL. Στο παρακάτω URL η παράμετρος ανακατεύθυνσης είναι ισοδύναμη με την κωδικοποιημένη τιμή για το http://evilsite.com.

```
http://trustedbank.com/offers?redirect=
http:%2F%2%65%76%69%6C%73%69%74%65%2E%63%6F%6D
```

Εικόνα 33 Παράδειγμα URL ανακατεύθυνσης σε κακόβουλη ιστοσελίδα με κωδικοποιημένη τιμή

Καθώς το domain όνομα στο URL http://trustedbanksite.com είναι γνωστό και οικείο, οι χρήστες συνήθως κλικ άρουν χωρίς πριν να ερευνήσουν το μονοπάτι ανακατεύθυνσης (redirect path). Εφόσον το θύμα βρεθεί στην κακόβουλη σελίδα, ένας επιτιθέμενος έχει πολλές επιλογές για να προκαλέσει ζημία, συμπεριλαμβάνοντας και την περίπτωση να κατεβάσει και να εγκαταστήσει κακόβουλο λογισμικό στο μηχάνημα του χρήστη.

Σενάριο 2: Ανακατεύθυνση σε μία ψεύτικη σελίδα σύνδεσης

Όταν ένας χρήστης κάνει κλικ σε έναν σύνδεσμο μετά το χρονικό διάστημα σύνδεσης (session timeout), ο χρήστης υποχρεούται να επιβεβαιώσει πρώτα την ταυτότητά του πριν μπορέσει να αποκτήσει πρόσβαση στην απαιτούμενη σελίδα. Για μια τέτοια εφαρμογή, ο διακομιστής πρέπει να θυμάται τη σελίδα που ο χρήστης ήθελε να έχει πρόσβαση και να τη παρουσιάσει μετά την επιτυχή πιστοποίηση. Ένας πιθανός τρόπος για την υλοποίησή του είναι να συμπεριληφθεί μια διαδρομή στην ζητούμενη σελίδα σε μια query παράμετρο της διεύθυνσης URL της αίτησης σύνδεσης και της αίτησης υποβολής φόρμας σύνδεσης. Αφού ο χρήστης συνδεθεί με επιτυχία, η υπηρεσία σύνδεσης στον διακομιστή χρησιμοποιεί τη διαδρομή που καθορίζεται από την παράμετρο ερωτήματος για την ανακατεύθυνση του χρήστη. Αν και αυτό είναι ένα χαρακτηριστικό γνώρισμα ευκολίας για τους χρήστες και σχετικά απλό να εφαρμοστεί στο διακομιστή, οι εισβολείς μπορούν να εκμεταλλευτούν τέτοιες ανακατευθύνσεις υποκαθιστώντας την τιμή παραμέτρου αίτησης σε οποιαδήποτε εξωτερική κακόβουλη ιστοσελίδα, όπως φαίνεται στην ακόλουθη διεύθυνση URL:

```
http://trustedsite.com/login?onautheticate=http://evilsite.com
```

Εικόνα 34 Ανακατεύθυνση σε κακόβουλη ιστοσελίδα μέσω παραμέτρου αίτησης

Ένας εισβολέας μπορεί στη συνέχεια να δημοσιεύσει μία τέτοια ειδικά σχεδιασμένη διεύθυνση URL χρησιμοποιώντας ένα ηλεκτρονικό ταχυδρομείο ηλεκτρονικού "ψαρέματος" (phishing email) ή σε κοινωνικά μέσα (media). Στον προορισμό ανακατεύθυνσης, ένας εισβολέας μπορεί να φιλοξενήσει μια ψεύτικη σελίδα σύνδεσης που ταιριάζει ακριβώς με εκείνη του trustedsite.com στο προηγούμενο παράδειγμα. Μετά την επιτυχή σύνδεση στο trustedsite.com, οι χρήστες ανακατευθύνονται στην κατασκευασμένη σελίδα σύνδεσης. Είναι πολύ πιθανό οι χρήστες να εισάγουν και πάλι τα διαπιστευτήριά τους σε αυτήν τη

σελίδα, πιστεύοντας ότι κάτι πήγε στραβά την πρώτη φορά και μη παρατηρώντας την αλλαγή στο όνομα domain στη γραμμή διευθύνσεων. Έτσι, οι επιτιθέμενοι θα μπορούσαν να καταχραστούν τέτοιες ανακατευθύνσεις για τη συλλογή διαπιστευτηρίων του χρήστη.

3.5 DOS (Denial Of Service)

Μία επίθεση άρνησης εξυπηρέτησης (Denial Of Service) είναι μία επίθεση που αποσκοπεί να τερματίσει τη λειτουργία μίας μηχανής ή ενός δικτύου, καθιστώντας την απρόσιτη για τους χρήστες τους οποίους προορίζεται. Οι επιθέσεις DOS επιτυγχάνουν το σκοπό τους πλημμυρίζοντας τον στόχο με κίνηση (traffic), ή στέλνοντας μία πληροφορία που θα πυροδοτήσει ένα crash. Και στις δύο περιπτώσεις οι επιθέσεις DOS στερούν από τους νόμιμους χρήστες τις υπηρεσίες ή τους πόρους που αναμένουν.

Μία συγκεκριμένη κατηγορία επιθέσεων DOS είναι οι ReDos (regular expression denial of service) επιθέσεις. Οι συγκεκριμένες επιθέσεις είναι επιθέσεις όπου εκμεταλλεύονται το γεγονός ότι οι υπολογισμοί κανονικών εκφράσεων μπορεί να φτάσουν ακραίες καταστάσεις με αποτέλεσμα η εφαρμογή να δουλεύει πολύ αργά ή ακόμα και να τερματίσει τη λειτουργία της μη μπορώντας να εξυπηρετήσει άλλα αιτήματα. Οι συγκεκριμένες επιθέσεις σχετίζονται κυρίως με τις εισόδους από τον χρήστη. Ένας επιτιθέμενος μπορεί μέσω ενός πεδίου εισόδου να βλάψει ένα πρόγραμμα χρησιμοποιώντας τις κανονικές εκφράσεις ώστε να φτάσει σε αυτές τις ακραίες καταστάσεις υπολογισμού και να τερματίσει το πρόγραμμα.

Μηχανισμοί επίθεσης

Όταν μία μη έμπιστη είσοδος εκτελείται σαν κανονική έκφραση, μπορεί να εκμεταλλευτεί ευάλωτα μονοπάτια τρέχοντας πολύ μεγάλους υπολογισμούς ώστε να ταιριάξει το εισερχόμενο string. Για το Node.js συγκεκριμένα, αυτό είναι πολύ σημαντικό εξαιτίας της single-threaded event-loop αρχιτεκτονικής του, το οποίο σημαίνει ότι η κύρια λειτουργία του Node.js εμποδίζεται από το να εξυπηρετήσει άλλα αιτήματα.

3.6 CSRF (Cross-Site Request Forgery) attacks

Η αδυναμία Cross-Site Request Forgery (CSRF) δίνει τη δυνατότητα σε έναν εισβολέα να εκμεταλλευτεί το επικυρωμένο session ενός χρήστη και να τον εξαπατήσει στο να εκτελέσει οποιαδήποτε αλλαγή κατάστασης λειτουργίας για την οποία ο χρήστης-θύμα είναι εξουσιοδοτημένος να κάνει. Ουσιαστικά μία επίθεση CSRF εξαναγκάζει τον περιηγητή του χρήστη να στείλει ένα κατασκευασμένο HTTP αίτημα (request), το οποίο συμπεριλαμβάνει και το session cookie του θύματος και οποιαδήποτε άλλη πληροφορία σχετική με την ταυτοποίηση, σε μία ευάλωτη διαδικτυακή εφαρμογή. Αυτό επιτρέπει στον επιτιθέμενο να εξαναγκάσει τον περιηγητή του θύματος να δημιουργήσει αιτήματα που αντιμετωπίζει η ευάλωτη εφαρμογή ως αιτήματα από το χρήστη.

Μηχανισμοί επίθεσης

Όπως δηλώνεται από το όνομα, ένας επιτιθέμενος προσελκύει έναν χρήστη να επισκεφτεί ένα cross-site, δηλαδή μία κακόβουλη εξωτερική σελίδα, η οποία συνήθως περιέχει μία κατασκευασμένη φόρμα HTML με κρυφά πεδία που ταιριάζουν με την ιστοσελίδα που παρέχεται από την ίδια την εφαρμογή. Στο παρακάτω παράδειγμα φαίνεται μία κακόβουλη ιστοσελίδα που στοχεύει ένα χρήστη διαχειριστή μίας ευάλωτης εφαρμογής. Επιτρέπει στον χρήστη να κάνει κλικ σε ένα κουμπί, το οποίο ενεργοποιεί την υποβολή της φόρμας. Επειδή

η απάντηση του διακομιστή πηγαίνει σε ένα κρυφό iframe, ο χρήστης θύμα δεν έχει καμία ένδειξη σχετικά με τη συναλλαγή.

```
<form method="POST" action="http://example.com/admin/changeRole"
   target="iframeHidden">
  <h1> You are about to win a brand new iPhone!</h1>
  <h2> Click on the win button to claim it...</h2>
  <input type="hidden" name="accountId" value="67887"/>
  <input type="hidden" name="newRoleId" value="1"/>
  <input type="submit" value="Win !!!"/>
</form>
<iframe name="iframeHidden" width="1" height="1"/>
```

Εικόνα 35 Κακόβουλη ιστοσελίδα για CSRF επίθεση

Στο παραπάνω παράδειγμα όταν ένας χρήστης διαχειριστής ο οποίος έχει εξουσιοδότηση για να μεταβάλει τον ρόλο των άλλων χρηστών, κάνει κλικ στο κουμπί, ο χρήστης με το αναγνωριστικό που καθορίστηκε στο κρυφό πεδίο του λογαριασμού λαμβάνει έναν προνομιούχο ρόλο με roleId 1. Αυτή η επίθεση είναι πιθανή, διότι μαζί με το αίτημα (request) ο περιηγητής του χρήστη στέλνει αυτόματα τα cookies, που έχουν οριστεί αρχικά από την εφαρμογή, συμπεριλαμβανομένου του session cookie. Επειδή ένα session cookie αποτελεί τον μόνο τρόπο με τον οποίο ένας διακομιστής εντοπίζει τον χρήστη, η παρουσία του στο αίτημα καθιστά το διακομιστή να το διαχειριστεί ως ένα νόμιμο αίτημα και έτσι ο επιτιθέμενος καταφέρνει να εκτελέσει τη συναλλαγή με το διακομιστή. Αν και ένας επιτιθέμενος απαιτείται να περάσει από μερικά εμπόδια (απαιτείται από έναν χρήστη να έχει ενεργό session, και να προσελκύσει τους χρήστες να κάνουν κλικ σε μία εξωτερική σελίδα), η συγκεκριμένη μορφή επίθεσης θεωρείται πολύ συχνή. Ακόμη και οι συναλλαγές πολλών σταδίων αποτυγχάνουν να αποκλείσουν αυτό το είδος επίθεσης. Εφόσον ένας επιτιθέμενος μπορεί να προβλέψει ή να συμπεράνει το ωφέλιμο φορτίο για κάθε βήμα της συναλλαγής, είναι δυνατή η επίθεση [19] CSRF.

ΚΕΦΑΛΑΙΟ 4

ΣΥΓΧΡΟΝΗ ΒΙΒΛΙΟΓΡΑΦΙΑ

Σε αυτό το κεφάλαιο παρουσιάζονται και εξετάζονται ορισμένες μελέτες που έχουν γίνει πάνω στη στατική ανάλυση και ιδιαίτερα από τη σκοπιά της ασφάλειας καθώς και ο ρόλος που κατέχουν οι linters πάνω σε αυτό το κομμάτι, ώστε να αναδειχθεί η σπουδαιότητα του θέματος που πραγματεύεται η παρούσα διπλωματική αλλά και το πόσο επίκαιρο είναι.

4.1 Άλλες τεχνικές με στόχο την ασφάλεια

Defensive Programming

Ο αμυντικός προγραμματισμός είναι μία μορφή αμυντικού σχεδιασμού που έχει σκοπό να εξασφαλίσει τη συνεχή λειτουργικότητα ενός λογισμικού υπό απρόβλεπτες συνθήκες. Οι πρακτικές αμυντικού προγραμματισμού χρησιμοποιούνται όταν απαιτείται υψηλή ασφάλεια. Επίσης ο αμυντικός προγραμματισμός αποτελεί μία προσέγγιση για τη βελτίωση του λογισμικού και του πηγαίου κώδικα, όσον αφορά τα παρακάτω:

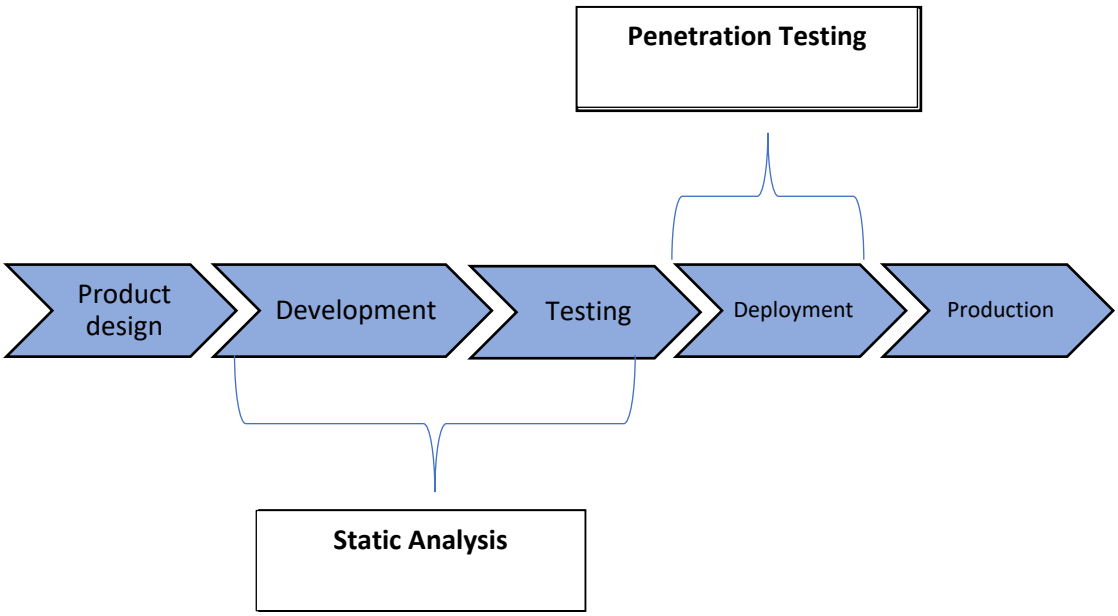
- **Γενική ποιότητα:** Μείωση του αριθμού των σφαλμάτων λογισμικού και των προβλημάτων
- **Κάνοντας τον πηγαίο κώδικα πιο κατανοητό:** Ο πηγαίος κώδικας θα πρέπει να είναι ευανάγνωστος και κατανοητός ώστε να εγκρίνεται σε έναν πιθανό έλεγχο κώδικα

Ωστόσο, η υπερβολική χρήση του αμυντικού προγραμματισμού μπορεί να προστατέψει από λάθη που δεν θα προκύψουν ποτέ, δημιουργώντας έτσι κόστος χρόνου λειτουργίας και συντήρησης. Υπάρχει επίσης ο κίνδυνος ο κώδικας να παγιδεύει ή να εμποδίζει πάρα πολλές εξαιρέσεις, οι οποίες ενδεχομένως να οδηγούν σε απαρατήρητα λανθασμένα αποτελέσματα.

Penetration Testing

Η δοκιμή διείσδυσης ή αλλιώς penetration testing, είναι η πρακτική του ελέγχου σε ένα σύστημα, δίκτυο ή μία εφαρμογή ώστε να βρεθούν αδυναμίες σε σχέση με την ασφάλεια, που ένας επίδοξος χάκερ θα μπορούσε να εκμεταλλευτεί. Το penetration testing μπορεί να γίνεται αυτοματοποιημένα μέσω εφαρμογών λογισμικού ή να εκτελεστεί χειροκίνητα. Με οποιονδήποτε από τους δύο τρόπους η διαδικασία περιλαμβάνει τη συλλογή πληροφοριών σχετικά με το στόχο πριν τη δοκιμή, τον εντοπισμό πιθανών σημείων εισόδου, και την προσπάθεια να “σπάσει” το σύστημα, κυριολεκτικά ή όχι, και να αναφέρει τα πορίσματα. Ο κύριος στόχος του penetration testing είναι ο προσδιορισμός των αδυναμιών ασφάλειας. Αυτός ο τύπος ελέγχου μπορεί επίσης να χρησιμοποιηθεί για να δοκιμαστεί η πολιτική ασφάλειας ενός οργανισμού, η τήρηση των απαιτήσεων συμμόρφωσης, η ευαισθητοποίηση των εργαζομένων σχετικά με την ασφάλεια και η ικανότητα του οργανισμού να εντοπίσει και να ανταποκριθεί σε συμβάντα σχετικά με την ασφάλεια. Τυπικά η όλη πληροφορία σχετικά με τις αδυναμίες ασφαλείας που εντοπίζονται από το penetration testing συγκεντρώνονται και προωθούνται στα τμήματα IT των εταιριών, ώστε να κατασκευάσουν διάφορες τεχνικές και στρατηγικές με στόχο των ασφάλεια.

Static Analysis **vs** Penetration Testing



Διάγραμμα 1 Static analysis vs Penetration testing

Παρακάτω παρουσιάζονται τα πλεονεκτήματα της στατικής ανάλυσης έναντι του penetration testing: [20]

- **Επιστροφή της επένδυσης:** Το penetration testing αποτελεί μία κουραστική διαδικασία που πρέπει να εκτελεστεί σε διάφορους κύκλους για να είναι πραγματικά αποτελεσματική ως μία αυτόνομη λύση ασφάλειας εφαρμογών. Ένα άλλο ζήτημα με αυτήν τη στρατηγική, είναι ότι η δοκιμή μπορεί να ξεκινήσει μόνο μετά την έναρξη της εφαρμογής. Αυτό σημαίνει ότι εάν εντοπιστούν ευπάθειες, πρέπει ταυτόχρονα να αντιμετωπιστούν οι καθυστερήσεις του χρονοδιαγράμματος, όσον αφορά την εξέλιξη της εφαρμογής. Οι οργανισμοί που εμπιστεύονται τα penetration tests ως πρωταρχική γραμμή άμυνας οφείλουν να εξετάζουν τις οικονομικές επιπτώσεις και τα τεχνικά προβλήματα που μπορεί να προκύψουν. Τα εργαλεία στατικής ανάλυσης (SAST) προσφέρουν καλύτερη απόδοση επένδυσης από τη στιγμή που εφαρμόζονται στην αρχή της ανάπτυξης (development stage) και εντοπίζουν γρήγορα και έγκαιρα τρωτά σημεία.
- **Μικρότερη ανάγκη για ανθρώπινο δυναμικό:** Τα Penetration test συχνά πραγματοποιούνται από εξειδικευμένο προσωπικό και έτσι εξαναγκάζονται οι επιχειρήσεις να προσλαμβάνουν υπαλλήλους που έχουν την απαιτούμενη γνώση πάνω σε θέματα ασφάλειας. Τη στιγμή που τα τεστ παράγουν τις αναφορές και τα αποτελέσματα οι υπάλληλοι ξεκινούν να εργάζονται ώστε να βρουν ακριβώς τα σημεία στα οποία υπάρχουν οι αδυναμίες, έτσι ώστε να τα αναφέρουν στην αντίστοιχη ομάδα ανάπτυξης λογισμικού. Αυτή η όλη διαδικασία απαιτεί πολύ χρόνο. Με την υλοποίηση των εργαλείων στατικής ανάλυσης για ασφάλεια, απαιτείται ελάχιστο ανθρώπινο δυναμικό. Ο κώδικας της εφαρμογής σαρώνεται αυτόματα με τις όποιες αδυναμίες πάνω σε θέματα ασφάλειας να εντοπίζονται αρκετά νωρίς στον κύκλο εργασίας.
- **Μεγαλύτερη ακρίβεια:** Τα Penetration test είναι όσο αποτελεσματικός είναι ο άνθρωπος που τα εκτελεί και τα εργαλεία που έχει στη διάθεση του. Συχνά οι γνώσεις του πάνω στις αδυναμίες και την ασφάλεια είναι ανεπαρκείς. Αυτό έχει ως αποτέλεσμα να παράγονται πολλά false negatives (FN) κατά την διαδικασία. Επίσης

ο άνθρωπος ή το εργαλείο που βρίσκεται πίσω από τα Penetration tests δεν έχει πρόσβαση στον κώδικα της εφαρμογής με αποτέλεσμα να μην μπορεί να δει τις αδυναμίες. Αντίθετα μέσω της στατικής ανάλυσης σκανάρεται η εφαρμογή περισσότερο αποτελεσματικά και τα αποτελέσματα παρέχονται πριν ακόμα τελειώσει η όλη διαδικασία.

- **Εκπαιδευτική αξία για τους προγραμματιστές:** Τα εργαλεία στατικής ανάλυσης έχουν το πάνω χέρι σε αυτό τον τομέα καθώς βοηθούν στην εξέλιξη του προγραμματιστή. Η λύση είναι ενσωματωμένη στο περιβάλλον των προγραμματιστών και επιτρέπει την εξαγωγή και τον έλεγχο του ευρήματος εκτός σύνδεσης, βελτιώνοντας ουσιαστικά τις γνώσεις του προγραμματιστή στις πρακτικές που αφορούν την ασφάλεια.

4.2 Εργαλεία στατικής ανάλυσης στη βιομηχανία

Έχουν αναπτυχθεί διάφορα SATs (static analysis tools) με ποικίλα χαρακτηριστικά και εφαρμογές, όπου τα συστήματα λογισμικού έχουν διαφορετικές απαιτήσεις για αυτά τα εργαλεία. Πολλά εργαλεία στατικής ανάλυσης χρησιμοποιούνται ή έχουν χρησιμοποιηθεί από εταιρίες λογισμικού όπως η Google, η Facebook και η Microsoft. Οι δημιουργοί (Bessay at al) [21] του Coverity (ενός στατικού εργαλείου ανάλυσης κώδικα) βίωσαν από πρώτο χέρι πόσο δύσκολο είναι να δημιουργηθεί ένα τέτοιο στατικό εργαλείο που να ταιριάζει στις ανάγκες των μεγάλων επιχειρήσεων, οι οποίες διαθέτουν μεγάλες βάσεις δεδομένων. Οι τεχνικές απαιτήσεις τέτοιων εταιριών μπορεί να ποικίλουν πολύ όπως για παράδειγμα οι διαφορετικές εσωτερικές διαδικασίες και τα διαφορετικά περιβάλλοντα που χρησιμοποιεί η κάθε εταιρία μπορεί να απαιτούν από το εργαλείο διαφορετική συμπεριφορά. Επίσης το μέγεθος και η φύση του εκάστοτε λογισμικού μπορεί να συνεπάγονται διαφορετικές απαιτήσεις για το SAT, με αποτέλεσμα η Google να καταφύγει στη δημιουργία δικού της SAT, καθώς κανένα από τα ήδη υπάρχοντα δεν ήταν ικανά να διαχειριστεί την τεράστια βάση του κώδικα της εταιρίας. Η Google πράγματι έχει δοκιμάσει διάφορα στατικά εργαλεία όπως το FindBugs και το Coverity και έχει δώσει feedback όσον αφορά τα συγκεκριμένα εργαλεία. Παρά την μερική επιτυχία αυτών των εργαλείων, δεν υιοθετήθηκαν εν τέλει, λόγω των πολλών ψευδών συναγερμών (FP) που παράγουν, της έλλειψης επεκτασιμότητας και της δυσκολίας να ενσωματωθούν από την εταιρία. Τελικώς δημιούργησαν και ανέπτυξαν το δικό τους εργαλείο μέσα από μία ομάδα προγραμματιστών, (Tricorder) το οποίο δημιουργεί και εξετάζει τους custom analyzers. Το συγκεκριμένο εργαλείο προωθεί και εμφανίζει μόνο τις προειδοποιήσεις (warnings) υψηλής προτεραιότητες από αναλυτές-εργαλεία (analyzers) που έχουν λάβει καλή κριτική από άλλους προγραμματιστές, σε μία προσπάθεια να γίνουν περισσότερο πρόθυμοι όσον αφορά το εργαλείο και να μειώσουν της μικρής προτεραιότητας προειδοποιήσεις αλλά και τους ψευδείς συναγερμούς (FP). Το eBay ανέφερε σχετικά την προσπάθεια τους να επιλέξουν το πιο πολύτιμο και κατάλληλο εργαλείο για αυτούς. Αναφέρουν μία μέθοδο αξιολόγησης αυτών των εργαλείων, όπου τελικά επέλεξαν το εργαλείο FindBugs στην αναπτυξιακή τους διαδικασία. Για το δικό τους λογισμικό αναφέρουν ότι είναι πολύ σημαντικό το εργαλείο που χρησιμοποιούν να δέχεται επεκτασιμότητα (δηλαδή να επιτρέπει τη δημιουργία επιπλέον κανόνων). Αναφέρουν επίσης ότι η προσαρμογή και η ιεράρχηση προτεραιοτήτων αποτελούν σημαντικά βήματα για την αξιολόγηση ενός εργαλείου, καθώς μπορούν να μειώσουν τους ψευδείς συναγερμούς και να κάνουν τους προγραμματιστές πιο πρόθυμους να χρησιμοποιήσουν το εργαλείο. [22]

4.3 Εργαλεία στατικής ανάλυσης με στόχο την ασφάλεια στην Javascript

Retire.js

Όπως αναφέρθηκε και στα προηγούμενα κεφάλαια πλέον ο κώδικας έχει γίνει κατά ένα πολύ μεγάλο βαθμό open source. Ιδιαίτερα όταν αναφερόμαστε στην Javascript και πιο συγκεκριμένα σε εφαρμογές κτισμένες στο Node.js υπάρχουν πολλές βιβλιοθήκες για χρήση από τους προγραμματιστές. Αρκεί να σκεφτεί κάποιος τις δυνατότητες που προσφέρει το npm. Το πλεονέκτημα με τις εξωτερικές βιβλιοθήκες είναι ότι απλοποιούν την όλη διαδικασία, αλλά ταυτόχρονα απαιτούν από τον προγραμματιστή και όλη την ομάδα να μένει ενημερωμένη όσον αφορά καινούργιες εκδόσεις αλλά και πιθανές αδυναμίες που μπορεί να υπάρχουν σε κάποιες βιβλιοθήκες. Συγκεκριμένα αποτελεί τεράστιο πρόβλημα για τον προγραμματιστή να γνωρίζει ανά πάσα στιγμή αν όλες οι βιβλιοθήκες που χρησιμοποιεί είναι ενημερωμένες ή έχουν κάποια γνωστή αδυναμία [23]. Το Retire.js είναι ένα πρόγραμμα που βοηθάει τον προγραμματιστή να εντοπίσει τις βιβλιοθήκες οι οποίες έχουν αδυναμίες που αφορούν την ασφάλεια. Πιο συγκεκριμένα το πρόγραμμα αποτελείται από τρία μέρη:

- Έναν command line scanner
- Ένα Chrome Extension
- Ένα grunt plugin

Command line Scanner

Σκανάρει όλες τις ιστοσελίδες στις οποίες έχει γίνει επίσκεψη για αναφορές σε μη ασφαλείς βιβλιοθήκες, και θέτει μία προειδοποίηση στην κονσόλα του προγραμματιστή. Επίσης θέτει ένα εικονίδιο στην διεύθυνση της μπάρας, εάν εντοπισθεί ότι φορτώθηκε μία βιβλιοθήκη με αδυναμίες στην ασφάλεια

Grunt Plugin

Σκανάρει το συνολικό έργο (project) για να βρει αρχεία Javascript τα οποία περιέχουν κάποιες γνωστές αδυναμίες

The Source

Επίσης διαθέτει ένα JSON αρχείο το οποίο διατηρείται σε αποθετήριο του GitHub και διαθέτει γνωστές αδυναμίες. Δημιουργήθηκε ψάχνοντας σημειώσεις και μέσω διάφορων ανιχνευτών προβλημάτων (issue trackers) για τα πιο γνωστά frameworks.

Detection

Για να εντοπίσει μία έκδοση από ένα πακέτο, το συγκεκριμένο πρόγραμμα χρησιμοποιεί το όνομα του φακέλου (filename) ή το URL. Εάν αυτό αποτύχει, κατεβάζει/ανοίγει τον φάκελο και ψάχνει για συγκεκριμένα πακέτα εντός του φακέλου. Εάν και αυτό αποτύχει, υπάρχει πιθανότητα να χρησιμοποιήσει hashes για τους minified φακέλους. Τέλος εάν αποτύχουν όλα τα παραπάνω, το Chrome plugin τρέχει τον κώδικα σε μία εικονική μηχανή (sandbox, virtual machine) για να εντοπίσει το πακέτο και την έκδοση [24].

JSPrime

Το JSPrime είναι ένα στατικό εργαλείο για την JavaScript και στοχεύει στη βελτίωση της ασφάλειας των εφαρμογών. Είναι βασισμένο στον ιδιαίτερα δημοφιλή Esprima ECMAsprit αναλυτή (parser) και μερικά από τα χαρακτηριστικά που προσφέρει είναι τα παρακάτω:

- Επίβλεψη του JavaScript πηγαίου κώδικα και των βιβλιοθηκών.
- Ενώ οι περισσότεροι στατικοί αναλυτές (static analyzers) έχουν φτιαχτεί έτσι ώστε να υποστηρίζουν την απλή Javascript, το συγκεκριμένο εργαλείο προσφέρει ανάλυση για διάφορα περιβάλλοντα της JavaScript όπως είναι η jQuery και το YUI. Καταφέρνει έτσι να περιορίσει τα false positives-false negatives που παράγουν τα περισσότερα εργαλεία καθώς προσφέρει ανάλυση πάνω στη λειτουργικότητα αυτών των framework.
- Εντοπισμός και έλεγχος συναρτήσεων και μεταβλητών.
- Περιορισμός των false positive συναγερμών.
- Υποστήριξη Minified JavaScript.
- Βελτιωμένη απόδοση.

ESLint Security Plugin

Σε επίπεδο Linter και πιο συγκεκριμένα στον ESLint έχει δημιουργηθεί ήδη ένα plugin με στόχο τη στατική ανάλυση κώδικα των εφαρμογών και την ασφάλεια τους. Το συγκεκριμένο plugin είναι διαθέσιμο μέσω του npm όπου μπορεί κάποιος να το εγκαταστήσει γράφοντας στο τερματικό του: `npm install --save-dev eslint-plugin-security` και να το θέσει σε λειτουργία κάνοντας την παρακάτω διαμόρφωση στον.eslintrc φάκελο

```
"plugins": [
  "security"
],
"extends": [
  "plugin:security/recommended"
]
```

Εικόνα 36 Διαμόρφωση του ESLint security plugin

Περιέχει ένα σετ κανόνων όπου μερικοί από τους οποίους είναι οι παρακάτω:

- **Detect-pseudorandomBytes:** Ανιχνεύει την ύπαρξη της μεθόδου pseudoRandomBytes() η οποία είναι προσβάσιμη μέσω του core πακέτου του Node.js και η οποία δεν δίνει πλήρως την τυχαιότητα που ίσως περιμένει και χρειάζεται ο εκάστοτε προγραμματιστής.
- **Detect-object-injection:** Ανιχνεύει την ύπαρξη περιπτώσεων όπως variable[key] δηλαδή της ανάθεσης των ιδιοτήτων μέσω των brackets (αγκυλών) και όχι μέσω της κλασικής μεθόδου variable.key, όπου εάν το property key είναι ελέγξιμο από ένα χρήστη μπορεί να προκαλέσει ζημία στην εφαρμογή όπως κάνοντας overwrite τα proto του αντικειμένου.
- **Detect-unsafe-regEx:** Ανιχνεύει την ύπαρξη κανονικών εκφράσεων στην εφαρμογή και μέσω ενός τρίτου πακέτου (safe-regex) το οποίο είναι διαθέσιμο μέσω του npm,

ελέγχει εάν οι κανονικές εκφράσεις που χρησιμοποιούνται είναι ασφαλείς σε επιθέσεις DOS.

- **Detect-new-buffer:** Ανιχνεύει την ύπαρξη instances new buffer.
- **Detect-require-with-non-Literal-arguments:** Ανιχνεύει την μέθοδο require() που δε διαθέτει literal (όρισμα μέσα σε εισαγωγικά) ορίσματα , καθώς εάν η μεταβλητή που υπάρχει στη μέθοδο είναι ελεγχόμενη από κάποιον χρήστη μπορεί να εκτελέσει κακόβουλες εντολές.

Το συγκεκριμένο plugin όπως αναφέρεται και από τους δημιουργούς του διαθέτει πολλά false-positives δηλαδή περιπτώσεις όπου ο χρήστης ενημερώνεται από τον linter μέσω ενός μηνύματος ότι βρέθηκαν περιπτώσεις πιθανής επίθεσης, ενώ στην πραγματικότητα δεν υπάρχει τέτοιος κίνδυνος. Αυτό οφείλεται σε διάφορους λόγους, καθώς όλα τα εργαλεία στατικής ανάλυσης είναι επιρρεπή σε περιπτώσεις false-positive, όπως έχει ήδη αναφερθεί, πράγμα το οποίο είναι αναπόφευκτο, αλλά επίσης οφείλεται και σε περιπτώσεις κακής υλοποίησης όπως για παράδειγμα στον κανόνα για την εύρεση μη Literal ορισμάτων στην μέθοδο require, όπου ο συγκεκριμένος κανόνας ενεργοποιείται και σε περιπτώσεις όπως οι παρακάτω:

- require(__dirname + '/lib/….') καθώς και,
- require(`/lib/…`)

Οι παραπάνω περιπτώσεις ανήκουν στην κατηγορία των false positive καθώς δεν γίνεται έλεγχος εάν ο προγραμματιστής χρησιμοποιεί backticks αντί για εισαγωγικά (quotes). Αυτή η περίπτωση δεν αποτελεί ικανή συνθήκη ώστε να πρέπει να ενεργοποιηθεί ο κανόνας, καθώς είναι ένας διαφορετικός τρόπος να δηλώσει ο προγραμματιστής τα Literal ορίσματα. Επίσης διαθέτει περιπτώσεις κανόνων όπου ανιχνεύουν την ύπαρξη μεθόδων όπου στις πιο πρόσφατες εκδόσεις του Node.js δεν είναι διαθέσιμες καθώς έχουν αφαιρεθεί, όπως αναφέρεται στο εγχειρίδιο [25]. Τέτοιες περιπτώσεις είναι οι παρακάτω:

- detect instances of new buffer και detect pseudoRandomBytes method από το βασικό πακέτο του Node.js.

4.4 Άλλα εργαλεία στατικής ανάλυσης με στόχο την ασφάλεια

Παρακάτω παρουσιάζονται κάποια από τα πολλά εργαλεία στατικής ανάλυσης με στόχο την ασφάλεια, που δημιουργήθηκαν στο πέρασμα των χρόνων από διάφορους ερευνητές, εργαστήρια και πανεπιστήμια, με σκοπό την βελτίωση της ασφάλειας των εφαρμογών σε γλώσσες προγραμματισμού διαφορετικές από την JavaScript.

Πίνακας 1 Λοιπά εργαλεία στατικής ανάλυσης με στόχο την ασφάλεια

Εργαλεία στατικής ανάλυσης με στόχο την ασφάλεια
ARCHER
tool BOON
CQual
Eau Claire Tool
LAPSE
MOPS
SATURN
Splint
Pixy
xg++

Ακολουθεί μία σύντομη περιγραφή των εργαλείων αυτών και του πεδίου δράσης τους.

- **ARCHER:** Πρόκειται για ένα εργαλείο στατικής ανάλυσης για τον έλεγχο των ορίων του πίνακα. Τα αρχικά του ονόματος προκύπτουν από το Array CHeckER. Το συγκεκριμένο εργαλείο χρησιμοποιεί έναν custom-built solver έτσι ώστε να εκτελέσει μία διαδικτυακή ανάλυση των προγραμμάτων C. Χρησιμοποιήθηκε για να βρει πάνω από δώδεκα αδυναμίες ασφάλειας στα LINUX, ενώ ταυτόχρονα βρήκε εκατοντάδες σχετικά προβλήματα στα όρια των πινάκων στα OpenBSD, Sendmail και PostgreSQL [Xie et al, 2003].
- **Tool BOON:** Το συγκεκριμένο εργαλείο εφαρμόζει ακέραια ανάλυση εύρους για να καθορίσει εάν ένα πρόγραμμα σε C είναι ικανό για indexing ενός πίνακα εκτός των ορίων του. Παρότι μπορεί να βρει πολλά λάθη που η λεξική ανάλυση αδυνατεί να βρει, ο ελεγκτής θεωρείται ανακριβής. Δεν μπορεί να μοντελοποιήσει τις εξαρτήσεις (dependencies), ενώ αγνοεί τελείως τους pointers.
- **CQual:** Πρόκειται για ένα εργαλείο το οποίο χρησιμοποιεί προεπιλεγμένους τύπους (type qualifiers) για να εκτελέσει ανάλυση για την ανίχνευση τρωτών σημείων μορφοποίησης στα προγράμματα C. Το συγκεκριμένο εργαλείο απαιτεί από τον προγραμματιστή να σημειώσει ένα μικρό αριθμό από μεταβλητές ως “μολυσμένες ή μη” και στη συνέχεια να εφαρμόσει ορισμένους κανόνες (μαζί με τις ήδη υπάρχουσες βιβλιοθήκες του συστήματος) ώστε να εξάγει ορισμένα συμπεράσματα. Μετά από αυτά. το σύστημα μπορεί να εντοπίσει format string αδυναμίες εφαρμόζοντας έλεγχο του τύπου (type checking)
- **Eau Claire:** Το συγκεκριμένο εργαλείο χρησιμοποιεί ένα θεώρημα για να δημιουργήσει ένα γενικό περιβάλλον ελέγχου για προγράμματα C. Μπορεί να χρησιμοποιηθεί για να βρει συχνά προβλήματα ασφάλειας όπως “buffer overflows”, “file access race conditions” και “format string bugs”. Το σύστημα ελέγχει τη χρήση συναρτήσεων οι οποίες προέρχονται από την κύρια βιβλιοθήκη χρησιμοποιώντας κάποιες ήδη υπάρχουσες προδιαγραφές. Οι προγραμματιστές μπορούν να χρησιμοποιήσουν προδιαγραφές έτσι ώστε να ελέγξουν ότι όλες οι συναρτήσεις έχουν την αναμενόμενη συμπεριφορά.
- **LAPSE:** του οποίου τα αρχικά σημαίνουν Lightweight Analysis for Program Security in Eclipse (ελαφρύ πρόγραμμα ανάλυσης για ασφάλεια στον Eclipse), είναι ένα plugin στον Eclipse που στοχεύει στο να ανιχνεύει αδυναμίες ασφάλειας σε εφαρμογές J2EE. Μπορεί να ανιχνεύσει αδυναμίες όπως SQL injection, XSS (cross-site scripting), cookie poisoning και parameter manipulation.
- **MOPS:** Τα αρχικά του σημαίνουν Model checking programs for Security properties (Μοντέλο ελέγχου προγράμματος για ιδιότητες ασφάλειας) και λαμβάνει μία προσέγγιση ενός μοντέλου ελέγχου έτσι ώστε να ερευνήσει πιθανές αδυναμίες στις προσωρινές ιδιότητες ασφάλειας στην C. Οι προγραμματιστές μπορούν να ορίσουν τις δικές τους ιδιότητες ασφάλειας (safety properties) και το πρόγραμμα χρησιμοποιείται για να ανιχνεύσει πιθανές παραβιάσεις σε σφάλματα διαχείρισης, και πρόσβασης.
- **SATURN:** Το συγκεκριμένο εργαλείο είναι παρόμοιο με το MOPS και χρησιμοποιήθηκε για να βρει περισσότερες από εκατό διαρροές μνήμης (memory leaks) στα Linux.
- **Splint:** Το εργαλείο αυτό επεκτείνει την έννοια του lint όσον αφορά την ασφάλεια. Χωρίς την προσθήκη σχολιασμών (annotations), οι προγραμματιστές μπορούν να εκτελέσουν βασικούς ελέγχους τύπου lint. Ενεργοποιώντας τα annotations οι προγραμματιστές μπορούν μέσω του εργαλείου να εντοπίσουν γενικές παραβιάσεις,

απροειδοποίητες τροποποιήσεις στις global μεταβλητές και πιθανά σφάλματα πριν την αρχικοποίηση.

- **Pixy:** Το εργαλείο αυτό ανιχνεύει Cross-site-scripting αδυναμίες στα PHP προγράμματα. Οι συγγραφείς του συγκεκριμένου εργαλείου υποστηρίζουν ότι η flow-sensitive ανάλυση μπορεί να εφαρμοστεί εύκολα σε άλλου είδους αδυναμίες όπως το SQL injection και command injection.
- **xg++:** Το συγκεκριμένο εργαλείο χρησιμοποιεί ένα template-driven compiler για να βρει αδυναμίες στο kernel στα Linux αλλά και σε άλλα λειτουργικά όπως το OpenBSD. Ουσιαστικά το εργαλείο χρησιμοποιεί τοποθεσίες για να βρει τα σημεία όπου χρησιμοποιούνται μη έμπιστα δεδομένα από μη έμπιστη πηγή χωρίς πρώτα να ελεγχθούν. Τέτοιες περιπτώσεις είναι όταν ο χρήστης μπορεί να προκαλέσει στο kernel να δεσμεύσει μνήμη και να μην την απελευθερώσει.

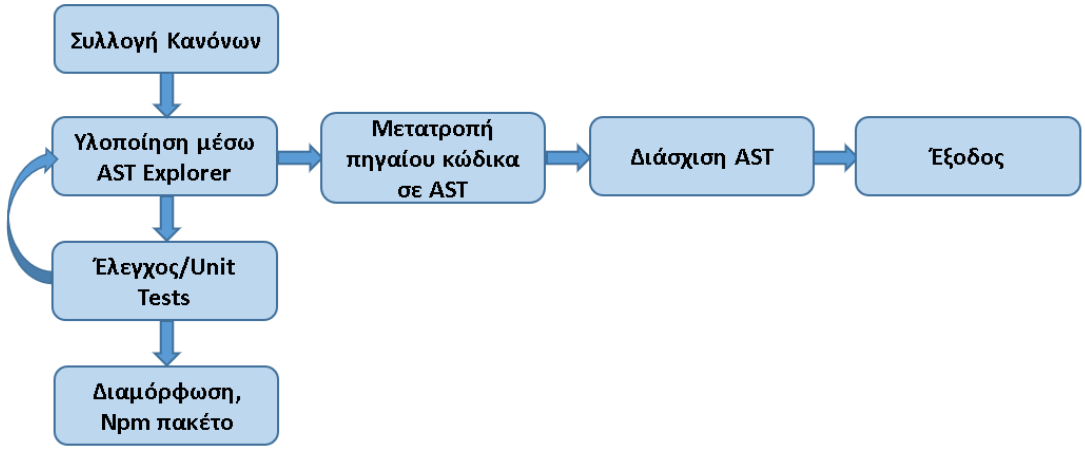
ΚΕΦΑΛΑΙΟ 5

ΜΕΘΟΔΟΛΟΓΙΑ

Σε αυτό το κεφάλαιο αναλύεται η μεθοδολογία που ακολουθήθηκε στην παρούσα διπλωματική για την κατασκευή του Plugin στο ESLint με στόχο την ασφάλεια. Παρουσιάζεται η δημιουργία του plugin μέσω του yeoman generator, το σύνολο των κανόνων που υλοποιήθηκαν, η λογική πίσω από κάθε κανόνα, ο τρόπος υλοποίησης τους, τα προβλήματα που επιλύουν, καθώς και οι έλεγχοι (unit tests) που υλοποιήθηκαν μέσω του mocha που παρέχει το ESLint στο σύνολο των κανόνων για τον έλεγχο της σωστής λειτουργίας. Τέλος αναφέρεται η τροποποίηση του plugin σε npm πακέτο και ο τρόπος εγκατάστασης από τον χρήστη.

ΓΕΝΙΚΑ

Για τη συλλογή της πληροφορίας όσον αφορά τα είδη των επιθέσεων, τις αδυναμίες των εφαρμογών και ιδιαίτερα αυτών που βασίζονται σε Node.js έγινε αναζήτηση σε πλατφόρμες όπως το snyk [26] για την καταγραφή αδυναμιών στα πιο γνωστά πακέτα του npm, στη βάση του OWASP με τις πιο γνωστές αδυναμίες και κατηγορίες επιθέσεων [27], καθώς και στο CVE details [28], όπου περιέχονται όλες οι καταγεγραμμένες αδυναμίες του Node.js από τη δημιουργία του μέχρι σήμερα. Παρακάτω παρουσιάζεται ένα διάγραμμα ροής με τη συλλογιστική πορεία που ακολουθήθηκε για την εκπόνηση της εργασίας, ενώ το περιεχόμενο αποτελεί οδηγό για την κατανόηση των Κεφαλαίων 5 & 6.



Διάγραμμα 2 Διάγραμμα ροής μεθοδολογίας

5.1 Δημιουργία plugin

Η δημιουργία ενός plugin στο ESLint γίνεται μέσω του **yeoman generator**. Ο συγκεκριμένος generator είναι ουσιαστικά ένα plugin που μπορεί να τρέξει με την εντολή 'yo' για να γεφυρώσει ολοκληρωμένα έργα ή χρήσιμα κομμάτια αυτών. Αρχικά έγινε εγκατάσταση του συγκεκριμένου generator μέσω του npm και των παρακάτω εντολών.

```
npm install -g yo
```

Εικόνα 37 Εγκατάσταση του yeoman generator

Και στη συνέχεια εγκατάσταση του yeoman για το ESLint,

```
npm install -g generator-eslint
```

Εικόνα 38 Εγκατάσταση του yeoman generator για το ESLint

Η δημιουργία του plugin επιτυγχάνεται με την παρακάτω εντολή μέσω του yeoman,

```
$ yo eslint:plugin
```

Εικόνα 39 Δημιουργία plugin στη γραμμή εντολών

όπου στη συνέχεια απεικονίζεται η βασική διαμόρφωση για το plugin.

```
What is the plugin ID? security
Type a short description of this plugin: a security plugin for ESLint
Does this plugin contain custom ESLint rules? Yes
Does this plugin contain one or more processors? No
 create package.json
 create lib/index.js
 create README.md
```

Εικόνα 40 Παραγόμενα αρχεία από τη δημιουργία του plugin

Στην παραπάνω εικόνα φαίνονται τα τρία αρχεία που δημιουργήθηκαν μετά την εκτέλεση της παραπάνω εντολής για τη δημιουργία του plugin μέσω του yeoman.

- **README.md**: Ένα αρχείο το οποίο περιέχει τις βασικές πληροφορίες για το plugin.
- **package.json:** Ένα αρχείο για τη διαμόρφωση των κανόνων και του plugin όπως αναφέρθηκε στο Κεφάλαιο 2 και πιο συγκεκριμένα στην παράγραφο (Διαμόρφωση και Διαμόρφωση Κανόνων και Plugin)
- **lib/index.js:** Ένα αρχείο για την εισαγωγή των κανόνων.

5.2 Δημιουργία κανόνων

Η δημιουργία ενός κανόνα επιτυγχάνεται με την παρακάτω εντολή μέσω του yeoman:

```
$ yo eslint:rule
```

Εικόνα 41 Δημιουργία κανόνα στη γραμμή εντολών

```
Where will this rule be published? ESLint Plugin
What is the rule ID? detect-child-process
Type a short description of this rule: detect exec method of child_process module with non-Literal arguments
Type a short example of the code that will fail:
 create docs/rules/detect-child-process.md
 create lib/rules/detect-child-process.js
 create tests/lib/rules/detect-child-process.js
```

Εικόνα 42 Παραγόμενα αρχεία από τη δημιουργία του κανόνα

Στην παραπάνω εικόνα παρουσιάζονται τα αρχεία που δημιουργήθηκαν μετά την εκτέλεση της εντολής για τη δημιουργία του κανόνα (στη συγκεκριμένη περίπτωση του detect-child-process).

- **docs/rules/detect-child-process.md:** Ένα αρχείο markdown το οποίο περιέχει πληροφορίες για τον εκάστοτε κανόνα.
- **lib/rules/detect-child-process.js:** Ο πηγαίος φάκελος που περιέχει την υλοποίηση του κανόνα
- **tests/lib/rules/detect-child-process.js:** Ένα αρχείο για τον έλεγχο του κανόνα που περιέχει έγκυρες και μη έγκυρες περιπτώσεις εφαρμογής του.

5.3 Λογική των κανόνων

Παρακάτω παρουσιάζεται ο πίνακας με τους κανόνες που δημιουργήθηκαν, ενώ αναλύεται και η λογική πίσω από κάθε κανόνα, δηλαδή σε ποιες αδυναμίες στοχεύουν και τα προβλήματα που επιλύουν.

Πίνακας 2 Λίστα κανόνων

ΛΙΣΤΑ ΚΑΝΟΝΩΝ
detect buffer unsafe allocation
detect child process exec method with non-Literal argument
detect crlf attack
detect timing attacks
detect absence of option: name in cookie of express session module
detect dangerous redirects
detect eval with expression
detect html injection
detect insecure randomness
detect non-Literal arguments in require method
detect no-SQL injection
detect option: multipleStatements in mysql.js module
detect option:rejectUnauthorized in https core module
detect option:unsafe in serialize JavaScript module
detect non-Literal argument in runInThisContext method of vm.js core module
detect security misconfiguration in cookie of express session module
detect SQL injection
detect non-Literal reg-exp
detect-helmet-without-noCache
detect disable of ssl across node server

detect buffer unsafe allocation (Ανίχνευση μη ασφαλούς δέσμευσης μνήμης)

Η κλάση Buffer του Node.js ανήκει στο Global Scope όποτε μπορεί κανείς να τη χρησιμοποιήσει κατευθείαν, χωρίς να χρειαστεί να κάνει require το core module. Στις προηγούμενες εκδόσεις του node.js τα buffer instances δημιουργούνταν μέσω του buffer constructor της κλάσης buffer για τη δέσμευση της μνήμης. Η μέθοδος new Buffer() αφαιρέθηκε από τις επόμενες εκδόσεις του node.js επειδή δεν θεωρούνταν σταθερή. Για παράδειγμα εάν ένας επιτιθέμενος μπορούσε να προκαλέσει σε μία εφαρμογή να δεχθεί έναν αριθμό αντί για string που περίμενε σαν όρισμα, τότε η εφαρμογή θα καλούσε τη

μέθοδο new Buffer(100) αντί για new Buffer('100'), με αποτέλεσμα να γίνει δέσμευση 100 byte buffer αντί για δέσμευση 3 byte buffer με περιεχόμενο '100'. Έτσι ο 100 byte buffer μπορεί να περιείχε αυθαίρετα προϋπάρχοντα στη μνήμη δεδομένα, με αποτέλεσμα να χρησιμοποιηθεί για την διαρροή ευαίσθητων δεδομένων μνήμης σε έναν επιτιθέμενο. Στις επόμενες εκδόσεις του Node (>8.00), η έκθεση της μνήμης δεν μπορεί να συμβεί επειδή τα δεδομένα είναι αρχικοποιημένα με μηδέν (zero-filled). Ωστόσο άλλες επιθέσεις είναι πιθανές, όπως η δέσμευση πολύ μεγάλων buffer, έτσι ώστε να οδηγήσουν σε πιθανά προβλήματα όσον αφορά την απόδοση ή ακόμα και να εξαντλήσουν την μνήμη. Για τους παραπάνω λόγους οι διάφορες μορφές του constructor new Buffer() αντικαταστάθηκαν από διαφορετικές μεθόδους όπως οι Buffer.from(), Buffer.alloc() και Buffer.allocUnsafe().

- **Buffer.allocUnsafe(size):** Επιστρέφει έναν νέο **μη** αρχικοποιημένο Buffer έχοντας το μέγεθος που έχει οριστεί.

Παρακάτω στην εικόνα φαίνεται η χρήση της συγκεκριμένης μεθόδου.

```
const buf = Buffer.allocUnsafe(10);

console.log(buf);
// Prints (contents may vary) CHECK WHAT BUFFER CONTAINTS ->
// <Buffer a0 8b 28 3f 01 00 00 00 50 32>
```

Εικόνα 43 Μέθοδος Buffer.allocUnsafe

Αυτό που κάνει τη συγκεκριμένη μέθοδο 'επικίνδυνη' είναι ότι όταν καλείται το πεδίο της δεσμευμένης μνήμης, αυτό είναι μη αρχικοποιημένο (δεν περιέχει μηδενικά). Ενώ αυτός ο σχεδιασμός κάνει τη δέσμευση μνήμης αρκετά γρήγορη, το δεσμευμένο πεδίο μνήμης μπορεί να περιέχει παλιά δεδομένα τα οποία είναι ευαίσθητα. Η χρήση ενός Buffer που δημιουργήθηκε από τη παραπάνω μέθοδο χωρίς να γίνει επικάλυψη της μνήμης μπορεί να επιτρέψει τη διαρροή αυτών των παλιών-ευαίσθητων δεδομένων όταν διαβάζεται η μνήμη. Ο προγραμματιστής για να αποφύγει αυτή την ανεπιθύμητη λειτουργία μπορεί να χρησιμοποιήσει την Buffer.alloc() μέθοδο ώστε να αρχικοποιήσει τον Buffer με μηδενικά και να εξαλείψει αυτή την περίπτωση διαρροής της μνήμης. Το πρόβλημα είναι ότι η δέσμευση αποτελεί μία σύγχρονη λειτουργία και σε συνδυασμό με την αρχιτεκτονική του Node.js ως μονού νήματος, αυτό δεν έχει πάντα τα επιθυμητά αποτελέσματα. Η μη ασφαλής δέσμευση είναι πολύ γρηγορότερη από την ασφαλή, επειδή η αρχικοποίηση που γίνεται στην ασφαλή δέσμευση με μηδενικά παίρνει πολύ περισσότερο χρόνο. Συνεπώς υπάρχει το τίμημα της απόδοσης. Η δημιουργία του κανόνα detect buffer unsafe allocation αποσκοπεί στην εύρεση της μεθόδου Buffer.allocUnsafe μέσα στον κώδικα του προγραμματιστή έτσι ώστε να ειδοποιηθεί μέσω ενός μηνύματος για τη χρήση της.

detect child process with exec method with non-Literal argument (Ανίχνευση της μεθόδου exec από την ενότητα child_process του Node.js με μη κυριολεκτικά ορίσματα)

Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση της μεθόδου exec του core module child_process του Node.js η οποία διαθέτει non-Literal (που δεν είναι string-δε βρίσκεται μέσα σε εισαγωγικά) πρώτο όρισμα. Η λειτουργία της συγκεκριμένης μεθόδου καθώς και το είδος της επίθεσης που μπορεί να πραγματοποιηθεί από έναν επιτιθέμενο αναλύθηκε στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο 3.1.1 (Command Injection Attacks). Η άλλη λειτουργία του συγκεκριμένου κανόνα αφορά την μέθοδο **execFile**, η οποία προτείνεται στο Κεφάλαιο 3 σαν εναλλακτική μέθοδος αντί της **exec**, καθώς θεωρείται πιο ασφαλής με

την έννοια ότι δεν δημιουργεί καινούργιο shell εξορισμού και θεωρείται αποτελεσματικότερη από την **exec**.

- **child_process.execFile(file[,args][,options][,callback])**

Παρακάτω παρουσιάζεται ένας πίνακας με τα διάφορα options που διαθέτει η συγκεκριμένη μέθοδος.

Πίνακας 3 options of execFile method

options
cwd <string>
env <Object>
encoding <string>
timeout <number>
maxBuffer <number>
killSignal <string | integer>
uid <number>
gid <number>
windowsHide <boolean>
windowsVerbatimArguments <boolean>
shell <boolean>

- **shell <boolean>**: Εάν είναι **true**, η εντολή εκτελείται μέσα σε ένα shell ακριβώς όπως την περίπτωση της exec μεθόδου. Χρησιμοποιεί το '/bin/sh' στο UNIX. Συνεπώς οι κίνδυνοι για command injection είναι ακριβώς οι ίδιοι όπως και στην περίπτωση της exec που αναλύθηκε στο Κεφάλαιο 3, Παράγραφος 3.1.1 (Command injection attacks).

Η υλοποίηση του κανόνα αφορά την αναζήτηση της συγκεκριμένης επιλογής στα ορίσματα της μεθόδου και την προειδοποίηση του για την ύπαρξη της.

Detect crlf attack (Ανίχνευση επιθέσεων καταγραφής)

Οι αδυναμίες που αφορούν τη διάχυση καταγραφής (log injection) επιτρέπουν σε έναν επιτιθέμενο να κατασκευάσει και να παραβιάσει τα αρχεία καταγραφής μίας εφαρμογής (application's logs). Σαν μηχανισμός επίθεσης θα μπορούσε να αναφερθεί η προσπάθεια ενός επιτιθέμενου να κατασκευάσει ένα κακόβουλο αίτημα (malicious request) που μπορεί να αποτύχει σκόπιμα, και το οποίο η εφαρμογή θα καταγράψει. Στην περίπτωση που η είσοδος από τον χρήστη-επιτιθέμενο δεν έχει ελεγχθεί, το payload αποστέλλεται όπως είναι στην εγκατάσταση καταγραφής. Οι αδυναμίες ενδέχεται να διαφέρουν ανάλογα με την εγκατάσταση της εφαρμογής. Σαν παράδειγμα θα μπορούσε να αναφερθεί η καταγραφή μίας αποτυχημένης προσπάθειας από την εφαρμογή.

```
var userName = req.body.userName;
console.log('Error: attempt to login with invalid user: ', userName);
```

Εικόνα 44 Καταγραφή αποτυχημένης προσπάθειας

Στην παραπάνω εικόνα φαίνεται ακριβώς αυτό το παράδειγμα, δηλαδή η έξοδος από μία αποτυχημένη προσπάθεια η οποία θα μπορούσε να καταγράφεται στη συνέχεια σε ένα αρχείο καταγραφής. Όταν η είσοδος του χρήστη δεν ελέγχεται και ο μηχανισμός εξόδου

είναι ένα συνηθισμένο τερματικό stdout, τότε η εφαρμογή είναι ευάλωτη σε CRLF injection, όπου ένας επιτιθέμενος μπορεί να δημιουργήσει ένα κακόβουλο payload όπως το επόμενο:

```
curl http://localhost:4000/login -X POST --data 'userName=vyva%0aError: alex moldovan fa
iled $1,000,000 transaction&password=Admin_123&_csrf='
```

Εικόνα 45 malicious payload in CRLF injection

Όπου η παράμετρος **username** κωδικοποιεί στο request το σύμβολο LF (line feed), το οποίο θα έχει ως αποτέλεσμα τη δημιουργία μίας νέας γραμμής, όπως φαίνεται παρακάτω:

```
Error: attempt to login with invalid user:  vyva
Error: alex moldovan failed $1,000,000 transaction
```

Εικόνα 46 CRLF injection

Η παραπάνω επίθεση αναφέρεται στις καταγραφές, όμως θα μπορούσε να γίνει πολύ πιο επικίνδυνη αν ένας επιτιθέμενος στόχευε σε ολόκληρο το σύστημα καταγραφής μέσω μίας σταδιακής επίθεσης. Για παράδειγμα εάν μία εφαρμογή διαθέτει ένα γραφείο το οποίο διαχειρίζεται, και ελέγχει τις καταγραφές, τότε ένας επιτιθέμενος θα μπορούσε να στείλει ένα XSS στην καταγραφή, το οποίο θα στόχευε σε ολόκληρο το σύστημα καταγραφής. Ο συγκεκριμένος κανόνας αναζητεί την ύπαρξη της μεθόδου **console.log** με non-Literal όρισμα, το οποίο θα μπορούσε να σημαίνει είσοδο από κάποιο χρήστη και σηματοδοτεί μία προειδοποίηση προς τον προγραμματιστή.

Detect timing attacks (Ανίχνευση επιθέσεων χρονισμού)

Μία επίθεση χρονισμού (timing attack), είναι μία αδυναμία ασφάλειας (security vulnerability) η οποία επιτρέπει σε έναν επιτιθέμενο να ανακαλύψει διάφορα κενά σε ένα σύστημα δικτύου μελετώντας τον ακριβή χρόνο που παίρνει στο σύστημα να απαντήσει σε διαφορετικές εισόδους. Μία επίθεση χρονισμού μπορεί να συμβεί κάθε φορά που μία είσοδος επηρεάζει τον χρόνο επεξεργασίας και απάντησης του συστήματος. Ο επιτιθέμενος έχοντας κάποιες γνώσεις της τεχνολογίας που χρησιμοποιεί το σύστημα της εφαρμογής, όπως ο επεξεργαστής ή ακόμα και διάφορες λεπτομέρειες σχετικά με τον διακομιστή, και του χρόνου jitter (δηλαδή της διακύμανσης των χρονικών καθυστερήσεων σε χιλιοστά του δευτερολέπτου μεταξύ των πακέτων δεδομένων σε ένα δίκτυο) μπορεί να απομονώσει το χρόνο επεξεργασίας και απάντησης από τον διακομιστή προς την πλευρά του πελάτη σε διάφορα αιτήματα, έτσι ώστε να εξάγει χρήσιμες πληροφορίες για διάφορα ευαίσθητα δεδομένα.

Η περίπτωση της επίθεσης που πραγματεύεται και προσπαθεί να εντοπίσει ο συγκεκριμένος κανόνας αφορά τις φυσικές συγκρίσεις τύπου string (native string comparisons) μέσα σε καταστάσεις if (if statements). Μία εφαρμογή που χρησιμοποιεί φυσικές συγκρίσεις για να ελέγξει την εγκυρότητα ευαίσθητων δεδομένων όπως κωδικούς, tokens, κλειδιά hash, πιστοποιητικά είναι ευάλωτη σε επιθέσεις χρονισμού. Αυτό συμβαίνει γιατί οι τελεστές που χρησιμοποιούνται σε φυσικές συγκρίσεις όπως (!==, == , = = = , !=) συγκρίνουν δύο όμοια string ελέγχοντας ένα χαρακτήρα κάθε φορά. Για παράδειγμα η σύγκριση μεταξύ των παρακάτω string “foo” & “bar” θα διαρκέσει πολύ λιγότερο από τη σύγκριση των “foo” & “fox”, καθώς στην πρώτη περίπτωση η διαδικασία θα σταματήσει στη σύγκριση του πρώτου χαρακτήρα ενώ στη δεύτερη στη σύγκριση του τρίτου χαρακτήρα. Εάν

ένας επιτιθέμενος ελέγχει την είσοδο της μεταβλητής η οποία θα μπει στη διαδικασία σύγκρισης μέσω φυσικού τρόπου, μπορεί μεταβάλλοντας τις εισόδους και καλύπτοντας διάφορες περιπτώσεις να εξάγει μέσω του χρόνου επεξεργασίας και απάντησης από τον διακομιστή χρήσιμα συμπεράσματα όσον αφορά ευαίσθητα δεδομένα όπως οι κωδικοί πρόσβασης. Κάθε έγκυρος χαρακτήρας που δίνεται στην είσοδο παίρνει στον διακομιστή περισσότερο χρόνο να απαντήσει καθώς πρέπει να λάβει υπόψιν μία έγκυρη σύγκριση παραπάνω. Οι φυσικές συγκρίσεις θεωρούνται αρκετά γρήγορες σε υλοποίηση αλλά και ταυτόχρονα αρκετά αργές. Έχει διαπιστωθεί από την έρευνα των Scott A. Crosby & Dan S. Wallach [29] στις τεχνικές των επιθέσεων χρονισμού ότι ένας επιτιθέμενος μπορεί να μετρήσει γεγονότα (events) με ακρίβεια 15-100μs στο διαδίκτυο. Ο συγκεκριμένος κανόνας αναζητεί φυσικές συγκρίσεις string μέσα σε **if** καταστάσεις στις οποίες τουλάχιστον το ένα από τα δύο ορίσματα που συγκρίνονται αντιστοιχεί στον πίνακα με τις λέξεις κλειδιά που έχουν οριστεί (('password', 'secret', 'api', 'apiKey', 'token', 'auth', 'pass', 'hash').

Detect absence of option: name in cookie of express session module (Ανίχνευση της απουσίας της ιδιότητας name στο πακέτο express session)

Ο συγκεκριμένος κανόνας αναζητεί τις ιδιότητες (properties) του αντικειμένου cookie στο express session πακέτο του express και ενημερώνει το χρήστη για την έλλειψη της ιδιότητας του ονόματος (property:name). Η σημασία της συγκεκριμένης ιδιότητας και τα προβλήματα που μπορεί να προκληθούν από την έλλειψη της αναφέρονται αναλυτικά στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο3.3 (Broken authentication and session management), όπου περιγράφεται η σημασία της διαχείρισης των cookies και του session γενικότερα.

Detect dangerous redirects (Ανίχνευση επικίνδυνων ανακατευθύνσεων)

Ο συγκεκριμένος κανόνας αποσκοπεί στη εύρεση της μεθόδου **res.redirect()** που δεν περιέχει Literal ορίσματα και θα μπορούσαν να υποδηλώνουν είσοδο από χρήστη. Η σημασία του συγκεκριμένου κανόνα και οι πιθανές επιθέσεις που θα μπορούσαν να προκύψουν αναλύθηκαν στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο 3.4 (Unvalidated redirects and forwards).

Detect eval with Expression (Ανίχνευση της μεθόδου eval με έκφραση σαν όρισμα)

Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση της μεθόδου **eval(),** όταν αυτή χρησιμοποιείται με non-Literal ορίσματα, που μπορεί να υποδηλώνουν είσοδο από τον χρήστη. Οι επιθέσεις που μπορεί να γίνουν σε αυτή την περίπτωση αναλύονται στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο 3.1.1 (Command Injection Attacks).

Detect insecure randomness (Ανίχνευση μη ασφαλούς τυχαιότητας)

Η συνάρτηση **Math.random()** επιστρέφει έναν ψευδοτυχαίο αριθμό που κυμαίνεται στην περιοχή 0-1 (συμπεριλαμβανομένου του 0 αλλά όχι του 1), με περίπου ομοιόμορφη κατανομή σε αυτό το εύρος, ο οποίος στη συνέχεια μπορεί να κλιμακωθεί στο επιθυμητό εύρος του προγραμματιστή. Η υλοποίηση επιλέγει τον αρχικό 'σπόρο' για τον αλγόριθμο δημιουργίας τυχαίων αριθμών, ο οποίος δεν μπορεί να επιλεγεί ή να επαναληφθεί από τον προγραμματιστή. Η συγκεκριμένη συνάρτηση χρησιμοποιείται συχνά για τη δημιουργία μη προβλέψιμων αριθμών, όπως τα τυχαία tokens, τα IDs ή τα UUIDs. Ωστόσο η συγκεκριμένη συνάρτηση θεωρείται κρυπτογραφικά μη ασφαλής. Μπορεί να παράγει προβλέψιμες τιμές

και ως εκ τούτου δεν είναι ασφαλής στη χρήση σε θέματα ασφάλειας. Ο συγκεκριμένος κανόνας ανιχνεύει τη χρήση της συγκεκριμένης συνάρτησης με σκοπό να προειδοποιήσει τον προγραμματιστή.

Detect non-Literal arguments in require method (Ανίχνευση δυναμικών κλήσεων της μεθόδου require)

Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση της μεθόδου require() (συνάρτηση για τη εισαγωγή module που βρίσκονται σε ξεχωριστά αρχεία) η οποία διαθέτει non-Literal (όρισμα που δεν είναι string) ορίσματα που μπορεί να υποδηλώνουν είσοδο από χρήστη. Η βασική λειτουργία αυτής της συνάρτησης είναι ότι διαβάζει ένα JavaScript αρχείο, το εκτελεί και στη συνέχεια προωθεί το επιστρεφόμενο αντικείμενο. Εάν ο προγραμματιστής επιτρέπει τη λειτουργία των δυναμικών require, τότε το σύστημα μπορεί να αποκτήσει ανεπιθύμητη συμπεριφορά. Υπάρχουν πολλοί τρόποι με τους οποίους ένας επιτιθέμενος μπορεί να εκμεταλλευτεί τη συγκεκριμένη αδυναμία και ο πιο συνηθισμένος είναι να εξαναγκάσει την εφαρμογή να φορτώσει ένα αρχείο το οποίο δεν υπάρχει, έχοντας ως αποτέλεσμα η εφαρμογή να καταρρεύσει. Η περίπτωση αυτή ανήκει στην κατηγορία των DOS (Denial Of Service) επιθέσεων.

```
var user_input;
var dynamic_require = require(user_input)
```

Εικόνα 47 Dynamic require

Στην παραπάνω εικόνα φαίνεται η περίπτωση μίας δυναμικής κλήσης της μεθόδου require. Το Node.js δέχεται οποιοδήποτε file extension μέσα στην require χωρίς να επικυρώσει ότι το συγκεκριμένο αρχείο είναι js. Συνεπώς εάν η μεταβλητή της εικόνας είναι ελεγχόμενη από τον χρήστη τότε μπορεί ο επιτιθέμενος να φορτώσει .png, .jpeg αρχεία τα οποία περιέχουν κακόβουλη JavaScript.

Detect no-SQL injection (Ανίχνευση no-SQL διάχυσης)

Ένας από τους τρόπους με τους οποίους ένας επιτιθέμενος μπορεί να κατασκευάσει μία επίθεση σε βάσεις όπως η MongoDB είναι μέσω του $where operator. Ο μηχανισμός επίθεσης θα μπορούσε να είναι η περίπτωση όπου ο συγκεκριμένος τελεστής (operator) λαμβάνει μία JavaScript συνάρτηση και επεξεργάζεται εισόδους χρήστη μέσα σε αυτή. Στο παρακάτω παράδειγμα απεικονίζεται ένα query το οποίο είναι ευάλωτο σε no-SQL injection.

```
db.myCollection.find({
 active: true,
 $where: function() { return obj.age < post_user_input; }
});
```

Εικόνα 48 vulnerable to no-sql injection query

Στο συγκεκριμένο παράδειγμα η είσοδος του χρήστη επιτιθέμενου (post_user_input) χρησιμοποιείται με μη ασφαλή τρόπο. Εάν ένας επιτιθέμενος περάσει μία έγκυρη JavaScript πρόταση (statement), όπως **0; while(1);** σαν τιμή για τη συγκεκριμένη μεταβλητή, το query θα τρέξει σε έναν ατέρμων βρόχο, κάνοντας τη βάση ανίκανη να απαντήσει. Το μειονέκτημα του $where σαν τελεστή είναι ότι παρέχει στον επιτιθέμενο τη δυνατότητα να κατασκευάσει

μία κακόβουλη είσοδο χρησιμοποιώντας JavaScript, το οποίο του παρέχει αυτόματα τεράστια ευελιξία και επιλογές. Ο συγκεκριμένος κανόνας αναζητεί τη χρήση του $where operator σε no-SQL queries έτσι ώστε να προειδοποιήσει τον χρήστη.

Detect option: multipleStatements in mysql.js module (Ανίχνευση της επιλογής multipleStatements στο πακέτο mysql.js)

Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση της επιλογής multipleStatements στο MySQL.js πακέτο.

```
var connection = mysql.createConnection({multipleStatements: true});
```

Εικόνα 49 Ενεργοποίηση πολλαπλών καταστάσεων στην SQL

Οι πολλαπλές καταστάσεις (multiple statements) σε ένα SQL query είναι ιδιαίτερα ευάλωτες σε SQL injection επιθέσεις, όπως αναλύθηκε στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο 3.1.2 (Database Injection Attacks).

Detect option :rejectUnauthorized in https core module (Ανίχνευση της επιλογής rejectUnauthorized στο πακέτο https)

Το rejectUnauthorized είναι μία ιδιότητα (property option) της μεθόδου request() η οποία ανήκει στο https βασικό πακέτο του Node.js.

```
var req = https.request({
    hostname: 'example.com',
    port: 443,
    path: '/',
    method: 'GET',
    rejectUnauthorized: false
}, function() {});
```

Εικόνα 50 option rejectUnauthorized:false

Εάν η συγκεκριμένη ιδιότητα είναι αληθής (true) τότε το πιστοποιητικό (certificate) του διακομιστή πιστοποιείται σε σχέση με την λίστα των παρεχόμενων Cas (Certificate authority). Σε αντίθετη περίπτωση, δηλαδή εάν είναι ψευδής (false), δεν επικυρώνεται η ταυτότητα του διακομιστή με αποτέλεσμα η εφαρμογή να είναι ευάλωτη σε MITM (Man In The Middle) επιθέσεις.

Detect option:unsafe in serialize JavaScript module (Ανίχνευση της επιλογής unsafe στο πακέτο serialize)

Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση της ιδιότητας (property) unsafe που παρέχεται στις επιλογές (options) του πακέτου serialize-JavaScript.

```
serialize(obj, {unsafe: true});
```

Εικόνα 51 Επιλογή unsafe στο serialize module

Με την ενεργοποίηση της συγκεκριμένης ιδιότητας (unsafe:true), η μέθοδος serialize() λαμβάνει το σήμα ότι ο προγραμματιστής θέλει να κάνει απευθείας μετατροπή (straight

conversion) χωρίς να προστατεύεται από XSS. Συνεπώς οι χαρακτήρες HTML και τα JavaScript line terminators δεν ελέγχονται.

Detect non-Literal argument in runInThisContext method of vm.js core module (Ανίχνευση της μεθόδου runInThisContext του πακέτου vm με μη κυριολεκτικά ορίσματα)

Το **vm** (virtual machine) core module του Node.js επιτρέπει στον προγραμματιστή να τρέξει κώδικα JavaScript σε μία νέα Node διαδικασία που δεν έχει πρόσβαση στην standard node βιβλιοθήκη (δηλαδή δεν υπάρχει πρόσβαση στο process ή στο console). Στο παρακάτω παράδειγμα απεικονίζεται αυτή η διαδικασία.

```
let a = 0;
let result = vm.runInNewContext('a += 1', {a})

console.log(a) // 0
```

Εικόνα 52 runInNewContext method

Στην παραπάνω εικόνα φαίνεται ότι η μεταβλητή a, δεν μεταβλήθηκε, συνεπώς δεν υπάρχει πρόσβαση στο παρόν context. Όμως στην περίπτωση που ένας προγραμματιστής θέλει να έχει πρόσβαση στο παρόν context, μπορεί να χρησιμοποιήσει την μέθοδο **runInThisContext**, όπως απεικονίζεται στην παρακάτω εικόνα.

```
a = 0;
vm.runInThisContext('a += 1')

console.log(a) // 1
```

Εικόνα 53 runInThisContext method

Όταν χρησιμοποιείται το παρόν context εκτέλεσης, ο κώδικας που εκτελείται μέσα στην εικονική μηχανή (virtual machine) έχει πρόσβαση στις global ορισμένες μεταβλητές. Αυτό έχει σαν αποτέλεσμα τα προβλήματα να παραμένουν ίδια με την eval. Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση της συνάρτησης **runInThisContext** που λαμβάνει non-Literal ορίσματα τα οποία μπορεί να υποδηλώνουν είσοδο από χρήστη. Τα προβλήματα που μπορεί να αποφέρει η συγκεκριμένη περίπτωση είναι ίδια με την περίπτωση της eval όπου αναλύθηκε στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο 3.1.1 (Command Injection Attacks).

Detect security misconfiguration in cookie of express session module (Ανίχνευση λανθασμένης διαμόρφωσης στο cookie του πακέτου express session)

Ο συγκεκριμένος κανόνας αποσκοπεί στην αναζήτηση της ύπαρξης της ιδιότητας (property) secure του αντικειμένου cookie στο express-session πακέτο. Θέτοντας τη συγκεκριμένη ιδιότητα (property) ως αληθής (true) όπως αναφέρεται και στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο 3.3 (Broken Authentication and session management) πραγματοποιούνται μόνο HTTPS συνδέσεις.

Detect SQL injection (Ανίχνευση SQL διάχυσης)

Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση queries στην SQL τα οποία δεν είναι παραμετροποιημένα, αλλά κτίζονται δυναμικά και με non-Literal ορίσματα τα οποία μπορεί

να υποδηλώνουν είσοδο από τον χρήστη. Οι επιπτώσεις μίας τέτοιας αδυναμίας και οι επιθέσεις που μπορούν να πραγματοποιηθούν σε μία τέτοια περίπτωση αναλύθηκαν στο Κεφάλαιο 3 και πιο συγκεκριμένα στην Παράγραφο 3.1.2 (Database Injection Attacks).

Detect non-Literal reg-exp (Ανίχνευση της μεθόδου RegExp με μη κυριολεκτικά ορίσματα)

Ο συγκεκριμένος κανόνας αποσκοπεί στην εύρεση της μεθόδου RegExp() που διαθέτει non-Literal ορίσματα και μπορεί να υποδηλώνουν είσοδο από τον χρήστη. Ο constructor RegExp δημιουργεί μία κανονική έκφραση για την αντιστοίχιση κειμένου με ένα συγκεκριμένο πρότυπο.

```
var user_input;
var regex1 = /\w+/;
var regex2 = new RegExp('\\w+' + user_input);
```

Εικόνα 54 Regular expression with user input

Στην παραπάνω εικόνα απεικονίζεται η δημιουργία μίας κανονικής έκφρασης η οποία παράγεται σε συνδυασμό με μία είσοδο από τον χρήστη. Στη συγκεκριμένη περίπτωση ένας επιτιθέμενος θα μπορούσε να προκαλέσει DOS (Denial Of Service) επίθεση όπως εξηγήθηκε στο Κεφάλαιο 3 με τις κατηγορίες των πιο γνωστών επιθέσεων και πιο συγκεκριμένα στην Παράγραφο 3.5 (DOS).

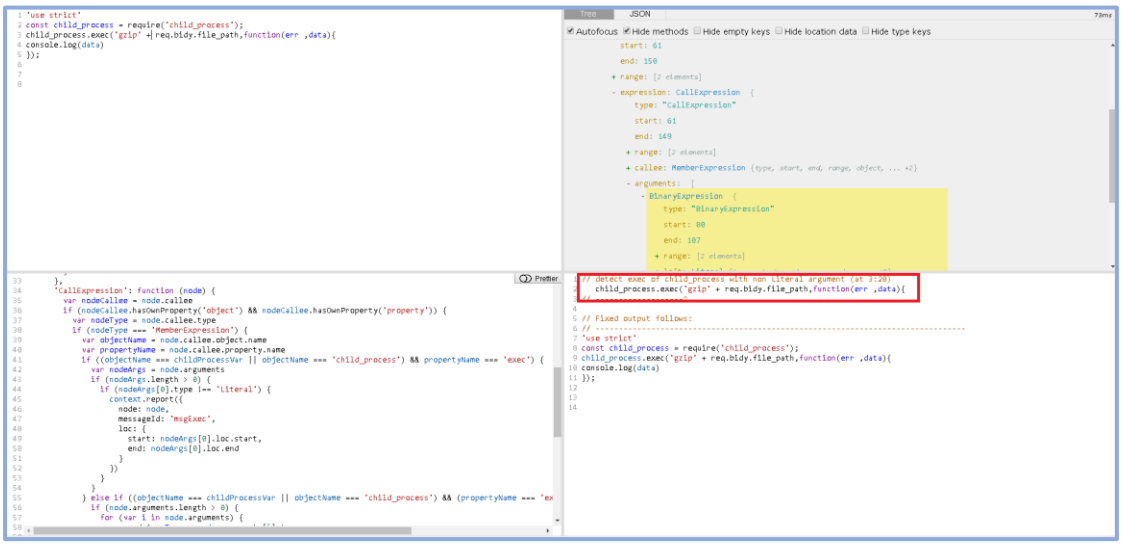
Detect helmet without noCache (Ανίχνευση χρήσης του helmet χωρίς τη χρήση του noCache)

Το helmet είναι ένα middleware του express module το οποίο συνεισφέρει στην ασφάλεια της εφαρμογής θέτοντας διάφορες HTTP κεφαλίδες (headers). Η εξορισμού διαμόρφωση του συγκεκριμένου middleware δεν συμπεριλαμβάνει όλα τα headers που είναι χρήσιμα για την ασφάλεια της εφαρμογής. Ο συγκεκριμένος κανόνας αναζητεί τη χρήση του middleware από τον προγραμματιστή (app.use(helmet()), με την εξορισμού του διαμόρφωση (default configuration), και ενημερώνει το χρήστη για τα middleware που αφορούν την ασφάλεια και δεν βρίσκονται στην εξορισμού διαμόρφωση, όπως το noCache όπου χρησιμεύει στην απενεργοποίηση της Cache στη μεριά του χρήστη.

5.4 Υλοποίηση μέσω AST explorer

Η υλοποίηση των κανόνων έγινε μέσω του εργαλείου AST explorer το οποίο αναλύθηκε στο ΚΕΦΑΛΑΙΟ 2 και πιο συγκεκριμένα στην παράγραφο (2.4.1 AST explorer). Παρακάτω παρουσιάζεται η διαδικασία υλοποίησης του κανόνα detect-child-process (η λογική του οποίου αναλύθηκε παραπάνω) μέσω του εργαλείου AST explorer.

Εικόνα 55 λειτουργία του AST explorer



Εικόνα 56 υλοποίηση του κανόνα detect child_process στο AST explorer

Στην Εικόνα 56 παρουσιάζονται τα βήματα για την υλοποίηση ενός κανόνα μέσω του AST explorer, ενώ στην Εικόνα 57 παρουσιάζεται η υλοποίηση του κανόνα detect-child-process (που αναλύθηκε στην προηγούμενη παράγραφο), όπως έγινε στο εργαλείο. Σε κόκκινο πλαίσιο φαίνεται η έξοδος που παράγεται, από την παραβίαση του κανόνα.
Στο πάνω αριστερά πλαίσιο της Εικόνας 57 απεικονίζεται ο πηγαίος κώδικας που πρόκειται να ελεγχθεί από τον κανόνα όπως φαίνεται παρακάτω. Στη συγκεκριμένη περίπτωση αφορά τον κανόνα detect-child-process όπου πρόκειται να ελεγχθεί η μέθοδος exec για πιθανό non-Literal πρώτο όρισμα.

```
'use strict'
const child_process = require('child_process');
child_process.exec('gzip' + req.bidy.file_path,function(err ,data){
console.log(data)
});

```

Εικόνα 57 Λειτουργία AST 1ο στάδιο

Στο πάνω δεξιά πλαίσιο της Εικόνας 57 απεικονίζεται ο πηγαίος κώδικας όπως έχει αναλυθεί από τον parser espree και ο σχηματισμός του AST.

Εικόνα 58 Απεικόνιση συντακτικού δέντρου στον AST explorer

Εικόνα 59 Λειτουργία AST 2ο στάδιο

Στο κάτω δεξιά πλαίσιο της Εικόνας 57 απεικονίζεται η υλοποίηση του κανόνα και το visitor pattern δηλαδή η διάσχιση του AST με σκοπό τον εντοπισμό του λάθους.

```
'use strict'

module.exports = {
  meta: {
    type: 'suggestion',
    messages:
      {
        msgExec: 'detect exec of child_process with non Literal argument',
        msgShell: 'detect option shell:true in execFile or spawn of child_process'
      },
    docs: {
      description: 'detect exec with non Literal argument',
      category: 'possible errors',
      recommended: true
    },
    fixable: null
  },

  create: function (context) {
    var childProcessVar

    return {
      'VariableDeclaration': function (node) {
        if (node.declarations[0].init.hasOwnProperty('callee')) {
          if (node.declarations[0].init.callee.hasOwnProperty('name')) {
            if (node.declarations[0].init.callee.name === 'require' &&
          node.declarations[0].init.arguments[0].type === 'Literal' &&
          node.declarations[0].init.arguments[0].value === 'child_process') {
              childProcessVar = node.declarations[0].id.name
            }
          }
        }
      },
      'CallExpression': function (node) {
        var nodeCallee = node.callee
        if (nodeCallee.hasOwnProperty('object') && nodeCallee.hasOwnProperty('property')) {
          var nodeType = node.callee.type
          if (nodeType === 'MemberExpression') {
            var objectName = node.callee.object.name
            var propertyName = node.callee.property.name
            if ((objectName === childProcessVar || objectName === 'child_process') && propertyName === 'exec') {
              var nodeArgs = node.arguments
              if (nodeArgs.length > 0) {
                if (nodeArgs[0].type !== 'Literal') {
                  context.report({
                    node: node,
                    messageId: 'msgExec',
                    loc: {
                      start: nodeArgs[0].loc.start,
                      end: nodeArgs[0].loc.end
                    }
                  })
                }
              }
```

Εικόνα 60 Υλοποίηση κανόνα detect child_process στον AST explorer

Εικόνα 61 Λειτουργία AST 3ο στάδιο

Τέλος στο κάτω δεξιά πλαίσιο της Εικόνας 57 απεικονίζεται η έξοδος, δηλαδή η περίπτωση που ο κανόνας ενεργοποιείται και το μήνυμα που λαμβάνει ο προγραμματιστής μέσω του ESLint

```
1 // detect exec of child_process with non Literal argument (at 3:20)
2    child_process.exec('gzip' + req.bidy.file_path,function(err ,data){
3 // -------------------^
```

Εικόνα 62 Παραγόμενη έξοδος από την ενεργοποίηση του κανόνα

Εικόνα 63 Λειτουργία AST 4ο στάδιο

5.5 Unit Tests

Ο έλεγχος για τη σωστή λειτουργία των κανόνων γίνεται μέσω των unit tests. Στο αρχείο tests/lib/rules υπάρχει ένα αρχείο ελέγχου που αντιστοιχεί σε κάθε κανόνα. Ο έλεγχος γίνεται μέσω του **eslint.RuleTestser** το οποίο αποτελεί ένα βοηθητικό εργαλείο για την υλοποίηση του ελέγχουν των κανόνων του ESLint. Χρησιμοποιείται εσωτερικά για τους κανόνες που συνοδεύουν το ESLint (bundled Rules), ενώ μπορεί να χρησιμοποιηθεί και για τα plugin. Το αντικείμενο ruleTester δημιουργείται μέσω του constructor RuleTester(). Η μέθοδος RuleTester.run() χρησιμοποιείται για την εκτέλεση του ελέγχου (tests), και δέχεται τα παρακάτω ορίσματα:

- Το όνομα του κανόνα για τον οποίο θα γίνει ο απαιτούμενος έλεγχος για τη σωστή λειτουργία του
- Το αντικείμενο του κανόνα το οποίο εξάγεται από τον πηγαίο φάκελο που περιέχει την υλοποίηση του κανόνα
- Ένα αντικείμενο το οποίο περιέχει σαν ιδιότητες δύο πίνακες (valid, invalid) οι οποίοι περιέχουν τις υποθέσεις ελέγχου (test cases.)

Παρακάτω παρουσιάζεται η περίπτωση χρήσης του RuleTester για τον κανόνα detect-dangerous-redirects.

```
ruleTester.run('detect-dangerous-redirects', rule, {

  valid: [
    {
      code: valid
    }
  ],

  invalid: [
    {
      code: invalid,
      errors: [{
        message: ERROR_MSG
      }]
    },
    {
      code: invalidSecond,
      errors: [{
        message: ERROR_MSG
      }]
    }
  ]
})
```

Εικόνα 64 RuleTester

invalid: `const invalid = 'res.redirect(foo + "foo");`

valid: `const valid = 'res.redirect("foo");'`

ERROR_MSG: `const ERROR_MSG = 'detect res.redirect() with non literal argument'.`

Στην παραπάνω περίπτωση σαν μη έγκυρη (invalid) θεωρείται η περίπτωση στην οποία γίνεται κλήση της μεθόδου require() χωρίς αυτή να περιέχει non-Literal ορίσματα. Στο συγκεκριμένο παράδειγμα το όρισμα αποτελεί μία δυαδική έκφραση (Binary Expression), στο οποίο η μεταβλητή foo θα μπορούσε να σηματοδοτεί είσοδο από το χρήστη.

5.6 Npm πακέτο

Το σύνολο των κανόνων (plugin) μεταμορφώθηκε σε npm πακέτο, έτσι ώστε να είναι διαθέσιμο για χρήση. Η εγκατάσταση του μπορεί να γίνει μέσω της παρακάτω εντολής.

- `npm install –save-dev eslint-plugin-security-node`

Για τη χρήση του πακέτου πρέπει να προστεθεί η παραπάνω διαμόρφωση στο αρχείο διαμόρφωσης (eslintrc) του ESLint (Κεφάλαιο 2, Παράγραφος 2.3.3).

```
"plugins": [
 "security-node"
],
"extends": [
 "plugin: security-node/recommended"
 ]
```

Εικόνα 65 διαμόρφωση του αρχείου eslintrc

Στο πεδίο “extends” ο χρήστης μπορεί να συμπεριλάβει την προτεινόμενη διαμόρφωση όσον αφορά την καλύτερη χρήση των κανόνων έτσι όπως αυτή έχει υλοποιηθεί στο αρχείο index.js. Η προτεινόμενη διαμόρφωση προβλέπει την ενεργοποίηση όλων των κανόνων σαν προειδοποιήσεις (warnings) και απεικονίζεται στην παρακάτω εικόνα.

```
configs: {
  recommended: {
    plugins: [
      'security-node'
    ],
    rules: {
      'security-node/detect-absence-of-name-option-in-exrpress-session': 'warn',
      'security-node/detect-buffer-unsafe-allocation': 'warn',
      'security-node/detect-child-process': 'warn',
      'security-node/detect-crlf': 'warn',
      'security-node/detect-dangerous-redirects': 'warn',
      'security-node/detect-eval-with-expr': 'warn',
      'security-node/detect-helmet-without-nocache': 'warn',
      'security-node/detect-html-injection': 'warn',
      'security-node/detect-insecure-randomness': 'warn',
      'security-node/detect-non-literal-require-calls': 'warn',
      'security-node/detect-nosql-injection': 'warn',
      'security-node/detect-option-multiplestatements-in-mysql': 'warn',
      'security-node/detect-option-rejectunauthorized-in-nodejs-httpsrequest': 'warn',
      'security-node/detect-option-unsafe-in-serialize-javascript-npm-package': 'warn',
      'security-node/detect-possible-timing-attacks': 'warn',
      'security-node/detect-runinthiscontext-method-in-nodes-vm': 'warn',
      'security-node/detect-security-missconfiguration-cookie': 'warn',
      'security-node/detect-sql-injection': 'warn',
      'security-node/disable-ssl-across-node-server': 'warn',
      'security-node/non-literal-reg-expr': 'warn'
    }
  }
}
```

Εικόνα 66 Προτεινόμενη διαμόρφωση των κανόνων στο αρχείο index.js

Για την υλοποίηση της προτεινόμενης διαμόρφωσης προς τον χρήστη (αρχείο index.js) όλοι οι κανόνες στο αντικείμενο rules έχουν σαν πρόθεμα το όνομα του πακέτου, ενώ για τη διάκριση τους χρησιμοποιείται το iD του κάθε κανόνα. Τέλος για την ενεργοποίηση τους σαν ειδοποιήσεις (warnings) λαμβάνουν την τιμή ‘warn’.

ΚΕΦΑΛΑΙΟ 6

ΑΠΟΤΕΛΕΣΜΑΤΑ

Σε αυτό το κεφάλαιο παρουσιάζονται αναλυτικά τα αποτελέσματα ύστερα από την εφαρμογή του npm πακέτου σε διάφορα αποθετήρια της JavaScript από το GitHub.

6.1 Αποθετήρια που ελέγχθηκαν και προϋποθέσεις

Παρακάτω παρουσιάζεται ο πίνακας με τα αποθετήρια που χρησιμοποιήθηκαν για τον έλεγχο του συνόλου των κανόνων, καθώς και τα διάφορα περιβάλλοντα που χρησιμοποιούν, τα οποία είναι απαραίτητα για την ενεργοποίηση ορισμένων κανόνων.

Πίνακας 4 Αποθετήρια που χρησιμοποιήθηκαν

Αποθετήρια GitHub	tags
axios	JavaScript/Node.js
bookshelf	JavaScript/Node.js/MySQL
express Cart	JavaScript/Node.js
hexo	JavaScript/Node.js
Hakathon-starter	JavaScript/Node.js/express
Keystone	JavaScript/Node.js
Kraken-js	JavaScript/Node.js/express
mean	JavaScript/Node.js/express/mongo DB
Mongo-express	JavaScript/Node.js/express/mongo DB
NodeBB	JavaScript/Node.js/mongo DB
N-blog	JavaScript/Node.js/express/mongo DB
Node-elm	JavaScript/Node.js/express/mongo DB
Node Goat	JavaScript/Node.js
Node-postgres	JavaScript/Node.js
objectionjs	JavaScript/Node.js/MySQL
pupeeter	JavaScript/Node.js
sails	JavaScript
sequelize	JavaScript/MySQL
vault-express	JavaScript/Node.js/express/helmet
express	JavaScript/Node.js/express
express-admin	JavaScript/Node.js/express/MySQL
Node.js-learning-guide	JavaScript/Node.js/express

Στη συνέχεια στον Πίνακα 5 παρουσιάζονται οι προϋποθέσεις (με βάση τα tags που αναφέρθηκαν παραπάνω) τις οποίες θα πρέπει να πληρούν τα αποθετήρια ώστε να ενεργοποιηθούν ορισμένοι κανόνες.

Σημείωση: Εξαίρεση αποτελεί το αποθετήριο Node Goat το οποίο έχει κατασκευαστεί διαθέτοντας σκοπίμως αρκετές αδυναμίες με αποτέλεσμα να είναι υποψήφιο για ενεργοποίηση πολλών κανόνων.

Πίνακας 5 Προϋποθέσεις ενεργοποίησης ορισμένων κανόνων

Tags	Κανόνες
express	1)detect-security-misconfiguration-cookie 2)detect-absence-of-name-option-in-express-session
helmet	detect-helmet-without-nocache
MySQL	1)detect-sql-injection 2)detect-option-multiple-statements-in-MySQL
Mongo DB	detect-no-sql-injection

6.2 Διαδικασία συλλογής αποτελεσμάτων

Για τον έλεγχο της λειτουργίας των κανόνων έγινε εγκατάσταση του npm πακέτου (Παράγραφος 5.6) και προστέθηκε στο αρχείο διαμόρφωσης eslintrc του κάθε αποθετηρίου το περιεχόμενο της Εικόνας 66 για την ενεργοποίηση του συνόλου των κανόνων με την προτεινόμενη χρήση. Παρακάτω παρουσιάζεται ένα κομμάτι των αποτελεσμάτων από το τερματικό που προέκυψε ύστερα από τον έλεγχο του αποθετηρίου NodeBB μέσω του npm πακέτου.

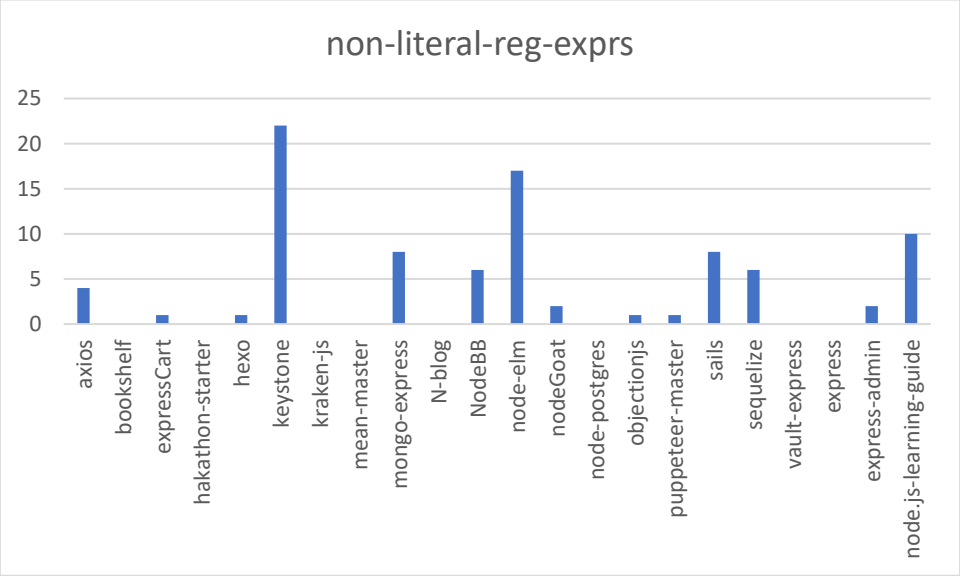
```
/home/giannis/testMyPlugin/NodeBB-master/src/routes/feeds.js
  52:2  warning  Potential timing attack, right side: token  security-node/detect-possible-timing-attacks

/home/giannis/testMyPlugin/NodeBB-master/src/routes/index.js
  160:16  warning  detect res.redirect() with non literal argument  security-node/detect-dangerous-redirects
  178:16  warning  detect res.redirect() with non literal argument  security-node/detect-dangerous-redirects

/home/giannis/testMyPlugin/NodeBB-master/src/socket.io/index.js
  175:24  warning  Found require with non-literal argument  security-node/detect-non-literal-require-calls

/home/giannis/testMyPlugin/NodeBB-master/src/upgrade.js
   37:30  warning  Found require with non-literal argument  security-node/detect-non-literal-require-calls
   38:30  warning  Found require with non-literal argument  security-node/detect-non-literal-require-calls
   56:33  warning  Found require with non-literal argument  security-node/detect-non-literal-require-calls
  154:32  warning  Found require with non-literal argument  security-node/detect-non-literal-require-calls

/home/giannis/testMyPlugin/NodeBB-master/src/webserver.js
  158:18  warning  detect absence of option:name in express-session                 security-node/detect-absence-of-name-option-in-exrpress-session
  158:18  warning  detect absence of option cookie:secure in cookie express-session  security-node/detect-security-missconfiguration-cookie

/home/giannis/testMyPlugin/NodeBB-master/test/mocks/databasemock.js
  231:25  warning  Found require with non-literal argument  security-node/detect-non-literal-require-calls

/home/giannis/testMyPlugin/NodeBB-master/test/socket.io.js
  6:1  warning  detect process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0"  security-node/disable-ssl-across-node-server
```

Εικόνα 67 Μέρος των αποτελεσμάτων στο τερματικό ύστερα από τον έλεγχο του αποθετηρίου NodeBB

Όπως φαίνεται στην εικόνα οι κανόνες που ενεργοποιήθηκαν στο συγκεκριμένο κομμάτι του αποθετηρίου NodeBB έχουν τη μορφή προειδοποίησης (warning). Ο χρήστης έχει τη δυνατότητα της πλήρης διαμόρφωσης του συνόλου των κανόνων έτσι όπως προβλέπεται από τον ESLint και αναλύθηκε στο Κεφάλαιο 2, Παράγραφος 2.3.3.

6.3 Ανάλυση αποτελεσμάτων για κάθε κανόνα

Παρακάτω παρουσιάζονται οι πίνακες και τα διαγράμματα των αποτελεσμάτων για κάθε κανόνα σε όλα τα αποθετήρια, έτσι όπως προέκυψαν από τον έλεγχο του συνόλου των κανόνων. Στον οριζόντιο άξονα αναπαρίστανται τα αποθετήρια στα οποία έγινε ο έλεγχος του πακέτου, ενώ στον κάθετο, οι φορές που ενεργοποιήθηκε ο εκάστοτε κανόνας σε καθένα από αυτά.

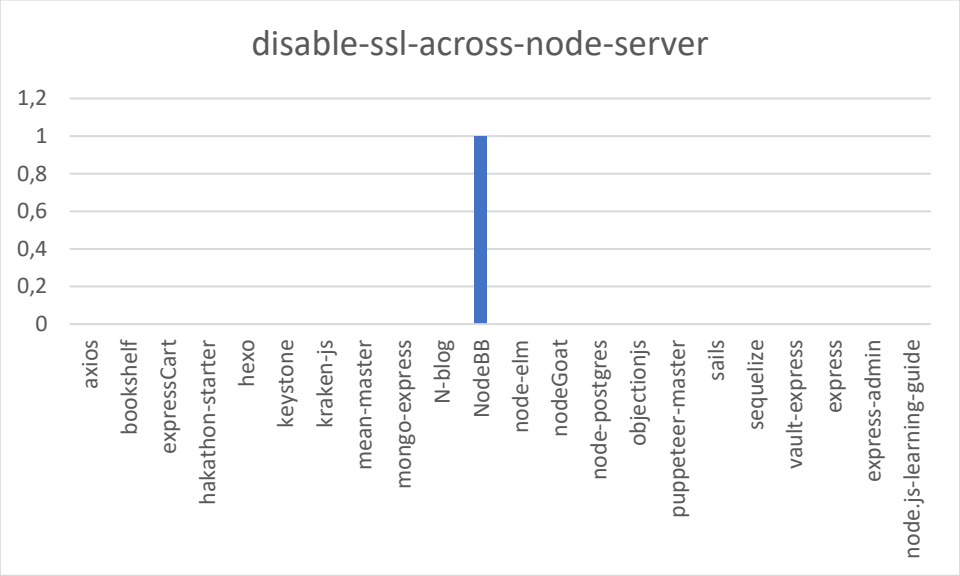


Διάγραμμα 3 Αποτελέσματα κανόνα non-literal-reg-exprs

Πίνακας 6 Αποτελέσματα κανόνα non-literal-reg-exprs

non-literal-reg-exprs	
axios	4
bookshelf	0
express Cart	1
hexo	0
Hakathon-starter	1
Keystone	22
Kraken-js	0
mean	0
Mongo-express	8
NodeBB	6
N-blog	0
Node-elm	17
Node Goat	2
Node-postgres	0
objectionjs	0
pupeeter	1
sails	8
sequelize	6
vault-express	0
express	0
express-admin	2
Node.js-learning-guide	10

non-literal-reg-exprs: Ο συγκεκριμένος κανόνας δεν είχε κάποιο περιορισμό από κάποια τρίτη βιβλιοθήκη όπως οι κανόνες στον Πίνακα 5. Εμφανίστηκε σαν προειδοποίηση στα περισσότερα αποθετήρια που ελέγχθηκαν, ενώ όπως φαίνεται από τη συχνότητα του είναι επιρρεπής στις **fp** (false-positive) περιπτώσεις.



Διάγραμμα 4 Αποτελέσματα κανόνα disable-ssl-across-node-server

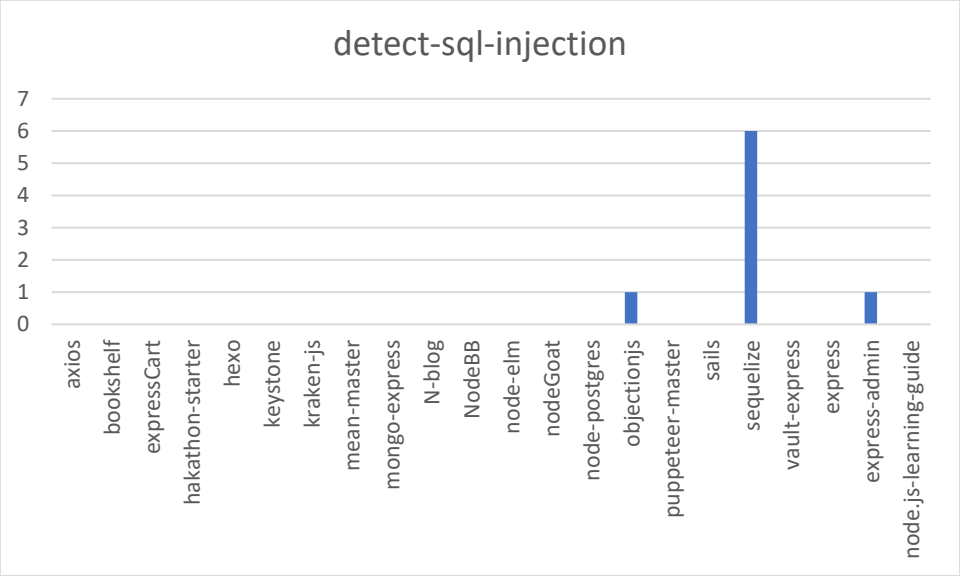
Πίνακας 7 Αποτελέσματα κανόνα disable-ssl-across-node-server

disable-ssl-across-node-server	
Axios	0
bookshelf	0
express Cart	0
Hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
Mean	0
Mongo-express	0
NodeBB	1
N-blog	0
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	0
pupeeter	0
Sails	0
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	0

disable-ssl-across-node-server: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Εμφανίστηκε μόνο μία φορά στο αποθετήριο NodeBB και το κομμάτι του κώδικα που προκάλεσε την ενεργοποίηση του απεικονίζεται στην παρακάτω εικόνα.

```
process.env.NODE_TLS_REJECT_UNAUTHORIZED = '0';

var assert = require('assert');
var async = require('async');
var nconf = require('nconf');
var request = require('request');
var cookies = request.jar();
```

Εικόνα 68 Εμφάνιση του κανόνα disable-ssl-across-node-server στο αποθετήριο NodeBB

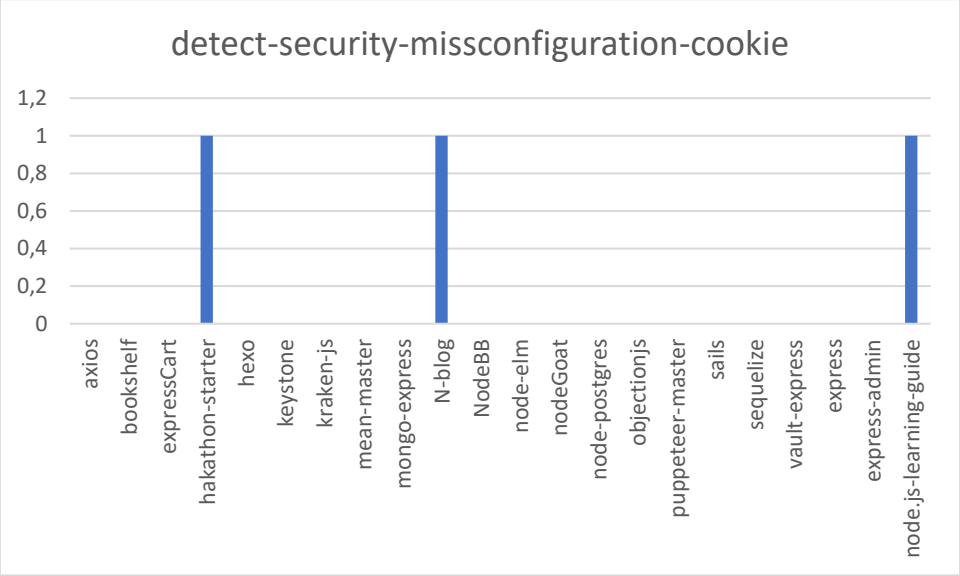


Διάγραμμα 5 Αποτελέσματα κανόνα detect-sql-injection

Πίνακας 8 Αποτελέσματα κανόνα detect-sql-injection

detect-sql-injection	
Axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	0
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	1
pupeeter	0
sails	0
sequelize	6
vault-express	0
express	0
express-admin	1
Node.js-learning-guide	0

detect-sql-injection: Ο συγκεκριμένος κανόνας όπως φανερώνεται από τον Πίνακα 5 απαιτεί την ύπαρξη του πακέτου mysql για την πιθανή ενεργοποίηση του. Από τα αποθετήρια που χρησιμοποιούν την mysql (objectionjs, sequelize, express-admin, Bookshelf), ο κανόνας ενεργοποιήθηκε στα 3/4 από αυτά, που σημαίνει ότι βρέθηκε η χρήση δυναμικών queries.



Διάγραμμα 6 Αποτελέσματα κανόνα detect-security-misconfiguration-cookie

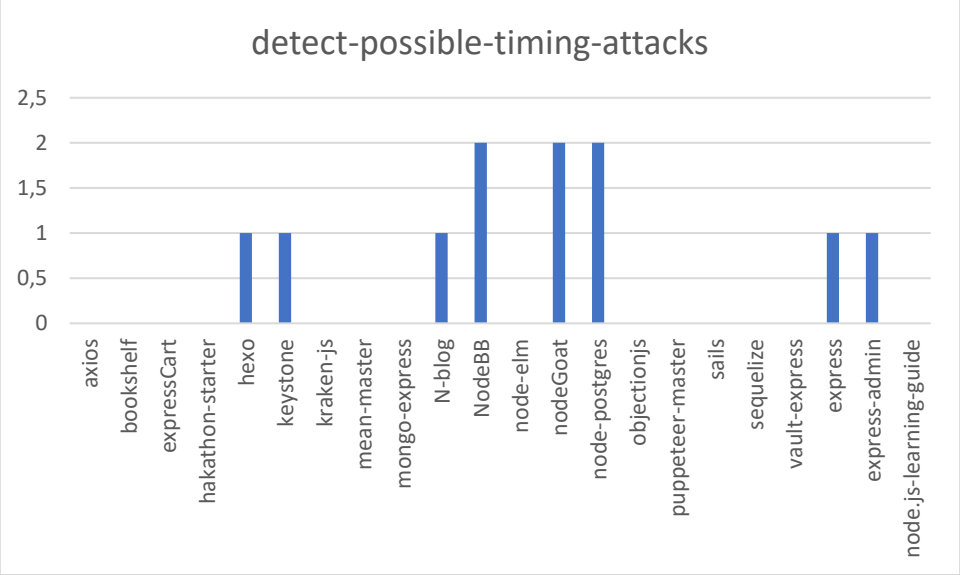
Πίνακας 9 Αποτελέσματα κανόνα detect-security-misconfiguration-cookie

detect-security-misconfiguration-cookie	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	1
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	1
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	1

detect-security-misconfiguration-cookie: Ο συγκεκριμένος κανόνας όπως φανερώνεται από τον Πίνακα 5 απαιτεί την ύπαρξη της βιβλιοθήκης express και πιο συγκεκριμένα του πακέτου express-session για την πιθανή ενεργοποίηση του. Από τα αποθετήρια που χρησιμοποιούν τη συγκεκριμένη βιβλιοθήκη (πίνακας 4) Hakathon-starter, Mongo-Express, N-Blog, Node-elm, vault-express, express-admin και Node.js-learning-guide ο κανόνας ενεργοποιήθηκε στα 3 από αυτά. Στην παρακάτω εικόνα απεικονίζεται η μία από τις 3 περιπτώσεις του διαγράμματος στο αποθετήριο hakathon-starter.

```
app.use(session({
  resave: true,
  saveUninitialized: true,
  secret: process.env.SESSION_SECRET,
  cookie: { maxAge: 1209600000 }, // two weeks in milliseconds
  store: new MongoStore({
    url: process.env.MONGODB_URI,
    autoReconnect: true,
  })
}));
```

Εικόνα 69 Εμφάνιση του κανόνα detect-security-misconfiguration-cookie στο αποθετήριο hakathon-starter



Διάγραμμα 7 Αποτελέσματα κανόνα detect-possible-timing-attacks

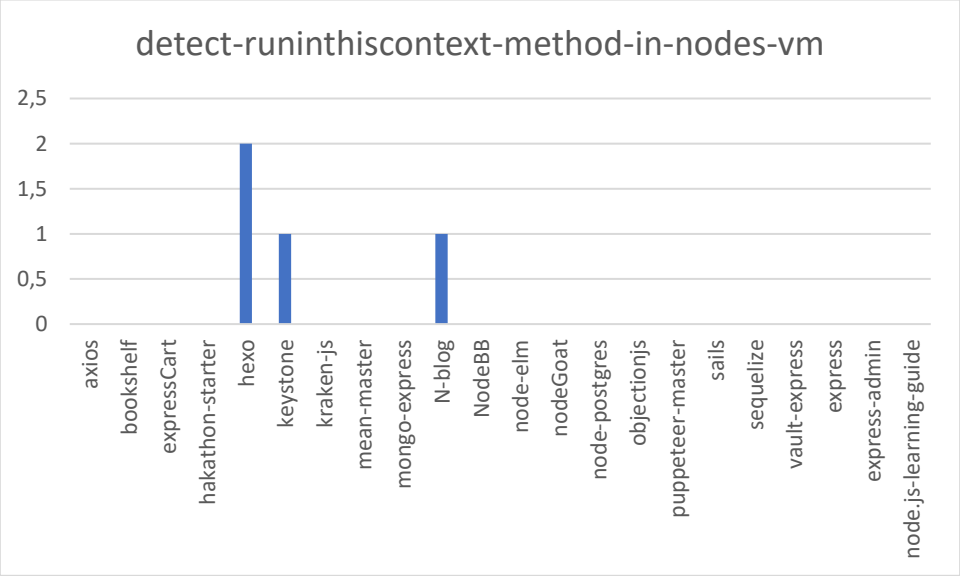
Πίνακας 10 Αποτελέσματα κανόνα detect-possible-timing-attacks

detect-potential-timing-attacks	
axios	0
bookshelf	0
express Cart	0
hexo	1
Hakathon-starter	0
Keystone	1
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	2
N-blog	1
Node-elm	2
Node Goat	2
Node-postgres	2
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	1
express-admin	1
Node.js-learning-guide	0

detect-potential-timing-attacks: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Ενεργοποιήθηκε σε 8 από τα 22 αποθετήρια που ελέγχθηκε, εντοπίζοντας περιπτώσεις επικίνδυνων συγκρίσεων όπως φανερώνεται στην παρακάτω εικόνα στο αρχείο feed.js του αποθετηρίου NodeBB.

```
    const userToken = await db.getObjectField('user:' + uid, 'rss_token');
    if (userToken !== token) {
        await user.auth.logAttempt(uid, req.ip);
        return helpers.notAllowed(req, res);
    }
```

Εικόνα 70 Εμφάνιση του κανόνα detect-potential-timing-attacks στο αποθετήριο NodeBB

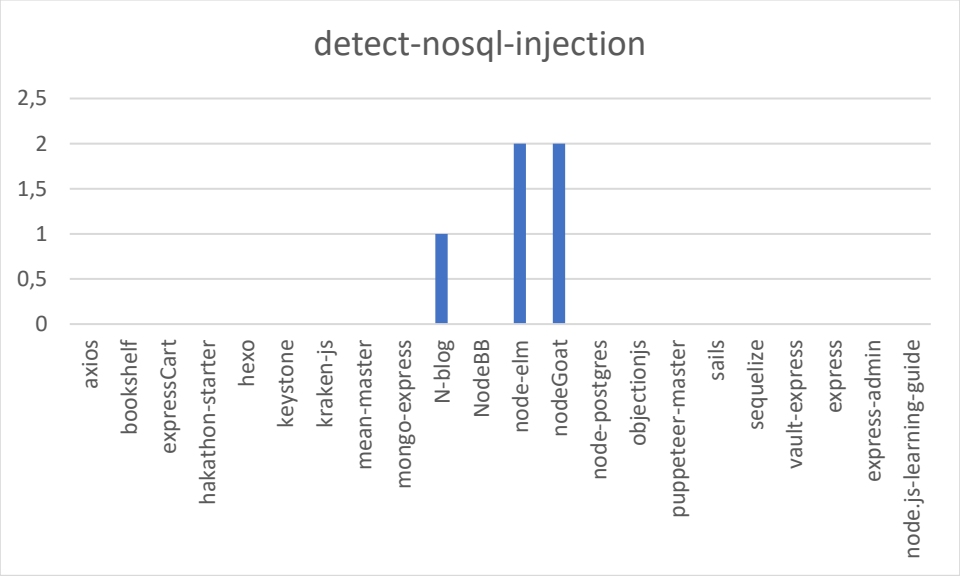


Διάγραμμα 8 Αποτελέσματα κανόνα detect-runinthiscontext-method-in-nodes-vm

Πίνακας 11 Αποτελέσματα κανόνα detect-runinthiscontext-method-in-nodes-vm

detect-runinthiscontext-method-in-nodes-vm	
axios	0
bookshelf	0
express Cart	0
hexo	2
Hakathon-starter	0
Keystone	1
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	1
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	0

detect-runinthiscontext-method-in-nodes-vm: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Ενεργοποιήθηκε μόλις 4 φορές στα 3/22 αποθετήρια που ελέγχθηκαν.

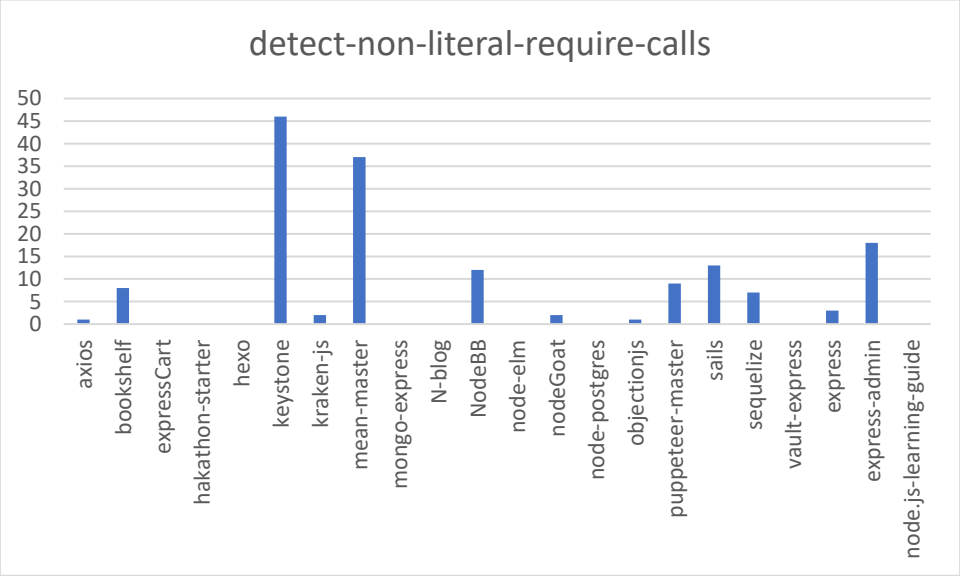


Διάγραμμα 9 Αποτελέσματα κανόνα detect-nosql-injection

Πίνακας 12 Αποτελέσματα κανόνα detect-nosql-injection

detect-nosql-injection	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	1
Node-elm	2
Node Goat	2
Node-postgres	0
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	0

detect-nosql-injection: Ο συγκεκριμένος κανόνας όπως φανερώνεται από τον Πίνακα 5 απαιτεί την ύπαρξη της βιβλιοθήκης MongoDB για την πιθανή ενεργοποίηση του. Ενεργοποιήθηκε στα 3 από τα 5 αποθετήρια που χρησιμοποιούν MongoDB εντοπίζοντας περιπτώσεις επικίνδυνης χρήσης του τελεστή $where.

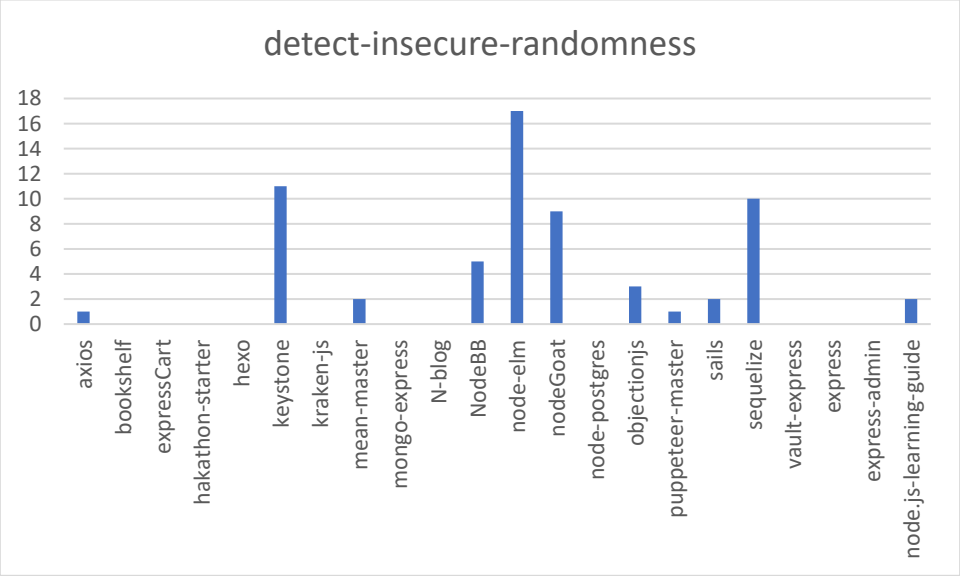


Διάγραμμα 10 Αποτελέσματα κανόνα detect-non-literal-require-calls

Πίνακας 13 Αποτελέσματα κανόνα detect-non-literal-require-calls

detect-non-literal-require-calls	
axios	1
bookshelf	8
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	46
Kraken-js	2
mean	37
Mongo-express	0
NodeBB	12
N-blog	0
Node-elm	0
Node Goat	2
Node-postgres	0
objectionjs	1
pupeeter	9
sails	13
sequelize	7
vault-express	0
express	3
express-admin	18
Node.js-learning-guide	0

detect-non-literal-require-calls: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Όπως απεικονίζεται και στο διάγραμμα ο κανόνας ενεργοποιήθηκε στα περισσότερα αποθετήρια στα οποία έγινε έλεγχος, ενώ εντόπισε 159 περιπτώσεις δυναμικών κλήσεων της μεθόδου require στο σύνολο των αποθετηρίων που ελέγχθηκαν. Αυτό έχει ως αποτέλεσμα να είναι επιρρεπής σε περιπτώσεις **fp** (false-positives).

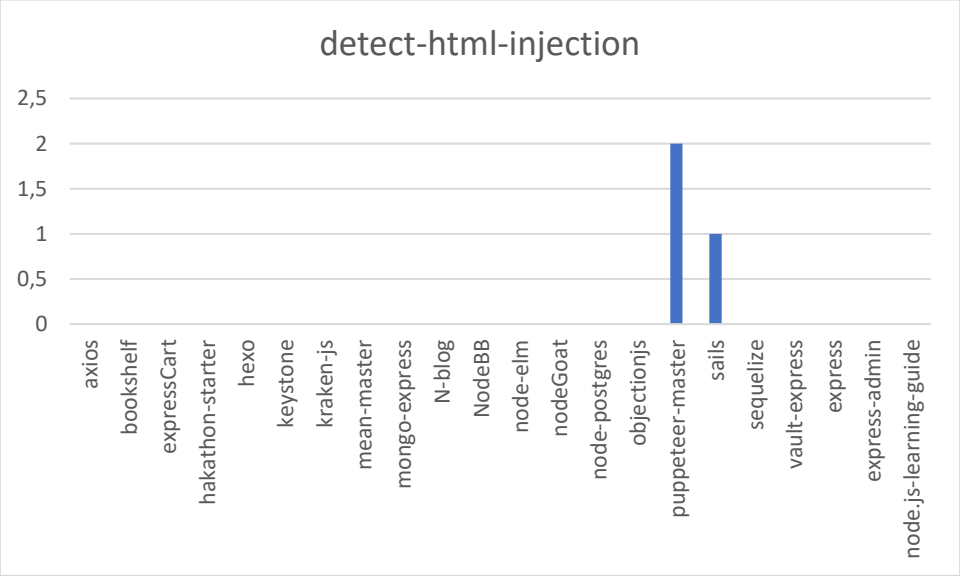


Διάγραμμα 11 Αποτελέσματα κανόνα detect-insecure-randomness

Πίνακας 14 Αποτελέσματα κανόνα detect-insecure-randomness

detect-insecure-randomness	
axios	1
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	11
Kraken-js	0
mean	2
Mongo-express	0
NodeBB	5
N-blog	0
Node-elm	17
Node Goat	9
Node-postgres	0
objectionjs	3
pupeeter	1
sails	2
sequelize	10
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	2

detect-insecure-randomness: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Ενεργοποιήθηκε στα μισά αποθετήρια στα οποία έγινε έλεγχο (11/22) εντοπίζοντας περιπτώσεις επικίνδυνης χρήσης της μεθόδου Math.random.



Διάγραμμα 12 Αποτέλεσμα κανόνα detect-html-injection

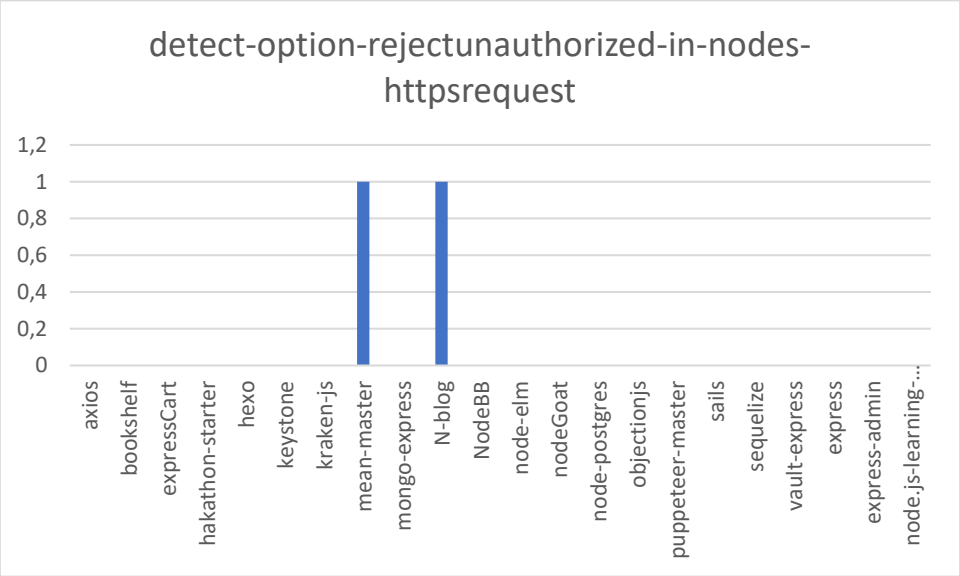
Πίνακας 15 Αποτελέσματα κανόνα detect-html-injection

detect-html-injection	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	0
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	0
pupeeter	2
sails	1
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	0

detect-html-injection: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Εντόπισε συνολικά 3 περιπτώσεις επικίνδυνης χρήσης της μεθόδου document.write όπως φαίνεται και στην παρακάτω εικόνα στο αρχείο DOM world.js του αποθετηρίου pupeeter.

```
await this.evaluate(html => {
  document.open();
  document.write(html);
  document.close();
}, html);
```

Εικόνα 71 Εμφάνιση του κανόνα detect-html-injection στο αποθετήριο pupeeter

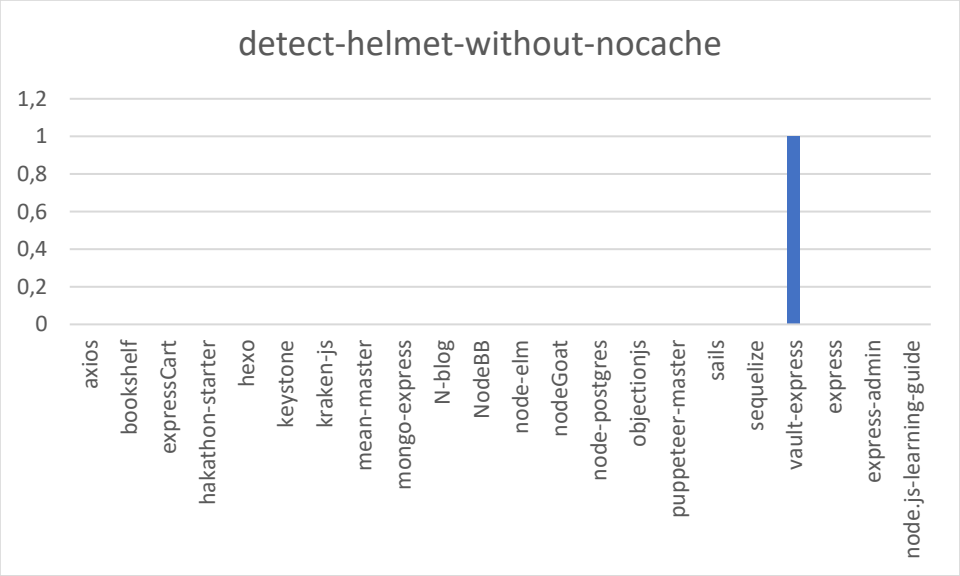


Διάγραμμα 13 Αποτελέσματα κανόνα detect-option-rejectunauthorized-in-nodes-httpsrequest

Πίνακας 16 Αποτελέσματα κανόνα detect-option-rejectunauthorized-in-nodes-httpsrequest

detect-option-rejectunauthorized-in-nodes-httpsrequest	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	1
Mongo-express	0
NodeBB	0
N-blog	1
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	0

detect-option-rejectunauthorized-in-nodes-httpsrequest: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Εντόπισε στο σύνολο των αποθετηρίων μόνο 2 περιπτώσεις (N-blog, mean), στις οποίες η ιδιότητα rejectunauthorized της μεθόδου httpsrequest είναι ψευδής.



Διάγραμμα 14 Αποτελέσματα κανόνα detect-helmet-without-nocache

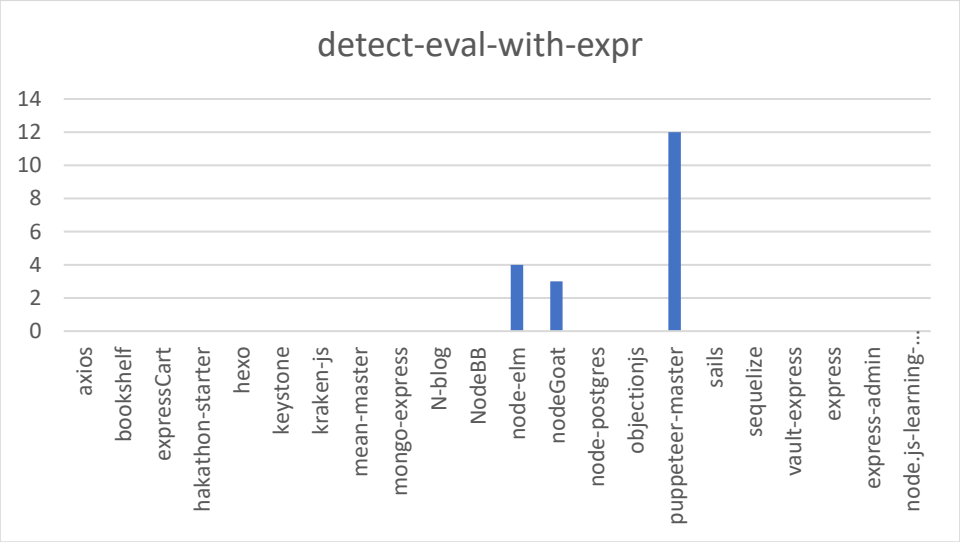
Πίνακας 17 Αποτελέσματα κανόνα detect-helmet-without-nocache

detect-helmet-without-nocache	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	0
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	1
express	0
express-admin	0
Node.js-learning-guide	0

detect-helmet-without-nocache: Ο συγκεκριμένος κανόνας όπως φανερώνεται από τον Πίνακα 5 απαιτεί την ύπαρξη της βιβλιοθήκης express και πιο συγκεκριμένα του πακέτου express-helmet για την πιθανή ενεργοποίηση του. Στο μοναδικό αποθετήριο που γινόταν χρήση του helmet ο κανόνας εντόπισε τη χρήση του helmet με τις εξορισμού κεφαλίδες χωρίς να συμπεριλαμβάνεται η κεφαλίδα noCache, όπως φαίνεται και στην παρακάτω εικόνα από το αρχείο server.js του αποθετηρίου vault-express.

```
app.use(helmet());
app.use(express.static(path.join(__dirname, 'public')));
app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
```

Εικόνα 72 Εμφάνιση του κανόνα detect-helmet-without-nocache στο αποθετήριο vault-express



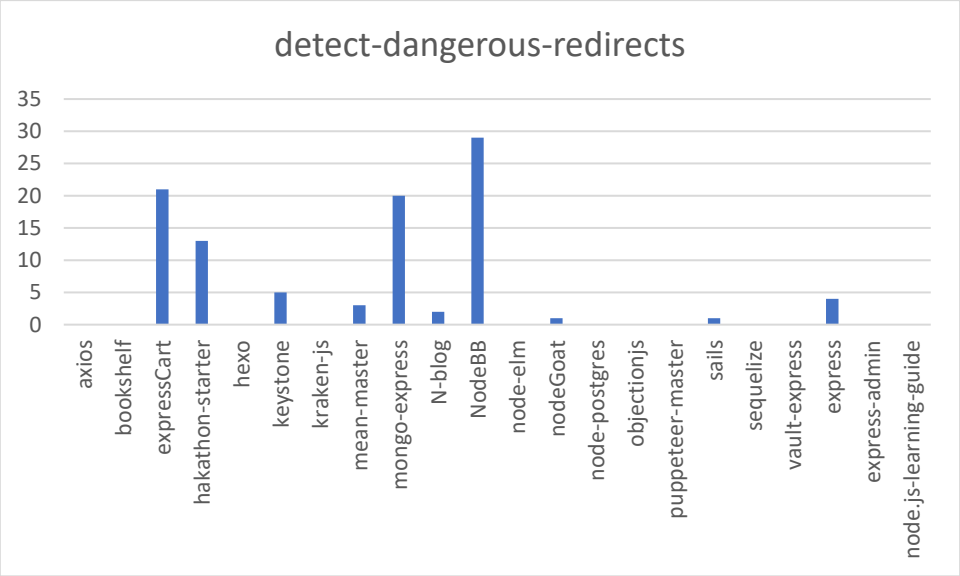
Διάγραμμα 15 Αποτελέσματα κανόνα detect-eval-with-expr

Πίνακας 18 Αποτελέσματα κανόνα detect-eval-with-expr

detect-eval-with-expr	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	0
Node-elm	4
Node Goat	3
Node-postgres	0
objectionjs	0
pupeeter	12
sails	0
sequelize	0
vault-express	0
express	0
express-admin	0

Node.js-learning-guide	0

detect-eval-with-expr: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Εντόπισε συνολικά 17 περιπτώσεις επικίνδυνης χρήσης της eval και ενεργοποιήθηκε στα 3 από τα 22 αποθετήρια που ελέγχθηκαν.

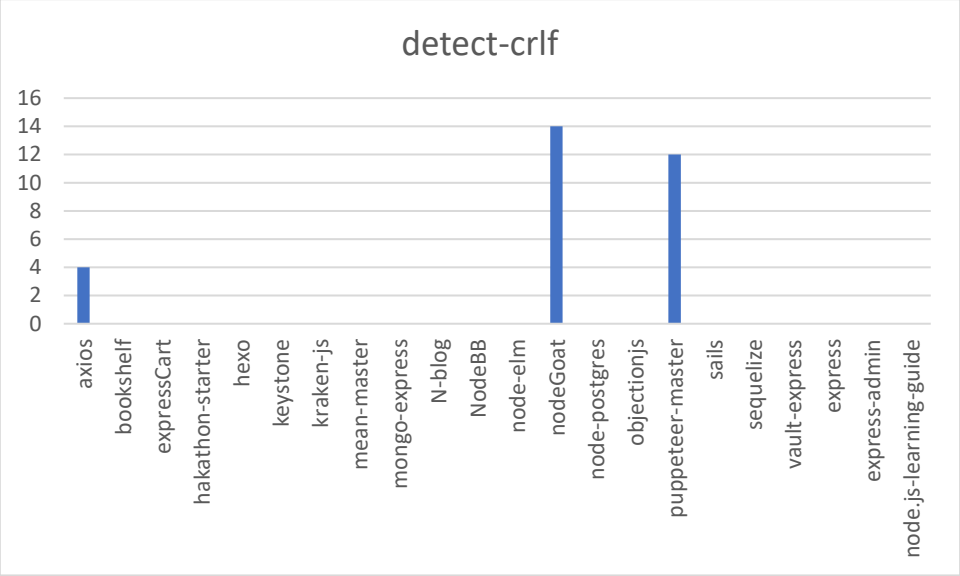


Διάγραμμα 16 Αποτελέσματα κανόνα detect-dangerous-redirects

Πίνακας 19 Αποτελέσματα κανόνα detect-dangerous-redirects

detect-dangerous-redirects	
axios	0
bookshelf	0
express Cart	21
hexo	0
Hakathon-starter	13
Keystone	5
Kraken-js	0
mean	3
Mongo-express	20
NodeBB	29
N-blog	2
Node-elm	0
Node Goat	1
Node-postgres	0
objectionjs	0
pupeeter	0
sails	1
sequelize	0
vault-express	0
express	4
express-admin	0
Node.js-learning-guide	0

detect-dangerous-redirects: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Εντόπισε συνολικά 99 περιπτώσεις χρήσης της μεθόδου redirect χωρίς Literal ορίσματα στο σύνολο των 22 αποθετηρίων, και είναι ιδιαίτερα επιρρεπής στις περιπτώσεις FP.



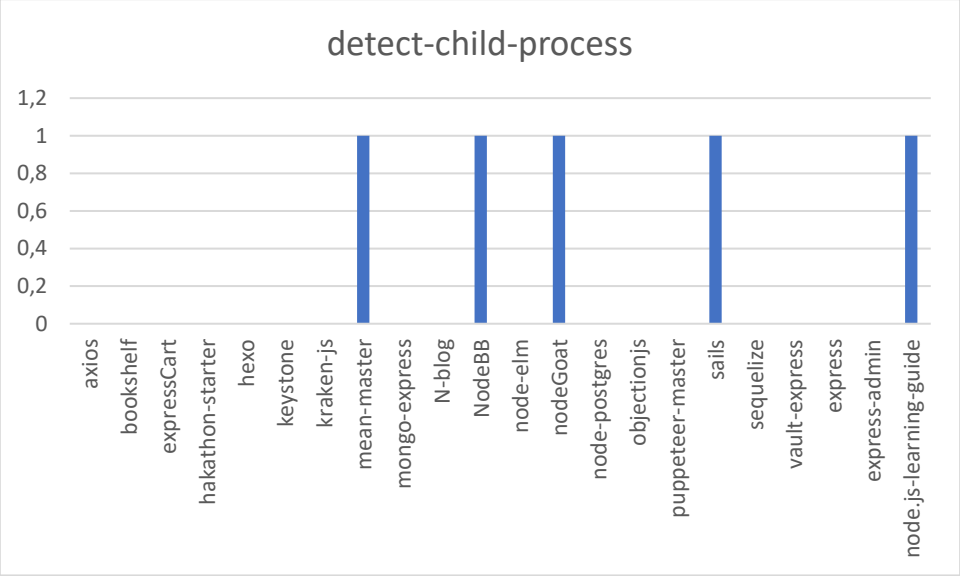
Διάγραμμα 17 Αποτελέσματα κανόνα detect-crlf

Πίνακας 20 Αποτελέσματα κανόνα detect-crlf

detect-crlf	
axios	4
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	0
Node-elm	0
Node Goat	14
Node-postgres	0
objectionjs	0
pupeeter	12
sails	0
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	0

detect-crlf: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του. Έγινε έλεγχος

του κανόνα στα 3 πρώτα αποθετήρια και στη συνέχεια ο παρόν κανόνας απενεργοποιήθηκε θέτοντας τη τιμή του "off" στο αρχείο διαμόρφωσης του ESLint λόγω της πολύ συχνής εμφάνισης του. Ο συγκεκριμένος κανόνας είναι ο πιο επιρρεπής στις περιπτώσεις **fp** (false-positives).



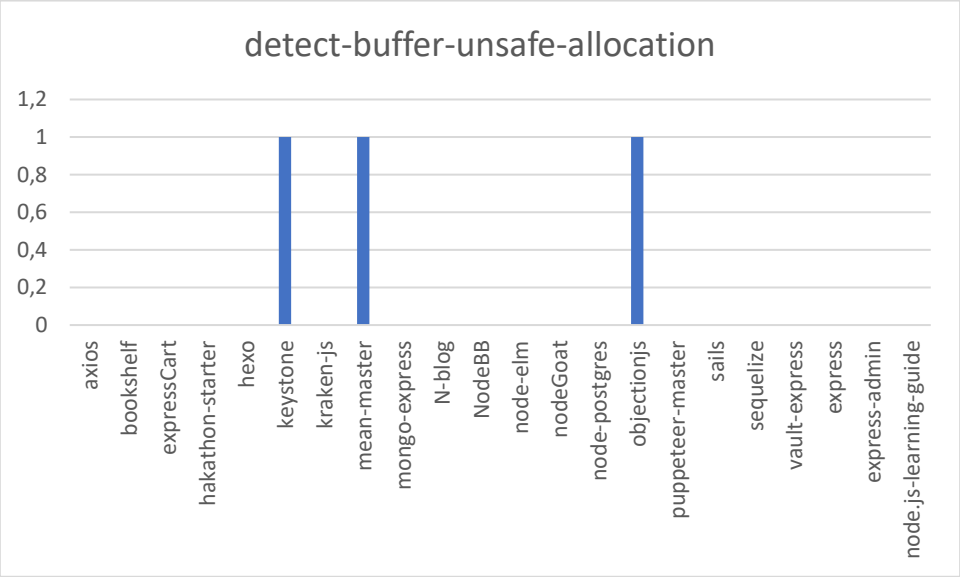
Διάγραμμα 18 Αποτελέσματα κανόνα detect-child-process

Πίνακας 21 Αποτελέσματα κανόνα detect-child-process

detect-child-process	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	1
Mongo-express	0
NodeBB	1
N-blog	0
Node-elm	0
Node Goat	1
Node-postgres	0
objectionjs	0
pupeeter	0
sails	1
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	1

detect-child-process: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την ενεργοποίηση του.

Εντόπισε συνολικά 5 περιπτώσεις χρήσης της μεθόδου exec από το πακέτο child_process του Node.js με μη Literal πρώτο όρισμα. Η δεύτερη περίπτωση του κανόνα (χρήση της execFile μεθόδου με την ιδιότητα shell:true) δεν ενεργοποιήθηκε σε κανένα από τα 22 αποθετήρια.



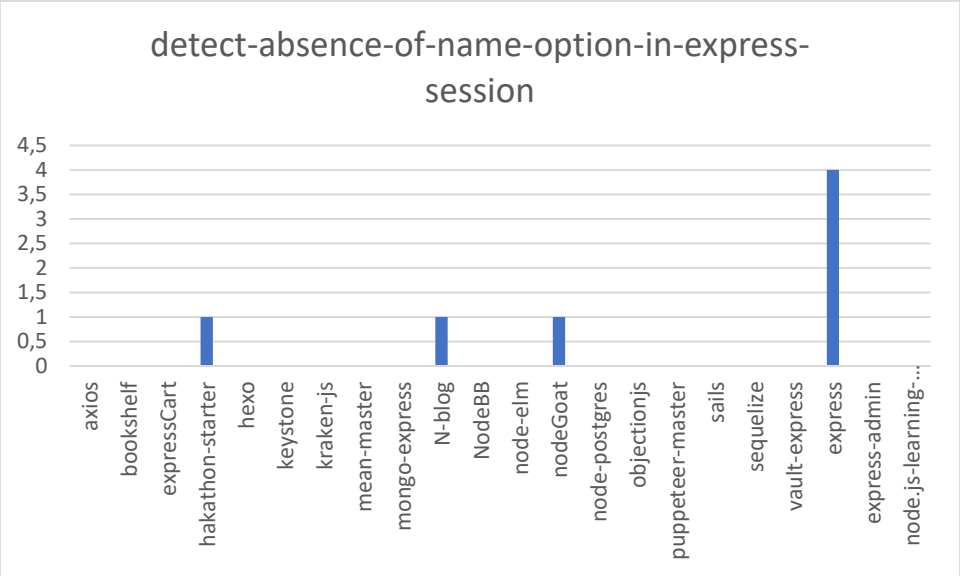
Διάγραμμα 19 Αποτελέσματα κανόνα detect-buffer-unsafe-allocation

Πίνακας 22 Αποτελέσματα κανόνα detect-buffer-unsafe-allocation

detect-buffer-unsafe-allocation	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	1
Kraken-js	0
mean	1
Mongo-express	0
NodeBB	0
N-blog	0
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	1
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	0
express-admin	0
Node.js-learning-guide	0

detect-buffer-unsafe-allocation: Ο συγκεκριμένος κανόνας όπως φαίνεται και από τον Πίνακα 5 δεν είχε κάποιο περιορισμό όσον αφορά κάποια τρίτη βιβλιοθήκη για την

ενεργοποίηση του. Εντόπισε 3 περιπτώσεις χρήσης της μεθόδου allocUnsafe της κλάσης Buffer στο σύνολο των 22 αποθετηρίων που ελέγχθηκαν.



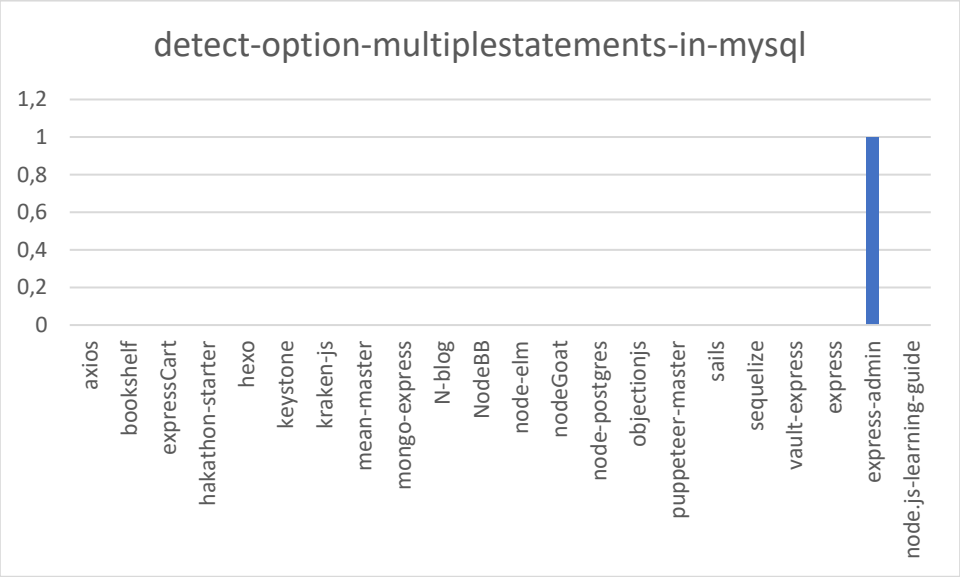
Διάγραμμα 20 Αποτελέσματα κανόνα detect-absence-of-name-option-in-express-session

Πίνακας 23 Αποτελέσματα κανόνα detect-absence-of-name-option-in-express-session

detect-absence-of-name-option-in-express-session	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	1
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	1
Node-elm	0
Node Goat	1
Node-postgres	0
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	4
express-admin	0
Node.js-learning-guide	0

detect-absence-of-name-option-in-express-session: Ο συγκεκριμένος κανόνας όπως φανερώνεται από τον Πίνακα 5 απαιτεί την ύπαρξη της βιβλιοθήκης express και πιο συγκεκριμένα του πακέτου express-session για την πιθανή ενεργοποίηση του. Από τα

αποθετήρια τα οποία κάνουν χρήση της βιβλιοθήκης express (express-admin, express, vault-express, Mongo-express, Hakathon-starter, N-blog, Node.js-learning-guide) εντόπισε συνολικά 7 περιπτώσεις μη χρήσης της ιδιότητας name στο express-session.



Διάγραμμα 21 Αποτελέσματα κανόνα detect-option-multipleStatements-in-sql

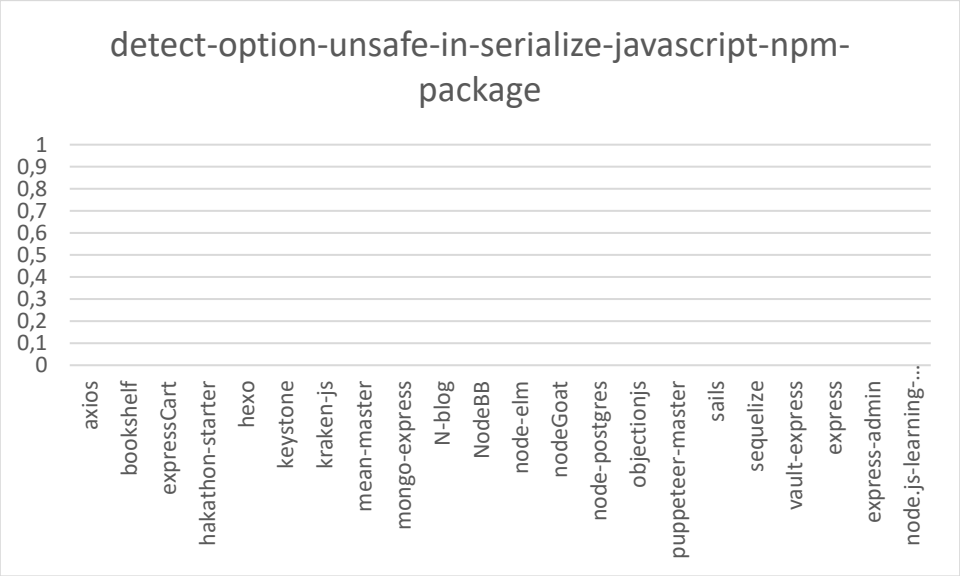
Πίνακας 24 Αποτελέσματα κανόνα detect-option-multipleStatements-in-sql

detect-option-multiplestatements-in-mysql	
axios	0
bookshelf	0
express Cart	0
hexo	0
Hakathon-starter	0
Keystone	0
Kraken-js	0
mean	0
Mongo-express	0
NodeBB	0
N-blog	0
Node-elm	0
Node Goat	0
Node-postgres	0
objectionjs	0
pupeeter	0
sails	0
sequelize	0
vault-express	0
express	0
express-admin	1
Node.js-learning-guide	0

detect-option-multiplestatements-in-mysql: Ο συγκεκριμένος κανόνας όπως φανερώνεται από τον Πίνακα 5 απαιτεί την ύπαρξη του πακέτου mysql για την πιθανή ενεργοποίηση του.

Από τα αποθετήρια που χρησιμοποιούν την mysql (objectionjs, sequelize, express-admin, bookshelf), ο κανόνας εντόπισε 1 περίπτωση χρήσης της ιδιότητας multiplestatements, που σημαίνει πιθανή χρήση queries με multiplestatements τα οποία είναι ιδιαίτερα επικίνδυνα για SQL injection. Η παρακάτω εικόνα απεικονίζει τη συγκεκριμένη περίπτωση στο αρχείο import.js του αποθετηρίου express-admin.

```
var connection = mysql.createConnection({
    user: 'liolio',
    password: 'karamba',
    multipleStatements: true
});
```



Διάγραμμα 22 Αποτελέσματα κανόνα detect-option-unsafe-in-serialize-javascript-npm-package

detect-option-unsafe-in-serialize-javascript-npm-package: Ο συγκεκριμένος κανόνας δεν ενεργοποιήθηκε για κανένα από τα 22 αποθετήρια που ελέγχθηκαν.

Σημείωση: Ο κώδικας με την υλοποίηση των κανόνων, τα αρχεία ελέγχου (test files), τα αρχεία τεκμηρίωσης (documentation files), και τις οδηγίες χρήσης του πακέτου (README) είναι διαθέσιμα μέσω του GitHub στον παρακάτω σύνδεσμο:

https://github.com/gkouziik/eslint-plugin-security-node

Ενώ το πακέτο npm που δημιουργήθηκε μαζί με τις οδηγίες εγκατάστασης του είναι διαθέσιμο στον παρακάτω σύνδεσμο:

https://www.npmjs.com/package/eslint-plugin-security-node

7.4 Περιορισμός false-negative περιπτώσεων

Όπως αναλύθηκε στο Κεφάλαιο 4 το μεγαλύτερο μειονέκτημα των εργαλείων στατικής ανάλυσης είναι η μεγάλη ευαισθησία τους στις περιπτώσεις FP & FN. Για το συγκεκριμένο σύνολο κανόνων σαν fp ορίζονται εκείνες οι περιπτώσεις στις οποίες ένας κανόνας ενεργοποιείται χωρίς η συνάρτηση, η μεταβλητή ή η έκφραση για την οποία ενεργοποιήθηκε να αποτελεί αδυναμία για την επίτευξη κάποιας επίθεσης. Αντίθετα σαν FN ορίζονται εκείνες

οι περιπτώσεις στις οποίες ένας κανόνας έπρεπε να ενεργοποιηθεί για τον έλεγχο κάποιας συνάρτησης ή μεταβλητής ή έκφρασης, η οποία αποτελεί αδυναμία για την επίτευξη κάποιας επίθεσης και δεν ενεργοποιήθηκε.

Οι περιπτώσεις FN αποτελούν πολύ μεγαλύτερο πρόβλημα από αυτές των FP από τη σκοπιά της ασφάλειας. Για τη μείωση των FN περιπτώσεων στο σύνολο των κανόνων, στις περιπτώσεις στις οποίες η ενεργοποίηση του κανόνα εξαρτάται από το όνομα της μεταβλητής που έχει χρησιμοποιήσει για την εισαγωγή κάποιου πακέτου (είτε του Node.js , είτε τρίτο) ο προγραμματιστής, η αρχική προσέγγιση τροποποιήθηκε. Για παράδειγμα στον κανόνα detect-child-process όπου απαιτείται η κλήση του module child_process του Node.js λαμβάνονταν υπόψιν μόνο η περίπτωση που η μεταβλητή στην οποία αποθηκεύονταν το επιστρεφόμενο αντικείμενο από την μέθοδο require('child_process') συνέπιπτε με το όνομα του module, όπως απεικονίζεται στην παρακάτω εικόνα.

```
var child_process = require('child_process');
var user_input;
child_process.exec('ls-la' + user_input, function(err,data){
   console.log(data);
});
```

Εικόνα 73 Αρχική υλοποίηση του κανόνα detect-child-process

Η παραπάνω υλοποίηση είναι ευάλωτη σε περιπτώσεις fn όπως απεικονίζεται στην παρακάτω εικόνα.

```
var childP = require('child_process');
var user_input;
childP.exec('ls-la' + user_input, function(err,data){
   console.log(data);
});
```

Εικόνα 74 Περίπτωση false negative

Ενώ όπως φαίνεται στην έξοδο του εργαλείου AST explorer, ο κανόνας για την προηγούμενη περίπτωση δεν ενεργοποιείται.

```
// Lint rule not fired.

// Fixed output follows:
// ------------------------------------------------------

var childP = require('child_process');
var user_input;
childP.exec('ls-la' + user_input, function(err,data){
   console.log(data);
});
```

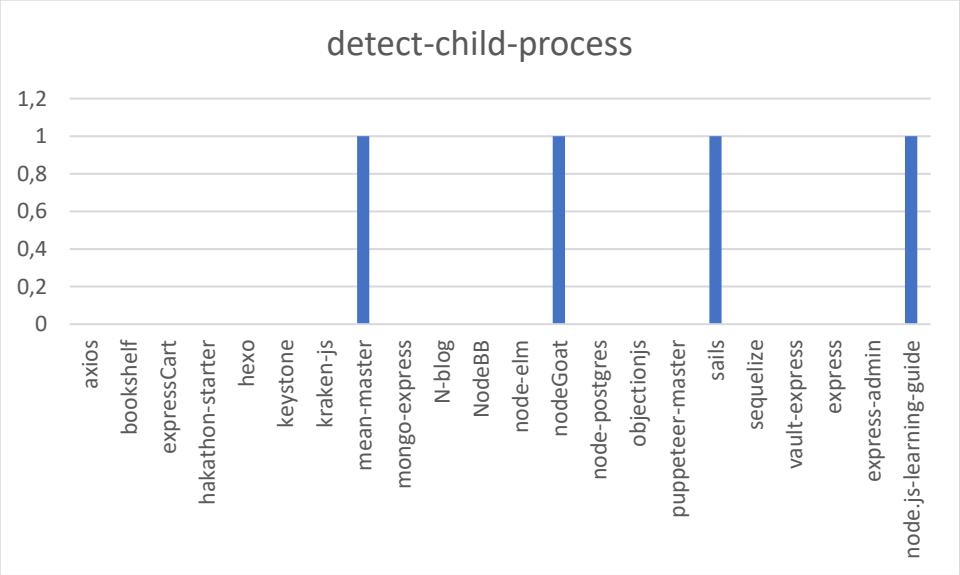
Εικόνα 75 Μη ενεργοποίηση κανόνα σε περίπτωση fn

Η υλοποίηση που πραγματοποιήθηκε και εν τέλει υιοθετήθηκε για την μείωση των FN περιπτώσεων στους κανόνες που εξαρτώνται από μεταβλητές των οποίων το όνομα μεταβάλλεται ανάλογα τον προγραμματιστή (όπως η παραπάνω περίπτωση), απεικονίζεται παρακάτω (συγκεκριμένα για την περίπτωση του κανόνα detect-child-process)

```
'VariableDeclaration': function (node) {
        if (node.declarations[0].init.hasOwnProperty('callee')) {
            if (node.declarations[0].init.callee.name === 'require' &&
          node.declarations[0].init.arguments[0].type === 'Literal' &&
          node.declarations[0].init.arguments[0].value === 'child_process') {
              childProcess = node.declarations[0].id.name
          }
        }
      }
```

Εικόνα 76 Υλοποίηση για τη μείωση των fn στον κανόνα detect-child-process

Στην παραπάνω εικόνα αναπαρίσταται το κομμάτι κώδικα που προστέθηκε στον πηγαίο κώδικα του κανόνα detect-child-process για την εξάλειψη της προηγούμενης περίπτωσης FN. Πιο συγκεκριμένα απεικονίζεται η σάρωση κάθε δήλωσης μεταβλητής και η αποθήκευση εκείνης που αφορά την κλήση στο module child_process ώστε να αποφευχθεί η περίπτωση FN της **Εικόνας 76**. Στο παρακάτω διάγραμμα απεικονίζεται η ενεργοποίηση του κανόνα detect-child-process στα αποθετήρια που ελέγχθηκαν με βάση την αρχική του υλοποίηση.



Διάγραμμα 23 Διάγραμμα κανόνα detect-child-process με την ευάλωτη σε FN υλοποίηση

Στο παραπάνω διάγραμμα και σε σύγκριση με το **Διάγραμμα 18** (το οποίο περιέχει τα αποτελέσματα της τελικής υλοποίησης) φαίνεται η μη ενεργοποίηση του κανόνα στο αποθετήριο NodeBB, όπου αποτελεί μία περίπτωση fn όπως αυτή που απεικονίζεται στην **Εικόνα 76**. Η ίδια ακριβώς διαδικασία ακολουθήθηκε για όλους τους κανόνες που είχαν αντίστοιχες περιπτώσεις fn με τον κανόνα detect-child-process, οι οποίοι απεικονίζονται στον παρακάτω πίνακα.

Detect-security-misconfiguration-cookie
Detect-option-unsafe-in-serialize-javascript-npm-package
Detect-option-reject-unauthorized-in-nodejs-https-request
Detect-multiplestatements-in-mysql
Detect-absence-of-name-option-in-express-session

Σχόλια: Για τους κανόνες του παραπάνω πίνακα δεν παρατηρήθηκε κάποια μεταβολή στα διαγράμματα της αρχικής υλοποίησης σε σχέση με εκείνα της τελικής (τα οποία αναπαρίστανται στην προηγούμενη παράγραφο) για τη μείωση των FN περιπτώσεων, όπως στον κανόνα detect-child-process.

ΚΕΦΑΛΑΙΟ 7

Συμπεράσματα και Μελλοντικές Επεκτάσεις

Στο κεφάλαιο αυτό παρουσιάζονται τα συμπεράσματα που προέκυψαν από την έρευνα για τη συλλογή και την εφαρμογή των κανόνων καθώς και τα αποτελέσματα ύστερα από τον έλεγχο του πακέτου στα διάφορα αποθετήρια της JavaScript. Τέλος προτείνονται κάποιες πιθανές μελλοντικές επεκτάσεις για τη βελτίωση και την εξέλιξη του παρόντος συνόλου κανόνων.

7.1 Συμπεράσματα

Η ασφάλεια αποτελεί αναπόσπαστο κομμάτι των διαδικτυακών εφαρμογών τόσο στα αρχικά στάδια ανάπτυξης τους όσο και στα τελικά. Τα εργαλεία στατικής ανάλυσης προσφέρουν τη δυνατότητα για καλύτερη εποπτεία της εφαρμογής όσον αφορά την ασφάλεια από τα πρώτα στάδια κατασκευής της. Για το σκοπό αυτό συλλέχθηκε και υλοποιήθηκε ένα σύνολο κανόνων με σκοπό να προβλέψει και να αποτρέψει ορισμένες επιθέσεις, το οποίο είναι διαθέσιμο σαν plugin μέσω του πιο διάσημου Linter της JavaScript, τον ESLint. Για το πρώτο στάδιο της υλοποίησης, το οποίο ήταν η συλλογή του συνόλου των κανόνων λήφθηκαν υπόψιν τα παρακάτω κριτήρια.

- Δεν εντάχθηκαν κανόνες οι οποίοι απαιτούν την εισαγωγή κάποιας τρίτης βιβλιοθήκης μέσω του npm για την υλοποίησης τους . Ο λόγος είναι ότι το κυριότερο πρόβλημα που αντιμετωπίζουν οι εφαρμογές που είναι κτισμένες σε Node.js είναι το δέντρο εξαρτήσεων της εφαρμογής από βιβλιοθήκες τρίτων. Η μεγάλη πολυπλοκότητα των εξαρτήσεων διογκώνει τον κίνδυνο και κάνει την εφαρμογή περισσότερο ευάλωτη.
- Τα εργαλεία στατικής ανάλυσης εξορισμού είναι ευάλωτα στις περιπτώσεις FP όπως αναφέρθηκε. Για το λόγο αυτό δεν εντάχθηκαν κανόνες οι οποίοι είναι ιδιαίτερα επιρρεπείς στις περιπτώσεις αυτές.
- Το σύνολο των κανόνων χωρίζεται σε δύο υποσύνολα:
    1. Στους κανόνες που σκοπός τους είναι η εύρεση και ο εντοπισμός περιπτώσεων που αποτελούν άμεσο κίνδυνο για την εφαρμογή .
    2. Στους κανόνες που αφορούν τη διαμόρφωση ορισμένων μεθόδων (οι οποίες παρέχονται από τρίτες βιβλιοθήκες), με σκοπό την καλύτερη οχύρωση και προστασία της εφαρμογής.

Για το δεύτερο στάδιο το οποίο αφορά την υλοποίηση των κανόνων, χρησιμοποιήθηκε η πλατφόρμα AST explorer για την μετατροπή του πηγαίου κώδικα σε συντακτικό δέντρο αλλά και στη συνέχεια για τη διάσχιση αυτού, με σκοπό την υλοποίηση του κανόνα. Για τον έλεγχο της σωστής λειτουργίας των κανόνων αναπτύχθηκαν τα αρχεία ελέγχου που παρέχονται από τον ESLint με έγκυρες και μη έγκυρες περιπτώσεις εφαρμογής του εκάστοτε κανόνα.

Για το τρίτο στάδιο, το σύνολο των κανόνων ομαδοποιήθηκε και μετατράπηκε σε ένα npm πακέτο με σκοπό τον έλεγχο του σε διάφορα αποθετήρια της JavaScript και την εξαγωγή συμπερασμάτων.

Για το τέταρτο στάδιο, το παρόν πακέτο ελέγχθηκε σε διάφορα αποθετήρια της JavaScript, όπου διαπιστώθηκε ότι το σύνολο των κανόνων εντόπισε αρκετές αδυναμίες (ανάλογα το σκοπό του εκάστοτε κανόνα) στα διάφορα αποθετήρια, ενώ ανταποκρίθηκε πολύ καλά στις

περιπτώσεις FP με εξαίρεση ορισμένους κανόνες (detect-non-literal-require-calls). Αντίθετα εντοπίστηκαν ορισμένες περιπτώσεις FN, οι οποίες αποτελούν το μεγαλύτερο κίνδυνο και για το λόγο αυτό τροποποιήθηκε η υλοποίηση ορισμένων κανόνων με σκοπό τη μείωση τους. Τέλος το σύνολο των κανόνων και η υλοποίηση τους, καθώς και τα διάφορα αρχεία τεκμηρίωσης (documentation files) είναι διαθέσιμα στο GitHub, ενώ το πακέτο npm είναι διαθέσιμο για χρήση μέσω της πλατφόρμας του npm.

7.2 Μελλοντικές Επεκτάσεις

Όπως αναφέρθηκε παραπάνω η υλοποίηση του συνόλου των κανόνων είναι open-source, και μπορεί να μελετηθεί και να αξιοποιηθεί περαιτέρω από την κοινότητα. Μελλοντικά είναι δυνατό να προστεθούν περισσότεροι κανόνες οι οποίοι καλύπτουν μεγαλύτερο εύρος επιθέσεων και περιπτώσεων, ενώ ταυτόχρονα μπορεί να βελτιωθεί ακόμα περισσότερο το παρόν σύνολο κανόνων όσον αφορά τις **FP** & **FN** περιπτώσεις.

Επιπλέον μία πιθανή βελτίωση θα ήταν οι μελλοντικοί κανόνες που θα προστεθούν να μην επικεντρώνονται μόνο στην παρουσία μεθόδων και συναρτήσεων και το περιεχόμενο των ορισμάτων τους, αλλά στην σωστή διαμόρφωση ορισμένων μεθόδων, εντοπίζοντας την έλλειψη κρίσιμων επιλογών και ιδιοτήτων που αφορούν την ασφάλεια (όπως οι περιπτώσεις των κανόνων detect-helmet-without-nocache, detect-absence-of-name-option-in-express-session και detect-security-misconfiguration-cookie). Τέλος μία σημαντική βελτίωση θα αποτελούσε η υλοποίηση κανόνων που επικεντρώνονται στην ύπαρξη αδυναμιών όσον αφορά τις μεθόδους τρίτων πακέτων (npm), έτσι ώστε ο προγραμματιστής να έχει πλήρη επίγνωση των αδυναμιών των εξωτερικών πακέτων που χρησιμοποιεί.

ΒΙΒΛΙΟΓΡΑΦΙΑ

[1] Berners-Lee, «info.sern.cz,» August 6 1991. [Ηλεκτρονικό]. Available: info.sern.cz.

[2] «Tuttorials teacher,» [Ηλεκτρονικό]. Available: https://www.tutorialsteacher.com/nodejs/what-is-nodejs.

[3] «codeburst.io,» [Ηλεκτρονικό]. Available: https://codeburst.io/how-node-js-single-thread-mechanism-work-understanding-event-loop-in-nodejs-230f7440b0ea.

[4] «the crazy programmer,» [Ηλεκτρονικό]. Available: https://www.thecrazyprogrammer.com/2016/03/top-7-reasons-node-js-popular.html.

[5] H. H. AlBreiki, «Evaluation of Static Analysis Tools for Software Security».

[6] J. W. Brian Chess, Secure programming with static analysis.

[7] «codeGuru,» [Ηλεκτρονικό]. Available: https://www.codeguru.com/tools/the-top-five-javascript-linting-tools.html.

[8] «ESLint,» [Ηλεκτρονικό]. Available: https://eslint.org/docs/about/.

[9] «ESLint,» [Ηλεκτρονικό]. Available: https://eslint.org/docs/developer-guide/architecture.

[10] «ESLint,» [Ηλεκτρονικό]. Available: https://eslint.org/docs/user-guide/configuring.

[11] «ESLint,» [Ηλεκτρονικό]. Available: https://eslint.org/docs/developer-guide/working-with-rules.

[12] J. JONES, «Abstract syntax tre implementation Idioms,» University of Alabama.

[13] T. Kuhn, «Abstract Syntax Tree and Java Code Manipulation in the Eclipse».

[14] B. Sullivan, «Server side Javascript Injection».

[15] J. Erickson, The art of exploitation.

[16] «OWASP cheatsheet,» OWASP, [Ηλεκτρονικό]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html.

[17] «Wikipedia,phising,» [Ηλεκτρονικό]. Available: https://el.wikipedia.org/wiki/Phishing.

[18] «enterprise verizon,» 2016. [Ηλεκτρονικό]. Available: https://enterprise.verizon.com/resources/reports/DBIR_2016_Report.pdf.

[19] K. Duuna, Secure your Node.js web application.

[20] CYBERSECURITY, protecting critical infrastructures from Cyber attack and cyber warfare, Thomas A. Johnson.

[21] M. C. a. C. Bird, «What developpers want and need from program analysis: an empirical sudy,» 2016.

[22] K. F. Tomasdottir, «Why and how JavaScript Developers use Linters».

[23] OWASP. [Ηλεκτρονικό]. Available: https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities.

[24] [Ηλεκτρονικό]. Available: https://github.com/RetireJS/retire.js/blob/master/docs/blogpost.md.

[25] «Node.js Documentaion,» [Ηλεκτρονικό]. Available: https://nodejs.org/api/deprecations.html.

[26] «snyk,» [Ηλεκτρονικό]. Available: https://snyk.io/.

[27] «OWASP,» [Ηλεκτρονικό]. Available: https://www.owasp.org/index.php/Category:Attack.

[28] «CVE details,» [Ηλεκτρονικό]. Available: https://www.cvedetails.com/product/30764/Nodejs-Node.js.html?vendor_id=12113.

[29] S. A. C. &. D. S. Wallach, «Opportunities and Limits of Remote Timing Attacks,» 2009.

[30] O'reilly, Securing Node applications, Oreilly Media.