Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής

Στατική Ανάλυση Προγραμμάτων για τον Εντοπισμό Προβλημάτων Ασφαλείας

Επιβλέπων Καθηγητής: Κατσαρός Παναγιώτης

Επιμέλεια: Χατζηδημητρίου Ανδρέας

Θεσσαλονίκη, Σεπτέμβριος 2009

Στατική Ανάλυση Προγραμμάτων για τον Εντοπισμό Προβλημάτων Ασφαλείας

Επιβλέπων Καθηγητής: Κατσαρός Παναγιώτης

Επιμέλεια: Χατζηδημητρίου Ανδρέας

Αριθμός Ειδικού Μητρώου: 1374

Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Τμήμα Πληροφορικής

Θεσσαλονίκη 2009

Περιεχόμενα

1 Εισαγωγή 1

2 Στατική Ανάλυση 3

2.1 Κώδικας προς ανάλυση 4

2.2 Χτίσιμο μοντέλου 5

2.2.1 Λεξική Ανάλυση 5

2.2.2 Συντακτική Ανάλυση 5

2.2.3 Σημασιολογική Ανάλυση 6

2.2.4 Γράφος Ροής Ελέγχου 7

2.2.5 Γράφος Κλήσεων 8

2.3 Αλγόριθμοι ανάλυσης 9

2.4 Γνώση θεμάτων ασφαλείας 10

2.5 Αποτελέσματα ανάλυσης 11

3 Ανάλυση Ροής Δεδομένων 13

3.1 Βασικές έννοιες ανάλυσης ροής δεδομένων 13

3.2 Σχήμα Ανάλυσης Ροής Δεδομένων 15

3.3 Θεμελιώσεις ανάλυσης ροής δεδομένων 20

3.4 Αλγόριθμος ανάλυσης ροής δεδομένων 22

4 Κενά Ασφαλείας και Taint Analysis 24

4.1 Εισαγωγή κακόβουλων δεδομένων 25

4.1.1 Πλαστογραφία παραμέτρων (parameter tampering) 25

4.1.2 Πλαστογραφία URL (URL tampering) 25

4.1.3 Διαχείριση κρυφών πεδίων (hidden field manipulation) 26

4.1.4 Διαχείριση κεφαλίδας HTTP (HTTP header manipulation) 26

4.1.5 Δηλητηρίαση cookie (cookie poisoning) 26

4.2 Διαχείριση εφαρμογής 27

4.2.1 Εμβολιασμός SQL εντολών (SQL injection) 27
4.2.2 Επίθεση XSS (cross-site scripting).. 28
4.2.3 Διαμοιρασμός απάντησης HTTP (HTTP response splitting).................. 29
4.2.4 Διάσχιση μονοπατιού (path traversal) 30
4.2.5 Εμβολιασμός εντολής (command injection) 30
5 FindBugs.. 32
5.1 Αρχιτεκτονική 32
5.2 Ανιχνευτές.. 34
5.2.1 Visitor based ανιχνευτές.. 34
5.2.2 CFG based ανιχνευτές 35
5.3 Ανάλυση Ροής Δεδομένων στο FindBugs 36
6 Taint analysis σε multi-applet JavaCard εφαρμογή 38
6.1 Ανιχνευτής taint analysis 38
6.1.1 Αναλυτής 39
6.1.2 Βάσεις Δεδομένων... 41
6.1.3 Ανιχνευτές 42
6.2 Παρουσίαση JavaCard εφαρμογής 46
7 Επίλογος 50
Παράρτημα: Επισημειώσεις 51
Αναφορές... 53

1 Εισαγωγή

Καθώς οι σύγχρονες εφαρμογές λογισμικού προσφέρουν όλο και περισσότερες λειτουργίες, η δομή τους τείνει να γίνεται όλο και πιο περίπλοκη. Η εύρεση προγραμματιστικών λαθών σε τέτοια συστήματα καθίσταται δυσκολότερη εξαιτίας της αυξημένης πολυπλοκότητας. Παρόλο που η παραδοσιακή τεχνική της επισκόπησης του κώδικα από ειδικούς εξακολουθεί να παίζει σημαντικό ρόλο στην διασφάλιση της ποιότητας, οι αυτοματοποιημένες τεχνικές αποσφαλμάτωσης θεωρούνται απαραίτητο συμπλήρωμα για την διευκόλυνση και τη βελτίωση της διαδικασίας.

Υπάρχουν αρκετές ευρέως διαδεδομένες τεχνικές που χρησιμοποιούνται στην πράξη και οι οποίες μπορούν να κατηγοριοποιηθούν σε δύο ομάδες: τις *δυναμικές* και τις *στατικές* προσεγγίσεις. Στην δυναμική προσέγγιση η επιβεβαίωση του εάν ένα συγκεκριμένο τμήμα κώδικα δουλεύει ορθά πραγματοποιείται συγκρίνοντας την συμπεριφορά της εφαρμογής σε περιπτώσεις ελέγχου (test cases), με τα αναμενόμενα αποτελέσματα. Πιθανά λάθη εντοπίζονται όταν κατατίθεται κάποια εξαίρεση σε χρόνο εκτέλεσης ή όταν η έξοδος που παράγει το πρόγραμμα δεν συμφωνεί με την αναμενόμενη έξοδο. Επομένως, η αποτελεσματικότητα του ελέγχου εξαρτάται σε μεγάλο βαθμό από την ποιότητα των περιπτώσεων ελέγχου. Χαρακτηριστικό είναι ότι στο πρότυπο (1), δεν συνιστάται η επικύρωση της ορθής λειτουργίας μιας εφαρμογής με βάση αποκλειστικά τον έλεγχο τμημάτων (unit testing), της πιο διαδεδομένης τεχνικής δυναμικού ελέγχου, καθώς η εξέταση όλων των πιθανών συνδυασμών εισόδου σε μη-τετριμμένες εφαρμογές είναι ανέφικτη. Αντίθετα, η στατική ανάλυση υπόσχεται τον αυτοματοποιημένο εντοπισμό σφαλμάτων με την εφαρμογή τυπικών μεθόδων και πολύπλοκων τεχνικών ανάλυσης απευθείας στον κώδικα.

Στην εργασία αυτή παρουσιάζουμε το αντικείμενο της στατικής ανάλυσης τόσο από θεωρητική όσο και από πρακτική σκοπιά. Σε ότι αφορά το θεωρητικό μέρος, εξετάζουμε αρχικά την καταλληλότητα της στατικής ανάλυσης για τον εντοπισμό κενών ασφαλείας και στη συνέχεια ακολουθούμε την λειτουργία των εργαλείων στατικής ανάλυσης εξετάζοντας τα επιμέρους βήματα (χτίσιμο μοντέλου, ανάλυση, παρουσίαση αποτελεσμάτων).

Η στατική ανάλυση έχει πλήθος εφαρμογών με σημαντικότερες την βελτιστοποίηση του κώδικα και τον εντοπισμό σφαλμάτων στον κώδικα. Παρόλο που

θα επικεντρωθούμε στην δεύτερη, η εξέταση βασικών τεχνικών, όπως η ανάλυση ροής δεδομένων, που χρησιμοποιούνται κατά την βελτιστοποίηση είναι αναπόφευκτη καθώς αποτελούν τη βάση για την ανάπτυξη πιο σύνθετων μεθόδων ανάλυσης.

Για την μελέτη των πρακτικών θεμάτων που άπτονται του αντικειμένου, θα επικεντρωθούμε στα προβλήματα ασφαλείας που μπορούν να μοντελοποιηθούν ως προβλήματα taint analysis. Αφού εξετάσουμε συγκεκριμένα παραδείγματα τέτοιων προβλημάτων, θα κάνουμε μια σύντομη παρουσίαση του εργαλείου FindBugs το οποίο παρέχει τη δυνατότητα εντοπισμού προβλημάτων ασφαλείας με χρήση στατικής ανάλυσης και θα παρουσιάσουμε έναν ανιχνευτή που μπορεί να ενσωματωθεί ως πρόσθετο στο FindBugs για να επιτρέψει την εκτέλεση taint analysis. Τέλος, εφαρμόζουμε τον ανιχνευτή αυτό σε μια πραγματική multi-applet JavaCard εφαρμογή και παρουσιάζουμε τα αποτελέσματα που προκύπτουν από την ανάλυση.

2 Στατική Ανάλυση

Γενικά, με τον όρο στατική ανάλυση αναφερόμαστε σε μια ευρεία οικογένεια τεχνικών για την ανάλυση του κώδικα, χωρίς την εκτέλεσή του. Αρχικά, οι τεχνικές στατικής ανάλυσης που αναπτύσσονταν, προορίζονταν για χρήση από τους μεταγλωττιστές ώστε να βελτιστοποιηθεί ο παραγόμενος κώδικας. Στην πορεία έγινε αντιληπτό ότι οι τεχνικές αυτές ήταν δυνατό να επεκταθούν ώστε να επιτρέπουν την ανίχνευση σφαλμάτων στον κώδικα. Έτσι, έχουν αναπτυχθεί διάφορες προσεγγίσεις που στηρίζονται στην στατική ανάλυση για τον εντοπισμό προβλημάτων ασφαλείας όπως: υπερχείλιση ενδιάμεσης μνήμης (buffer overflow) (2), αναφορά σε κενό δείκτη (null pointer dereference) (3) (4) και «εμβολιασμός» εντολής SQL (SQL injection) (5).

Σε γενικές γραμμές, η στατική ανάλυση υπερτερεί των άλλων τεχνικών στον εντοπισμό προβλημάτων ασφάλειας για τους παρακάτω λόγους:

- Τα εργαλεία στατικής ανάλυσης εφαρμόζουν εκτεταμένο και συνεπή έλεγχο. Ο κώδικας ελέγχεται σε όλη την έκτασή του, σε αντίθεση με τον δυναμικό έλεγχο όπου είναι πιθανόν να μην ελεγχθούν όλες οι ακραίες καταστάσεις. Ο έλεγχος είναι συνεπής, απουσιάζει δηλαδή η μεροληψία που θα είχε ένας προγραμματιστής ως προς το ποια τμήματα κώδικα ενδέχεται να αξίζουν έλεγχο για θέματα ασφάλειας.
- Με τον έλεγχο του κώδικα καθ' αυτού τα εργαλεία στατικής ανάλυσης εντοπίζουν το πρόβλημα στη ρίζα του, υποδεικνύοντας πιθανά *ελαττώματα* (bugs). Αντίθετα με τον δυναμικό έλεγχο εντοπίζονται *σφάλματα* (errors) τα οποία ίσως είναι δύσκολο να αντιστοιχηθούν στο ελάττωμα που είναι υπεύθυνο.
- Η στατική ανάλυση μπορεί να αποκαλύψει προβλήματα στα αρχικά στάδια της διαδικασίας ανάπτυξης, ακόμη και πριν το πρόγραμμα εκτελεστεί για πρώτη φορά. Ο γρήγορος εντοπισμός προβλημάτων συνεπάγεται και λιγότερο κόστος για την επιδιόρθωσή τους. Ταυτόχρονα η έγκαιρη γνώση από την πλευρά του προγραμματιστή μπορεί να οδηγήσει σε μελλοντική αποφυγή τους.
- Η αυτοματοποίηση της διαδικασίας στατικής ανάλυσης επιτρέπει στην ομάδα ανάπτυξης να εκτελεί εύκολα ελέγχους σε μεγάλα τμήματα κώδικα. Ακόμη και αν ενσωματωθούν πρόσφατα ανακαλυφθέντα προβλήματα ασφάλειας στο εργαλείο στατικής ανάλυσης, μπορεί σχετικά εύκολα να γίνει επανέλεγχος του κώδικα.

2.1 Κώδικας προς ανάλυση

Τα περισσότερα εργαλεία στατικής ανάλυσης εξετάζουν το πρόγραμμα αναλύοντας τον πηγαίο κώδικα, ενώ μερικά επιλέγουν την ανάλυση του εκτελέσιμου (ή ενδιάμεσου) κώδικα. Η ανάλυση εκτελέσιμου κώδικα υπερτερεί της ανάλυσης πηγαίου κώδικα στα εξής:

- Το εργαλείο στατικής ανάλυσης δεν χρειάζεται να μαντέψει το πώς ο μεταγλωττιστής θα «ερμηνεύσει» τον κώδικα αφού ο μεταγλωττιστής έχει ήδη ολοκληρώσει τη δουλειά του. Έτσι αποφεύγεται η περίπτωση ασάφειας.
- Μερικές φορές ο πηγαίος κώδικας δεν είναι διαθέσιμος. Μια εφαρμογή είναι δυνατόν να χρησιμοποιεί διάφορες βιβλιοθήκες ή να αλληλεπιδρά με άλλα προγράμματα των οποίων ο πηγαίος κώδικας δεν είναι διαθέσιμος. Σε αυτές της περιπτώσεις η μόνη λύση είναι να γίνει ανάλυση του εκτελέσιμου ή ενδιάμεσου κώδικα.

Αντίθετα η ανάλυση του πηγαίου κώδικα παρουσιάζει τα ακόλουθα πλεονεκτήματα έναντι της ανάλυσης εκτελέσιμου κώδικα:

- Ο κώδικας που παράγει ο μεταγλωττιστής είναι πιθανόν να κάνει πιο δύσκολη τη διαδικασία ανάλυσης. Ειδικά στην περίπτωση του εκτελέσιμου κώδικα, πρέπει να υπάρχει συμβατότητα μεταξύ των εντολών μηχανής που παράγονται και του επεξεργαστή για τον οποίο προορίζεται το πρόγραμμα. Επιπρόσθετα, η βελτιστοποίηση που εκτελεί ο μεταγλωττιστής περιπλέκει ακόμη περισσότερο τα πράγματα. Στην περίπτωση του ενδιάμεσου κώδικα, όπως για παράδειγμα στην Java, τα προβλήματα αυτά είναι λιγότερο έντονα. Και σε αυτή την περίπτωση όμως, οι μετασχηματισμοί που πραγματοποιούνται στον πηγαίο κώδικα είναι δυνατόν να αποκρύψουν τις προθέσεις του προγραμματιστή για το τι ακριβώς πρέπει να κάνει ένα τμήμα κώδικα.
- Η ανάλυση εκτελέσιμου κώδικα καθιστά πιο δύσκολη και την αναφορά των ευρημάτων της ανάλυσης. Συνήθως τα ευρήματα της ανάλυσης παρουσιάζονται στον χρήστη/προγραμματιστή ως σημεία στον πηγαίο κώδικα, όπως ακριβώς παρουσιάζονται και τα ευρήματα της διαδικασίας αποσφαλμάτωσης (debugging). Για να επιτευχθεί αυτό, δεδομένου ότι το εργαλείο αναλύει τον πηγαίο κώδικα, θα

πρέπει να γίνει κάποιο είδους αντιστοίχιση από το σημείο στον εκτελέσιμο κώδικα όπου εντοπίστηκε το πρόβλημα, στο αντίστοιχο σημείο του πηγαίου κώδικα. Η διαδικασία αυτή μπορεί να αποφευχθεί με την εξαρχής ανάλυση του πηγαίου κώδικα.

Για διερμηνευόμενες (interpreted) γλώσσες, όπως η Java, δεν υπάρχει σαφής προτίμηση μεταξύ των δύο εναλλακτικών προσεγγίσεων. Ο πηγαίος κώδικας περιέχει σαφώς περισσότερη πληροφορία, αλλά ο ενδιάμεσος κώδικας είναι πάντοτε διαθέσιμος. Αντίθετα, στην περίπτωση γλωσσών όπως η C και C++ τα πλεονεκτήματα που προκύπτουν από την ανάλυση του πηγαίου κώδικα υπερτερούν των μειονεκτημάτων, καθώς η ανάλυση πηγαίου κώδικα είναι ευκολότερη και πιο αποδοτική.

2.2 Χτίσιμο μοντέλου

Το πρώτο πράγμα που χρειάζεται να κάνει ένα εργαλείο στατικής ανάλυσης είναι να μετατρέψει τον προς ανάλυση κώδικα σε ένα *μοντέλο προγράμματος*, ένα σύνολο δομών δεδομένων που αντιπροσωπεύουν τον κώδικα. Για το σκοπό αυτό, τα περισσότερα εργαλεία δανείζονται πολλές τεχνικές από τον κόσμο των μεταγλωττιστών (6). Στη συνέχεια εξετάζουμε συνοπτικά τις βασικές τεχνικές που χρησιμοποιούνται.

2.2.1 Λεξική ανάλυση

Σε πρώτη φάση εκτελείται λεξική ανάλυση στον κώδικα. Δηλαδή, ο πηγαίος κώδικας μετατρέπεται από μια ροή χαρακτήρων σε μια ροή *αναγνωριστικών* (tokens) σύμφωνα με τους λεξικούς κανόνες της γλώσσας που πρόκειται να αναλυθεί. Με την εφαρμογή λεξικής ανάλυσης μπορεί να γίνει διαχωρισμός μεταξύ σχολίων, αλφαριθμητικών, δηλώσεων και κλήσεων σε μεθόδους. Η λειτουργία αυτή είναι επαρκής για τα απλούστερα των εργαλείων όπως τα ITS4[1] , RATS[2] και Flawfinder[3].

2.2.2 Συντακτική ανάλυση

Η συντακτική ανάλυση είναι σαφώς πιο δύσκολη από την λεξική ανάλυση και στόχο έχει τον μετασχηματισμό της ροής αναγνωριστικών σε *συντακτικό δένδρο* (syntax

---

[1] http://www.citigal.com/its4
[2] http://www.fortify.com/security-resources/rats.jsp
[3] http://www.dwheeler.com/flawfinder

tree) με βάση μια *γραμματική χωρίς συμφραζόμενα* (context-free grammar – CFG) που περιγράφει τη γλώσσα του προς ανάλυση κώδικα. Γραμματική χωρίς συμφραζόμενα καλείται ένα σύνολο κανόνων παραγωγής που περιγράφουν τη δομή μιας γλώσσας και ένα μέρος της στατικής σημασίας της γλώσσας (π.χ. προτεραιότητα και προσεταιριστικότητα τελεστών). Σε μερικές περιπτώσεις προτιμάται η ενδιάμεση δημιουργία του *παράγωγου δένδρου* (parse tree) καθώς αναπαριστά λεπτομερέστερα τον κώδικα και τους κανόνες παραγωγής, επιτρέποντας έτσι την ευκολότερη εφαρμογή απλών μεθόδων στιλιστικών ελέγχων (stylistic checks). Πάντοτε όμως το τελικό παράγωγο είναι το συντακτικό δένδρο, το οποίο αποτελεί συμπαγή δενδρική αναπαράσταση του πηγαίου κώδικα και επιτρέπει την εφαρμογή σύνθετων μεθόδων ανάλυσης.

2.2.3 Σημασιολογική ανάλυση

Ταυτόχρονα με το χτίσιμο του συντακτικού δένδρου, τα πλείστα εργαλεία εφαρμόζουν και σημασιολογική ανάλυση στον κώδικα. Η σημασιολογική ανάλυση αφορά στην εφαρμογή της *γραμματικής ιδιοτήτων* (attribute grammar) της γλώσσας για την εξαγωγή ενός *πίνακα συμβόλων* (symbol table). Γραμματική ιδιοτήτων καλείται κάθε γραμματική χωρίς συμφραζόμενα, της οποίας οι κανόνες παραγωγής συνοδεύονται από σημασιολογικούς κανόνες που περιγράφουν τη στατική σημασία της γλώσσας. Το τελικό παράγωγο της ανάλυσης, ο πίνακας συμβόλων, αποτελεί μια δομή όπου καταγράφονται όλοι οι προσδιοριστές (identifiers), ο τύπος καθώς και δείκτης στη δήλωση του καθενός.

Με την ολοκλήρωση και της σημασιολογικής ανάλυσης το εργαλείο στατικής ανάλυσης είναι εφοδιασμένο με όλες τις απαραίτητες πληροφορίες που απαιτούνται για εφαρμογή πιο σύνθετων ελέγχων (π.χ. έλεγχος τύπων – type checking). Στην πράξη οι περισσότεροι έλεγχοι που αφορούν τη σημασιολογία της γλώσσας εκτελούνται από τον μεταγλωττιστή. Τα εργαλεία στατικής ανάλυσης είναι δυνατόν να εκτελέσουν επιπλέον ελέγχους σε περιπτώσεις που ο μεταγλωττιστής δεν αναφέρει πρόβλημα ενώ υπάρχει.

Επιπλέον, τα εργαλεία στατικής ανάλυσης είναι σημαντικό να εφοδιαστούν με αυτές τις πληροφορίες ειδικά σε περίπτωση που θα γίνει ανάλυση αντικειμενοστραφούς γλώσσας ώστε να είναι εφικτός ο προσδιορισμός των μεθόδων που επιτρέπεται να καλέσει ένα αντικείμενο. Η ανάγκη αυτή έγκειται σε θεμελιώδης αρχές των αντικειμενοστραφών γλωσσών όπως η κληρονομικότητα και ο πολυμορφισμός.

2.2.4 Γράφος ροής ελέγχου

Για την εφαρμογή των αλγορίθμων ανάλυσης απαιτείται συνήθως η εξερεύνηση των διαφορετικών *μονοπατιών εκτέλεσης* (execution path) που μπορούν να ληφθούν κατά την εκτέλεση μιας μεθόδου. Για το λόγο αυτό τα εργαλεία στατικής ανάλυσης χτίζουν ένα *γράφο ροής ελέγχου* (control flow graph). Οι κόμβοι του γράφου αναπαριστούν βασικά μπλοκ, ακολουθίες δηλαδή εντολών που θα εκτελεστούν αδιάσπαστα. Οι ακμές είναι κατευθυνόμενες και αναπαριστούν ροές ελέγχου μεταξύ βασικών μπλοκ.

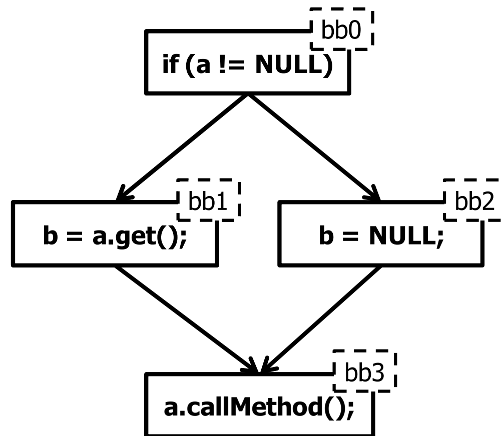
Παράδειγμα 2.1

Στο παρακάτω τμήμα κώδικα, αν και φαινομενικά δεν υπάρχει πρόβλημα εντούτοις η κλήση της μεθόδου `callMethod()` στο αντικείμενο `a` είναι δυνατόν να οδηγήσει σε null pointer dereference εάν η εκτέλεση ακολουθήσει το `else` μονοπάτι της διακλάδωσης.

```
if (a != NULL) {
   b = a.get();
} else {
   b = NULL;
}
a.callMethod();
```

Ο γράφος ροής ελέγχου που αντιστοιχεί στον παραπάνω κώδικα παρουσιάζεται στο Σχήμα 2.1. Η εντολή διακλάδωσης `if` αποτελεί ένα βασικό μπλοκ του γράφου από το οποίο προκύπτουν δύο ροές ελέγχου, μια για κάθε δυνατό μονοπάτι εκτέλεσης. Αφού εκτελεστεί το ανάλογο βασικό μπλοκ σύμφωνα με την συνθήκη της εντολής `if`, η ροή συνεχίζεται στην τελευταία εντολή του κώδικα. Από τον γράφο προκύπτουν δύο ακολουθίες ροής ελέγχου: [bb0, bb1, bb3] και [bb0, bb2, bb3].

Παρόλο που σε χρόνο εκτέλεσης είναι δυνατόν να εκτελεστεί η πρώτη ακολουθία και να μην υπάρξει πρόβλημα, ένα εργαλείο στατικής ανάλυσης είναι σε θέση να εξετάσει και τις δύο ακολουθίες. Από τον έλεγχο της ακολουθίας [bb0, bb2, bb3] προκύπτει ότι η μεταβλητή `a` δεν περιέχει αναφορά σε κάποιο αντικείμενο και επομένως η κλήση της μεθόδου `callMethod()` θα προκαλέσει null pointer dereference.



Σχήμα 2.1 Γράφος ροής ελέγχου για το Παράδειγμα 2.1

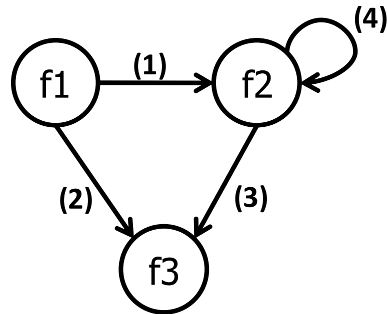
2.2.5 Γράφος κλήσεων

Σε αντίθεση με τον γράφο ροής ελέγχου που αναπαριστά την ροή εκτέλεσης εντός μιας συγκεκριμένης μεθόδου, ο *γράφος κλήσεων* (call graph) αναπαριστά την ροή εκτέλεσης μεταξύ μεθόδων. Οι κόμβοι του γράφου αναπαριστούν μεθόδους του προγράμματος ενώ οι κατευθυνόμενες ακμές αναπαριστούν τη ροή ελέγχου μεταξύ αυτών των μεθόδων. Με το χτίσιμο του γράφου κλήσεων το εργαλείο στατικής ανάλυσης εφοδιάζεται με μια συνοπτική περιγραφή των εξαρτήσεων που υπάρχουν μεταξύ των μεθόδων ώστε να είναι δυνατή η εφαρμογή αλγορίθμων ανάλυσης σε εμβέλεια προγράμματος.

Παράδειγμα 2.2

Στο Σχήμα 2.2 παρουσιάζεται ένα απόσπασμα κώδικα και ο αντίστοιχος γράφος κλήσεων. Οι τρεις μέθοδοι που φαίνονται στον κώδικα αποτελούν τους κόμβους του γράφου ενώ οι κλήσεις μεθόδων αντιστοιχούν σε κατευθυνόμενες ακμές στον γράφο. Συγκεκριμένα από την μέθοδο `f1` σχεδιάζουμε δύο ακμές, προς τη μέθοδο `f2` και την μέθοδο `f3` αντίστοιχα. Παρατηρούμε ότι σε αντίθεση με τον γράφο ροής ελέγχου, οι ακμές εδώ είναι ανεξάρτητες του μονοπατιού εκτέλεσης και αφορούν μόνο στις συσχετίσεις μεταξύ των μεθόδων. Κατά τον ίδιο τρόπο, οι κλήσεις στις μεθόδους `f3` και `f2` στο σώμα της μεθόδου `f2` εμφανίζονται στο γράφο ως ακμές προς τους αντίστοιχους κόμβους.

```
void  f1(int v) {
   if (v)
      f2(1);    (1)
   else
      f3();     (2)
}
int  f2(int v) {
   if (v) {
      f3();     (3)
      f2(0);    (4)
   } else {
      f3();
   }
}
int f3() {
   /* empty */
}
```



Σχήμα 2.2 Κώδικας και αντίστοιχος γράφος ροής ελέγχου για το Παράδειγμα 2.2

2.3 Αλγόριθμοι ανάλυσης

Στα εργαλεία στατικής ανάλυσης επιβάλλεται η χρήση εξειδικευμένων και πολύπλοκων αλγορίθμων ανάλυσης ώστε να γίνει όσο το δυνατόν πιο ακριβής εκτίμηση της επικινδυνότητας του κώδικα. Για να επιτευχθεί ο στόχος αυτός θα πρέπει οι αλγόριθμοι να είναι σε θέση να «κατανοούν» τις συνθήκες κάτω από τις οποίες εκτελείται ένα συγκεκριμένο τμήμα κώδικα. Για παράδειγμα, είναι γενικά εύκολο να εντοπίσουμε όλες τις κλήσεις `strcpy()` που υπάρχουν στο πρόγραμμα και να πούμε ότι πρέπει να αντικατασταθούν. Όμως, είναι πολύ πιο δύσκολο να εντοπίσουμε μόνο τις κλήσεις `strcpy()` που μπορούν να επιτρέψουν σε έναν επιτιθέμενο να υπερχειλίσει κάποια ενδιάμεση μνήμη (buffer).

Όλες οι εξελιγμένες στρατηγικές ανάλυσης αποτελούνται από τουλάχιστον δύο επίπεδα ανάλυσης: *εντός των μεθόδων* (intraprocedural) και *μεταξύ των μεθόδων* (interprocedural)[4]. Οι αλγόριθμοι που εκτελούν ανάλυση εντός των μεθόδων αξιοποιούν συνήθως τη δομή του συντακτικού δένδρου και αυτή του γράφου ροής ελέγχου. Στην περίπτωση των αλγορίθμων που λαμβάνουν υπόψη και τις συσχετίσεις μεταξύ των μεθόδων είναι απαραίτητη επιπλέον και η χρήση γράφου κλήσεων.

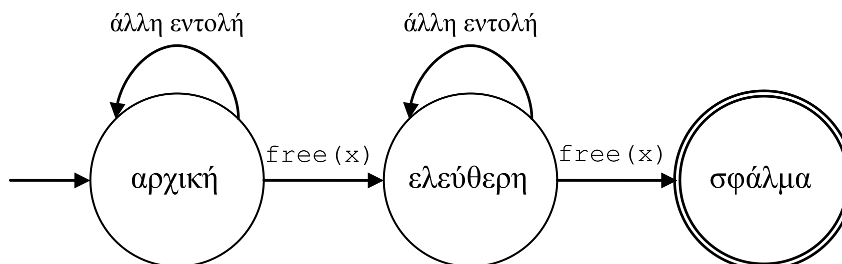
---

[4] Στη βιβλιογραφία χρησιμοποιείται μερικές φορές και ο όρος καθολική ανάλυση (global analysis) αντί του όρου interprocedural analysis.

Μια απλή σχετικά μορφή ανάλυσης είναι ο *έλεγχος μοντέλου* (model checking). Σύμφωνα με την προσέγγιση αυτή, το προς ανάλυση πρόγραμμα μετασχηματίζεται σε ένα αυτόματο το οποίο καλείται μοντέλο και συγκρίνεται ως προς ένα σύνολο προκαθορισμένων ιδιοτήτων τις οποίες δέχεται ως προδιαγραφές. Με τον όρο «ιδιότητα» αναφερόμαστε σε μια συνθήκη που αφορά στην ασφάλεια του κώδικα. Εάν η συνθήκη αυτή δεν ισχύει, τότε εντοπίζεται σφάλμα στο πρόγραμμα. Τέτοιες ιδιότητες μπορεί για παράδειγμα να ορίζουν ότι «η μνήμη πρέπει να ελευθερώνεται μόνο μια φορά» και ότι «αναφορές πρέπει να γίνονται μόνο σε δείκτες που δεν είναι null».

Παράδειγμα 2.3

Έστω ότι επιθυμούμε να ελέγξουμε την ιδιότητα «η μνήμη πρέπει να ελευθερώνεται μόνο μια φορά». Μπορούμε να κατασκευάσουμε ένα πεπερασμένο αυτόματο που να περιγράφει αυτή την ιδιότητα, όπως το αυτόματο στο Σχήμα 2.3. Εάν ο ελεγκτής μοντέλου μπορεί να βρει μια μεταβλητή και ένα μονοπάτι στο πρόγραμμα το οποίο θα προκαλέσει το πεπερασμένο αυτόματο να περιέλθει σε κατάσταση σφάλματος, τότε έχει εντοπιστεί ένα ενδεχόμενο κενό ασφαλείας στο πρόγραμμα.



Σχήμα 2.3 Πεπερασμένο αυτόματο ιδιότητας για το Παράδειγμα 2.3

2.4 Γνώση θεμάτων ασφαλείας

Κάθε εργαλείο στατικής ανάλυσης περιλαμβάνει κωδικοποιημένη γνώση, *πρότυπα*[5], για το πώς μοιάζει ένα τμήμα κώδικα το οποίο είναι επίφοβο για την εμφάνιση σφάλματος. Οι οντότητες που υλοποιούν αυτά τα πρότυπα καλούνται *ανιχνευτές* (detectors).

---

[5] Στη σχετική βιβλιογραφία συναντάται και ο όρος «κανόνες» ο οποίος προέρχεται από την ορολογία ενός αρκετά διαδεδομένου εμπορικού εργαλείου που ονομάζεται Fortify Source Code Analysis.

Πιο ειδικά, τα πρότυπα οδηγούν τον αναλυτή δίνοντάς του ειδικές πληροφορίες για το πώς είναι ένα σφάλμα, όπως για παράδειγμα κάποια συγκεκριμένη αλληλουχία κώδικα που είναι επίφοβη. Είναι επίσης πιθανό να δίνει την πληροφορία ποιες τιμές μεταβλητών είναι δυνατόν να δημιουργήσουν πρόβλημα και να ερευνάται στη συνέχεια αν κάτι τέτοιο ισχύει στατικά. Μια ακόμη περίπτωση είναι να δίνει στοιχεία που έχουν σχέση με την ροή εκτέλεσης του προγράμματος και να γίνεται ανάλυση σε αυτή την κατεύθυνση.

Οι παράγοντες που χαρακτηρίζουν ένα πρότυπο είναι η πολυπλοκότητα, ο βαθμός ακρίβειας και η ταχύτητα της ανάλυσης. Ακόμη και απλά από άποψη πολυπλοκότητας πρότυπα, είναι δυνατόν να εντοπίσουν μεγάλο πλήθος ελαττωμάτων στον κώδικα (7). Η ακρίβεια ενός προτύπου εξαρτάται άμεσα από τη σχεδίασή του. Έτσι, ένα γενικό πρότυπο μπορεί να στερείται ακρίβειας ενώ ένα ειδικότερο πρότυπο μπορεί να χαρακτηρίζεται από αυξημένη ακρίβεια σε ότι αφορά τα σφάλματα που έχει σχεδιαστεί να εντοπίζει. Συνήθως τα χαρακτηριστικά αυτά είναι αλληλένδετα. Ένα πολύπλοκο πρότυπο μπορεί μεν να παρουσιάζει αυξημένη ακρίβεια, αλλά να υστερεί σε ταχύτητα εκτέλεσης.

Παράδειγμα 2.4

```
if (listeners == null)
    listeners.remove(listener);
```

Ο παραπάνω κώδικας περιέχει ένα απλό σφάλμα: γίνεται έλεγχος αν μια μεταβλητή είναι κενή (`null`) και εάν ναι, χρησιμοποιείται. Ένα πιθανό πρότυπο θα μπορούσε να ορίζει ότι πρέπει να αναφερθεί ως σφάλμα η περίπτωση κατά την οποία «μια μεταβλητή που είναι κενή χρησιμοποιείται». Ένα πιο εξειδικευμένο πρότυπο θα όριζε την αναφορά σφάλματος μόνο όταν «μια μεταβλητή που ελέγχεται αν είναι κενή, χρησιμοποιείται μέσα στο μπλοκ του κώδικα όπου γίνεται ο έλεγχος». Για τον συγκεκριμένο κώδικα και τα δύο πρότυπα είναι έγκυρα, αλλά το δεύτερο είναι πιο ειδικό και επομένως δυσκολότερο να υλοποιηθεί.

2.5 Αποτελέσματα ανάλυσης

Τα αποτελέσματα της ανάλυσης κατατάσσονται σε δύο κατηγορίες: τα *έγκυρα θετικά* (true positive) και τα *μη έγκυρα θετικά* (false positives). Έγκυρα θετικά θεωρούνται τα αποτελέσματα που αντιστοιχούν σε πραγματικά σφάλματα στον κώδικα.

Στο Παράδειγμα 2.4, εάν το εργαλείο στατικής ανάλυσης τελικά αναφέρει ότι εντοπίζεται πρόβλημα στον κώδικα, τότε το αποτέλεσμα είναι έγκυρο αφού πράγματι η αναφορά σε κενή μεταβλητή αποτελεί σφάλμα.

Τα μη έγκυρα θετικά (false positives) αφορούν σε περιπτώσεις που το εργαλείο εντοπίζει σφάλμα που στην πραγματικότητα δεν υφίσταται. Επιπλέον, υπάρχει και μια τρίτη κατηγορία που περιλαμβάνει τα *μη έγκυρα αρνητικά* (false negatives). Στην κατηγορία αυτή κατατάσσονται τα σφάλματα που ενώ υπάρχουν στον κώδικα, εντούτοις, δεν εντοπίζονται και άρα δεν αναφέρονται από το εργαλείο στατικής ανάλυσης.

Και οι δύο τελευταίες κατηγορίες αποτελεσμάτων είναι ανεπιθύμητες από την πλευρά των χρηστών/προγραμματιστών. Πληθώρα μη έγκυρων θετικών αποτελεσμάτων απαιτεί επιπλέον κόπο για την ανασκόπησή τους από τον προγραμματιστή ενώ ταυτόχρονα υπάρχει το ενδεχόμενο να παραβλεφθεί ένα έγκυρο θετικό αποτέλεσμα που βρίσκεται «θαμμένο» στα μη έγκυρα. Σημαντικότερη όμως είναι η περίπτωση των μη έγκυρων αρνητικών αποτελεσμάτων. Αυτό συνεπάγεται ότι υπάρχει κάποιο σφάλμα στον κώδικα που όμως το εργαλείο στατικής ανάλυσης δεν κατάφερε να εντοπίσει. Έτσι, ενώ το πρόγραμμα έχει κάποιο κενό ασφαλείας, ο προγραμματιστής πιστεύει ότι είναι ασφαλές αφού το εργαλείο δεν υπέδειξε κάποιο πρόβλημα.

3 Ανάλυση Ροής Δεδομένων

Η ανάλυση ροής δεδομένων (data flow analysis) είναι μια ευρεία οικογένεια τεχνικών οι οποίες εξάγουν πληροφορία σχετικά με τη ροή των δεδομένων στα διάφορα *μονοπάτια εκτέλεσης* (execution path) του προγράμματος. Στην οικογένεια αυτή κατατάσσονται αλγόριθμοι που απαντούν σε ερωτήματα όπως: «ποιοι δείκτες δείχνουν στην ίδια θέση μνήμης;» και «είναι δυνατόν δεδομένα που εισάγονται από το εξωτερικό περιβάλλον να επηρεάσουν την ασφάλεια της εφαρμογής;».

3.1 Βασικές έννοιες ανάλυσης ροής δεδομένων

Η εκτέλεση ενός προγράμματος μπορεί να θεωρηθεί ως μια σειρά μετασχηματισμών της κατάστασης του προγράμματος, η οποία αποτελείται από τις τιμές όλων των μεταβλητών του προγράμματος. Με την εκτέλεση κάθε εντολής, η κατάσταση εισόδου μετασχηματίζεται σε μια νέα κατάσταση εξόδου. Η κατάσταση εισόδου συσχετίζεται με το *σημείο* του προγράμματος ακριβώς πριν από την εντολή, ενώ η κατάσταση εξόδου αφορά στο σημείο του προγράμματος ακριβώς μετά την εντολή που εκτελέστηκε. Για την ανάλυση της συμπεριφοράς του προγράμματος πρέπει να ληφθούν υπόψη όλες οι πιθανές ακολουθίες σημείων που μπορεί να ακολουθήσει η εκτέλεση του προγράμματος.

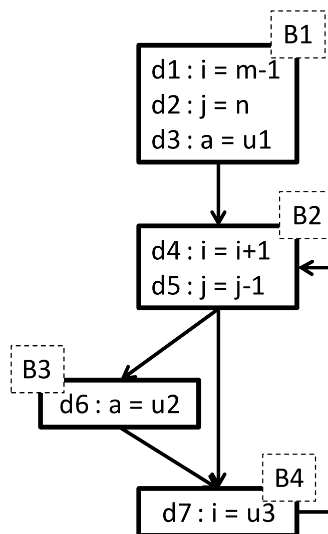
Γενικά, το πλήθος των πιθανών μονοπατιών εκτέλεσης είναι άπειρο ενώ ταυτόχρονα δεν ορίζεται πεπερασμένο άνω όριο στο μήκος ενός μονοπατιού. Για να το δούμε αυτό ας υποθέσουμε ότι στο πρόγραμμα υπάρχει ένα βρόγχος while η συνθήκη του οποίου ορίζει ότι ο βρόγχος θα εκτελείται όσο ο χρήστης εισάγει αριθμό διάφορο του μηδενός από τη γραμμή εντολών. Εξετάζοντας το πρόγραμμα στατικά, ο βρόγχος μπορεί να μην εκτελεστεί εάν ο χρήστης εισάγει εξαρχής το μηδέν ενώ μπορεί να εκτελεστεί θεωρητικά άπειρες φορές, όσο ο χρήστης εισάγει διαφορετικούς αριθμούς.

Στην πράξη δεν είναι εφικτό να καταγράφεται η πλήρης κατάσταση του προγράμματος. Αντίθετα, επιλέγεται η καταγραφή συγκεκριμένων δεδομένων που είναι απαραίτητα για το σκοπό της ανάλυσης που εκτελείται. Επιπλέον, προτιμάται συνήθως η καταγραφή της κατάστασης στα όρια των βασικών μπλοκ καθώς είναι εύκολο να υπολογιστεί από αυτήν η κατάσταση σε κάθε σημείο μέσα στο μπλοκ.

Παράδειγμα 3.1

Στο σημείο αυτό αξίζει να μελετήσουμε μια εφαρμογή της ανάλυσης ροής δεδομένων που συνήθως χρησιμοποιείται στους μεταγλωττιστές, ώστε να αποσαφηνιστούν τα όσα έχουν αναφερθεί. Το πρόβλημα που θα εξετάσουμε καλείται φάσμα ορισμών (reaching definitions) και σκοπός είναι για κάθε σημείο του προγράμματος να υπολογίσουμε το σύνολο των ορισμών που είναι δυνατόν να προσεγγίσουν το σημείο αυτό. Για την απλοποίηση του προβλήματος, θα υπολογίσουμε το φάσμα ορισμών στο σημείο εισόδου στο δεύτερο βασικό μπλοκ (B2) στο Σχήμα 3.1.

Λέμε ότι ένας ορισμός d προσεγγίζει ένα σημείο p εάν υπάρχει μονοπάτι από το σημείο ακριβώς μετά τον d προς το σημείο p, έτσι ώστε ο d να μην εξουδετερώνεται κατά μήκος αυτού του μονοπατιού. Στο γράφο του Σχήμα 3.1 ο ορισμός d1 προσεγγίζει το σημείο πριν από τον ορισμό d4 αλλά όχι το σημείο πριν από τον ορισμό d5 αφού ο d4 εξουδετερώνει τον d1.



Σχήμα 3.1 Γράφος ροής ελέγχου για το πρόβλημα του φάσματος ορισμών

Με βάση αυτά, ας εξετάσουμε τώρα τον γράφο ροής ελέγχου του σχήματος. Όλοι οι ορισμοί του μπλοκ B1 (d1, d2, d3) προσεγγίζουν την αρχή του μπλοκ B2. Ο ορισμός d5: j = j-1 στο μπλοκ B2 επίσης προσεγγίζει την αρχή του B2, επειδή δεν υπάρχει άλλος ορισμός για την μεταβλητή j στον βρόγχο που οδηγεί πίσω στο B2 και άρα ο d5 δεν εξουδετερώνεται. Εντούτοις, ο ορισμός αυτός εξουδετερώνει το ορισμό d2: j = n, αποτρέποντας τον από το να προσεγγίσει τα μπλοκ B3 και B4. Ο ορισμός d4: i = i+1 στο B2 δεν προσεγγίζει την αρχή του B2, καθώς η μεταβλητή i επαναορίζεται

στον ορισμό d7, όποιο μονοπάτι και αν ακολουθηθεί. Τέλος, ο ορισμός d6: a = u2 προσεγγίζει επίσης την αρχή του μπλοκ B2. Προκύπτει λοιπόν ότι το φάσμα ορισμών για το σημείο εισόδου στο B2 είναι το {d1, d2, d3, d5, d6, d7}.

Από το παραπάνω παράδειγμα προκύπτουν τα εξής:

- Τα πιθανά μονοπάτια εκτέλεσης ενός προγράμματος είναι, θεωρητικά, άπειρα. Ακόμη και για αυτό το απλό παράδειγμα ο βρόγχος που αρχίζει από το μπλοκ B2 μπορεί να εκτελεστεί ενδεχομένως άπειρες φορές παράγοντας έτσι άπειρα μονοπάτια εκτέλεσης.
- Το μήκος ενός μονοπατιού εκτέλεσης δεν έχει πεπερασμένο άνω όριο. Στο συγκεκριμένο παράδειγμα το μήκος του μονοπατιού εξαρτάται από το πλήθος των επαναλήψεων του βρόγχου. Όπως έχει ήδη αναφερθεί, αυτό μπορεί να είναι θεωρητικά άπειρο και επομένως δεν ορίζεται άνω φράγμα για το μήκος του μονοπατιού.
- Για την ανάλυση ενός συγκεκριμένου προβλήματος δεν απαιτείται η καταγραφή της πλήρους κατάστασης σε κάθε σημείο του προγράμματος. Στο πρόβλημα του φάσματος ορισμών είναι αρκετή η καταγραφή μόνο των μεταβλητών στο αριστερό τμήμα των ορισμών και η πληροφορία αυτή μπορεί να καταγράφεται μόνο για την είσοδο στα βασικά μπλοκ.

3.2 Σχήμα ανάλυσης ροής δεδομένων

Για να γίνει δυνατή η μελέτη όλων των εφαρμογών χρειάζεται να καθοριστεί ένα ευρύτερο πλαίσιο που να καλύπτει την ανάλυση ροής δεδομένων. Έτσι, για κάθε εφαρμογή, συσχετίζουμε σε κάθε σημείο του προγράμματος μια *τιμή ροής δεδομένων* (data flow value) που αναπαριστά το σύνολο όλων των πιθανών καταστάσεων του προγράμματος που μπορούν να παρατηρηθούν στο σημείο αυτό. Το σύνολο όλων τον δυνατών τιμών ροής δεδομένων καλείται *πεδίο ορισμού* (domain) της εφαρμογής. Για παράδειγμα, το πεδίο ορισμού στο πρόβλημα του φάσματος ορισμών είναι το σύνολο όλων των υποσυνόλων των ορισμών που υπάρχουν στο πρόγραμμα. Μια συγκεκριμένη τιμή ροής δεδομένων είναι ένα σύνολο ορισμών, και στόχος είναι να συσχετίσουμε κάθε σημείο του προγράμματος με αυτό ακριβώς το σύνολο ορισμών που μπορεί να προσεγγίσει το σημείο.

Συμβολίζουμε τις τιμές ροής δεδομένων πριν και μετά μια εντολή s ως IN[s] και OUT[s], αντίστοιχα. Το πρόβλημα πλέον, μετατρέπεται στο να βρούμε μια λύση στο σύνολο των περιορισμών που ορίζεται από τα IN[s] και OUT[s], για όλες τις εντολές s. Υπάρχουν δύο κατηγορίες περιορισμών: αυτοί που βασίζονται στη σημασιολογία των εντολών (συναρτήσεις μετάβασης) και αυτοί που στηρίζονται στη ροή ελέγχου.

Οι τιμές ροής δεδομένων πριν και μετά από κάποια εντολή περιορίζονται από τη σημασιολογία της εντολής. Για παράδειγμα, ας υποθέσουμε ότι η ανάλυση στοχεύει στον προσδιορισμό της τιμής των μεταβλητών. Αν η μεταβλητή a έχει τιμή k πριν την εκτέλεση της εντολής b = a, τότε και οι δύο μεταβλητές a και b θα έχουν τιμή k μετά την εκτέλεση της εντολής. Αυτή η σχέση μεταξύ των τιμών ροής δεδομένων πριν και μετά την εντολή ανάθεσης είναι μια *συνάρτηση μετάβασης*. Άρα, η συνάρτηση μετάβασης μιας εντολής s, την οποία συμβολίζουμε f_s, παίρνει την τιμή ροής δεδομένων πριν από την εντολή και παράγει μια νέα τιμή ροής δεδομένων μετά την εντολή. Συμβολικά μπορούμε να γράψουμε: $OUT[s] = f_s(IN[s])$.

Η δεύτερη κατηγορία περιορισμών οφείλεται στην ροή ελέγχου. Μέσα σε ένα βασικό μπλοκ, η ροή είναι απλή. Εάν το μπλοκ B αποτελείται από τις εντολές $s_1,s_2,\ldots,s_n$ σε αυτή τη σειρά, τότε η τιμή ροής δεδομένων μετά την εντολή s_i είναι ίδια με την τιμή πριν από την εντολή s_{i+1}. Άρα, $IN[s_{i+1}] = OUT[s_i]$ για κάθε $i = 1,2,\ldots,n-1$.

Ωστόσο, οι ακμές στον γράφο ροής ελέγχου μεταξύ των βασικών μπλοκ δημιουργούν πιο σύνθετους περιορισμούς μεταξύ της τελευταίας εντολής ενός βασικού μπλοκ και της πρώτης εντολής του μπλοκ που ακολουθεί. Για παράδειγμα, στο πρόβλημα του φάσματος ορισμών, το σύνολο των ορισμών που προσεγγίζουν την πρώτη εντολή ενός μπλοκ B είναι η ένωση των ορισμών αμέσως μετά τις τελευταίες εντολές όλων των μπλοκ που οδηγούν άμεσα στο B.

Όπως έχουμε ήδη αναφέρει, είναι επαρκής η καταγραφή της κατάστασης μόνο στα όρια των βασικών μπλοκ. Επομένως μπορούμε να διατυπώσουμε τους πιο πάνω ορισμούς με βάση αυτή τη θεώρηση. Έτσι, συμβολίζουμε τις τιμές ροής δεδομένων ακριβώς πριν και ακριβώς μετά από κάθε βασικό μπλοκ B με IN[B] και OUT[B], αντίστοιχα. Εάν το μπλοκ B αποτελείται από τις εντολές $s_1,s_2,\ldots,s_n$, τότε ισχύουν οι σχέσεις:

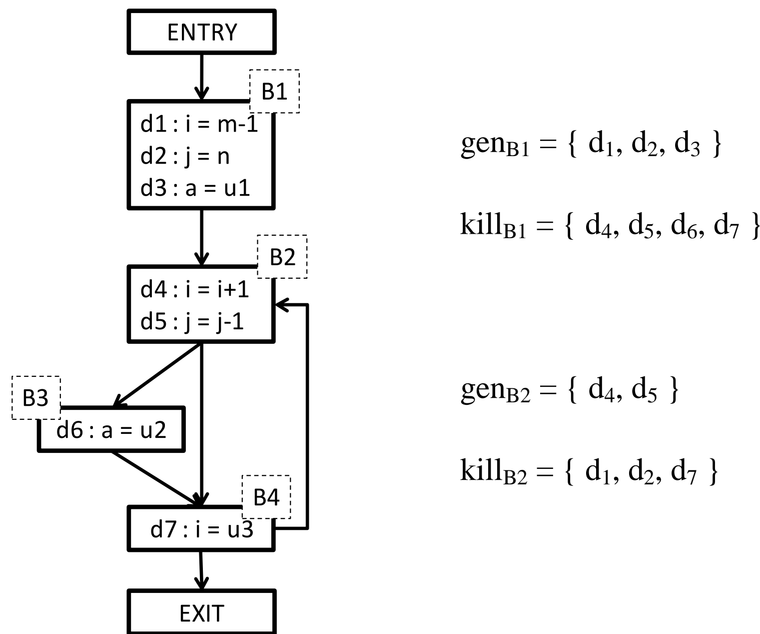
$IN[B] = IN[s_1]$ και $OUT[B] = OUT[s_n]$

Η συνάρτηση μετάβασης ενός βασικού μπλοκ Β, που συμβολίζεται με f_B, μπορεί να εξαχθεί από την σύνθεση των επιμέρους εντολών του μπλοκ ως εξής:

$$f_B = f_{s_n} \circ \ldots \circ f_{s_2} \circ f_{s_1}$$

Η σχέση μεταξύ της αρχής και του τέλους του μπλοκ ορίζεται από τη σχέση

$$OUT\ [B] = f_B(IN[B])$$.



Σχήμα 3.2 Γράφος ροής ελέγχου και αντίστοιχα σύνολα για το Παράδειγμα 3.2

Παράδειγμα 3.2

Στο σημείο αυτό αξίζει να επανεξετάσουμε το πρόβλημα του φάσματος ορισμών χρησιμοποιώντας την ορολογία που ορίσαμε. Αρχικά πρέπει να κατασκευάσουμε τους περιορισμούς του προβλήματος. Γενικά, ένας ορισμός έχει τη μορφή d: u = v + w , όπου + ένας δυαδικός τελεστής. Η εντολή αυτή παράγει τον ορισμό d της μεταβλητής u και εξουδετερώνει όλους τους προηγούμενους ορισμούς για την μεταβλητή u. Η συνάρτηση μεταφοράς μπορεί να εκφραστεί ως

$$f_d(x) = gen_d \cup (x - kill_d)$$

όπου $gen_d = \{d\}$, το σύνολο των ορισμών που παράγονται από την εντολή, και $kill_d$ το σύνολο των ορισμών που εξουδετερώνονται από τον ορισμό d.

Όπως έχουμε δει, η συνάρτηση μεταφοράς ενός βασικού μπλοκ παράγεται από τη σύνθεση των συναρτήσεων μεταφοράς των επιμέρους εντολών του μπλοκ. Αποδεικνύεται ότι στην περίπτωση αυτή η συνάρτηση μεταφοράς ενός μπλοκ B είναι:

$$f_B(x) = gen_B \cup (x - kill_B)$$

όπου

$$kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$$

και

$$gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \\ \ldots \cup (gen_1 - kill_2 - kill_3 - \ldots - kill_n)$$

Ακολούθως, πρέπει να σχηματίσουμε τους περιορισμούς που οφείλονται στην ροή ελέγχου μεταξύ των βασικών μπλοκ. Όπως έχει ήδη αναφερθεί, το σύνολο των ορισμών που προσεγγίζουν την πρώτη εντολή ενός μπλοκ B είναι η ένωση των ορισμών αμέσως μετά τις τελευταίες εντολές όλων των μπλοκ που οδηγούν άμεσα στο B. Επεκτείνοντας την παρατήρηση αυτή σε επίπεδο μπλοκ μπορούμε να πούμε ότι

$$IN[B] = \cup_{P\ a\ predecessor\ of\ B} OUT[P]$$

Υποθέτουμε ότι κάθε γράφος ροής ελέγχου έχει δύο άδεια βασικά μπλοκ, έναν κόμβο εισόδου, και έναν κόμβο εξόδου. Καθώς δεν υπάρχουν ορισμοί που να προσεγγίζουν την αρχή του γράφου, η συνάρτηση μετάβασης για το μπλοκ εισόδου είναι μια απλή συνάρτηση που επιστρέφει πάντοτε Ø ως απάντηση. Ο προσαρμοσμένος γράφος με τα αντίστοιχα σύνολα gen και kill για κάθε βασικό μπλοκ, φαίνεται στο Σχήμα 3.2.

Τελικά το πρόβλημα του φάσματος ορισμών ορίζεται από τις εξισώσεις:

$$OUT[ENTRY] = \emptyset$$

$$OUT[B] = gen_B \cup (IN[B] - kill_B)$$

$$IN[B] = \cup_{P\ a\ predecessor\ of\ B} OUT[P]$$

Ο αλγόριθμος που επιλύει τις εξισώσεις είναι ο ακόλουθος:

```
1.  OUT[ENTRY]= Ø;
2.  for (each basic block B other than ENTRY) OUT[B]= Ø;
3.  while (changes to any OUT occur)
4.      for (each basic block B other than ENTRY){
5.              IN[B] = ∪ P a predecessor of B OUT[P];
6.              OUT[B]=genB ∪(IN[B]-killB);
7.      }
```

Για την εφαρμογή του αλγορίθμου ας υποθέσουμε ότι αναπαριστούμε τους επτά ορισμούς $d_1,d_2,\ldots,d_7$ του γράφου ροής ελέγχου με ένα δυαδικό διάνυσμα, όπου το δυαδικό ψηφίο i από τα αριστερά αναπαριστά τον ορισμό d_i. Η ένωση συνόλων υπολογίζεται λαμβάνοντας το λογικό OR των αντίστοιχων διανυσμάτων. Η διαφορά δύο συνόλων S-T υπολογίζεται συμπληρώνοντας το διάνυσμα του T και λαμβάνοντας στη συνέχεια το λογικό AND του συμπληρώματος, με το διάνυσμα του S.

Ο Πίνακας 3.1 παρουσιάζει τις τιμές των συνόλων IN και OUT κατά την εκτέλεση του αλγορίθμου. Οι αρχικές τιμές (στήλη $OUT[B]^0$) αντιστοιχούν στην εντολή 2 του αλγορίθμου, με το κενό σύνολο να αναπαριστάται με το διάνυσμα 000 0000. Οι επόμενες στήλες αντιστοιχούν στα σύνολα που υπολογίζονται κατά την εκτέλεση του βρόγχου for του αλγορίθμου, ενώ η επιγραφή σε κάθε σύνολο υποδηλώνει την επανάληψη του βρόγχου while στην οποία υπολογίζεται το συγκεκριμένο σύνολο.

Block B	$OUT[B]^0$	$IN[B]^1$	$OUT[B]^1$	$IN[B]^2$	$OUT[B]^2$
B1	000 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	111 0000	001 1100	111 0111	001 1110
B3	000 0000	001 1100	000 1110	001 1110	000 1110
B4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

Πίνακας 3.1 Υπολογισμός συνόλων IN και OUT αλγορίθμου

Ας υποθέσουμε ότι ο βρόγχος for εκτελείται με το B να λαμβάνει τις τιμές B1, B2, B3, B4, EXIT. Για B=B1, αφού OUT[ENTRY]= Ø, από την γραμμή 5 του αλγορίθμου προκύπτει ότι $IN[B1]^1$=Ø ενώ από την γραμμή 6 προκύπτει $OUT[B1]^1=gen_{B1}=\{d_1, d_2, d_3\}$=[111 0000]. Συνεχίζοντας, B=B2 και άρα

$IN[B2]^1 = OUT[B1]^1 \cup OUT[B4]^0$

= [111 0000] + [000 0000] = [111 0000]

$OUT[B2]^1 = gen_{B2} \cup (IN[B2]^1 - kill_{B2})$

= [000 11000] + ([111 0000] - [110 0001]) = [001 1100]

Συνολικά εκτελούνται δύο επαναλήψεις του βρόγχου for του αλγορίθμου, αφού κατά τον υπολογισμό της δεύτερης επανάληψης δεν προκύπτουν αλλαγές στα σύνολα OUT. Εξετάζοντας τα τελικά αποτελέσματα, παρατηρούμε ότι κατά την είσοδο στο βασικό μπλοκ B2 το φάσμα ορισμών είναι το [111 0111] ή αλλιώς το σύνολο $\{ d_1, d_2, d_3, d_5, d_6, d_7 \}$ όπως αναμενόταν και από το αποτέλεσμα που προέκυψε στο Παράδειγμα 3.1.

3.3 Θεμελιώσεις ανάλυσης ροής δεδομένων

Ένα *πλαίσιο ανάλυσης ροής δεδομένων* $(D, V, \wedge, F)$ αποτελείται από

- Την κατεύθυνση D της ανάλυσης, που είναι είτε Forward[6] ή Backward.
- Ένα ημιπλέγμα (semilattice) το οποίο περιλαμβάνει το πεδίο ορισμού V και τον τελεστή συνένωσης (meet operator) $\wedge$.
- Μια οικογένεια F από συναρτήσεις μετάβασης από το V στο V.

Ημιπλέγμα καλείται ένα σύνολο V με έναν δυαδικό τελεστή συνένωσης $\wedge$ έτσι ώστε για κάθε x, y και z που ανήκουν στο V:

- $x \wedge x = x$ (ταυτοδυναμία)
- $x \wedge y = y \wedge x$ (αντιμεταθετικότητα)
- $x \wedge (y \wedge z) = (x \wedge y) \wedge z$ (προσεταιριστικότητα)
- $\exists \top \in V : \top \wedge p = p \quad \forall p \in V$ (στοιχείο κορυφή, $\top$)
- $\exists \bot \in V : \bot \wedge p = \bot \quad \forall p \in V$ (στοιχείο βάση, $\bot$)

[6] Στην ενότητα αυτή ασχολούμαστε για λόγους συντομίας μόνο με forward ανάλυση. Όλοι οι ορισμοί όμως επεκτείνονται εύκολα και για backward ανάλυση.

Ο τελεστής συνένωσης ενός ημιπλέγματος ορίζει μια *μερική διάταξη* στις τιμές του πεδίου ορισμού. Μια σχέση ≤ είναι *μερική διάταξη* στο σύνολο V εάν για όλα τα x, y και z του V ισχύει:

- $x \leq x$ (αντανακλαστική)
- Αν $x \leq y$ και $y \leq x$, τότε $x = y$ (αντισυμμετρική)
- Αν $x \leq y$ και $y \leq z$, τότε $x \leq z$ (μεταβατική)

Το ζεύγος (V, ≤) καλείται *σύνολο μερικής διάταξης*. Στην περίπτωση του ημιπλέγματος (V, $\wedge$) η μερική διάταξη ≤ ορίζεται ως εξής: για κάθε x και y που ανήκουν στο V ισχύει $x \leq y$ αν και μόνον αν $x \wedge y = x$.

Αύξουσα αλυσίδα σε ένα σύνολο μερικής διάταξης καλείται μια ακολουθία για την οποία $x_1 \leq x_2 \leq \ldots \leq x_n$. Το *ύψος* ενός ημιπλέγματος ορίζεται ως ο μέγιστος αριθμός από ≤ σχέσεις σε κάθε αύξουσα αλυσίδα, δηλαδή, το ύψος είναι κατά ένα μικρότερο από το πλήθος των στοιχείων της αλυσίδας. Προφανώς, ένα ημιπλέγμα που αποτελείται από πεπερασμένο σύνολο τιμών θα έχει πεπερασμένο ύψος. Επιπλέον, είναι πιθανόν ένα ημιπλέγμα με άπειρο πλήθος τιμών να έχει πεπερασμένο ύψος.

Λέμε ότι ένα πλαίσιο ανάλυσης ροής δεδομένων (D, V, $\wedge$, F) είναι *μονότονο* αν για οποιεσδήποτε δύο τιμές του V, έστω x και y με x μικρότερο ή ίσο του y, η εφαρμογή οποιασδήποτε συνάρτησης f του συνόλου F στις δύο αυτές τιμές θα δώσει αποτελέσματα για τα οποία θα ισχύει f(x) μικρότερο ή ίσο του f(y). Ισοδύναμα, ισχύει ότι εάν πάρουμε την συνένωση δύο τιμών και εφαρμόσουμε σε αυτήν τη συνάρτηση f, το αποτέλεσμα ποτέ δεν είναι μεγαλύτερο από το αν εφαρμόζαμε την f στις δύο τιμές ξεχωριστά και μετά να παίρναμε την συνένωση των επιμέρους αποτελεσμάτων. Συνοπτικά μπορούμε να γράψουμε ότι για να είναι ένα πλαίσιο ανάλυσης ροής δεδομένων *μονότονο* θα πρέπει να ισχύει ένα από τα εξής:

- για κάθε x και y στο V και f στο F, αν $x \leq y$ τότε $f(x) \leq f(y)$
- για κάθε x και y στο V και f στο F, $f(x \wedge y) \leq f(x) \wedge f(y)$

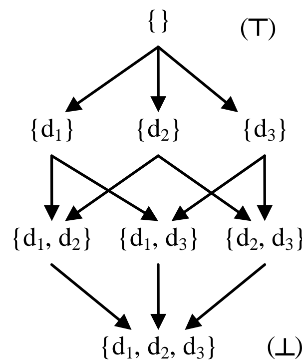
Παράδειγμα 3.3

Στο πρόβλημα του φάσματος ορισμών ο *τελεστής συνένωσης* είναι η ένωση συνόλου. Γι' αυτό και στο Παράδειγμα 3.2 για τον υπολογισμό του συνόλου IN[B], παίρναμε την ένωση των συνόλων OUT[P] των κόμβων P από τους οποίους άρχιζε

ακμή προς τον κόμβο B. Για την ένωση συνόλων, το *στοιχείο κορυφή* είναι το Ø ενώ το *στοιχείο βάση* είναι το σύνολο *U*, που περιλαμβάνει όλους τους ορισμούς του προγράμματος. Αυτό αποδεικνύεται αφού, για κάθε υποσύνολο x του *U*, ισχύει $\emptyset \cup x = x$ και $U \cup x = U$.

Για όλα τα x και y που ανήκουν στο V, η σχέση $x \cup y = x$ συνεπάγεται ότι $x \supseteq y$, επομένως, η *μερική διάταξη* που επιβάλλει η ένωση συνόλου είναι η σχέση του υπερσυνόλου. Αυτό σημαίνει ότι, για την πράξη της ένωσης συνόλων, ένα σύνολο με περισσότερα στοιχεία θεωρείται μικρότερο από ένα σύνολο με λιγότερα στοιχεία στην μερική διάταξη.

Στο Σχήμα 3.3 παρουσιάζεται το *διάγραμμα πλέγματος* για ένα πρόβλημα φάσματος ορισμών που περιλαμβάνει τρεις ορισμούς: d_1, d_2 και d_3. Το στοιχείο κορυφή εμφανίζεται στην κορυφή του διαγράμματος και αντίστοιχα, το στοιχείο βάση εμφανίζεται στη βάση του διαγράμματος. Το ύψος το ημιπλέγματος είναι ίσο με 3, αφού στην αύξουσα αλυσίδα $\{d_1, d_2, d_3\} \supseteq \{d_1, d_2\} \supseteq \{d_1\} \supseteq \{\}$ υπάρχουν τρεις σχέσεις υπερσυνόλου. Το αποτέλεσμα της εφαρμογής του τελεστή συνένωσης (ένωση συνόλου στο συγκεκριμένο παράδειγμα) σε δύο στοιχεία του πεδίου ορισμού φαίνεται στο διάγραμμα ακολουθώντας τις ακμές που αρχίζουν από τα δύο αυτά στοιχεία. Έτσι, για παράδειγμα, $\{d_1\} \cup \{d_2\} = \{d_1, d_2\}$.



Σχήμα 3.3 Διάγραμμα πλέγματος για πρόβλημα φάσματος ορισμών

3.4 Αλγόριθμος ανάλυσης ροής δεδομένων

Πλέον, είμαστε σε θέση να γενικεύσουμε τον αλγόριθμο που είδαμε στο Παράδειγμα 3.2 ώστε να ανταποκρίνεται στο γενικότερο πλαίσιο που έχει οριστεί.

Έτσι, ο αλγόριθμος αυτός θα μπορεί να εφαρμοστεί σε ένα πλήθος προβλημάτων ροής δεδομένων.

Ως είσοδο, ο αλγόριθμος λαμβάνει τα ακόλουθα:

1. Ένα γράφο ροής ελέγχου με επιπρόσθετους κόμβους εισόδου-εξόδου.
2. Ένα σύνολο τιμών V
3. Έναν τελεστή συνένωσης $\wedge$
4. Ένα σύνολο συναρτήσεων F, όπου f_B στο F είναι η συνάρτηση μεταφοράς για το μπλοκ B
5. Μια σταθερή τιμή u_{ENTRY} που ανήκει στο V και αντιπροσωπεύει την συνοριακή συνθήκη στον κόμβο εισόδου[7].

Η έξοδος του αλγορίθμου είναι οι τιμές των συνόλων IN[B] και OUT[B] για κάθε βασικό μπλοκ B του γράφου.

Τα βήματα του αλγορίθμου είναι τα ακόλουθα:

```
1.  OUT[ENTRY]= u_ENTRY;
2.  for (each basic block B other than ENTRY) OUT[B]= ⊤;
3.  while (changes to any OUT occur)
4.        for (each basic block B other than ENTRY){
5.              IN[B] = ∧ P a predecessor of B OUT[P];
6.              OUT[B]= f_B (IN[B]);
7.        }
```

Αξίζει να σημειωθούν δύο βασικές ιδιότητες του αλγορίθμου:

- Εάν ο αλγόριθμος συγκλίνει, το αποτέλεσμα είναι μια λύση των εξισώσεων που ορίζουν το συγκεκριμένο πρόβλημα ανάλυσης ροής δεδομένων.
- Εάν το πλέγμα του προβλήματος είναι μονότονο και πεπερασμένου ύψους, ο αλγόριθμος εγγυημένα θα συγκλίνει.

[7] Όπως είδαμε στο Παράδειγμα 3.2 η συνοριακή τιμή για το πρόβλημα του φάσματος ορισμών είναι το κενό σύνολο. Η τιμή αυτή όμως διαφέρει ανάλογα με το εκάστοτε πρόβλημα.

4 Κενά Ασφαλείας και Taint Analysis

Η *επικύρωση εισόδου* (input validation) αφορά στα προβλήματα που προκαλούνται όταν ο κώδικας «εμπιστεύεται» είσοδο που δεν έχει προηγουμένως ελεγχθεί για την εγκυρότητα της. Στην πιο απλή μορφή, ενώ αναμένεται από τον χρήστη να εισάγει κάποιον αριθμό, αυτός δίνει ως είσοδο μια γραμματοσειρά και το πρόγραμμα αδυνατώντας να συνεχίσει την επεξεργασία τερματίζει. Όταν όμως η είσοδος σχεδιαστεί κατάλληλα από κάποιον κακόβουλο χρήστη, τότε μπορούν να προκληθούν σοβαρότερα προβλήματα όπως υπερχείλιση προσωρινής μνήμης (buffer overflow) και «εμβολιασμός» SQL εντολής (SQL injection). Κενά ασφαλείας αυτής της μορφής συναντώνται κατά κόρον στις εφαρμογές ιστού (8) αλλά δεν περιορίζονται σε αυτές (9).

Για να επωφεληθεί αυτών των κενών ασφαλείας, ένα επιτιθέμενος θα πρέπει να φέρει εις πέρας δύο στόχους:

- Να εισάγει κακόβουλα δεδομένα στην εφαρμογή.
- Να καταλάβει τον έλεγχο της εφαρμογής αξιοποιώντας τα εισαχθέντα δεδομένα.

Με τον όρο taint analysis, αναφερόμαστε στην μεθοδολογία για τον εντοπισμό τέτοιων προβλημάτων. Επομένως, αυτό που μας ενδιαφέρει είναι να εξετάσουμε εάν μη έμπιστα δεδομένα τα οποία λαμβάνονται από τον χρήστη είναι δυνατόν να επηρεάσουν άλλα δεδομένα τα οποία το σύστημα εμπιστεύεται. Για να δοθεί λύση στο πρόβλημα αυτό, είναι προφανές ότι, χρειάζεται να καθοριστούν οι *πηγές* (sources) τέτοιων δεδομένων καθώς και οι *ευαίσθητες* (sensitive) μέθοδοι. Ως πηγή ορίζουμε το σημείο του προγράμματος στο οποίο γίνεται εισαγωγή δεδομένων τα οποία δίνει ο χρήστης ενώ ως ευαίσθητη ορίζουμε μια μέθοδο η οποία πρέπει απαραίτητα να ενεργεί με έμπιστα μόνο δεδομένα. Πλέον, το πρόβλημα μετασχηματίζεται στην *διάδοση* (propagation) των tainted δεδομένων από τις πηγές σε όλη την έκταση του προγράμματος, και η διασφάλιση ότι tainted δεδομένα δεν προσεγγίζουν ευαίσθητες μεθόδους. Για το λόγο αυτό χρησιμοποιείται συχνά και ο όρος *πρόβλημα διάδοσης tainted δεδομένων* (tainted propagation problem).

4.1 Εισαγωγή κακόβουλων δεδομένων

Η προστασία των εφαρμογών από τρωτά σημεία που οφείλονται σε μη επικυρωμένα δεδομένα εισόδου είναι δύσκολη, καθώς οι εφαρμογές μπορούν να αξιοποιήσουν πλήθος διαφορετικών μεθόδων για την λήψη της πληροφορίας από τον χρήστη. Έτσι, θα πρέπει να γίνει συστηματικός έλεγχος όλων των πηγών από όπου ο χρήστης μπορεί να εισάγει δεδομένα, όπως για παράδειγμα τα πεδία στις φόρμες HTTP, οι κεφαλίδες αιτημάτων HTTP και τα cookies. Μερικές από τις πιο συνηθισμένες πρακτικές για εισαγωγή κακόβουλων δεδομένων παρατίθενται στη συνέχεια.

4.1.1 Πλαστογραφία παραμέτρων (parameter tampering)

Η πιο συνηθισμένη μέθοδος για τη λήψη παραμέτρων από εφαρμογές Ιστού είναι οι φόρμες HTML. Όταν καταχωρείται μια φόρμα, οι παράμετροι αποστέλλονται ως μέρος του αιτήματος HTTP. Ένας επιτιθέμενος μπορεί εύκολα να εισάγει κατάλληλα σχεδιασμένα δεδομένα εισόδου μέσω των πεδίων των HTML φορμών.

4.1.2 Πλαστογραφία URL (URL tampering)

Στις περιπτώσεις όπου χρησιμοποιείται η μέθοδος GET για την καταχώριση φορμών HTML, οι παράμετροι των φορμών καθώς και οι τιμές τους εμφανίζονται ως τμήμα του URL. Ένας επιτιθέμενος μπορεί να τροποποιήσει απευθείας το αλφαριθμητικό του URL, εισάγοντας κακόβουλα δεδομένα σε αυτό και στη συνέχεια να προσπελάσει το νέο URL καταχωρώντας τα δεδομένα αυτά στην εφαρμογή.

Για παράδειγμα, ας υποθέσουμε ότι μια τράπεζα παρέχει τη δυνατότητα σε έναν εξουσιοδοτημένο χρήστη να επιλέξει έναν από τους λογαριασμούς του και να κάνει ανάληψη 100 ευρώ από αυτόν. Όταν η φόρμα καταχωρηθεί, στον φυλλομετρητή του χρήστη εμφανίζεται το ακόλουθο URL:

http://www.mybank.com/myacc.php?accno=12345&debit_amount=100.

Στην περίπτωση αυτή και δεδομένου ότι δεν λαμβάνεται κάποια επιπλέον πρόνοια από την εφαρμογή που λαμβάνει την αίτηση, η πρόσβαση στο URL

http://www.mybank.com/myacc.php?accno=12345&debit_amount=-5000

ενδεχομένως να αυξήσει το υπόλοιπο του λογαριασμού.

4.1.3 Διαχείριση κρυφών πεδίων (hidden field manipulation)

Για σκοπούς διατήρησης της κατάστασης, πολλές εφαρμογές Ιστού κάνουν χρήση κρυφών πεδίων. Τα κρυφά πεδία δεν είναι τίποτε άλλο από συνηθισμένα πεδία φορμών τα οποία παραμένουν αόρατα στον τελικό χρήστη. Για παράδειγμα, μια φόρμα παραγγελίας είναι πιθανόν να περιέχει ένα κρυμμένο πεδίο για την αποθήκευση της τιμής των προϊόντων που βρίσκονται στην κάρτα αγορών: `<input type="hidden" name="total_price" value="25.00">`. Σε αντίθεση με τα συνηθισμένα πεδία, τα κρυμμένα πεδία δεν μπορούν να τροποποιηθούν άμεσα καταχωρώντας τιμές στην φόρμα HTML. Ωστόσο, από τη στιγμή που τα πεδία αυτά εξακολουθούν να αποτελούν τμήμα του πηγαίου κώδικα της ιστοσελίδας, ένας κακόβουλος χρήστης είναι σε θέση να αποθηκεύσει την ιστοσελίδα, να τροποποιήσει τις τιμές των κρυμμένων πεδίων και να επαναφορτώσει την ιστοσελίδα. Με τον τρόπο αυτό οι νέες τιμές των κρυφών πεδίων μπορούν να καταχωρηθούν τελικά στην εφαρμογή.

4.1.4 Διαχείριση κεφαλίδας HTTP (HTTP header manipulation)

Οι κεφαλίδες των αιτημάτων HTTP παραμένουν αόρατες συνήθως στον χρήστη ενώ αξιοποιούνται μόνο από τον φυλλομετρητή και τον εξυπηρετητή Ιστού. Εντούτοις, μερικές εφαρμογές αξιοποιούν την πληροφορία των κεφαλίδων, και άρα ένας επιτιθέμενος μπορεί να εισάγει μέσω αυτών κακόβουλα δεδομένα στην εφαρμογή. Αν και ένας συνηθισμένος φυλλομετρητής δεν επιτρέπει την παραποίηση των εξερχόμενων αιτήσεων, υπάρχουν διάφορα εργαλεία που μπορούν να χρησιμοποιηθούν για τον σκοπό αυτό.

4.1.5 Δηλητηρίαση cookie (cookie poisoning)

Οι περισσότερες εφαρμογές Ιστού κάνουν χρήση των cookies για να αποθηκεύσουν διάφορες πληροφορίες στον υπολογιστή του χρήστη (π.χ. αναγνωριστικό και συνθηματικό χρήστη). Με τον τρόπο αυτό η εφαρμογή μπορεί να προσπελάσει πληροφορία που έχει σωθεί σε προηγούμενη εκτέλεση διατηρώντας έτσι ένα είδος κατάστασης. Παρόλο που τα cookies είναι πρακτικά αόρατα στον μέσο χρήστη της εφαρμογής, ένας κακόβουλος χρήστης μπορεί να τα εντοπίσει στον υπολογιστή και να τροποποιήσει κατά βούληση το περιεχόμενό τους.

4.2 Διαχείριση εφαρμογής

Από τη στιγμή που τα δεδομένα εισάγονται στην εφαρμογή, ο επιτιθέμενος μπορεί να χρησιμοποιήσει διάφορες τεχνικές για να τροποποιήσει την λειτουργία της εφαρμογής αξιοποιώντας τα δεδομένα αυτά. Οι βασικότερες τεχνικές παρουσιάζονται στη συνέχεια.

4.2.1 Εμβολιασμός SQL εντολών (SQL injection)

Τα κενά ασφαλείας αυτού του είδους προκαλούνται από μη επικυρωμένα δεδομένα εισόδου τα οποία προωθούνται για εκτέλεση στη βάση δεδομένων της εφαρμογής. Εάν τα δεδομένα αυτά σχεδιαστούν κατάλληλα είναι πιθανόν να εκτελεστούν επιπλέον εντολές SQL με πλήθος συνεπειών, όπως η αποκάλυψη της δομής της βάσης δεδομένων, η εισαγωγή επιπλέον χρηστών στη βάση ή ακόμη και η αποκάλυψη των κωδικών που υπάρχουν αποθηκευμένοι.

Παράδειγμα 4.1

```
HttpServletRequest request = … ;
String userName = request.getParameter("name");
Connection con = … ;
String query = "SELECT * FROM Users" + "WHERE name = '" +
                userName + "'";
con.execute(query);
```

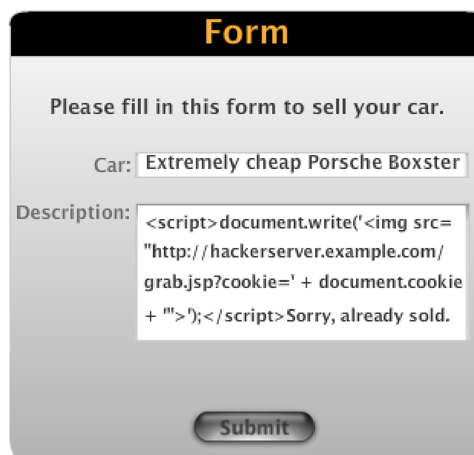
Ο πιο πάνω κώδικας ανακτά το αναγνωριστικό του χρήστη μέσω κλήσης στη μέθοδο `getParameter("name")` και στη συνέχεια το χρησιμοποιεί για να συνθέσει ένα ερώτημα που θα εκτελεστεί στη βάση δεδομένων μέσω της τελευταίας εντολής. Αν και φαινομενικά αθώος, ο παραπάνω κώδικας αφήνει ανοιχτό το ενδεχόμενο κάποιος επιτιθέμενος να αποκτήσει πρόσβαση σε δεδομένα για τα οποία δεν είναι εξουσιοδοτημένος. Αυτό μπορεί να γίνει, για παράδειγμα, εάν εισάγει ως αναγνωριστικό το αλφαριθμητικό: `'OR 1=1; --`. Οι δύο παύλες (--) χρησιμοποιούνται από την SQL για να υποδηλώσουν σχόλια με αποτέλεσμα η εισαγωγή του συγκεκριμένου αλφαριθμητικού να προκαλεί παράκαμψη του τμήματος WHERE της εντολής. Ως αποτέλεσμα, το ερώτημα θα επιστρέψει όλες τις εγγραφές των χρηστών που βρίσκονται καταχωρημένοι στη βάση.

Για να μοντελοποιήσουμε το παραπάνω παράδειγμα ως ένα πρόβλημα taint analysis αρκεί να θεωρήσουμε το σημείο όπου καλείται η getParameter ως πηγή και την

μέθοδο execute ως ευαίσθητη. Αρχικά, η επιστρεφόμενη τιμή της κλήσης getParameter θεωρείται tainted. Στη συνέχεια, η χρήση της μεταβλητής userName για τη δημιουργία της μεταβλητής query έχει ως αποτέλεσμα τον χαρακτηρισμό της τελευταίας ως tainted. Επομένως, η κλήση της ευαίσθητης μεθόδου execute με παράμετρο την tainted μεταβλητή query σηματοδοτεί ένα κενό ασφαλείας.

4.2.2 Επίθεση XSS (cross-site scripting)

Όταν δυναμικά παραγόμενες ιστοσελίδες εμφανίζουν είσοδο που δεν έχει επικυρωθεί κατάλληλα λέμε ότι εμφανίζεται τρωτότητα cross-site scripting. Ένας επιτιθέμενος είναι δυνατόν να ενσωματώσει κακόβουλα σενάρια JavaScript εκμεταλλευόμενος το κενό ασφαλείας. Όταν ο κώδικας αυτός εκτελεστεί στον υπολογιστή του χρήστη που προσπελαύνει την ιστοσελίδα, το σενάριο ενδεχομένως να υποκλέψει τα στοιχεία του χρήστη, να υποκλέψει πληροφορία από τα cookies ή ακόμη να προσθέσει μη επιθυμητό υλικό (π.χ. διαφημίσεις) στην ιστοσελίδα.



Σχήμα 4.1 Παράδειγμα κενού ασφαλείας τύπου cross-site scripting

Παράδειγμα 4.2

Έστω μια ιστοσελίδα που παρέχει σε εξουσιοδοτημένους χρήστες την δυνατότητα να καταχωρούν και να διαβάζουν αγγελίες για πωλήσεις αυτοκινήτων. Η καταχώρηση καινούριας αγγελίας γίνεται μέσω μιας φόρμας HTML όπου ο χρήστης δίνει έναν τίτλο και μια περιγραφή για την αγγελία που επιθυμεί να καταχωρήσει. Εάν η εφαρμογή αυτή εμπεριέχει κενό ασφαλείας cross-site scripting, δηλαδή δεν επικυρώνει τα δεδομένα που εισάγουν οι χρήστες και στη συνέχεια τα παρουσιάζει αυτούσια σε

άλλους χρήστες, ένας επιτιθέμενος μπορεί να εισάγει ένα σενάριο JavaScript στο πεδίο της περιγραφής, όπως φαίνεται στο Σχήμα 4.1. Όταν κάποιος ανυποψίαστος χρήστης επιχειρήσει να προσπελάσει τη συγκεκριμένη αγγελία, το cookie που έχει αποθηκευτεί με τα στοιχεία που τον εξουσιοδοτούν στην εφαρμογή θα αποσταλεί στον εξυπηρέτη του επιτιθέμενου.

4.2.3 Διαμοιρασμός απάντησης HTTP (HTTP response splitting)

Με τον όρο HTTP Response Splitting αναφερόμαστε σε μια γενική τεχνική που χρησιμοποιείται για να πραγματοποιηθούν επιθέσεις web cache poisoning, cross-user defacement και cross-site scripting. Σύμφωνα με την τεχνική αυτή ο επιτιθέμενος εισάγει χαρακτήρες αλλαγής γραμμής (CR - Carriage Return και LF - Line Feed) με αποτέλεσμα να παράγονται δύο απαντήσεις HTTP με διαμοιρασμό της αρχικής απάντησης HTTP.

Παράδειγμα 4.3

Ο ακόλουθος κώδικας χρησιμοποιείται για να διαβάσει το όνομα του συγγραφέα μιας εγγραφής ιστολογίου από ένα αίτημα (request) HTTP και να το θέσει στην κεφαλίδα του cookie μιας απάντησης (response) HTTP.

```
String author = request.getParameter(AUTHOR_PARAM);
...
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
```

Αν υποθέσουμε ότι στο αίτημα HTTP καταχωρείται ένα συνηθισμένο αλφαριθμητικό ως όνομα συγγραφέα (π.χ. “Jane Smith”), η απάντηση HTTP θα έχει την εξής μορφή:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

Όμως, επειδή η τιμή της μεταβλητής δεν επικυρώνεται, η μορφή της απάντησης θα έχει την προαναφερθείσα μορφή μόνο εάν στην τιμή που καταχωρείται στην AUTHOR_PARAM δεν περιέχονται χαρακτήρες αλλαγής γραμμής. Έτσι, εάν καταχωρηθεί κάποιο αλφαριθμητικό όπως το `"Wiley Hacker\r\nHTTP/1.1 200`

OK\r\n...", τότε η αρχική απάντηση HTTP θα διαμοιραστεί σε δύο απαντήσεις της ακόλουθης μορφής:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Wiley Hacker
```

απάντηση HTTP 1

```
HTTP/1.1 200 OK
...
```

απάντηση HTTP 2

Είναι προφανές ότι, η δεύτερη απάντηση είναι πλήρως ελεγχόμενη από τον επιτιθέμενο και μπορεί να δομηθεί με οποιαδήποτε κεφαλίδα και σώμα αυτός επιθυμεί.

4.2.4 Διάσχιση μονοπατιού (path traversal)

Τα κενά ασφαλείας αυτού του τύπου επιτρέπουν στον επιτιθέμενο να προσπελάσει και ενδεχομένως να τροποποιήσει αρχεία πέραν αυτών στα οποία θα έπρεπε να έχει πρόσβαση. Για να το επιτύχει αυτό ο επιτιθέμενος αναζητεί στην εφαρμογή απόλυτους συνδέσμους σε αρχεία που βρίσκονται αποθηκευμένα στον εξυπηρετητή. Τροποποιώντας μεταβλητές που αναφέρονται σε αρχεία και χρησιμοποιώντας την ακολουθία χαρακτήρων dot-dot-slash (../), είναι πιθανόν να επιτύχει προσπέλαση αρχείων και καταλόγων αποθηκευμένων στο σύστημα αρχείων του εξυπηρετητή, όπως για παράδειγμα αρχείων διαμόρφωσης (configuration files) και κρίσιμων αρχείων συστήματος.

Παράδειγμα 4.4

Έστω μια εφαρμογή ιστού που χρησιμοποιεί την μέθοδο GET για προσπέλαση ενός αρχείου με αποτέλεσμα να εμφανίζεται το ακόλουθο URL στον φυλλομετρητή:

http://some_site.com.br/get-files.jsp?file=report.pdf

Στην περίπτωση αυτή, είναι πιθανόν κάποιος να τροποποιήσει την παράμετρο file επιχειρώντας πρόσβαση σε αρχεία πέραν του καταλόγου της εφαρμογής, όπως για παράδειγμα το αρχείο συνθηματικών σε λειτουργικό UNIX:

http://some_site.com.br/../../../../etc/shadow

http://some_site.com.br/get-files?file=/etc/passwd

4.2.5 Εμβολιασμός εντολής (command injection)

Στην περίπτωση αυτή στόχος είναι η εισαγωγή και εκτέλεση εντολών συστήματος που επιλέγονται από τον επιτιθέμενο. Επειδή οι εντολές εκτελούνται μέσω

της εφαρμογής, ο επιτιθέμενος αποκτά αυτόματα τα ίδια δικαιώματα με αυτά τα οποία έχει η εφαρμογή. Τέτοια τρωτά σημεία εμφανίζονται σε εφαρμογές όπου δεν πραγματοποιείται ορθή επικύρωση των δεδομένων που λαμβάνονται από το εξωτερικό περιβάλλον της εφαρμογής και επιπλέον στον κώδικα χρησιμοποιούνται συναρτήσεις για εκτέλεση εντολών κελύφους ή δυναμικής φόρτωσης βιβλιοθηκών.

Παράδειγμα 4.5

Λόγω ανεπαρκούς επικύρωσης εισόδου σε ένα συμπληρωματικό σενάριο κελύφους που περιλαμβανόταν σε ορισμένες εκδόσεις του φυλλομετρητή ιστού Firefox, υπήρχε το ενδεχόμενο κάποιος μη εξουσιοδοτημένος επιτιθέμενος να εκτελέσει αυθαίρετες εντολές (10). Το σενάριο αυτό παρείχε σε εξωτερικές εφαρμογές τη δυνατότητα να «καλέσουν» τον Firefox δίνοντας σαν παράμετρο κάποιο URL. Έτσι, ενώ η εντολή

```
firefox http://local\`find\`host
```

θα έπρεπε να αναγνωρίζεται ως εσφαλμένη, στην πραγματικότητα, γινόταν κλήση της εντολής κελύφους find.

Αν και φαινομενικά διαφορετικές, όλες οι παραπάνω περιπτώσεις είναι εφικτές λόγω μη ορθής επικύρωσης των δεδομένων που λαμβάνονται από το εξωτερικό περιβάλλον των εφαρμογών και μπορούν να μοντελοποιηθούν ως προβλήματα taint analysis.

5 FindBugs

Η αυτοματοποιημένη ανάλυση προγραμμάτων για τον εντοπισμό ελαττωμάτων στον κώδικα είναι ένα ενεργό πεδίο έρευνας στην επιστημονική κοινότητα. Με το πέρας των χρόνων έχει αναπτυχθεί πλήθος τεχνικών προς αυτή την κατεύθυνση. Οι παραδοσιακές τεχνικές όμως, στηρίζονται συνήθως σε φορμαλιστικές μεθόδους και πολύπλοκους υπολογισμούς με αποτέλεσμα να υστερούν σε επεκτασιμότητα και απόδοση. Το FindBugs δημιουργήθηκε με σκοπό να υπερκεράσει αυτά τα προβλήματα.

5.1 Αρχιτεκτονική

Σε αντίθεση με τα προγενέστερα εργαλεία, το FindBugs στηρίζεται σε μια απλή αλλά ταυτόχρονα ισχυρή τεχνική για να διεξάγει στατική ανάλυση. Η βάση του συστήματος είναι ένα σύνολο *προτύπων ελαττωμάτων* (bug patterns), δηλαδή, ιδιωμάτων κώδικα που πιθανόν να αποτελούν λάθος. Ένας *ανιχνευτής* (detector) υλοποιείται με σκοπό να εντοπίζει ένα σύνολο τέτοιων προτύπων και να αναφέρει τις θέσεις στον κώδικα όπου αυτά εμφανίζονται. Σύμφωνα με τους κατασκευαστές του, τέτοιοι ανιχνευτές είναι εύκολο να υλοποιηθούν και ταυτόχρονα παρουσιάζουν ικανοποιητική αποτελεσματικότητα στην ανακάλυψη αληθινών ελαττωμάτων (7).

Από άποψη σχεδίασης, το FindBugs στηρίζεται στο πρότυπο επισκέπτη (visitor pattern). Η αρχιτεκτονική αυτή επιτρέπει την ενσωμάτωση προσθέτων (plugin) τα οποία υλοποιούν επιπλέον λειτουργικότητα. Στη συγκεκριμένη περίπτωση, ανεξάρτητοι κατασκευαστές μπορούν να υλοποιήσουν ανιχνευτές και να τους διαθέσουν ως πρόσθετα στους χρήστες του εργαλείου (11). Ο ρόλος της αρχιτεκτονικής αυτής θα γίνει σαφέστερος στη συνέχεια, όπου θα εξετάσουμε τον κύκλο λειτουργίας του FindBugs.

Σε τεχνικό επίπεδο, το FindBugs χρησιμοποιεί διάφορες έτοιμες βιβλιοθήκες αλλά και δικό του κώδικα, ώστε να αναπαραστήσει και να διαχειριστεί τα προγράμματα τα οποία θα αναλύσει. Καθώς η ανάλυση εφαρμόζεται σε επίπεδο ενδιάμεσου κώδικα (bytecode) Java, μια από τις σημαντικότερες βιβλιοθήκες που χρησιμοποιούνται είναι η BCEL (Byte Code Engineering Library). Η βιβλιοθήκη αυτή αναπαριστά τα στοιχεία της Java, όπως οι κλάσεις, οι μέθοδοι και οι μεταβλητές ενώ ταυτόχρονα παρέχει μια εύχρηστη διασύνδεση (API) για αλληλεπίδραση με τα στοιχεία αυτά (12).

Ο κύκλος λειτουργίας του FindBugs περιγράφεται από τα ακόλουθα βήματα:

1. Δημιουργία αντικειμένου AnalysisContext: το αντικείμενο αυτό χρησιμοποιείται σαν αποθήκη πληροφοριών που προκύπτουν κατά τη διάρκεια της ανάλυσης.
2. Δημιουργία και διαμόρφωση βάσεων δεδομένων εκπαίδευσης (training databases): σε κάθε τέτοια βάση δεδομένων καταγράφονται τα αποτελέσματα της ανάλυσης μιας συγκεκριμένης μεθόδου ή πεδίου για να μπορούν να επαναχρησιμοποιηθούν σε επόμενο κύκλο ανάλυσης. Αποτελούν έναν εύκολο τρόπο για πραγματοποίηση ανάλυσης μεταξύ των μεθόδων (interprocedural analysis).
3. Δημιουργία πλάνου εκτέλεσης (execution plan): το πλάνο εκτέλεσης οργανώνει τους ανιχνευτές σε περάσματα (passes) και καθορίζει τη σειρά εκτέλεσης των ανιχνευτών σε κάθε πέρασμα. Οι νεότερες εκδόσεις του εργαλείου επιτρέπουν τον καθορισμό περιορισμών σχετικά με τη σειρά εκτέλεσης των ανιχνευτών.
4. Εύρεση των προς ανάλυση κλάσεων: πραγματοποιείται αναζήτηση σε όλα τα αρχεία που δίνονται ως είσοδος στο εργαλείο. Για κάθε κλάση που εντοπίζεται δημιουργείται ένα αντικείμενο BCEL JavaClass με αποτέλεσμα όλες οι κλάσεις να βρίσκονται στη μνήμη κατά τη διάρκεια της ανάλυσης.
5. Εκτέλεση ανάλυσης: η ανάλυση εκτελείται σύμφωνα με τον ψευδοκώδικα που παρουσιάζεται στο Σχήμα 5.1.
6. Αναφορά ευρημάτων ανάλυσης: με την ολοκλήρωση της ανάλυσης όλα τα πιθανά σφάλματα που έχουν εντοπιστεί βρίσκονται αποθηκευμένα στο αντικείμενο BugReporter και μπορούν να αναφερθούν μέσω της μεθόδου reportQueuedErrors().

```
for each analysis pass in the execution plan do
    for each application class do
        for each detector in the analysis pass do
            request ClassContext for the class from the AnalysisContext
            apply the detector to the ClassContext
        end for
    end for
end for
```

Σχήμα 5.1 Ψευδοκώδικας εκτέλεσης ανάλυσης στο FindBugs

5.2 Ανιχνευτές

Οι ανιχνευτές που υλοποιούνται στο FindBugs μπορούν να χωριστούν σε δύο βασικές κατηγορίες, ανάλογα με την ανάλυση που επιτελούν:

- Visitor based
- Control Flow Graph (CFG) based

Θεωρητικά, το FindBugs δεν θέτει οποιουσδήποτε περιορισμούς σχετικά με την υλοποίηση των ανιχνευτών: μπορεί και θα πρέπει να χρησιμοποιηθεί οποιαδήποτε τεχνική ανάλυσης επιφέρει τα επιθυμητά αποτελέσματα. Εντούτοις, στην πράξη οι CFG based ανιχνευτές εκτελούν συνήθως πιο περίπλοκη ανάλυση σε σχέση με τους visitor based ανιχνευτές.

5.2.1 Visitor based ανιχνευτές

Στην οικογένεια αυτή ανήκουν οι ανιχνευτές που πραγματοποιούν γραμμική σάρωση του ενδιάμεσου κώδικα, αναζητώντας «ύποπτα» πρότυπα. Η λειτουργία των ανιχνευτών της συγκεκριμένης τάξης είναι στην ουσία η μοντελοποίηση ενός πεπερασμένου ντετερμινιστικού αυτόματου, του οποίου η τελική κατάσταση υποδηλώνει την παρουσία ελαττώματος. Βέβαια, είναι πιθανόν να υπάρχουν παραπάνω από μία τελικές καταστάσεις και συνεπώς ο ανιχνευτής να αναγνωρίζει πολλά διαφορετικά σφάλματα.

Γενικά, αυτοί οι ανιχνευτές αποτελούν υποκλάσεις της BytecodeScanningVisitor. Η βασική κλάση διασχίζει το αρχείο της προς ανάλυση κλάσης, αποκωδικοποιώντας όλες τις συμβολικές πληροφορίες σχετικά με τα στοιχεία της κλάσης (πεδία, μεθόδους κτλ). Για κάθε τέτοιο στοιχείο που εντοπίζεται, καλείται η αντίστοιχη μέθοδος της βασικής κλάσης που είναι υπεύθυνη για την ανάλυση στοιχείων αυτού του τύπου. Έτσι, για να συγκεκριμενοποιηθεί η εργασία ενός ανιχνευτή, αρκεί να γίνει υπέρβαση (override) αυτών των μεθόδων.

Χαρακτηριστικό της κατηγορίας αυτής είναι, εκτός της απλότητας των ανιχνευτών, η ταχύτητα εκτέλεσης. Αυτό οφείλεται στο γεγονός ότι δεν απαιτείται η κατασκευή πολύπλοκων δομών, όπως στην περίπτωση των CFG based ανιχνευτών. Ταυτόχρονα όμως, οι visitor based ανιχνευτές υστερούν στο εύρος σφαλμάτων που

μπορούν να εντοπίσουν. Για παράδειγμα, δεν είναι δυνατόν να ανακαλύψουν σφάλματα με πολύπλοκη δομή, όπως σφάλματα που έχουν σχέση με νήματα (threads).

for each method in the class **do**
request a CFG for the method from the ClassContext
request one or more analysis objects on the method from the ClassContext
for each location in the method **do**
get the dataflow facts at that location
inspect the dataflow facts
if a dataflow fact inticated an error **then**
report a worning
end if
end for
end for

Σχήμα 5.2 Ψευδοκώδικας λειτουργίας CFG based ανιχνευτών

5.2.2 CFG based ανιχνευτές

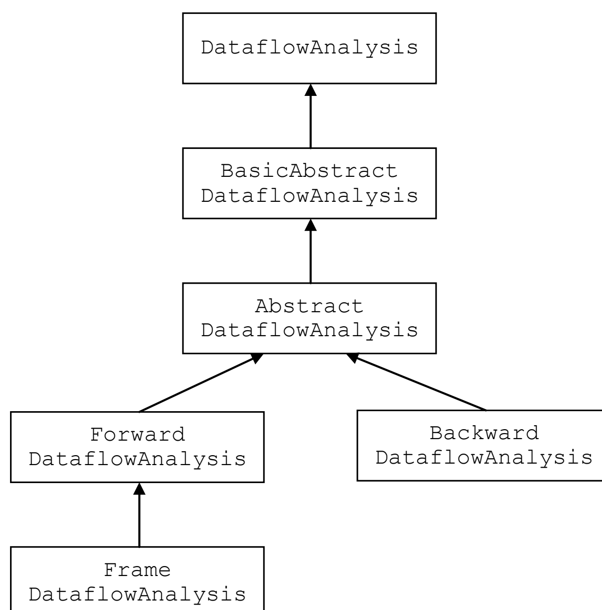
Οι ανιχνευτές αυτής της οικογένειας χρησιμοποιούν τη δομή του γράφου ροής ελέγχου για εκτέλεση πιο περίπλοκων μορφών ανάλυσης σε σχέση με τους visitor based ανιχνευτές. Σε αντίθεση με τους visitor based ανιχνευτές που κληρονομούν από την κλάση BytecodeScanningVisitor, οι CFG based ανιχνευτές υλοποιούν συνήθως απευθείας τη διασύνδεση Detector.

Η βασική ιδέα στην περίπτωση αυτή είναι να επισκεφτούμε με τη σειρά τις μεθόδους της προς ανάλυση κλάσης και να ζητάμε για κάθε μέθοδο κάποια αντικείμενα ανάλυσης. Ένα αντικείμενο ανάλυσης είναι το τελικό αποτέλεσμα μιας συγκεκριμένης ανάλυσης, δηλαδή, ένα αντικείμενο που καταγράφει γεγονότα[8] (facts) σχετικά με μια μέθοδο για μια συγκεκριμένη μορφή ανάλυσης. Στη συνέχεια, ο ανιχνευτής διασχίζει τις θέσεις (locations) του γράφου ροής ελέγχου και εξετάζει τα γεγονότα που αντιστοιχούν σε κάθε θέση για να δει εάν προκύπτει κάποιο ελάττωμα. Ο ψευδοκώδικας που περιγράφει την παραπάνω διαδικασία παρουσιάζεται στο Σχήμα 5.2.

---

[8] Ο όρος γεγονός (fact) χρησιμοποιείται συνώνυμα ως προς τον όρο τιμή ροής δεδομένων (data flow value) στο κεφάλαιο Ανάλυση Ροής Δεδομένων.

Οι ανιχνευτές αυτοί είναι χρήσιμοι για οποιοδήποτε είδος ανάλυσης μπορεί να μοντελοποιηθεί ως πρόβλημα ανάλυσης ροής δεδομένων. Αν και καλύπτουν ένα ευρύτερο φάσμα προβλημάτων, είναι πολύ πιο αργοί σε εκτέλεση συγκριτικά με τους visitor based ανιχνευτές.



Σχήμα 5.3 Βασικές κλάσεις για ανάλυση ροής δεδομένων

5.3 Ανάλυση ροής δεδομένων στο FindBugs

Το FindBugs παρέχει το απαραίτητο αλγοριθμικό πλαίσιο για την εκτέλεση ανάλυσης ροής δεδομένων. Η βασική κλάση για το σκοπό αυτό είναι η Dataflow η οποία παραμετροποιείται με την κλάση DataflowAnalysis που αναπαριστά τον τύπο της ανάλυσης που πρόκειται να πραγματοποιηθεί. Με τον τρόπο αυτό, παρέχεται η δυνατότητα ενσωμάτωσης επιπλέον τύπων ανάλυσης στο εργαλείο.

Στο Σχήμα 5.3 παρουσιάζονται οι βασικές κλάσεις που παρέχει το FindBugs για την υλοποίηση ανάλυσης ροής δεδομένων. Η διασύνδεση DataflowAnalysis προσδιορίζει την απαιτούμενη λειτουργικότητα που πρέπει να παρέχει οποιαδήποτε κλάση προορίζεται για εκτέλεση ανάλυσης ροής δεδομένων. Οι κλάσεις BasicAbstractDataflowAnalysis και AbstractDataflowAnalysis παρέχουν μέρος της απαιτούμενης λειτουργικότητας και μπορούν να χρησιμοποιηθούν για υλοποίηση ανάλυσης σε επίπεδο βασικών μπλοκ και εντολών, αντίστοιχα. Οι κλάσεις ForwardDataFlowAnalysis και BackwardDataflowAnalysis, όπως δηλώνουν και τα

ονόματα, είναι αφηρημένες κλάσεις που μπορούν να επεκταθούν για εκτέλεση ανάλυσης μιας εκ των δύο δυνατών κατευθύνσεων.

Ιδιαίτερο ενδιαφέρον παρουσιάζει η κλάση FrameDataflowAnalysis που αποτελεί τη βασική κλάση για την εκτέλεση ανάλυσης ροής δεδομένων όπου για γεγονότα χρησιμοποιούνται *πλαίσια* (frames). Η κλάση Frame αναπαριστά ένα *πλαίσιο σωρού* (stack frame) της Java και αποτελείται από *εσοχές* (slots) στις οποίες αποθηκεύονται συμβολικές τιμές που αντιστοιχούν στις τοπικές μεταβλητές και στις τιμές που βρίσκονται στον *σωρό τελεστέων* (operand stack).

Για παράδειγμα, το FindBugs παρέχει την κλάση IsNullValueAnalysis που προσδιορίζει εάν οι εσοχές των πλαισίων περιέχουν κενές (null) τιμές και χρησιμοποιείται από τον ανιχνευτή FindNullDeref για τον εντοπισμό αναφορών σε κενούς δείκτες. Για να το επιτύχει αυτό, η κλάση IsNullValueAnalysis επεκτείνει την FrameDataflowAnalysis και παραμετροποιείται με τις κλάσεις IsNullValue και IsNullValueFrame. Η κλάση IsNullValue αναπαριστά την συμβολική τιμή μιας μόνο εσοχής που μπορεί να είναι κενή, μη-κενή ή απροσδιόριστη. Με τη σειρά της η κλάση αυτή χρησιμοποιείται για το σχηματισμό της κλάσης IsNullValueFrame που επεκτείνει επιπλέον την κλάση Frame και αναπαριστά ένα ολόκληρο πλαίσιο σωρού. Η κλάση IsNullValueFrame μπορεί πλέον να χρησιμοποιηθεί για την αναπαράσταση γεγονότων στην ανάλυση ροής δεδομένων.

Συνοψίζοντας, για την υλοποίηση ανιχνευτή που στηρίζεται σε ανάλυση ροής δεδομένων απαιτούνται τα ακόλουθα βήματα:

1. Δημιουργία γεγονότος: κλάση για την καταγραφή πληροφοριών σχετικά με τα χαρακτηριστικά που μας ενδιαφέρουν (π.χ. IsNullValueFrame).
2. Δημιουργία ανάλυσης: κλάση που ορίζει την κατεύθυνση της ανάλυσης και μεθόδους για την υλοποίηση των συναρτήσεων μετάβασης της συγκεκριμένης ανάλυσης. Υλοποιεί άμεσα ή έμμεσα τη διασύνδεση DataflowAnalysis (π.χ. IsNullValueAnalysis)
3. Δημιουργία αναλυτή: κλάση που επεκτείνει άμεσα ή έμμεσα την DataFlow και μπορεί να χρησιμοποιηθεί για την ανάκτηση γεγονότων πριν και μετά από κάθε εντολή. (π.χ. IsNullValueDataflow)
4. Δημιουργία ανιχνευτή: CFG based ανιχνευτής που υλοποιεί τη διασύνδεση Detector (π.χ. FindNullDeref).

6 *Taint analysis σε multi-applet JavaCard εφαρμογή*

Για την εφαρμογή των όσων έχουν μέχρι τώρα αναφερθεί, παρουσιάζουμε στην ενότητα αυτή ένα παράδειγμα, όπου ένας ανιχνευτής αξιοποιείται για τον έλεγχο μιας multi-applet JavaCard εφαρμογής. Ο ανιχνευτής εφαρμόζει taint analysis και βρίσκεται υλοποιημένος ως πρόσθετο του εργαλείου FindBugs (12) ενώ η JavaCard εφαρμογή υλοποιήθηκε στα πλαίσια προηγούμενης πτυχιακής εργασίας (13) (14).

6.1 Ανιχνευτής taint analysis

Τα βασικά μέρη στην αρχιτεκτονική του ανιχνευτή είναι οι βάσεις δεδομένων, ο αναλυτής και τρείς επιμέρους ανιχνευτές. Οι βάσεις δεδομένων χρησιμοποιούνται για την αποθήκευση ενδιάμεσων αποτελεσμάτων που προκύπτουν κατά τη διαδικασία της ανάλυσης. Ο αναλυτής υλοποιεί taint analysis στηριζόμενος στην λειτουργικότητα που παρέχει το FindBugs για εκτέλεση ανάλυσης ροής δεδομένων. Οι τρείς ανιχνευτές είναι οι ακόλουθοι:

1. MethodAnnocationDetector: εντοπίζει τα επισημειωμένα στοιχεία και αποθηκεύει τις κατάλληλες πληροφορίες στις βάσεις δεδομένων.
2. TaintnessDetector: εφαρμογή του αναλυτή για διάδοση των tainted δεδομένων.
3. TaintedInjection: ανίχνευση των σημείων όπου tainted δεδομένα προσεγγίζουν ευαίσθητες μεθόδους.

Οι τρεις ανιχνευτές εκτελούνται με την πιο πάνω σειρά, όπως ορίζει και ο κώδικας του αρχείου findbugs.xml (Σχήμα 6.1). Με τον τρόπο αυτό αξιοποιούνται οι δυνατότητες που παρέχει το FindBugs για πραγματοποίηση καθολικής ανάλυσης καθώς, γίνεται εφικτή η αποθήκευση ενδιάμεσων αποτελεσμάτων που μπορούν στη συνέχεια να χρησιμοποιηθούν σε επόμενο πέρασμα.

```
<OrderingConstraints>
   <SplitPass>
      <Earlier class="...MethodAnnotationDetector"/>
      <Later class="...TaintnessDetector"/>
   </SplitPass>
   <SplitPass>
      <Earlier class="...TaintnessDetector"/>
      <Later class="...TaintedInjection"/>
   </SplitPass>
</OrderingConstraints>
```

Σχήμα 6.1 Κώδικας XML για καθορισμό προτεραιότητας μεταξύ των ανιχνευτών

6.1.1 Αναλυτής

Σκοπός του αναλυτή είναι να αξιοποιήσει τη λειτουργικότητα που παρέχει το FindBugs για εκτέλεση ανάλυσης ροής δεδομένων και να την επεκτείνει για την πραγματοποίηση taint analysis. Παρόλο που υλοποιείται τόσο forward όσο και backward ανάλυση στη συνέχεια θα ασχοληθούμε μόνο με την πρώτη καθώς η λογική είναι παρόμοια και για την περίπτωση της δεύτερης.

Όπως έχουμε αναφέρει στο κεφάλαιο 4, για να πραγματοποιήσουμε taint analysis είναι απαραίτητο να ορίσουμε τις πηγές και τις ευαίσθητες μεθόδους. Ο συγκεκριμένος ανιχνευτής αξιοποιεί για το σκοπό αυτό ένα χαρακτηριστικό της Java που καλείται επισημείωση (annotation)[9]. Οι επισημειώσεις που υποστηρίζονται είναι οι ακόλουθες:

- @Sensitive: μπορεί να εφαρμοστεί σε παραμέτρους μεθόδων για να υποδείξει ότι οι συγκεκριμένες παράμετροι είναι ευαίσθητες.
- @TaintedResult: εφαρμόζεται σε μεθόδους που επιστρέφουν tainted δεδομένα.
- @TaintnessValidator: μπορεί να εφαρμοστεί σε μέθοδο για να υποδηλώσει ότι η συγκεκριμένη μέθοδος επικυρώνει τα δεδομένα που δέχεται ως παράμετρο.

Με βάση αυτά, είναι προφανής η αντιστοιχία με την ορολογία που ορίσαμε στο κεφάλαιο 4. Συγκεκριμένα, το σημείο κλήσης μιας μεθόδου στην οποία εφαρμόζεται η επισημείωση @TaintedResult είναι αντίστοιχο της πηγής ενώ, ευαίσθητη είναι η μέθοδος της οποίας παράμετρος επισημειώνεται με @Sensitive. Επομένως, η ανάλυση θα πρέπει σε πρώτο στάδιο να μεταδίδει τα tainted δεδομένα και στη συνέχεια να αναζητεί περιπτώσεις όπου tainted δεδομένα προσεγγίζουν @Sensitive παραμέτρους.

[9] Περισσότερα σχετικά με τις επισημειώσεις στο Παράρτημα.

Επιπλέον, η επισημείωση @TaintnessValidator χρησιμοποιείται για να μετατρέψει tainted δεδομένα σε untainted. Αυτό πρακτικά σημαίνει ότι, τυχόν tainted δεδομένα που δίνονται ως ορίσματα σε @TaintnessValidator μεθόδους θα πρέπει να θεωρούνται ασφαλή έπειτα από το σημείο αυτό.

Η ανάλυση που υλοποιείται βασίζεται στη χρήση πλαισίων ως γεγονότα της ανάλυσης ροής δεδομένων (frame-based). Οι βασικές κλάσεις για το σκοπό αυτό είναι οι TaintValue, TaintValueFrame, TaintValueFrameModelingVisitor, TaintDataflow, και TaintAnalysis.

Από τα όσα αναφέρθηκαν στην ενότητα 3.3, είναι προφανές ότι μπορούμε να σχηματίσουμε ένα ημιπλέγμα το οποίο θα αποτελείται από το σύνολο V = {tainted, untainted} και έναν δυαδικό τελεστή συνένωσης ($\wedge$). Αν επιπλέον θεωρήσουμε τα στοιχεία κορυφής ($\top$) και βάσης ($\bot$), μπορούμε να ορίσουμε τον τελεστή συνένωσης ως εξής:

1. untainted $\wedge$ tainted = tainted
2. untainted $\wedge$ $\top$ = untainted
3. tainted $\wedge$ $\top$ = tainted
4. untainted $\wedge$ $\bot = \bot$
5. tainted $\wedge$ $\bot = \bot$

Μεταξύ των τιμών του συνόλου ορίζεται η μερική διάταξη untainted < tainted. Η κλάση TaintValue αναπαριστά μια τιμή του συνόλου {tainted, untainted} και εκτός από τη βασική λειτουργικότητα παρέχει επιπλέον μεθόδους για τη συνένωση δύο τιμών.

Η κλάση TaintValueFrame επεκτείνει την Frame και αναπαριστά ένα πλαίσιο το οποίο μπορεί να χρησιμοποιηθεί ως γεγονός (fact). Η TaintValueFrame παραμετροποιείται με την κλάση TaintValue που αντιπροσωπεύει τη συμβολική τιμή η οποία θα αποθηκεύεται στις εσοχές (slots) του πλαισίου. Επιπλέον, η TaintValueFrameModelingVisitor χρησιμοποιεί τις προαναφερθείσες κλάσεις και επεκτείνει την AbstractFrameModelingVisitor για να μοντελοποιήσει την επίδραση των διαφόρων εντολών στα περιεχόμενα των πλαισίων. Τέλος, η κλάση TaintAnalysis επεκτείνει την FrameDataflowAnalysis και υλοποιεί τις συναρτήσεις μετάβασης και

συνένωσης που απαιτούνται για την εφαρμογή του αλγοριθμικού πλαισίου της ανάλυσης ροής δεδομένων.

6.1.2 Βάσεις Δεδομένων

Για την δημιουργία των απαραίτητων βάσεων δεδομένων αξιοποιείται ο μηχανισμός που παρέχει το FindBugs για τον σκοπό αυτό. Συγκεκριμένα, όλες οι βάσεις δεδομένων που κατασκευάζονται, επεκτείνουν είτε την κλάση `MethodPropertyDatabase` ή την κλάση `AnnotationDatabase`. Η πρώτη αποσκοπεί στην καταγραφή ιδιοτήτων που σχετίζονται με μεθόδους ενώ η δεύτερη στην αποθήκευση επισημειωμένων στοιχείων για ένα συγκεκριμένο τύπο επισημείωσης.

Συνολικά κατασκευάζονται έξι βάσεις δεδομένων που αντιστοιχούν στις ακόλουθες κλάσεις:

1. `ParameterTaintnessPropertyDatabase`: καταγράφει ιδιότητες τύπου `ParameterTaintnessProperty`. Κάθε τέτοια ιδιότητα αναπαριστά τις παραμέτρους μιας συγκεκριμένης μεθόδου. Συγκεκριμένα περιέχει δύο διανύσματα των 32 bit όπου το ένα καταγράφει τις θέσεις των tainted παραμέτρων ενώ το άλλο τις θέσεις των παραμέτρων που πρέπει να είναι untainted.
2. `KeyIndicatorPropertyDatabase`: καταγράφει ιδιότητες τύπου `KeyIndicatorProperty`. Η ιδιότητα αυτή εκφράζει το γεγονός ότι μια συγκεκριμένη μέθοδος έχει επισημειωθεί ως επαληθευτής (validator).
3. `IsParameterTaintedPropertyDatabase`: καταγράφει ιδιότητες τύπου `IsParameterTaintedProperty`. Κάθε τέτοια ιδιότητα περιέχει πληροφορία για τις παραμέτρους μιας συγκεκριμένης μεθόδου που πρέπει να είναι untainted (ευαίσθητη μέθοδος).
4. `IsResultTaintedPropertyDatabase`: καταγράφει ιδιότητες τύπου `IsResultTaintedProperty`. Η ιδιότητα αυτή εκφράζει εάν η επιστρεφόμενη τιμή μιας συγκεκριμένης μεθόδου είναι tainted ή όχι.
5. `KeyIndicatorAnnotationDatabase`: χρησιμοποιείται για την αποθήκευση και ανάκτηση των στοιχείων που είναι επισημειωμένα ως `@TaintnessValidator`.
6. `TaintAnnotationDatabase`: χρησιμοποιείται για την αποθήκευση και ανάκτηση των στοιχείων που είναι επισημειωμένα ως @Sensitive ή @ReturnTainted.

6.1.3 Ανιχνευτές

Ο πρώτος ανιχνευτής που τίθεται σε λειτουργία είναι ο `MethodAnnotationDetector`. Στόχος του είναι να εντοπίσει τα επισημειωμένα στοιχεία που υπάρχουν στον κώδικα και να καταγράψει τις κατάλληλες πληροφορίες στις βάσεις δεδομένων. Ο ψευδοκώδικας της λειτουργίας του είναι ο εξής:

```
1.  initialize databases
2.  get AnalysisCache object
3.  get method’s TypeDataflow from AnalysisCache
4.  create an isResultTainter property
5.  if method is annotated as @TaintedResult then
6.          set property to true
7.  else
8.          set property to false
9.  end if
10. associate property with current method and save in isResultTainted db
11. if method is annotated as @TaintnessValidator then
12.         create a keyIndicator property
13.         set property to validator
14.         associate property with current method and save in keyIndicator  db
15. end if
16. create a parameterTaintness property
17. for each parameter in current method do
18.         if parameter is annotated as @Sensitive
19.                 set parameter’s untaint bit in property as true
20.         else
21.                 set parameter’s taint bit in property as true
22.         end if
23. end for
24. associate property with current method and save in parameterTaintness db
25. for each basic block in CFG do
26.         for each instruction in block do
27.                 if instruction instance of invoke virtual then
28.                         get location of instruction
29.                         get TypeFrame (fact) at current location
30.                         get all possible invoked methods
31.                         for each resolved method do
32.                                 create an isResultTainted property
33.                                 if method is annotated as @TaintedResult then
34.                                         set property to true
35.                                 else
36.                                         set property to false
37.                                 end if
38.                                 associate property with current method and save in db
39.                                 if method is annotated as @TaintnessValidator then
40.                                         create a keyIndicator property
```

```
41.                         set property to validator
42.                         associate property with current method and save in db
43.                     end if
44.                     create a parameterTaintness property
45.                     for each parameter in current method do
46.                         if parameter is annotated as @Sensitive
47.                             set parameter's untaint bit in property as true
48.                         else
49.                             set parameter's taint bit in property as true
50.                         end if
51.                     end for
52.                 associate property with current method and save in db
53.                 end for
54.             end if
55.         end for
56. end for
```

Στη συνέχεια, καλείται ο `TaintnessDetector` μέσω του οποίου εφαρμόζεται backward taint analysis (γραμμή 6) που έχει ως αποτέλεσμα την διάδοση των tainted δεδομένων. Ο ψευδοκώδικας του αναλυτή παρατίθεται ακολούθως:

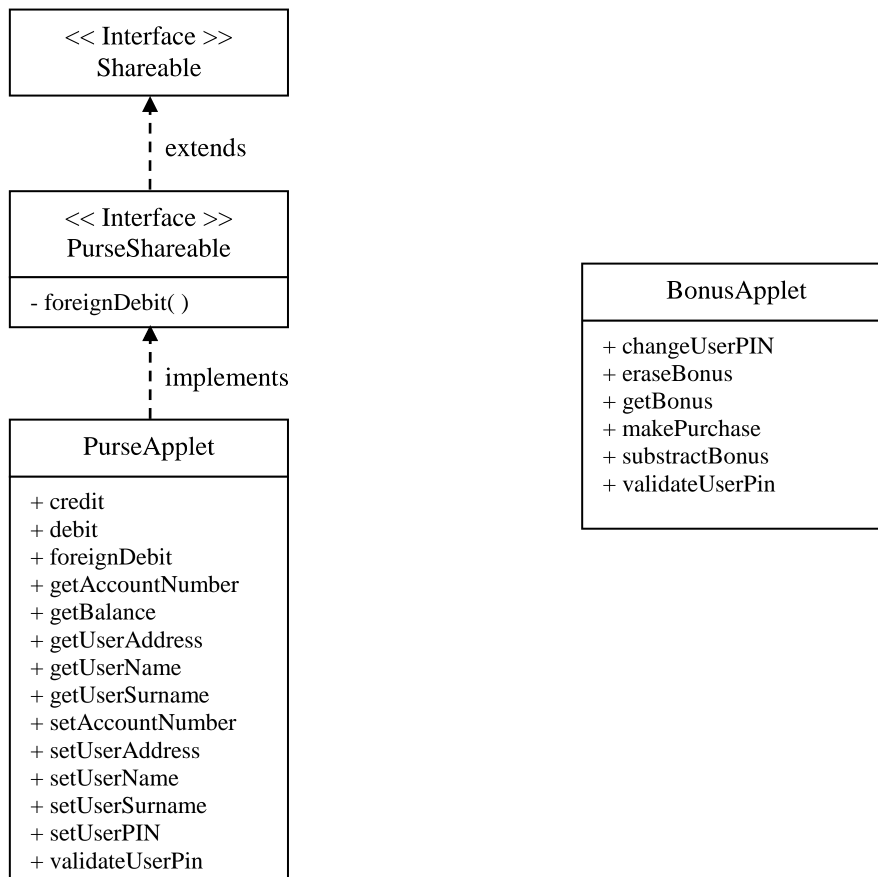
```
1.  retrieve TaintnessPropertyDatabase from AnalysisCache
2.  for each method in the class do
3.      if method is abstract || method is native || method !has code then
4.          continue
5.      end if
6.      get method's TaintnessDataflow (backward) from ClassContext
7.      get TaintnessFrame (fact) at the end of CFG's entry node
8.      get method's number of arguments
9.      create a new ParameterTaintnessProperty
10.     for each argument do
11.         get argument's TaintnessValue
12.         if argument is tainted then
13.             set argument's bit in property to true
14.         else
15.             set argument's bit in property to false
16.         end if
17.         if source-sink collision exists then
18.             save the source line in the property
19.         end if
20.     end for
21.     retrieve method's old property from the database
22.     merge the new property with the old one
```

23. save the result as method's property in the database
24. **end for**

Τέλος, ενεργοποιείται ο αναλυτής `TaintedInjection` ο οποίος αναζητεί περιπτώσεις όπου tainted δεδομένα προσεγγίζουν παραμέτρους μεθόδων που έχουν επισημειωθεί ως @Sensitive. Αναλυτικά η λειτουργία του φαίνεται στον παρακάτω ψευδοκώδικα:

1. **for each** method in the class **do**
2. **if** method is abstract || method is native || method !has code **then**
3. continue
4. **end if**
5. create a return locations list
6. initialize database for isResultTainted property
7. initialize database for parameterTaintness property
8. get method's TypeDataflow from ClassContext
9. get method's TaintDataflow (forward) from ClassContext
10. request a CFG for the method
11. **for each** basic block in the CFG **do**
12. **for each** instruction in the basic block **do**
13. **if** instruction instance of invoke instruction **then**
14. get location of instruction
15. get TypeFrame (fact) at current location
16. get all possible invoked methods
17. **for each** resolved method **do**
18. get TaintValueFrame (fact) at current location
19. get method's parameterTaintness property from db
20. **for each** argument **do**
21. get argument as TaintValue
22. **if** property specifies argument to be untainted **&&** argument is tainted **then**
23. report a bug
24. **end if**
25. **end for**
26. **end for**
27. **end if**
28. **end for**
29. get last instruction's opcode
30. **if** opcode is kind of return **then**
31. add instruction's location in the return locations list
32. **end if**

33. **end for**
34. **if** method's return type != void **then**
35. get method's isResultTainted property from db
36. get property as TaintValue (result)
37. **for each** location in return locations list **do**
38. get TaintValueFrame (fact) at current location
39. extract TaintValue from fact
40. meet both TaintValue's and save in result
41. **end for**
42. create a new isResultTainted property for the result
43. associate property with current method and save in db
44. **end if**
45. **end for**



Σχήμα 6.2 Μέθοδοι στα applet της εφαρμογής

6.2 Στατικός έλεγχος JavaCard εφαρμογής

Η εφαρμογή, όπως παρουσιάζεται στο Σχήμα 6.2, αποτελείται από δύο applets: ένα ηλεκτρονικό πορτοφόλι (purse) που αλληλεπιδρά με ένα δεύτερο applet το οποίο αποθηκεύει μονάδες επιβράβευσης (bonus). Δηλαδή, το δεύτερο applet είναι υπεύθυνο για την αποθήκευση των μονάδων bonus που μπορεί κανείς να κερδίσει ως επιβράβευση των συναλλαγών που πραγματοποιεί. Το ηλεκτρονικό πορτοφόλι προσθέτει και αφαιρεί μονάδες από ψηφιακά χρήματα και αποθηκεύει τα προσωπικά στοιχεία του ιδιοκτήτη της κάρτας. Τα δύο applet βρίσκονται απομονωμένα και η μεταξύ τους επικοινωνία είναι εφικτή μόνο μέσω μιας διαμοιραζόμενης διεπαφής. Αυτό στην πράξη σημαίνει ότι, μόνο οι μέθοδοι που ορίζονται σε interface που επεκτείνει το Shareable είναι διαθέσιμες μέσω του τείχους προστασίας (firewall).

Για να είναι δυνατή η ανάλυση της εφαρμογής θα πρέπει να προσθέσουμε τις απαιτούμενες επισημειώσεις στον πηγαίο κώδικα. Οι δύο μέθοδοι με τις οποίες θα ασχοληθούμε είναι η makePurchase(APDU apdu) της κλάσης BonusApplet και η foreignDebit(short amount) της διεπαφής PurseShareable.

Αρχικά, δημιουργούμε μια νέα κλάση, την TaintSource, η οποία θα αναπαριστά την πηγή tainted δεδομένων. Σε αυτήν ορίζουμε τη μέθοδο getAmount που παίρνει σαν όρισμα ένα αντικείμενο APDU από το οποίο εξάγει και επιστρέφει ένα αντικείμενο τύπου Short. Επισημειώνουμε με @TaintedResult την μέθοδο αυτή καθώς τα δεδομένα που επιστρέφει προέρχονται από το εξωτερικό προς την εφαρμογή περιβάλλον και δεν έχουν επικυρωθεί.

```
public class TaintSource {

      @TaintedResult()
      public Short getAmount(APDU apdu) {
            byte[] buffer = apdu.getBuffer();
            byte[] amount = new byte[2];
            Util.arrayCopy(buffer, (short) ISO7816.OFFSET_CDATA,
                           amount, (short) 0,(short) 2);
            Short amountShort = (short) (amount[1] & 0x0F);
            amountShort += (short) (amount[1] & 0xF0);
            amountShort += (short) (amount[0] << 8);
            return amountShort;
      }
}
```

Στη συνέχεια, χαρακτηρίζουμε την μέθοδο foreignDebit της διεπαφής PurseShareable ως ευαίσθητη. Για να το κάνουμε αυτό αρκεί να επισημειώσουμε τα ορίσματα που δεν θα πρέπει να λάβουν tainted δεδομένα, ως @Sensitive.

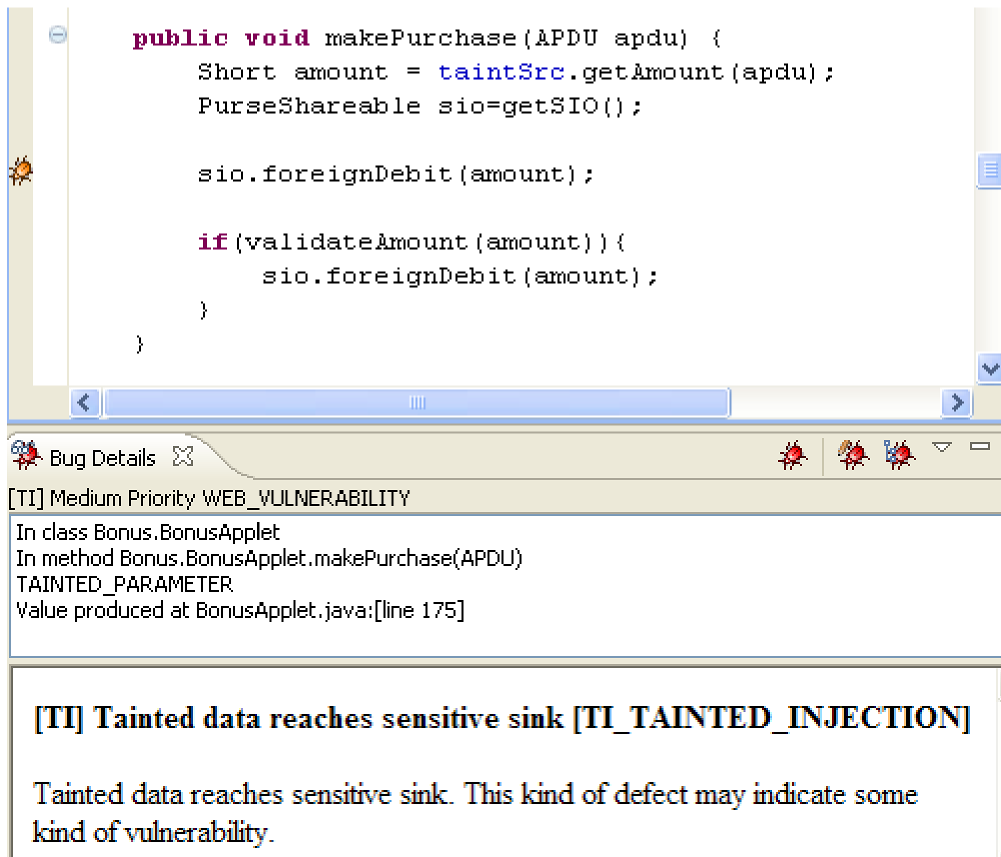
```
public interface PurseShareable extends Shareable {
      public void foreignDebit(@Sensitive Short amount);
}
```

Ακολούθως, τροποποιούμε τον κώδικα της κλάσης BonusApplet για να ανταποκρίνεται στις αλλαγές που έχουν προηγηθεί. Δημιουργούμε ένα νέο πεδίο στην κλάση το οποίο συσχετίζουμε με ένα αντικείμενο τύπου TaintSource για να το χρησιμοποιήσουμε για εισαγωγή tainted δεδομένων. Επίσης, κατασκευάζουμε μια μέθοδο επικύρωσης η οποία απλά επιβεβαιώνει ότι το ποσόν είναι μη αρνητικός αριθμός και διάφορο του μηδενός. Εφαρμόζουμε την επισημείωση @TaintnessValidator για να ορίσουμε ότι η μέθοδος αυτή επικυρώνει τα δεδομένα που δέχεται ως παράμετρο και άρα πρέπει να θεωρούνται untainted στη συνέχεια. Τέλος, στην μέθοδο makePurchase λαμβάνουμε το ποσό και το δίνουμε σαν παράμετρο στην κλήση της μεθόδου foreignDebit. Για τους σκοπούς του παραδείγματος ελέγχουμε τη συμπεριφορά του αναλυτή και σε περίπτωση όπου πραγματοποιείται επικύρωση των δεδομένων.

```
private TaintSource taintSrc = new TaintSource();

public void makePurchase(APDU apdu) {
      Short amount = taintSrc.getAmount(apdu);
      PurseShareable sio=getSIO();
      sio.foreignDebit(amount);

      if(validateAmount(amount)){
            sio.foreignDebit(amount);
      }
}

@TaintnessValidator
private boolean validateAmount(Short amount) {
      if (amount <= 0)
            return false;
      return true;
}
```

Στο Σχήμα 6.3 παρουσιάζεται το αποτέλεσμα της εφαρμογής του ανιχνευτή στην JavaCard εφαρμογή. Παρατηρούμε ότι ο ανιχνευτής εντοπίζει επικίνδυνη την κλήση στη μέθοδο foreignDebit καθώς η μεταβλητή amount είναι tainted ενώ

ταυτόχρονα στη δήλωση της μεθόδου επισημειώσαμε το όρισμα ως @Sensitive. Αξίζει να σημειώσουμε ότι η δεύτερη κλήση στην μέθοδο foreignDebit δεν αναφέρεται από τον ανιχνευτή καθώς η μεταβλητή amount έχει υποστεί επικύρωση.

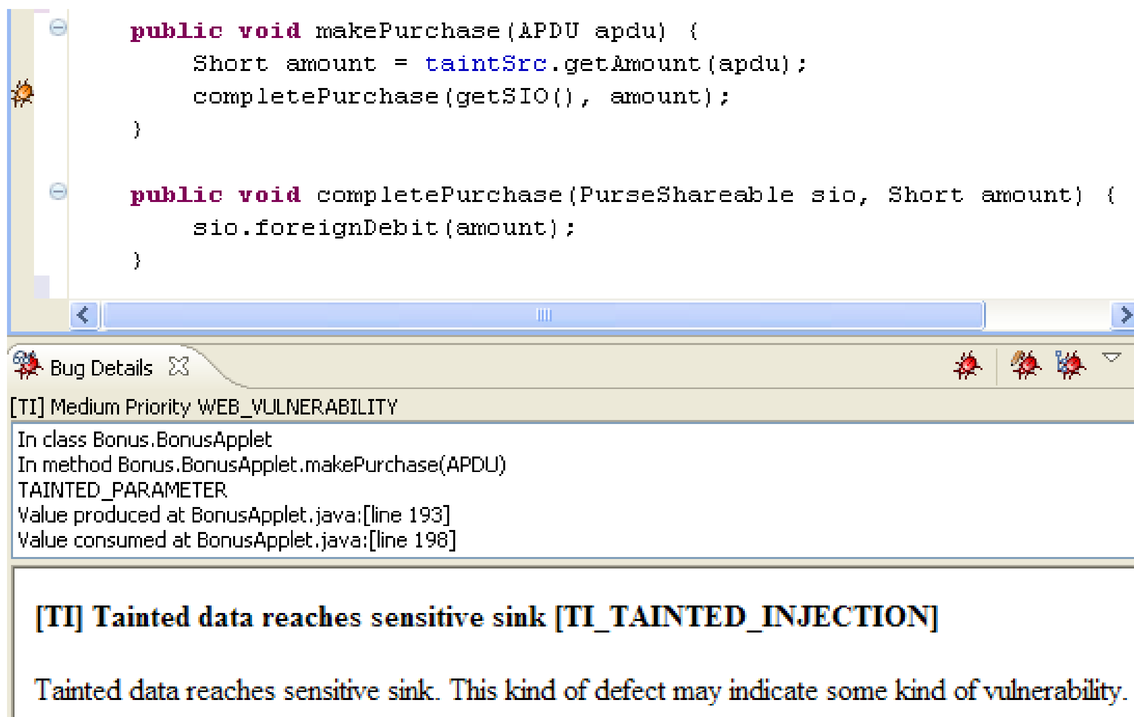


Σχήμα 6.3 Αποτέλεσμα εφαρμογής ανιχνευτή στην εφαρμογή

Στο σημείο αυτό αξίζει να δούμε ένα δεύτερο παράδειγμα όπου εξετάζουμε τη συμπεριφορά του ανιχνευτή σε διάδοση δεδομένων που δεν περιορίζονται στα πλαίσια μιας μόνο μεθόδου. Για την περίπτωση αυτή τροποποιούμε την μέθοδο makePurchase όπως φαίνεται παρακάτω και επιπλέον προσθέτουμε τη μέθοδο completePurchase στην οποία μεταφέρεται πλέον η κλήση στην ευαίσθητη μέθοδο foreignDebit.

```
public void makePurchase (APDU apdu) {
      Short amount = taintSrc.getAmount(apdu);
      completePurchase(getSIO(), amount);
}

public void completePurchase(PurseShareable sio, Short amount) {
      sio.foreignDebit(amount);
}
```

Και στην περίπτωση αυτή ο ανιχνευτής εντοπίζει επιτυχώς το ελάττωμα, όπως φαίνεται στο Σχήμα 6.4. Συγκεκριμένα, αναφέρει την κλήση στην μέθοδο completePurchase ως επικίνδυνη καθώς η tainted μεταβλητή amount μεταδίδεται από την μέθοδο makePurchase στην μέθοδο completePurchase και προσεγγίζει τελικά την ευαίσθητη μέθοδο foreignDebit.



Σχήμα 6.4 Εφαρμογή ανιχνευτή για interprocedural περίπτωση

7 Επίλογος

Η εργασία αυτή εξέτασε την επίλυση προβλημάτων taint analysis μέσω του εργαλείου FindBugs. Παρόλο που το πρόβλημα αυτό έχει μελετηθεί εκτενώς στα πλαίσια εφαρμογών Ιστού (5) (16), στην εργασία αυτή το εξετάσαμε στα πλαίσια multi-applet JavaCard εφαρμογών. Με δεδομένο το σημαντικό ρόλο που παίζει η ασφάλεια σε τέτοιες εφαρμογές, είναι απαραίτητο να διασφαλίσουμε την έλλειψη ροής μη επικυρωμένων δεδομένων σε κρίσιμες μεθόδους της εφαρμογής.

Η προσέγγιση που παρουσιάσαμε στηρίζεται στις επισημειώσεις της Java για να προσθέσει επιπλέον πληροφορία σχετικά με τις προθέσεις του προγραμματιστή. Θα ήταν χρήσιμο στο μέλλον να γίνει προσαρμογή του κώδικα του ανιχνευτή ώστε να συμβαδίζει με το JSR 305 (17), καθώς προσπαθεί να προσφέρει ένα ενιαίο πακέτο επισημειώσεων για χρήση από εργαλεία που προσανατολίζονται σε θέματα ασφάλειας.

Παράρτημα: Επισημειώσεις

Με τον όρο *επισημείωση* (annotation) αναφερόμαστε γενικά σε οποιαδήποτε σήμανση συνοδεύει μια μορφή πληροφορίας (π.χ. κείμενο, βίντεο). Στην περίπτωση του κώδικα οι επισημειώσεις χρησιμοποιούνται για τον εμπλουτισμό του κώδικα με πληροφορία η οποία δεν επηρεάζει άμεσα τη σημασιολογία του προγράμματος, δηλαδή, δεν τροποποιεί τις ενέργειες που εκτελεί ο κώδικας. Εντούτοις, η πληροφορία αυτή μπορεί να αξιοποιηθεί από διάφορα εργαλεία όπως για παράδειγμα μια γεννήτρια πηγαίου κώδικα ή ένα εργαλείο στατικής ανάλυσης κώδικα. Στην Java, παρόλο που προϋπήρχε η δυνατότητα χρήσης ενσωματωμένων επισημειώσεων όπως οι @Deprecated και @Override, η δυνατότητα δήλωσης νέων τύπων επισημειώσεων (ή μεταδεδομένων) προστέθηκε στην έκδοση J2SE 5 και υποστηρίζεται από τη βιβλιοθήκη java.lang.annotation (15).

Η δήλωση ενός τύπου επισημείωσης στηρίζεται στο μηχανισμό δήλωσης διεπαφών (interafaces) με χαρακτηριστικότερη ίσως διαφορά το σύμβολο @ που προηγείται της λέξης-κλειδί interface. Παρόλο που στην γενική περίπτωση είναι δυνατόν να δηλωθούν μέθοδοι μέλη στον τύπο επισημείωσης, στην περίπτωση που μας ενδιαφέρει αυτό δεν είναι αναγκαίο. Ένας τύπος επισημείωσης που δεν περιλαμβάνει μεθόδους μέλη καλείται *επισημείωση σήμανσης* (marker annotation).

Ο παρακάτω κώδικας παρουσιάζει τη δήλωση ενός τύπου επισημείωσης σήμανσης με την ονομασία Sensitive που μπορεί να χρησιμοποιηθεί για τον χαρακτηρισμό ευαίσθητων δεδομένων.

```
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.PARAMETER)
public @interface Sensitive {}
```

Παρατηρούμε ότι στην δήλωση του τύπου αυτού εφαρμόζονται δύο *μετα-επισημειώσεις*:

- Retention: Ορίζει το σημείο μέχρι το οποίο θα διατηρηθεί η επισημείωση και μπορεί να πάρει μια τιμή εκ των SOURCE, CLASS και RUNTIME. Στο συγκεκριμένο παράδειγμα επιλέγουμε τη διατήρηση των επισημειώσεων στα αρχεία .class που θα παραχθούν από τον μεταγλωττιστή.
- Target: Ορίζει τα στοιχεία στα οποία μπορεί να εφαρμοστεί η επισημείωση και μπορεί να πάρει μια τιμή εκ των ANNOTATION_TYPE, CONSTRUCTOR, FIELD,

LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER και TYPE. Στο συγκεκριμένο παράδειγμα δηλώνουμε ότι η επισημείωση Sensitive επιτρέπεται να εφαρμόζεται μόνο σε παραμέτρους μεθόδων. Σε κάθε άλλη περίπτωση ο μεταγλωττιστής θα αναφέρει σφάλμα.

Αφού δηλώσουμε έναν τύπο επισημείωσης μπορούμε να χρησιμοποιήσουμε την συγκεκριμένη επισημείωση για σήμανση κλάσεων, μεθόδων, πεδίων, παραμέτρων κ.ο.κ. Είναι βέβαια προφανές ότι, η χρήση επισημειώσεων πρέπει να περιοριστεί στα στοιχεία τα οποία ορίζει η μετα-επισημείωση Target του αντίστοιχου τύπου. Σε όλες τις περιπτώσεις η επισημείωση προηγείται της δήλωσης στην οποία θέλουμε να εφαρμόζεται. Σε συνέχεια του προηγούμενου παραδείγματος, ο ακόλουθος κώδικας παρουσιάζει την εφαρμογή της επισημείωσης @Sensitive στην παράμετρο value.

```
public void sensitive(@Sensitive String value) {
    System.out.println("sensitive got value: " + value);
}
```

Αναφορές

1. *IEEE Standard for Software Unit Testing: An American National Standard.* s.l. : The Institute of Electrical and Electronics Engineers, 1999. ANSI/IEEE Std 1008-1987.

2. *Statically detecting likely buffer overflow vulnerabilities.* **Larochelle, David and Evans, David.** Washington, D.C. : USENIX Association, 2001. SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium.

3. *Detecting memory errors via static pointer analysis.* **Dor, Nurit, Rodeh, Michael and Sagiv, Mooly.** Montreal : ACM Press, 1998. PASTE '98: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 27-34.

4. *Finding more null pointer bugs, but not too many.* **Hovemeyer, David and Pugh, William.** San Diego : ACM Press, 2007. PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 9-14.

5. *Finding Security Vulnerabilities in Java Applications with Static Analysis.* **Livshits, Benjamin and Lam, Monica.** 2005. Proceedings of the 14th USENIX Security Symposium.

6. **Ullman, Alfred V. Aho and Monica S. Lam and Ravi Sethi and Jeffrey D.** *Compilers: Principles, Techniques, and Tools.* 2nd Edition. Boston : Addison Wesley, 2006. ISBN 0321486811.

7. *Finding Bugs is Easy.* **Hovemeyer, David and Pugh, William.** 12, s.l. : ACM Press, 2004, SIGPLAN Not., Vol. 39, pp. 92-106.

8. Application Security Project. *The ten most critical Web application security vulnerabilitieσ.* [Online] 2004. http://umn.dl.sourceforge.net/sourceforge/ owasp/OWASPTopTen2004.pdf.

9. CWE/SANS. *Top 25 Most Dangerous Programming Errors.* [Online] 2009. http://cwe.mitre.org/top25.

10. US-CERT Vulnerability Notes Database. *Mozilla Firefox fails to properly sanitize user-supplied URIs via shell script.* [Online] 2005. http://www.kb.cert.org/vuls/id/914681.

11. **Zhang, Jerry.** Implementation of Customized FindBugs Detectors. [Online] http://people.cs.ubc.ca/~kdvolder/CPSC511/submissions_06_07/jerry.pdf.

12. **Dahm, Markus.** *Byte Code Engineering with the BCEL API.* 2001.

13. Teachable Static Analysis Workbench. [Online] http://code.google.com/p/teachablesa/.

14. **Λοϊζίδης, Αλέξανδρος.** Εφαρμογές Έξυπνων Καρτών Ανοικτής Σχεδίασης Java Card. Θεσσαλονίκη : s.n., 2008.

15. *Static program analysis for Java Card applets.* **Almaliotis, Vasilios, et al.** London : Springer-Verlag, 2008. CARDIS '08: Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications. pp. 17-31.

16. *Dynamic Taint Propagation for Java.* **Haldar, Vivek, Franz, Michael και Chandra, Deepak.** s.l. : IEEE Computer Society, 2005. ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference. σσ. 303-311.

17. **Pugh, William.** Java Specification Requests. *JSR 305: Annotations for Software Defect Detection.* [Ηλεκτρονικό] http://jcp.org/en/jsr/detail?id=305.

18. **Gosling, James, et al.** *Java Language Specification, 3rd Edition.* s.l. : Addison Wesley, 2005.