ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι. ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

<<Εύρεση Ευπαθειών Λογισμικού:

Κατανεμημένο Fuzzing>>

Του φοιτητή

Τσαούσογλου Βασίλη

Αρ. Μητρώου: 073255

Επιβλέπων καθηγητής

Ηλιούδης Χρήστος

Θεσσαλονίκη 2016

ΠΡΟΛΟΓΟΣ

Τα προγράμματα υπολογιστών δέχονται δεδομένα και παράγουν δεδομένα. Επιπλέον δεδομένα χρησιμοποιούνται εσωτερικά από τα προγράμματα με σκοπό να ρυθμίζεται επιτυχώς η ροή εκτέλεσης τους.

Λόγω του τρόπου υλοποίησης και εκτέλεσης των προγραμμάτων στα σύγχρονα συστήματα, σφάλματα κατά την επεξεργασία των δεδομένων εισόδου είναι πιθανόν να επηρεάσουν τα εσωτερικά δεδομένα ενός προγράμματος, επηρεάζοντας την ροή εκτέλεσης του με απροσδόκητο τρόπο.

Ορισμένα σφάλματα αυτού του είδους μπορούν να χρησιμοποιηθούν με τέτοιο τρόπο ώστε στέλνοντας τα κατάλληλα δεδομένα εισόδου να επηρεάζεται ελεγχόμενα η ροή εκτέλεσης του προγράμματος.

Στην περίπτωση όπου ένα τέτοιο σφάλμα κριθεί πως μπορεί να γίνει εκμεταλλεύσιμο από εξωτερικούς παράγοντες τότε αυτό χαρακτηρίζεται ως ευπάθεια, καθώς επιτρέπει σε εξωτερικούς παράγοντες να παραβιάσουν την ακεραιότητα, διαθεσιμότητα και εμπιστευτικότητα τόσο του ίδιου του προγράμματος όσο και των δεδομένων που αυτό διαχειρίζεται.

Λόγω του όλο και αυξανόμενου ρόλου της ασφάλειας στα σύγχρονα υπολογιστικά συστήματα η έρευνα για τον εντοπισμό ευπαθειών λογισμικού αποτελεί πλέον αντικείμενο έντονου ενδιαφέροντος.

Αναμφιβόλως η διεξοδική εξέταση του κώδικα ενός λογισμικού, είτε πρόκειται για τον πηγαίο κώδικα είτε για κώδικα μηχανής, είναι ο πληρέστερος τρόπος έρευνας για τον εντοπισμό σφαλμάτων και εν δυνάμει ευπαθειών σε αυτό. Η εύρεση ενός σφάλματος κατά αυτό τον τρόπο συνοδεύεται από μετέπειτα προσπάθειες να προσδιοριστούν, εφόσον είναι δυνατό, τα δεδομένα εισόδου που προκαλούν το σφάλμα κατά την διάρκεια εκτέλεσης του λογισμικού.

Εάν αντιστρέψουμε την λογική της παραπάνω μεθόδου τότε μπορούμε να καταλήξουμε σε μια διαφορετική προσέγγιση όπου πρώτα αναζητούμε δεδομένα εισόδου που προκαλούν κάποιο σφάλμα εν ώρα εκτέλεσης και έπειτα εντοπίζουμε το σφάλμα που προκλήθηκε.

Η αναζήτηση αυτών των δεδομένων υλοποιείται μέσω της συνεχούς αποστολής απροσδόκητων ή και τελείως τυχαίων δεδομένων στις εισόδους του λογισμικού μέχρις ότου αυτό να τερματίσει απροσδόκητα, σηματοδοτώντας πως κατά την επεξεργασία των απεσταλμένων δεδομένων υπήρξε κάποιο σφάλμα που επηρέασε την εκτέλεση του.

Σε αντίθεση με την προηγούμενη προσέγγιση η μετέπειτα ανάλυση στην περίπτωση αυτή στοχεύει στον εντοπισμό του σφάλματος στον κώδικα εξετάζοντας το στιγμιότυπο του τερματισμένου προγράμματος στην μνήμη.

Αυτή η προσέγγιση ονομάζεται fuzzing και η κατανεμημένη χρήση της είναι το αντικείμενο της παρούσας πτυχιακής εργασίας.

ΠΕΡΙΛΗΨΗ

Σκοπός της πτυχιακής εργασίας είναι ο σχεδιασμός και η υλοποίηση μιας υποδομής που επιτρέπει το κατανεμημένο fuzzing εφαρμογών καθώς και την κεντρικοποιημένη συλλογή αποτελεσμάτων και στατιστικών στοιχείων που προκύπτουν κατά την χρήση της.

Αρχικά δίνετε μια σύντομη περιγραφή των ευπαθειών λογισμικού και των υφιστάμενων μεθόδων εύρεσης τους.

Στη συνέχεια περιγράφεται η μέθοδος του fuzzing, παρουσιάζοντας αναλυτικά τα επιμέρους στάδια της, τους τρόπους εφαρμογής της για τον εκάστοτε τύπο λογισμικού καθώς και τις διαφορετικές προσεγγίσεις που δύναται να ακολουθηθούν ανά περίπτωση.

Έπειτα παρουσιάζονται τα μέλη της υποδομής που δημιουργήθηκε καθώς και τα αποτελέσματα που συλλέχθηκαν κατά την χρήση της για την εύρεση σφαλμάτων στο λογισμικό ανάγνωσης και επεξεργασίας PDF αρχείων, Foxit Reader.

Καθ όλη την έκταση της πτυχιακής εργασίας παρουσιάζεται ο κώδικας υλοποίησης ως συμπληρωματικό μέσο που, σε συνδυασμό με την λεκτική περιγραφή, υποβοηθά την κατανόηση της εργασίας από τον αναγνώστη, ενώ ταυτόχρονα παρουσιάζει το πρακτικό μέρος της εργασίας.

Τέλος αναφέρονται τα συμπεράσματα που εξήχθησαν κατά την εκπόνηση της εργασίας καθώς και οι μελλοντικές εργασίες που μπορούν να γίνουν όσον αφορά την επέκταση της υποδομής.

ABSTRACT

The discovery of exploitable software vulnerabilities is a time consuming and demanding task.

Fuzz testing (or fuzzing) is a collection of technologies and techniques that provide random (and carefully constructed random) data to applications as inputs with the goal of triggering not intended behavior and ultimately to the discovery of bugs.

The objective of this thesis is the design and implementation of an infrastructure that aids in the deployment of distributed fuzzing campaigns and the centralized storage of results and statistics. By automating phases of the fuzzing process we aim to reduce the administrative overhead. Moreover, our modular architecture allows for the continuous improvement and targeted customization of the fuzzing phases for each particular application under investigation.

The thesis is structured as follows.

Chapter 1 introduces the concepts of software vulnerabilities and how can their exploitation lead to the violation of trust boundaries. Existing vulnerability discovery methods are also presented therein.

Fuzzing, a method of vulnerability discovery, is thoroughly examined, in Chapter 2. The concepts presented in this chapter are further assisted by showcasing state-of-the art tools.

Chapter 3 presents the distributed fuzzing infrastructure that we have designed, implemented and deployed. A hands-on example on the use of our system is given at Chapter 4.

The process of setting up a distributed fuzzing campaign is practically presented, in relation to the theoretical concepts presented at Chapter 2.

We conclude with the strengths and shortcomings of our developed distributed fuzzing infrastructure and the future plans regarding its improvements.

Κατάλογος περιεχομένων

Ευρετήριο σχημάτων

1 Ευπάθειες Συστημάτων και Λογισμικού

1.1 Εισαγωγή

Στο πρώτο κεφάλαιο της εργασίας παρουσιάζονται οι όροι της ασφάλειας υπολογιστών των οποίων η γνώση είναι απαραίτητη για την κατανόηση των θεμάτων που παρουσιάζονται στην συνέχεια.

1.2 Ευπάθειες Υπολογιστικών Συστημάτων (System Vulnerabilities)

Στον τομέα της ασφάλειας υπολογιστών ως ευπάθεια ενός υπολογιστικού συστήματος ορίζεται οποιοδήποτε σφάλμα υπάρχει στον σχεδιασμό, την υλοποίηση ή την λειτουργία ενός συστήματος το οποίο επιτρέπει σε εξωτερικούς παράγοντες να παραβιάσουν την πολιτική ασφαλείας του [1].

Οι ευπάθειες αφορούν όλα τα μέλη από τα οποία αποτελείται ένα υπολογιστικό σύστημα καθώς και όλα τα εξωτερικά προς το σύστημα μέλη που αλληλεπιδρούν με αυτό και συνεπώς επηρεάζουν την λειτουργία του.

Μια ευπάθεια ενός συστήματος μπορεί να αφορά:

- Το φυσικό περιβάλλον του συστήματος.
- Το υλικό του συστήματος.
- Το λογισμικό του συστήματος.
- Το ανθρώπινο δυναμικό που διαχειρίζεται το σύστημα.

Το πεδίο της ασφάλειας υπολογιστών αφορά συνολικά την ασφάλεια ενός υπολογιστικού συστήματος, ωστόσο η έρευνα για τον εντοπισμό ευπαθειών σε οποιοδήποτε από τα επιμέρους μέλη ενός συστήματος αποτελεί ξεχωριστό πεδίο ερευνητικής δραστηριότητας.

Για τον αποτελεσματικό έλεγχο ολόκληρου του συστήματος είναι απαραίτητο να εκτελεστεί έρευνα σε κάθε ένα από τα μέλη του, καθώς ένα σύστημα θεωρείται όσο ασφαλές όσο το πιο επισφαλές μέλος του.

Η μέθοδος που παρουσιάζεται στην παρούσα εργασία ανήκει στο πεδίο της έρευνας για τον εντοπισμό ευπαθειών στο λογισμικό μέρος ενός υπολογιστικού συστήματος.

1.3 Ευπάθειες Λογισμικού (Software Vulnerabilities)

Επεκτείνοντας τον ορισμό της ευπάθειας ενός υπολογιστικού συστήματος, ορίζουμε ως ευπάθεια λογισμικού οποιοδήποτε σφάλμα υπάρχει στον σχεδιασμό, την υλοποίηση ή την λειτουργία ενός λογισμικού το οποίο επιτρέπει εξωτερικούς παράγοντες να παραβιάσουν την πολιτική ασφαλείας του.

Όπως απορρέει από τον παραπάνω ορισμό, οι ευπάθειες λογισμικού είναι ένα υποσύνολο των σφαλμάτων λογισμικού.

Πράγματι όλες οι ευπάθειες λογισμικού είναι σφάλματα λογισμικού(software bugs) ωστόσο δεν θεωρούνται όλα τα σφάλματα λογισμικού ευπάθειες. Για να χαρακτηριστεί ένα σφάλμα λογισμικού ως ευπάθεια θα πρέπει να αποδειχθεί πως η πιθανή εκμετάλλευση του από εξωτερικούς παράγοντες παραβιάζει την ακεραιότητα, εμπιστευτικότητα και διαθεσιμότητα του λογισμικού και των δεδομένων που αυτό διαχειρίζεται [2].

1.4 Όρια εμπιστοσύνης (Trust Boundaries)

Προτού εξεταστεί η εκμετάλλευση των ευπαθειών λογισμικού είναι σημαντικό να αναφέρουμε τον ρόλο των ορίων εμπιστοσύνης στην ασφάλεια υπολογιστών.

Τα όρια εμπιστοσύνης ορίζουν με σαφή τρόπο τα σύνορα μέσα στα οποία ένα σύστημα εμπιστεύεται όλα τα υποσυστήματα του, συμπεριλαμβανομένων και των δεδομένων που διαχειρίζεται [3].

Η εκτέλεση και τα δεδομένα ενός λογισμικού αλλάζουν όρια εμπιστοσύνης καθώς μεταβαίνουν από ένα υποσύστημα του λογισμικού σε άλλο. Για παράδειγμα, τα δεδομένα που εισέρχονται σε μια εφαρμογή δια μέσω δικτύου αλλάζουν όριο εμπιστοσύνης την στιγμή που εισέρχονται στην εφαρμογή από το δίκτυο. Κατά την αλλαγή του ορίου χρειάζεται να γίνουν οι απαραίτητοι έλεγχοι από την εφαρμογή προτού τα δεδομένα θεωρηθούν έμπιστα.

1.5 Εκμετάλλευση Ευπαθειών Λογισμικού (Exploitation)

Η εκμετάλλευση(exploitation) μιας ευπάθειας λογισμικού περιλαμβάνει όλες τις ενέργειες που απαιτείται να εκτελεστούν ώστε να παραβιαστεί η πολιτική ασφαλείας του λογισμικού. Αυτού τους είδους οι ενέργειες αποτελούν πάντα κάποια μορφή αποστολής δεδομένων στις εισόδους του λογισμικού όπου τα δεδομένα που εισέρχονται προέρχονται εκτός των ορίων εμπιστοσύνης του υποσυστήματος που τα επεξεργάζεται.

Το γεγονός πως εξωτερικοί παράγοντες μπορούν να εκμεταλλευθούν τις ευπάθειες λογισμικού απομακρυσμένα, με άμεσο τρόπο στην περίπτωση που το λογισμικό

επικοινωνεί με δίκτυα ή με έμμεσο τρόπο κάνοντας χρήση τεχνικών κοινωνικής μηχανικής, καθιστά τις εκμεταλλεύσεις ευπαθειών λογισμικού εξαιρετικά κρίσιμες για την ασφάλεια ολόκληρου του συστήματος και των δεδομένων του.

Ο βαθμός δυσκολίας της εκμετάλλευσης μιας ευπάθειας σε συνδυασμό με το τελικό αποτέλεσμα της πιθανής εκμετάλλευσής της κρίνει τον βαθμό κρισιμότητας ο οποίος εν τέλει θα της αποδοθεί.

Οι τεχνικές με τις οποίες δύναται να εκμεταλλευθεί κανείς μια ευπάθεια διαφέρουν, εξαρτώμενες από το υλικό και λειτουργικό σύστημα που φιλοξενεί το λογισμικό και πρωτίστως τον σχεδιασμό, την υλοποίηση και την διαμόρφωση του ίδιου του λογισμικού.

1.6 Κλάσεις Ευπαθειών Λογισμικού (Bug Classes)

Βασιζόμενοι στον ορισμό που δόθηκε στο κεφάλαιο 1.2, οι ευπάθειες λογισμικού μπορούν να ταξινομηθούν σε κλάσεις σύμφωνα με το στάδιο ζωής του λογισμικού [4] στο οποίο εμφανίζονται.

Αυτή η ταξινόμηση δεν είναι η μοναδική, καθώς οι ευπάθειες λογισμικού διέπονται από πολλά διαφορετικά χαρακτηριστικά και ως εκ τούτου μπορούν να ταξινομηθούν με ποικίλους τρόπους. Επιπλέον δεν είναι ούτε απολύτως ορθή καθώς μια ευπάθεια μπορεί να ανήκει σε περισσότερες από μια κλάσεις.

Η ταξινόμηση κατά αυτό τον τρόπο μας βοηθά, ωστόσο, να ξεχωρίσουμε το είδος των ευπαθειών που μπορούν να εντοπιστούν χρησιμοποιώντας την μέθοδο που εξετάζεται στην παρούσα εργασία.

1.6.1 Ευπάθειες Σχεδιασμού (Design Vulnerabilities)

Οι σχεδιαστικές ευπάθειες είναι αποτέλεσμα σφαλμάτων στον σχεδιασμό ενός λογισμικού. Οι εφαρμογές που υπόκεινται σε τέτοιου είδους σφάλματα είναι τρωτές σε εκμεταλλεύσεις ακόμη και στην περίπτωση όπου η υλοποίηση τους είναι αλάθητη.

Ένα παράδειγμα αυτού του είδους ευπάθειας είναι η χρήση αδύναμης κρυπτογράφησης κατά την μετάδοση ευαίσθητων δεδομένων μιας εφαρμογής. Σε αυτή την περίπτωση δίδεται η δυνατότητα σε εξωτερικούς παράγοντες να παραβιάσουν την εμπιστευτικότητα των δεδομένων της εφαρμογής.

Ο εντοπισμός των ευπαθειών αυτού του είδους είναι αποτέλεσμα κυρίως της σχολαστικής εξέτασης του σχεδιασμού το λογισμικού και όχι χρήσης fuzzing.

1.6.2 Ευπάθειες Διαμόρφωσης (Configuration Vulnerabilities)

Οι ευπάθειες διαμόρφωσης είναι αποτέλεσμα εσφαλμένης διαμόρφωσης των ρυθμίσεων του λογισμικού που βρίσκεται σε λειτουργία.

Παραδείγματα τέτοιων ευπαθειών αποτελούν η χρήση αδύναμων κωδικών αυθεντικοποίησης και η λανθασμένη διαμόρφωση των δικαιωμάτων πρόσβασης στο σύστημα αρχείων που χρησιμοποιεί το λογισμικό.

Όπως και στην περίπτωση των σχεδιαστικών ευπαθειών, το fuzzing δεν είναι η προσφιλέστερη μέθοδος εύρεσης αυτού του είδους ευπαθειών.

1.6.3 Ευπάθειες Υλοποίησης (Implementation Vulnerabilities)

Οι ευπάθειες υλοποίησης είναι αποτέλεσμα σφαλμάτων στην υλοποίηση του σχεδιασμού ενός λογισμικού. Τα σφάλματα αυτά, γνωστά και ως προγραμματιστικά λάθη, είναι συνήθως αποτέλεσμα της λανθασμένης χρήσης της ελευθερίας που παρέχουν στον προγραμματιστή οι γλώσσες προγραμματισμού όπως η C και η C++.

Στην κατηγορία αυτή ανήκουν τα σφάλματα που βρίσκονται μέσω της μεθόδου που παρουσιάζεται στην παρούσα εργασία και ως εκ τούτου είναι χρήσιμο να τα εξετάσουμε περαιτέρω .

1.6.4 Κατηγορίες Ευπαθειών Υλοποίησης σε C/C++

Για την εξέταση των ευπαθειών υλοποίησης θα εξετάσουμε παραδείγματα σφαλμάτων που παρουσιάζονται σε προγράμματα γραμμένα στις γλώσσες C και C++.

1.6.4.1 Υπερχείλιση Μνήμης (Buffer Overflows)

Η υπερχείλιση μνήμης είναι ο πιο συχνά εμφανιζόμενος τύπος σφάλματος. Η κρισιμότητα αυτών των σφαλμάτων κρίνεται από το είδος των δεδομένων τα οποία επικαλύπτει η υπερχείλιση. Τα σφάλματα υπερχείλισης μνήμης χωρίζονται στις παρακάτω δύο κατηγορίες βάση της περιοχής μνήμης στην οποία ενεργούν.

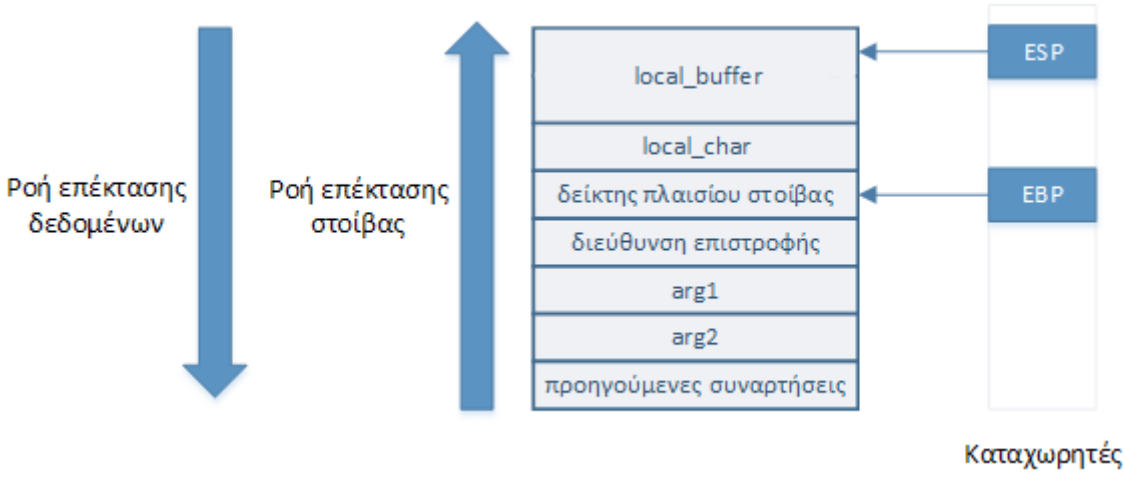
1.6.4.1.1 Υπερχείλιση Μνήμης Στοίβας (Stack-based Overflows)

Τα σφάλματα υπερχείλισης στοίβας θεωρούνται εξαιρετικά κρίσιμα καθώς η στοίβα μιας διεργασίας χρησιμοποιείται για την αποθήκευση δεδομένων που επηρεάζουν άμεσα την ροή εκτέλεσης του προγράμματος.[5]

Κύρια αιτία αυτών των σφαλμάτων είναι η λανθασμένη διαχείριση των τοπικών μεταβλητών μιας συνάρτησης, οι οποίες, λόγου του τρόπου υλοποίησης των προγραμμάτων, αποθηκεύονται στην στοίβα μαζί με κρίσιμα για την εκτέλεση δεδομένα.

```
void func(int arg1, char * arg2)
{
 char local_char;
 char local_buffer[512];
 strcpy(local_buffer, arg_2);
}
```

Λόγω του τρόπου λειτουργίας της στοίβας στην περίπτωση που το μέγεθος της μεταβλητής arg2 του παραπάνω παραδείγματος είναι μεγαλύτερο από 512 bytes η συνάρτηση strcpy θα επικαλύψει την διεύθυνση επιστροφής της συνάρτησης func με δεδομένα της arg2.



Εικόνα 1: Απεικόνιση στοίβας

1.6.4.1.2 Υπερχείλιση Μνήμης Σωρού (Heap-based Overflows)

Η υπερχείλιση μνήμης που ανήκει στον σωρό της διεργασίας αποτελεί την πιο συχνά εμφανιζόμενη περίπτωση υπερχείλισης μνήμης. Η εκμεταλλευσιμότητα αυτών των σφαλμάτων ποικίλει αναλόγως την εφαρμογή και την υλοποίηση του σωρού που χρησιμοποιεί.

Στο παρακάτω παράδειγμα αν το μέγεθος της πρώτης παραμέτρου ξεπεράσει τα 512 bytes τότε η συνάρτηση strcpy θα γράψει σε μνήμη που δεν ανήκει στην μεταβλητή input.

```
void func(char * arg1)
{
  char *input;
  input = malloc(512);
  strcpy(input, arg1);
}
```

1.6.4.2 Χρήση αποδεσμευμένης μνήμης (Use-after-free)

Τα σφάλματα χρήσης αποδεσμευμένης μνήμης συμβαίνουν όταν αποδεσμεύεται μνήμη και έπειτα χρησιμοποιείται εσφαλμένα ένας δείκτης αναφοράς σε αυτή. Η εκμεταλλευσιμότητα αυτών των σφαλμάτων εξαρτάται κυρίως από το εάν μπορούμε να δεσμεύσουμε την αποδεσμευμένη μνήμη πριν γίνει χρήση του δείκτη αναφοράς.

```
...
Obj* obj1 = new Obj();
...
delete obj1;
...
obj1.foo()
```

Στο παραπάνω παράδειγμα ο δείκτης αναφοράς obj1 χρησιμοποιείται εσφαλμένα μετά την αποδέσμευση του αντικειμένου.

1.6.4.3 Παρά-ένα λάθη (Off-by-one)

Τα παρά-ένα λάθη είναι λάθη κατά τα οποία ο υπολογισμός μεγέθους μιας περιοχής μνήμης είναι ανακριβής για ένα byte. Τα σφάλματα αυτά είναι εκμεταλλεύσιμα στην περίπτωση που μας επιτρέπουν να γράψουμε σε μνήμη που περιέχει δυναμικές δομές δεδομένων. [6] Στο παρακάτω παράδειγμα ο βρόγχος θα γράψει ένα παραπάνω μηδενικό σε θέση μνήμης που δεν ανήκει στον πίνακα arr.

```
void func(char * arg1)
{
  int i;
  int arr[4];
  for (i = 0; i <= 4; i++)
    arr[i] = 0;
}
```

1.6.4.4 Υπερχείλιση Ακεραίων Μεταβλητών (Integer Overflows)

Οι ακέραιες μεταβλητές έχουν την δυνατότητα να αποθηκεύσουν τιμές συγκεκριμένου εύρους. Στην περίπτωση που ξεπεραστεί η μέγιστη τιμή αποθήκευσης μιας μεταβλητής, πχ. ως αποτέλεσμα μιας πρόσθεσης, τότε έχουμε αναδίπλωση της τιμής.

Τα σφάλματα αυτού του είδους θεωρούνται κρίσιμα στις περιπτώσεις όπου η τιμή του υπερχειλισμένου ακεραίου χρησιμοποιηθεί αργότερα ως θέση στοιχείου σε πίνακα ή κατά τη δυναμική δέσμευση μνήμης.

Στην παρακάτω περίπτωση, εάν η τιμή του arg1 γίνει ίση με 0xFFFFFFFF (θεωρώντας την χρήση 32-bit ακεραίων) τότε η κλήση της malloc για δυναμική δέσμευση μνήμης θα γίνει με παράμετρο μεγέθους ίση με το μηδέν. Αυτό συμβαίνει καθώς προσθέτοντας ένα στην μέγιστη τιμή ενός ακεραίου αυτός αναδιπλώνεται στο μηδέν. (0xFFFFFFFF + 1 = 0)

```
void func(int arg1, char * arg2)
{
   char * text;
   int size;

   size = arg1 + 1;
   text = malloc(size);
   strcpy(text, arg2);
}
```

1.6.4.5 Σφάλματα Πρόσημου (Signedness Bugs)

Στα σφάλματα πρόσημου μια μεταβλητή δίχως πρόσημο ερμηνεύεται σαν να έχει πρόσημο ή το αντίστροφο. Τα σφάλματα αυτά παρουσιάζονται σε περιπτώσεις όπου ακέραιοι με πρόσημο χρησιμοποιούνται σε συγκρίσεις ή αριθμητικές πράξεις με ακεραίους δίχως πρόσημο, είτε στις περιπτώσεις όπου ένας ακέραιος υπερχειλίζει και αναδιπλώνεται αλλάζοντας πρόσημο.

Όπως και στην περίπτωση των σφαλμάτων υπερχείλισης ακεραίων μεταβλητών η εκμεταλλευσιμότητα αυτών των σφαλμάτων εξαρτάται από την μετέπειτα χρήση της τιμής του ακεραίου.

Στο παρακάτω παράδειγμα, εάν η τιμή του arg3 είναι κάτω από οκτώ τότε η κλήση της memcpy θα γίνει με αρνητική παράμετρο μεγέθους, την τιμή της οποίας η memcpy θα εκλάβει ως μεγάλο θετικό αριθμό προκαλώντας υπερχείλιση μνήμης.

```
void func(void *arg1, void *arg2, int arg3)
{
   int size;
   size =  arg3 - 8;
   memcpy(arg1, arg2, size);
}
```

1.6.4.6 Παράλειψη προσδιοριστή μορφής (Format String Vulnerabilities)

Η παράλειψη προσδιοριστή μορφής σε συναρτήσεις της οικογένειας συναρτήσεων printf μπορεί να έχει ως αποτέλεσμα την διαρροή δεδομένων από την στοίβα ή ακόμη και την αυθαίρετη επικάλυψη μνήμης σε αυτή.

Αυτό συμβαίνει λόγω του τρόπου υλοποίησης των συναρτήσεων της οικογένειας printf. Τα σφάλματα αυτού του είδους θεωρούνται αυτομάτως κρίσιμα [7].

```
void func(char * arg1)
{
    char text[512];
    strncpy(text, arg1, sizeof(dest));
    printf(text);
 }
```

1.6.4.7 Συνθήκες Ανταγωνισμού (Race Conditions)

Τα σφάλματα συνθηκών ανταγωνισμού παρουσιάζονται όταν δύο ή περισσότερες μονάδες εκτέλεσης ανταγωνίζονται για την πρόσβαση στο ίδιο πόρο. Στο παρακάτω παράδειγμα, εάν ένα διαφορετικό νήμα εκτέλεσης αλλάξει την μεταβλητή arg1.buffer_size μετά την εκτέλεση της κλήσης malloc, η κλήση της memcpy που ακολουθεί θα γίνει με διαφορετική παράμετρο μεγέθους.

```
void func(test_struct* arg1)
{
  void * buffer;
  buffer = malloc(arg1.buffer_size);
  // διαθέσιμο παράθυρο χρόνου αλλαγής του
  // arg1.buffer_size από κάποιο διαφορετικό νήμα
  memcpy(buffer, arg1.buffer, arg1.buffer_size);
  return buffer;
 }
```

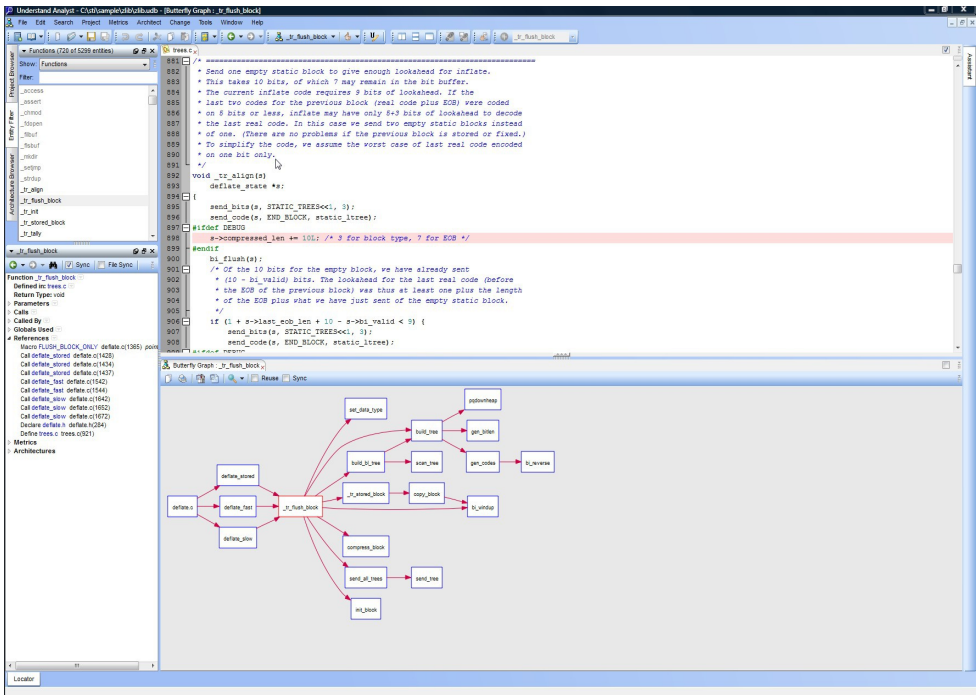
1.7 Μέθοδοι Εύρεσης Ευπαθειών Λογισμικού

Παρακάτω περιγράφονται συνοπτικά οι υφιστάμενοι μέθοδοι εύρεσης σφαλμάτων και εν δυνάμει ευπαθειών λογισμικού.

1.7.1 Εξέταση Πηγαίου Κώδικα (Code Auditing)

Η εξέταση του πηγαίου κώδικα μιας εφαρμογής αποτελεί την πληρέστερη μέθοδο εύρεσης ευπαθειών. Η εξέταση μπορεί να είναι αναλυτική εξετάζοντας τον κώδικα γραμμή προς γραμμή είτε μπορεί να γίνει αυτοματοποιημένα χρησιμοποιώντας εργαλεία ανάλυσης κώδικα για την αναζήτηση συγκεκριμένων τύπων σφαλμάτων.

Λόγω του μεγέθους και της πολυπλοκότητας του κώδικα των σύγχρονων λογισμικών η χρήση εργαλείων πλοήγησης κώδικα είναι επιβεβλημένη. Παραδείγματα αυτού του είδος εργαλείων είναι το OpenGrok[8] και το Understand[9].



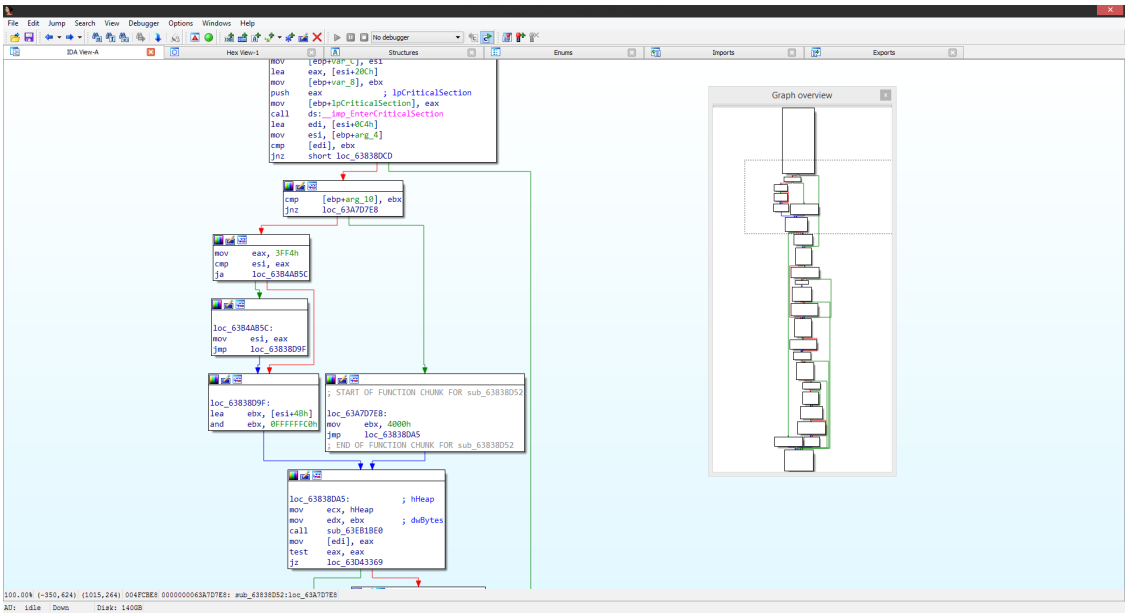
Εικόνα 2: Εξέταση κώδικα μέσω του Scitools Understand

Η δυναμική ανάλυση πηγαίου κώδικα μπορεί να αποφέρει αποτελέσματα αυτόματα, αναζητώντας γνωστά είδη σφαλμάτων στον κώδικα. Παραδείγματα εργαλείων δυναμικής ανάλυσης πηγαίου κώδικα είναι το Coverity[10] και το Joern[11].

1.7.2 Εξέταση Κώδικα Μηχανής(Binary Auditing)

Στην περίπτωση όπου δεν υπάρχει πρόσβαση στον πηγαίο κώδικα ενός λογισμικού εξετάζεται ο κώδικας μηχανής που παράγετε από τον πηγαίο κώδικα για την συγκεκριμένη αρχιτεκτονική που μας ενδιαφέρει. Ακόμη και στις περιπτώσεις όπου υπάρχει ο πηγαίος κώδικας, η εξέταση του κώδικα μηχανής μπορεί να αποκαλύψει σφάλματα που δεν είναι το ίδιο ευδιάκριτα κατά την εξέταση του πηγαίου κώδικα.

Απαραίτητο προαπαιτούμενο αυτής της μεθόδου είναι η χρήση disassembler για την μετατροπή του κώδικα μηχανής σε συμβολική γλώσσα. Επιβεβλημένη είναι επίσης και η χρήση εργαλείων πλοήγησης κώδικα. Το πιο γνωστό εργαλείο που συνδυάζει τα παραπάνω δύο προαπαιτούμενα είναι το IDA Pro[12].



Εικόνα 3: Εξέταση κώδικα μηχανής x86 με το IDA Pro

Αντίστοιχα εργαλεία αυτής της κατηγορίας είναι το radare2[13], το Hopper[14] και το BinNavi[15].

1.7.3 Fuzzing

Η μέθοδος που εξετάζεται στην παρούσα εργασία. Πρόκειται για μια δυναμική μέθοδο η οποία εστιάζει στην εύρεση σφαλμάτων μέσω της επαναλαμβανόμενης αποστολής μη αναμενόμενων δεδομένων στις εισόδους της εφαρμογής που εξετάζεται. Το fuzzing εξετάζεται αναλυτικά στο επόμενο κεφάλαιο.

1.7.4 Συμβολική Εκτέλεση (Symbolic Execution)

Η συμβολική εκτέλεση είναι μια μέθοδος ανάλυσης προγραμμάτων που στοχεύει στον εντοπισμό των δεδομένων εισόδου που προκαλούν την εκτέλεση ενός συγκεκριμένου σημείου κώδικα.

Κατά την συμβολική εκτέλεση ενός προγράμματος τα δεδομένα που εισέρχονται σε αυτό εκλαμβάνονται ως σύμβολα και όχι ως πραγματικές τιμές. Αυτό μας δίνει την δυνατότητα να εκφράσουμε τις τιμές των δεδομένων που απαιτούνται για να φτάσει η εκτέλεση στο σημείο του κωδικά που μας ενδιαφέρει ως ένα πρόβλημα ικανοποίησης περιορισμών. Στην συνέχεια οι περιορισμοί λύνονται χρησιμοποιώντας αλγόριθμους επίλυσης προβλημάτων περιορισμών (constraint solvers). Η λύση του προβλήματος μας παρέχει όλα τα πιθανά δεδομένα που οδηγούν στην εκτέλεση του συγκεκριμένου σημείου κώδικα.

Ενδιαφέρον παρουσιάζει ο συνδυασμός της συμβολικής εκτέλεσης με τις υπόλοιπες μεθόδους εύρεσης ευπαθειών. Στην περίπτωση της μεθόδου του fuzzing η συμβολική εκτέλεση μπορεί να χρησιμοποιηθεί για την παραγωγή test-case, ή, όπως στην περίπτωση του SAGE [16], για να εκτελεστεί η διαδικασία του fuzzing με ευφυή τρόπο.

Δυστυχώς η συμβολική εκτέλεση σύνθετου λογισμικού είναι ακόμη προβληματική και ως εκ τούτου βρίσκει περιορισμένη χρήση σε ρεαλιστικά σενάρια έρευνας.

Παραδείγματα εργαλείων που επιτρέπουν την συμβολική εκτέλεση προγραμμάτων είναι το Klee [17] και το Miasm2 [18].

1.7.5 Taint Analysis

Η μέθοδος του taint analysis επιχειρεί να προσδιορίσει ποια κομμάτια κώδικα επεξεργάζονται δεδομένα των οποίων οι τιμές προέρχονται από δεδομένα εισόδου. Κατά αυτό τον τρόπο είναι εύκολο να εντοπιστούν τα σημεία του κώδικα που επεξεργάζονται τιμές που προέρχονται από μη ασφαλή δεδομένα εισόδου.

Η δυναμική χρήση αυτής της μεθόδου μας επιτρέπει να γνωρίζουμε ποια δεδομένα προέρχονται από δεδομένα εισόδου μια ορισμένη χρονική στιγμή της εκτέλεσης του προγράμματος.

Όπως και στην περίπτωση της συμβολικής εκτέλεσης, η χρήση του taint analysis σε σύνθετες εφαρμογές είναι ακόμη προβληματική.

Ένα εργαλείο χρήσης δυναμικού taint analysis είναι το SemTrax [19].

1.8 Επίλογος

Στο πρώτο κεφάλαιο παρουσιάστηκαν συνοπτικά οι όροι των ευπαθειών συστημάτων και λογισμικού καθώς και οι τρόποι με τους οποίους η εκμετάλλευση τους υποβαθμίζει την ασφάλεια ενός υπολογιστικού συστήματος.

Οι ευπάθειες λογισμικού ταξινομήθηκαν σύμφωνα με το στάδιο ζωής λογισμικού στο οποίο εμφανίζονται με σκοπό να γίνει ευδιάκριτη η κατηγορία των ευπαθειών που εντοπίζεται κάνοντας χρήση της μεθόδου που εξετάζεται στην παρούσα εργασία.

Οι κατηγορίες σφαλμάτων που ανήκουν στην γενικότερη κατηγορία των σφαλμάτων υλοποίησης εξετάστηκαν περαιτέρω καθώς αποτελούν την πλειοψηφία των σφαλμάτων που θα μας απασχολήσουν.

Τέλος παρουσιάστηκαν οι υφιστάμενοι μέθοδοι εύρεσης ευπαθειών λογισμικού συμπεριλαμβανομένης και της μεθόδου του fuzzing, η οποία εξετάζεται αναλυτικά στο επόμενο κεφάλαιο.

2 Μέθοδος Εύρεσης Ευπαθειών: Fuzzing

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο περιγράφεται αναλυτικά η διαδικασία της εύρεσης ευπαθειών λογισμικού χρησιμοποιώντας την μεθόδου του fuzzing, εξετάζοντας ξεχωριστά κάθε ένα από τα στάδια της.

2.2 Ορισμός

Το fuzzing είναι μια μέθοδος εύρεσης σφαλμάτων λογισμικού. Πρόκειται για μια συνήθως αυτοματοποιημένη ή ήμι-αυτοματοποιημένη διαδικασία κατά την οποία μη αναμενόμενα, ή και τελείως τυχαία, δεδομένα αποστέλλονται στις εισόδους μιας εφαρμογής, ενώ ταυτόχρονα παρακολουθείται η πορεία εκτέλεσής της για τυχόν αποκλίσεις κατά την επεξεργασία των αποσταλθέντων δεδομένων.

2.3 Εισαγωγή Στο Fuzzing

Η μέθοδος του fuzzing είναι ένας εύκολος, αυτοματοποιημένος και αποτελεσματικός τρόπος εύρεσης σφαλμάτων σε λογισμικό. Ακόμη και στην πιο απλή μορφή του, το fuzzing μπορεί να αποκαλύψει σφάλματα χωρίς να έχει εξεταστεί ούτε μια γραμμή κώδικα.

Δείγμα της αποτελεσματικότητας του fuzzing είναι η προσθήκη της μεθόδου ως προαπαιτούμενο στάδιο στον κύκλο ανάπτυξης ασφαλούς λογισμικού της Microsoft [20].

Πέρα του τομέα ανάπτυξης λογισμικού, το fuzzing βρίσκει ευρεία χρήση και στο πεδίο της έρευνας ευπαθειών λογισμικού καθώς κάθε σφάλμα που εντοπίζεται είναι και εν δυνάμει ευπάθεια.

Αναλογιζόμενοι το παραπάνω γεγονός, μπορούμε να χαρακτηρίσουμε το fuzzing ως μέθοδο εύρεσης ευπαθειών επεκτείνοντας τον ρόλο του ως μέθοδος εύρεσης σφαλμάτων. Συγκεκριμένα το fuzzing θεωρείται μέθοδος εύρεσης ευπαθειών στις περιπτώσεις όπου τα δεδομένα στα οποία εφαρμόζεται εισέρχονται στην εφαρμογή από ένα όριο εμπιστοσύνης διαφορετικό από αυτό που ανήκει ο κώδικας που τα επεξεργάζεται.

Αξίζει να σημειωθεί πως παρ΄ όλη την αποτελεσματικότητα της, η χρήση της μεθόδου fuzzing δεν αντικαθιστά τις μεθόδους ενδελεχούς εξέτασης του λογισμικού και συνήθως χρησιμοποιείται παράλληλα με την εξέταση κώδικα κατά την διαδικασία της έρευνας ευπαθειών.

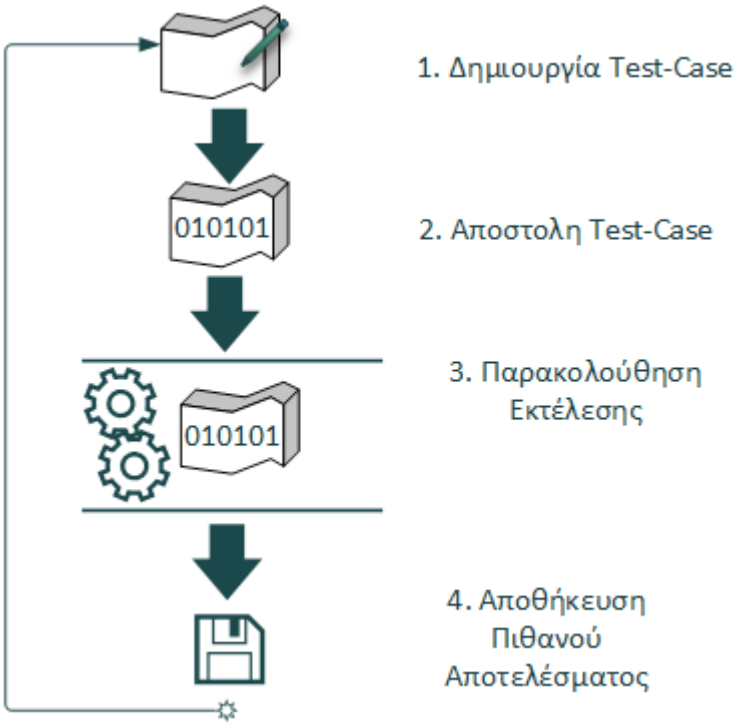
2.4 Διαδικασία Του Fuzzing

Η διαδικασία του fuzzing είναι μια επαναληπτική διαδικασία όπου σε κάθε επανάληψη εκτελούνται τα παρακάτω βήματα:

1. Ανάκτηση/Δημιουργία δεδομένων αποστολής (test-case).
2. Αποστολή του test-case στην εφαρμογή.
3. Παρακολούθηση της εκτέλεσης της εφαρμογής.
4. Αποθήκευση πιθανού αποτελέσματος.

Είναι σημαντικό όλα τα βήματα μιας επανάληψης να μπορούν να αυτοματοποιηθούν ώστε να αυτοματοποιηθεί συνολικά η διαδικασία, ελαχιστοποιώντας την ανάγκη για ανθρώπινη παρέμβαση κατά την διάρκεια της.

Το πρόγραμμα το οποίο υλοποιεί την διαδικασία του fuzzing ονομάζεται fuzzer.



Εικόνα 4: Διαδικασία fuzzing

Στην συνέχεια εξετάζονται ξεχωριστά τα βήματα που εκτελούνται σε κάθε επανάληψη.

2.5 Δημιουργία/Ανάκτηση Test-Case

Η δημιουργία ή ανάκτηση του test-case που πρόκειται να αποσταλεί στην εφαρμογή είναι το πρώτο βήμα που εκτελείτε σε κάθε επανάληψη της διαδικασίας.

Ένα test-case μπορεί να δημιουργείται δυναμικά σε κάθε επανάληψη ή μπορεί να έχει δημιουργηθεί πρωτύτερα και να ανακτάται στην αρχή της.

Η δημιουργία των test-case είναι το σημαντικότερο βήμα όσο αναφορά τις πιθανότητες εύρεσης αποτελεσμάτων.

Παρακάτω περιγράφονται οι πιθανοί μέθοδοι δημιουργίας test-case.

2.5.1 Τυχαία Δεδομένα

Η παραγωγή τυχαίων δεδομένων είναι η πιο απλή μέθοδος δημιουργίας ενός test-case.

Η χρήση αυτής της μεθόδου δεν συνίσταται καθώς οι πιθανότητες μια εφαρμογή να επεξεργάζεται επαρκώς δεδομένα που δεν διέπονται από κάποια λογική δομή είναι μικρές.

```
import random


def random_data(size):
  return "".join(chr(random.randint(0,0xff))
                 for _ in range(size))
...
>>> print_bytearray(random_data(0x20))
0e d7 61 20 13 5b 28 b9 61 bf df 2f ef 1e 79 b5
7f 23 a5 4f 6d 34 02 c0 61 92 ab ee 13 ae 42 f3
```

2.5.2 Μετάλλαξη Δεδομένων (Mutational Fuzzing)

Η μετάλλαξη δεδομένων είναι η πιο δημοφιλής μορφή δημιουργίας test-case καθώς συνδυάζει μικρό χρόνο υλοποίησης με καλές πιθανότητες παραγωγής αποτελεσμάτων.

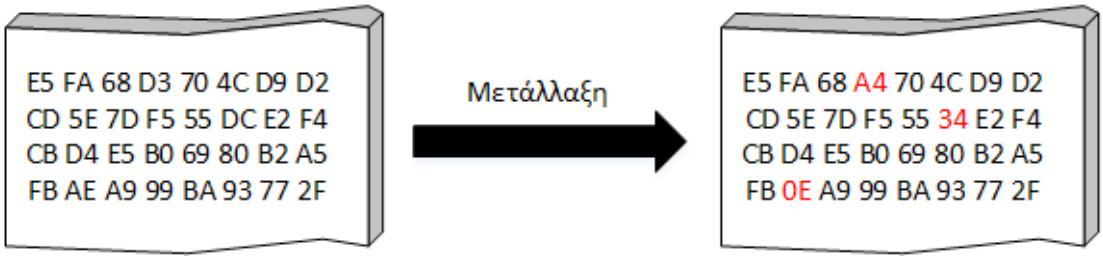
Με την χρήση αυτής της μεθόδου τα test-case που παράγονται είναι αποτέλεσμα μετάλλαξης ήδη υπαρχόντων δεδομένων(seeds), τα οποία γνωρίζουμε πως επεξεργάζεται επιτυχώς η εφαρμογή.

Το παραπάνω γεγονός εγγυάται πως τα παραγόμενα test-case διέπονται από την λογική δομή των δεδομένων που αναμένει η εφαρμογή. Το γεγονός αυτό με την σειρά του αυξάνει τις πιθανότητες η εφαρμογή να επεξεργαστεί επαρκώς το test-case, βελτιώνοντας άμεσα την πιθανότητα εύρεσης κάποιου σφάλματος κατά την επεξεργασία του.

Ο πιο σημαντικός παράγοντας όσο αναφορά την αποτελεσματικότητα της μετάλλαξης δεδομένων είναι η ποιότητα των seeds στα οποία εφαρμόζεται ο αλγόριθμος μετάλλαξης.

Η μετρική που χρησιμοποιείτε για να υπολογιστεί η ποιότητα ενός seed είναι το ποσοστό του κώδικα που εκτελείτε κατά την επεξεργασία του. Όσο περισσότερη κάλυψη κώδικα προκαλεί η επεξεργασία ενός seed τόσο μεγαλύτερο ποσοστό κώδικα αναμένεται να δοκιμαστεί κατά την επεξεργασία του test-case που προέρχεται από την μετάλλαξη του [21].

Οι αλγόριθμοι μετάλλαξης μπορούν να χωριστούν σε δύο κατηγορίες, στους αλγόριθμους που έχουν γνώση περί της δομής των δεδομένων που μεταλλάσσουν, γνωστοί ως αλγόριθμοι έξυπνης μετάλλαξης δεδομένων(smart/intelligent mutation), και σε αυτούς που δεν έχουν, γνωστοί ως αλγόριθμοι απλής μετάλλαξης δεδομένων(dumb mutation).



Εικόνα 5: Μετάλλαξη Δεδομένων

2.5.2.1 Αλγόριθμοι Απλής Μετάλλαξης Δεδομένων (Dumb Mutation)

Οι αλγόριθμοι απλής μετάλλαξης δεδομένων ενεργούν πάνω στα δεδομένα χωρίς να έχουν καμία γνώση για την λογική δομή που τα διέπει. Όλα τα δεδομένα εκλαμβάνονται ως μια σειρά από bytes(ή bits).

Η πιο δημοφιλής μέθοδος απλής μετάλλαξης δεδομένων είναι η τυχαία αλλαγή τιμών byte μέσα από τα αρχικά δεδομένα(random byte mutation).

Όπως παρουσιάζεται παρακάτω, η υλοποίηση αυτού του είδους αλγορίθμου είναι πολύ απλή.

```
def mutate(buf, num_mutations):
  # έλεγχος μεγέθους
  if len(buf) < num_mutations:
    num_mutations = len(buf)

  # επιλογή τυχαίων θέσεων μέσα από τα αρχικά δεδομένα
  off_mutations = random.sample(xrange(len(buf)),
                                num_mutations)
  # αλλαγή των τιμών των byte
  for offset in off_mutations:
    buf[offset] = random.getrandbits(8)

  return buf
```

Ο συγκεκριμένος αλγόριθμος, αν και απλοϊκός, έχει αποδειχθεί πως είναι ιδιαίτερα αποτελεσματικός στην εύρεση σφαλμάτων λογισμικού [22].

2.5.2.2 Αλγόριθμοι Εξυπνης Μετάλλαξης Δεδομένων (Intelligent Mutation)

Η έξυπνη μετάλλαξη δεδομένων είναι η λογική εξέλιξη των αλγορίθμων απλής μετάλλαξης. Οι έξυπνοι αλγόριθμοι μετάλλαξης επιδρούν στοχευμένα πάνω στα υπάρχοντα δεδομένα καθώς διαθέτουν γνώση περί της δομής τους.

Στις περισσότερες περιπτώσεις οι αλγόριθμοι έξυπνης μετάλλαξης αναμένεται να παράγουν περισσότερα και γρηγορότερα αποτελέσματα από τους αλγόριθμους απλής μετάλλαξης.

Όπως είναι φυσικό αυτού του είδους οι αλγόριθμοι απαιτούν περισσότερο χρόνο και προσπάθεια για να χρησιμοποιηθούν καθώς απαιτείται γνώση της δομής των δεδομένων από τον αλγόριθμο.

Μια συνήθης πρακτική είναι η τροφοδότηση του αλγορίθμου με την περιγραφή της δομής των δεδομένων που πρόκειται να μεταλλάξει. Παράδειγμα χρήσης αυτής της πρακτικής αποτελεί ο fuzzer Peach3 [23], όπου η δομή των δεδομένων περιγράφεται στον αλγόριθμο μέσω XML αρχείων.

Η περιγραφή των δεδομένων δεν είναι πάντοτε απαραίτητη καθώς ένας αλγόριθμος μπορεί να έχει ενσωματωμένη την γνώση περί της δομής των δεδομένων που μεταλλάσσει μέσα στην λογική του. Ορισμένοι αλγόριθμοι μάλιστα προσπαθούν να αναγνωρίσουν, ως ένα βαθμό, την δομή των δεδομένων αυτόματα χρησιμοποιώντας ευρεστικές μεθόδους. Ένα παράδειγμα χρήσης αυτής της προσέγγισης είναι ο fuzzer Radamsa [24] .

Στις περιπτώσεις όπου δεν διαθέτουμε τον πηγαίο κώδικα της εφαρμογής, υπάρχουν σενάρια όπου η χρήση έξυπνων έναντι απλών αλγορίθμων μετάλλαξης είναι επιβεβλημένη.

Ένα παράδειγμα αυτού του είδους σεναρίου είναι η ύπαρξη πεδίων ελέγχου ακεραιότητας(πχ. MD5, CRC32) στην δομή των δεδομένων. Μια εφαρμογή είναι πολύ πιθανό να εφαρμόζει τους ελέγχους ακεραιότητας προτού προβεί στη περαιτέρω επεξεργασία των δεδομένων.

Η χρήση αλγορίθμων απλής μετάλλαξης σε αυτή την περίπτωση οδηγεί στην δημιουργία test-case που δεν περνούν τους ελέγχους ακεραιότητας έχοντας ως τελικό αποτέλεσμα την μη επαρκή επεξεργασία τους από την εφαρμογή.

Αντιθέτως, ένας αλγόριθμος έξυπνης μετάλλαξης μπορεί να διορθώσει τις τιμές των πεδίων ακεραιότητας οδηγώντας στην επαρκή επεξεργασία τους.

Επιπλέον σενάρια αυτού του είδους είναι η μετάλλαξη κρυπτογραφημένων και συμπιεσμένων δεδομένων. Αυτά τα σενάρια ωστόσο αντιμετωπίζονται ευκολότερα στο στάδιο αποστολής των δεδομένων, κάνοντας χρήση in-memory fuzzing όπως θα δούμε αργότερα.

```
<!-- Defines the format of a WAV file -->
<DataModel name="Wav">
    <!-- wave header -->
    <String value="RIFF" token="true" />
    <Number size="32" />
    <String value="WAVE" token="true"/>
    <Choice maxOccurs="30000">
        <Block ref="ChunkFmt"/>
        <Block ref="ChunkData"/>
        <Block ref="ChunkFact"/>
        <Block ref="ChunkSint"/>
        <Block ref="ChunkWavl"/>
        <Block ref="ChunkCue"/>
        <Block ref="ChunkPlst"/>
        <Block ref="ChunkLtxt"/>
        <Block ref="ChunkSmpl"/>
        <Block ref="ChunkInst"/>
        <Block ref="Chunk"/>
    </Choice>
</DataModel>
```

Κείμενο 1: Περιγραφή δεδομένων κατάλληλη για χρήση με τον fuzzer Peach3

2.5.3 Παραγωγή Δεδομένων (Generational Fuzzing)

Μια εναλλακτική προσέγγιση στην δημιουργία test-cases είναι η χρήση αλγορίθμων παραγωγής δεδομένων. Αυτού του είδους οι αλγόριθμοι παράγουν όλους τους πιθανούς συνδυασμούς ενός τύπου δεδομένων ακολουθώντας έναν συγκεκριμένο ορισμό της δομής του.

Όπως και στην περίπτωση των αλγορίθμων έξυπνης μετάλλαξης, οι αλγόριθμοι παραγωγής δεδομένων μπορούν είτε να τροφοδοτούνται με την περιγραφή των δεδομένων τα οποία πρόκειται να παράγουν είτε να έχουν ενσωματωμένη την γνώση περί της δομής τους μέσα στην λογική τους.

Αν και θεωρητικά αυτή η μέθοδος είναι η βέλτιστη, στην πράξη είναι αδύνατο να δοκιμαστούν όλα τα πιθανά δεδομένα που μπορεί να δέχεται ένα πρόγραμμα σε ρεαλιστικό χρόνο.

Η πιθανή χρήση αυτής της μεθόδου προϋποθέτει τον προσδιορισμό ενός ρεαλιστικού υποσυνόλου το οποίο θα δοκιμαστεί. Το μέγεθος του υποσυνόλου εξαρτάται κυρίως από τον αριθμό των fuzzer που μπορούμε να διαθέσουμε και την ταχύτητα με την οποία εκτελείται η διαδικασία του fuzzing για την συγκεκριμένη εφαρμογή.

Ένα παράδειγμα fuzzer που κάνει χρήση της μεθόδου παραγωγής δεδομένων (generational fuzzer) είναι ο Peach3.

2.5.4 Χρήση Εξελικτικών Αλγορίθμων (Evolutionary/Genetic Algorithms)

Η διαδικασία δημιουργίας test-case μπορεί να βελτιωθεί σε μεγάλο βαθμό συνδυάζοντας την χρήση εργαλείων instrumentation[25] με την χρήση εξελικτικών αλγορίθμων.

Σε αυτή τη προσέγγιση, γνωστή και ως evolutionary fuzzing, τα test-case που παράγονται είναι ουσιαστικά ο πληθυσμός πάνω στον οποίο ο γενετικός αλγόριθμος εφαρμόζει την λογική του ελιτισμού.

Κατά αυτό τον τρόπο, σε κάθε επανάληψη δημιουργείτε μια νέα γενιά test-case που προέρχεται από τον ανασυνδυασμό ή την διασταύρωση των καλύτερων test-case της προηγούμενης γενιάς. Οι πληροφορίες που χρειάζεται ο γενετικός αλγόριθμος για την αξιολόγηση των test-cases προέρχονται από το εκάστοτε εργαλείο instrumentation που χρησιμοποιείται.

Παραδείγματα fuzzer που κάνουν χρήση γενετικών αλγορίθμων είναι ο Choronzon [26] και ο American Fuzzy Lop[27].

Η χρήση αυτής της μεθόδου απαιτεί την ύπαρξη του πηγαίου κώδικα της εφαρμογής.

2.6 Αποστολή Δεδομένων Στην Εφαρμογή

Επόμενο βήμα μετά την δημιουργία ενός test-case είναι η αποστολή του στην εφαρμογή. Η υλοποίηση αυτού του βήματος εξαρτάται αποκλειστικά από το είδος της εισόδου της εφαρμογής.

Παραδείγματα εισόδων δεδομένων μιας εφαρμογής είναι τα αρχεία, οι θύρες δικτύου και οι παράμετροι γραμμής εντολών.

Το πιο σημαντικό μέρος αυτού του σταδίου είναι η αυτοματοποίηση του. Αν δεν μπορεί να αυτοματοποιηθεί η αποστολή των δεδομένων στην εφαρμογή τότε η διαδικασία του fuzzing δεν μπορεί να εκκινήσει.

Γενικότερα θεωρούμε πως η αποστολή των δεδομένων μπορεί να αυτοματοποιηθεί στις περιπτώσεις που αυτή μπορεί να γίνει είτε μέσω γραμμής εντολών είτε προγραμματιστικά.

2.6.1 Αρχεία

Στην περίπτωση όπου η είσοδος της εφαρμογής που δοκιμάζεται δέχεται αρχεία τότε και τα test-case αποστέλλονται σε αυτή ως αρχεία.

Εφόσον η εφαρμογή δέχεται τα αρχεία εισόδου από την γραμμή εντολών τότε η αποστολή των test-case μπορεί να αυτοματοποιηθεί εύκολα.

Αντιθέτως αν η εφαρμογή δέχεται τα αρχεία εισόδου αποκλειστικά μέσω κάποιου γραφικού περιβάλλοντος τότε είναι απαραίτητη η συγγραφή κώδικα που θα αυτοματοποιεί την διαδικασία.

Παραδείγματα εφαρμογών που δέχονται δεδομένα μέσω αρχείων είναι οι εφαρμογές πολυμέσων, οι επεξεργαστές κειμένου, οι φυλλομετρητές και τα anti-virus.

2.6.2 Θύρες Δικτύου

Οι εφαρμογές που επικοινωνούν με δίκτυα δέχονται δεδομένα μέσω θυρών δικτύου. Επιπλέον οι θύρες δικτύου μπορούν να χρησιμοποιηθούν και για την επικοινωνία εντός του συστήματος, όπως για παράδειγμα γίνεται στην περίπτωση της επικοινωνίας μεταξύ διεργασιών.

Η αυτοματοποίηση της αποστολής των test-case μέσω θυρών δικτύου παρουσιάζει δυσκολίες στην περίπτωση όπου το πρωτόκολλο επικοινωνίας που χρησιμοποιείται απαιτεί sessions ή/και κρυπτογράφηση. Σε αυτές τις περιπτώσεις είναι αναγκαία η μελέτη του πρωτοκόλλου επικοινωνίας ώστε να αυτοματοποιηθεί η αποστολή των δεδομένων.

Παραδείγματα εφαρμογών που δέχονται δεδομένα μέσω θυρών δικτύου είναι οι client/server εφαρμογές που υλοποιούν κάποιο πρωτόκολλο όπως τα http, ftp, ssh, vpn, κτλ, οι φυλλομετρητές και οι εφαρμογές που υλοποιούν RPC.

2.6.3 Παράμετροι Γραμμής Εντολών

Δεδομένα εισόδου μπορούν να σταλούν σε μια εφαρμογή μέσω των παραμέτρων που αυτή δέχεται κατά την εκκίνηση της από την γραμμή εντολών. Σε αυτή την περίπτωση η αυτοματοποίηση της αποστολής των δεδομένων είναι εύκολη.

2.6.4 Δεδομένα Διαμόρφωσης

Μια επιπλέον είσοδος δεδομένων που διαθέτουν σχεδόν όλες οι εφαρμογές είναι οι πηγές από τις οποίες αντλούν τα δεδομένα διαμόρφωσης τους. Παραδείγματα δεδομένων που ανήκουν σε αυτή την κατηγορία είναι οι μεταβλητές περιβάλλοντος και κελύφους σε συστήματα UNIX και οι μεταβλητές που ανήκουν στην registry των συστημάτων Windows.

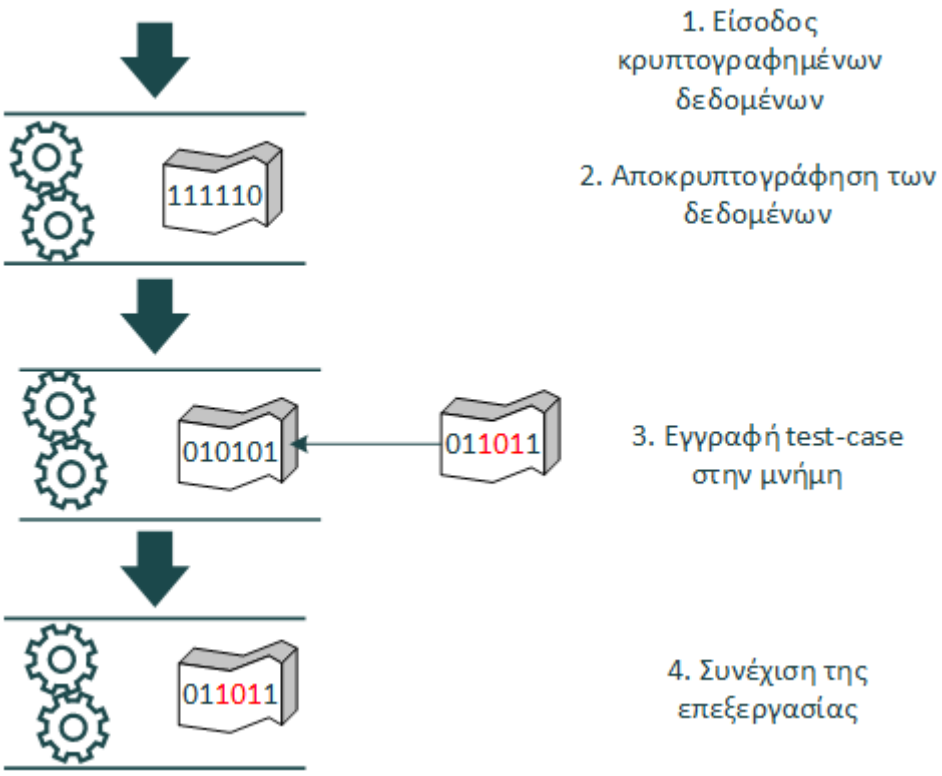
Η αυτοματοποίηση σε αυτές της περιπτώσεις είναι εύκολη καθώς αυτού του είδους τα δεδομένα μπορούν να αλλάξουν από την γραμμή εντολών.

2.6.5 In-memory

Προτού μια εφαρμογή επεξεργαστεί τα δεδομένα που της αποστέλλονται, τα αποθηκεύει στην μνήμη.

Στην περίπτωση που έχουμε εντοπίσει το σημείο στο οποίο η εφαρμογή αποθηκεύει τα δεδομένα που πρόκειται να επεξεργαστεί, μπορούμε να γράψουμε τα δεδομένα του test-case απευθείας στην μνήμη.

Μια περίπτωση όπου η συγκεκριμένη μέθοδος βρίσκει εφαρμογή είναι η αποστολή μη κρυπτογραφημένων δεδομένων σε μια εφαρμογή που αναμένει κρυπτογραφημένα δεδομένα. Εφόσον έχουμε εντοπίσει το σημείο όπου η εφαρμογή επεξεργάζεται την αποκρυπτογραφημένη μορφή των δεδομένων μπορούμε να γράψουμε το test-case που θέλουμε να επεξεργαστεί απευθείας στην μνήμη, αποφεύγοντας έτσι την εξέταση της κρυπτογράφησης που χρησιμοποιεί η εφαρμογή.



Εικόνα 6: Παράδειγμα in-memory fuzzing

2.7 Παρακολούθηση Εφαρμογής

Επόμενο βήμα μετά την αποστολή του test-case στην εφαρμογή είναι η παρακολούθηση της εκτέλεσης της κατά την επεξεργασία των αποσταλθέντων δεδομένων.

Οι ερωτήσεις που πρέπει να απαντηθούν σε αυτό το στάδιο είναι πρώτον, το πότε σταματά η επεξεργασία των αποσταλθέντων δεδομένων, και δεύτερον, σε περίπτωση εύρεσης ενός σφάλματος ποια δεδομένα ή αλληλουχία δεδομένων το προκάλεσαν. Η απάντηση του δεύτερου ζητήματος καθορίζεται σε μεγάλο βαθμό από την απάντηση του πρώτου.

Ο τρόπος παρακολούθησης της εφαρμογής διαφέρει ανά λειτουργικό σύστημα. Για παράδειγμα, στα λειτουργικά συστήματα Windows η παρακολούθηση υλοποιείται μέσω του Windows Debugger Engine API [28] ενώ σε λειτουργικά συστήματα UNIX μέσω των κλήσεων συστήματος wait() [29] και ptrace() [30].

Παρακάτω παρουσιάζεται ένα παράδειγμα κώδικα παρακολούθησης μιας εφαρμογής Windows. Η επικοινωνία με το Windows Debugger Engine API γίνεται μέσω της Python βιβλιοθήκης WinAppDbg [31].

```
def run_and_monitor(cmd, timeout=4):
    # αντικείμενα του WinAppDbg
    handler = DebugEventHandler()
    debug = Debug(handler, bKillOnExit=True)
    try:
        # ορισμός της εντολής εκτέλεσης του προγράμματος
        p = debug.execv(cmd)

        # δημιουργία νήματος παρακολούθησης
        cpu_mon_t = threading.Thread(target=cpu_timeout_mon,
                                     args=(p, timeout))

        # εκκίνηση νήματος παρακολούθησης
        cpu_mon_t.start()
        # εκτέλεση του προγράμματος
        debug.loop()
    finally:
        # τερματισμός του προγράμματος
        debug.stop()
        # έλεγχος πιθανού αποτελέσματος
        if handler.crash is not None:
            return handler.crash
```

2.7.1 Τερματική Συνθήκη Επεξεργασίας Δεδομένων

Το πιο σημαντικό ζήτημα κατά την παρακολούθηση της εφαρμογής είναι το πότε θεωρούμε ότι η εφαρμογή ολοκλήρωσε την επεξεργασία του test-case που αποστείλαμε. Αυτό το ζήτημα είναι ζωτικής σημασίας για την ταχύτητα την οποία εκτελείται η διαδικασία του fuzzing καθώς ορίζει το πότε τελειώνει η παρούσα επανάληψη και ξεκινάει η επόμενη.

Η διαδικασία του fuzzing μπορεί να υλοποιηθεί χωρίς σαφή τερματική συνθήκη επανάληψης, ωστόσο σε αυτή την περίπτωση θα είναι εξαιρετικά δύσκολο να εντοπιστεί το test-case που προκάλεσε το σφάλμα καθώς δεν θα υπάρχει σαφής διαχωρισμός μεταξύ των επαναλήψεων.

Η δυσκολία εντοπισμού της τερματικής συνθήκης ποικίλει από σενάριο σε σενάριο.

Ανεξαρτήτως σεναρίου πρέπει να είμαστε σε θέση να γνωρίζουμε ποια από τις εξής τρεις περιπτώσεις έχει συμβεί στο τέλος της επεξεργασίας των δεδομένων:

1. Δεν βρέθηκε σφάλμα
2. Βρέθηκε σφάλμα
3. Πέρασε το χρονικό όριο της επανάληψης (πιθανό σφάλμα)

Η περίπτωση εύρεσης σφάλματος εντοπίζεται εύκολα καθώς αντιστοιχίζεται με τον απρόσμενο τερματισμό του προγράμματος.

Το μέγιστο χρονικό όριο θέτεται σε κάθε σενάριο ώστε να αποφευχθεί η μονοπώληση της διαδικασίας του fuzzing από την παρούσα επανάληψη.

Σε πολλές περιπτώσεις η εξάντληση του χρονικού ορίου μπορεί να σημαίνει πως υπάρχει κάποιο σφάλμα ατέρμονου βρόγχου στην εφαρμογή που εξετάζουμε. Αν μας ενδιαφέρουν τα σενάρια παραβίασης της διαθεσιμότητας της εφαρμογής τότε μπορούμε να αποθηκεύσουμε το παρόν test-case.

Για την περίπτωση όπου η επεξεργασία των αποσταλθέντων δεδομένων δεν προκαλεί κάποιο σφάλμα, δηλαδή η πιο πιθανή περίπτωση, χρειάζεται να βρούμε έναν τρόπο ώστε να καταλαβαίνουμε το πότε τελειώνει η επεξεργασία.

2.7.2 Παραδείγματα Περιπτώσεων

Στις περιπτώσεις όπου η εφαρμογή δέχεται τα δεδομένα εισόδου και έπειτα επιστρέφει κατά την ολοκλήρωση της επεξεργασίας τους, όπως συμβαίνει για παράδειγμα στην πλειοψηφία των UNIX εφαρμογών, τότε είναι σαφές το πότε τερματίζει μια επανάληψη καθώς σίγουρα θα συμβεί ένα από τα παρακάτω:

1. Επέστρεψε η εφαρμογή (δεν βρέθηκε σφάλμα)
2. Η εφαρμογή τερμάτισε απροσδόκητα (βρέθηκε σφάλμα)
3. Πέρασε το χρονικό όριο της επανάληψης (πιθανό σφάλμα)

Υπάρχουν σενάρια όπου η συνθήκη τερματισμού της επανάληψης δεν είναι τόσο σαφής όσο στην παραπάνω περίπτωση. Εάν για παράδειγμα η εφαρμογή δέχεται δεδομένα εισόδου όμως δεν επιστρέφει όταν ολοκληρώσει την επεξεργασία τους, όπως συμβαίνει στις περισσότερες εφαρμογές με γραφικό περιβάλλον, τότε χρειάζεται να αποφασίσουμε πότε τελειώνει μια επανάληψη βασιζόμενοι σε άλλους παράγοντες. Μια πιθανή λύση είναι η εξής:

1. Η εφαρμογή χρησιμοποιεί 0% από τον χρόνο του επεξεργαστή

   (δεν βρέθηκε σφάλμα)
2. Η εφαρμογή τερμάτισε απροσδόκητα (βρέθηκε σφάλμα)
3. Πέρασε το χρονικό όριο της επανάληψης (πιθανό σφάλμα)

Η παραπάνω λύση βασίζεται στο γεγονός πως η εφαρμογή θα σταματήσει να χρησιμοποιεί τον επεξεργαστή όταν τελειώσει την επεξεργασία των αποσταλθέντων δεδομένων. Αυτή η λύση χρησιμοποιείται από fuzzers όπως ο FOE2 [32].

Όπως μπορεί κανείς να διαπιστώσει η τερματική συνθήκη εξαρτάται σε μεγάλο βαθμό από την ίδια την εφαρμογή που εξετάζεται. Τερματική συνθήκη μπορεί να θεωρηθεί για παράδειγμα η δημιουργία ενός νέου παραθύρου σε ένα γραφικό

περιβάλλον ή στην περίπτωση μιας εφαρμογής διακομιστή η απάντηση που στέλνει πίσω σε εμάς.

Μια γενική, πρακτική, ωστόσο χρονοβόρα λύση, είναι να εντοπίσουμε ένα σημείο του κώδικα το οποίο εκτελείτε μόνο εφόσον έχει τελειώσει η επεξεργασία. Κατά αυτό το τρόπο μπορούμε να γνωρίζουμε πως η επεξεργασία τελείωσε μόλις η εκτέλεση του προγράμματος φτάσει στο συγκεκριμένο σημείο.

1. Εκτελέστηκε το σημείο του κώδικα (δεν βρέθηκε σφάλμα)
2. Η εφαρμογή τερμάτισε απροσδόκητα (βρέθηκε σφάλμα)
3. Πέρασε το χρονικό όριο της επανάληψης (πιθανό σφάλμα)

2.7.3 Σύνδεση Test Case – Σφάλματος

Η αυτοματοποίηση της σύνδεσης test-case και σφάλματος εξαρτάται αποκλειστικά από το εάν υπάρχει σαφής διαχωρισμός μεταξύ των επαναλήψεων.

Στην περίπτωση που δεν έχει βρεθεί σαφή τερματική συνθήκη τότε υπάρχουν δυο επιλογές.

Μπορεί να θυσιαστεί η ταχύτητα της διαδικασίας παραλείποντας την ύπαρξη σαφούς τερματικής συνθήκης, έτσι ώστε κάθε επανάληψη που δεν βρίσκει σφάλμα να εκτελείτε πάντα στο μέγιστο χρονικό όριο:

1. Η εφαρμογή τερμάτισε απροσδόκητα (βρέθηκε σφάλμα)
2. Πέρασε το χρονικό όριο της επανάληψης (δεν βρέθηκε σφάλμα)

Σε αυτή την περίπτωση, εκτός από την αργή εκτέλεση της διαδικασίας του fuzzing αγνοούμε και τα πιθανά σφάλματα ατέρμων βρόγχων.

Εναλλακτικά τα test-case μπορούν να αποστέλλονται χωρίς σαφή διαχωρισμό μεταξύ των επαναλήψεων, διατηρώντας έτσι την ταχύτητα της διαδικασίας έναντι της δυνατότητας να συνδέονται αυτόματα τα πιθανά σφάλματα με τα test-case που τα προκαλούν .

Στην περίπτωση αυτή μπορούμε να κρατάμε έναν μεγάλο αριθμό test-case που έχουν αποσταλεί ώστε να έχουμε την δυνατότητα να επαναλάβουμε την διαδικασία αρχικά με τα ίδια test-case και στην συνέχεια με όλο και λιγότερα ελπίζοντας πως θα καταλήξουμε στην αλληλουχία δεδομένων που προκάλεσαν το σφάλμα.

2.7.4 Χρήση Εργαλείων Instrumentation

Όπως αναφέρθηκε στο κεφάλαιο 2.7.1, ο εντοπισμός σφαλμάτων κατά την διαδικασία του fuzzing αντιστοιχίζεται με τον απρόσμενο τερματισμό του προγράμματος που εξετάζεται.

Στις περισσότερες περιπτώσεις, η χρονική στιγμή την οποία εκτελείτε ο κώδικας που περιέχει το σφάλμα και η χρονική στιγμή του απρόσμενου τερματισμού της εφαρμογής δεν ταυτίζονται. Κατά την διάρκεια του fuzzing, αυτό το γεγονός οδηγεί στην παραγωγή φαινομενικά διαφορετικών αποτελεσμάτων, τα οποία όμως επί της ουσίας προέρχονται από το ίδιο σφάλμα.

Η χρήση εργαλείων instrumentation επιλύει το παραπάνω πρόβλημα, επιβάλλοντας τον απρόσμενο τερματισμό της εφαρμογής ακριβώς την στιγμή την οποία εκτελείται το σφάλμα.

Κατά αυτό τον τρόπο μειώνεται ο φόρτος της μετέπειτα ανάλυσης των αποτελεσμάτων από τον ερευνητή καθώς μειώνεται ο αριθμός των στιγμιοτύπων μνήμης που χρειάζεται να αναλύσει.

Παρακάτω παρουσιάζονται παραδείγματα εργαλείων instrumentation.

2.7.4.1 LLVM Sanitizers

Οι LLVM Sanitizers[33] είναι μια συλλογή εργαλείων instrumentation που βοηθούν στον έγκαιρο εντοπισμό σφαλμάτων σε προγράμματα μεταφρασμένα από τον μεταφραστή LLVM και GCC(για εκδόσεις μεγαλύτερες από 4.8).

Οι LLVM Sanitizers περιλαμβάνουν τα παρακάτω εργαλεία instrumentation:

- Address Sanitizer: Έγκαιρος εντοπισμός σφαλμάτων διαχείρισης μνήμης.
- Thread Sanitizer: Έγκαιρος εντοπισμός σφαλμάτων συνθηκών ανταγωνισμού.
- Memory Sanitizer: Έγκαιρος εντοπισμός σφαλμάτων προσπέλασης μη αρχικοποιημένων δεδομένων.

- Undefined Behavior Sanitizer: Έγκαιρος εντοπισμός σφαλμάτων που προκαλούν την μη ορισμένη συμπεριφορά του προγράμματος.

Παρακάτω παρουσιάζεται ένα παράδειγμα εντοπισμού σφάλματος χρήσης αποδεσμευμένης μνήμης από τον Address Sanitizer.

```
 ==9442== ERROR: AddressSanitizer heap-use-after-free on
address 0x7f7ddab8c084 at pc 0x403c8c bp 0x7fff87fb82d0
sp 0x7fff87fb82c8

READ of size 4 at 0x7f7ddab8c084 thread T0

    #0 0x403c8c in main example_UseAfterFree.cc:4

    #1 0x7f7ddabcac4d in __libc_start_main ??:0

0x7f7ddab8c084 is located 4 bytes inside of 400-byte
region [0x7f7ddab8c080,0x7f7ddab8c210)

freed by thread T0 here:

    #0 0x404704 in operator delete[](void*) ??:0

    #1 0x403c53 in main example_UseAfterFree.cc:4

    #2 0x7f7ddabcac4d in __libc_start_main ??:0

previously allocated by thread T0 here:

    #0 0x404544 in operator new[](unsigned long) ??:0

    #1 0x403c43 in main example_UseAfterFree.cc:2

    #2 0x7f7ddabcac4d in __libc_start_main ??:0

==9442== ABORTING
```

2.7.4.2 Valgrind

Το Valgrind[34] είναι ένα instrumentation framework για την υλοποίηση εργαλείων instrumentation σε Linux. Το Valgrind περιλαμβάνει έτοιμα προς χρήση εργαλεία όπως το Memcheck και το Helgrind.

Παρακάτω παρουσιάζεται ένα παράδειγμα εντοπισμού σφάλματος υπερχείλισης μνήμης από τον εργαλείο Memcheck.

```
  ==19182== Invalid write of size 4
  ==19182==    at 0x804838F: f (example.c:6)
  ==19182==    by 0x80483AB: main
(example.c:11)
  ==19182==  Address 0x1BA45050 is 0 bytes
after a block of size 40 alloc'd
  ==19182==    at 0x1B8FF5CD: malloc
(vg_replace_malloc.c:130)
  ==19182==    by 0x8048385: f (example.c:5)
  ==19182==    by 0x80483AB: main
(example.c:11)
```

2.7.4.3 PageHeap

Το PageHeap[35] είναι ένα εργαλείο instrumentation για προγράμματα που χρησιμοποιούν τον default διαχειριστή σωρού των Windows(Low Fragmentation Heap). Η χρήση του PageHeap επιτρέπει τον έγκαιρο εντοπισμό σφαλμάτων διαχείρισης της μνήμης σωρού.

Παρακάτω παρουσιάζεται ένα παράδειγμα εντοπισμού σφάλματος χρήσης αποδεσμευμένης μνήμης από το PageHeap.

```
DllGetClassObject+0x1e455:
10061011 663906  cmp word ptr [esi],ax ds:002b:007d0000=????


0:007> !heap -p -a esi
  address 007d0000 found in
  _DPH_HEAP_ROOT @ 3b1000
  in free-ed allocation(DPH_HEAP_BLOCK:  VirtAddr   VirtSize)
                        3b4e38           7d0000     2000
```

2.8 Αναφορά και Αποθήκευση Αποτελεσμάτων

Το τελευταίο βήμα μιας επανάληψης της διαδικασίας fuzzing αφορά την αναφορά και αποθήκευση των πιθανών αποτελεσμάτων. Αποτελέσματα θεωρούνται οι πληροφορίες από το στιγμιότυπο του τερματισμένου προγράμματος(crash) που μας ενδιαφέρουν, το test-case που προκάλεσε τον απρόσμενο τερματισμό καθώς και τα επιπλέον στατιστικά στοιχεία που επιθυμούμε να αποθηκεύσουμε.

Μπορούμε για παράδειγμα να αποθηκεύσουμε τον αλγόριθμο με τον οποίο δημιουργήθηκε το test-case, ή τον χρόνο που χρειάστηκε για να βρεθεί το σφάλμα.

Σε αυτό το στάδιο αποφασίζεται ποιες πληροφορίες θα αποθηκευτούν από το στιγμιότυπο του τερματισμένου προγράμματος στην μνήμη, καθώς και ο τρόπος με τον οποίο θα αποθηκευτούν.

Επιπλέον σε αυτό το στάδιο μπορεί να γίνει χρήση αυτοματοποιημένων εργαλείων που παράγουν επιπλέον πληροφορίες, με σκοπό να διευκολυνθεί η μετέπειτα διαδικασία ανάλυσης και ταξινόμησης του σφάλματος που εντοπίστηκε.

2.8.1 Δομή Των Αποτελεσμάτων

Έχοντας το στιγμιότυπο του τερματισμένου προγράμματος στην μνήμη μπορούμε να αποφασίσουμε ποιες πληροφορίες αξίζει να εξάγουμε από αυτό. Οι πληροφορίες που μας είναι χρήσιμες περιλαμβάνουν οι τιμές των καταχωρητών του επεξεργαστή, την συμβολική απεικόνιση του κώδικα(disassembly) στον οποίο σταμάτησε η εκτέλεση, το στιγμιότυπο της στοίβας και στην περίπτωση που διαθέτουμε τον πηγαίο κώδικα της εφαρμογής, το σημείο του πηγαίου κώδικα που σταμάτησε η εκτέλεση.

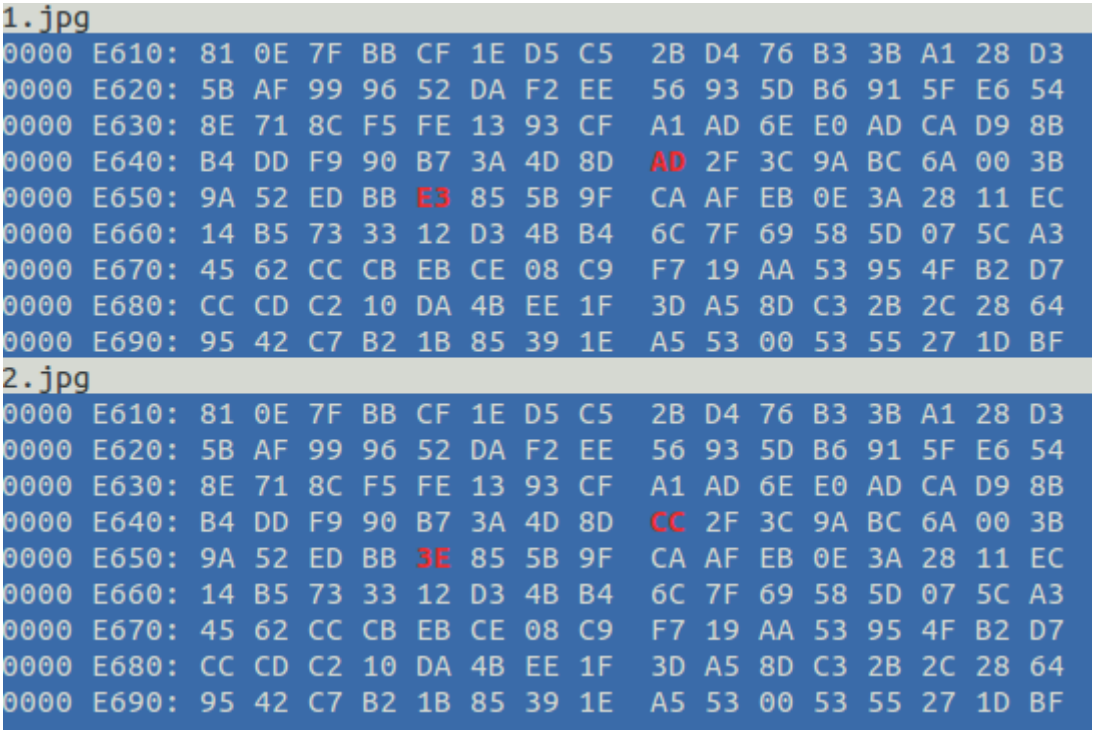
Ένα παράδειγμα δομής αποτελεσμάτων παρουσιάζεται παρακάτω.

```
Access violation (second chance) at start+0x56cc22

Registers:

eax=0b843dba ebx=ffffffff ecx=00000000 edx=01b96100
esi=07af6aa8 edi=00000000

eip=01b79a03 esp=0034f028 ebp=0034f028 iopl=0         no up
ei ng nz na pe cy

cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00010287

Code disassembly:

   01B799FB |   75f3 | jnz start+0x56cc0f

   01B799FD |   33c0 | xor eax, eax

   01B799FF |     c3 | ret

   01B79A00 |     55 | push ebp

   01B79A01 |   8bec | mov ebp, esp

 * 01B79A03 | 8b410c | mov eax, [ecx+0xc]

   01B79A06 |   85c0 | test eax, eax

   01B79A08 |   7412 | jz start+0x56cc3b

   01B79A0A | 8b4d08 | mov ecx, [ebp+0x8]

   01B79A0D | 8d4900 | lea ecx, [ecx+0x0]

   01B79A10 | 394824 | cmp [eax+0x24], ecx

Stack trace:

Frame    Origin

0034F028 01B8AC5A (start+0x57de79)
```

Στην περίπτωση όπου υπάρχει σαφής διαχωρισμός των επαναλήψεων της διαδικασίας του fuzzing, αποθηκεύεται και το test-case που προκάλεσε τον απρόσμενο τερματισμό του προγράμματος(trigger).

Εάν η παραγωγή του test-case είναι αποτέλεσμα αλγόριθμου μετάλλαξης τότε είναι χρήσιμο να αποθηκεύσουμε και τα αρχικά δεδομένα τα οποία μετάλλαχθηκαν ώστε να έχουμε την δυνατότητα να εντοπίσουμε τα ακριβή δεδομένα που προκαλούν το σφάλμα συγκρίνοντας τα δύο αρχεία μεταξύ τους.

Στην παρακάτω εικόνα παρουσιάζεται ένα παράδειγμα σύγκρισης αρχικών δεδομένων - test-case κάνοντας χρήση του προγράμματος vbindiff [36].



Εικόνα 7: Παράδειγμα χρήσης vbindiff

2.8.2 Επιπλέον Συλλογή Πληροφοριών

Επιπλέον πληροφορίες μπορούν να παραχθούν βασιζόμενοι στα αρχικά δεδομένα που εξάγουμε από το στιγμιότυπο μνήμης του τερματισμένου προγράμματος. Χρήσιμες πληροφορίες αυτού του είδους είναι η ταυτοποίηση του στιγμιότυπου μνήμης του τερματισμένου προγράμματος καθώς και ο υπολογισμός της κρισιμότητας του σφάλματος που το προκάλεσε.

2.8.2.1 Ταυτοποίηση/Μοναδικότητα Στιγμιότυπου

Η ταυτοποίηση ενός στιγμιότυπου μνήμης έχει ως σκοπό να καταλάβουμε εάν δύο διαφορετικά στιγμιότυπα του τερματισμένου προγράμματος αφορούν το ίδιο σφάλμα.

Η επιτυχημένη ταυτοποίηση ενός στιγμιότυπου μας αποτρέπει από το να αποθηκεύσουμε αποτελέσματα που αφορούν το ίδιο σφάλμα περισσότερες από μια φορές.

Η ταυτοποίηση ενός στιγμιότυπου είναι εξαιρετικά χρήσιμη στις περιπτώσεις όπου το fuzzing μιας εφαρμογής παράγει μεγάλο αριθμό αποτελεσμάτων.

Στην περίπτωση όπου χρησιμοποιούνται διαφορετικοί fuzzer κατά την διαδικασία του fuzzing η χρήση ενός εργαλείου ταυτοποίησης εγγυάται την ομοιογενή αποθήκευση των αποτελεσμάτων που παράγονται.

Ένα εργαλείο ταυτοποίησης στιγμιοτύπων για Windows είναι το BugId [37].

2.8.2.2 Κρισιμότητα Σφάλματος

Εφαρμόζοντας ευρεστικές μεθόδους στα δεδομένα που εξάγουμε από το στιγμιότυπο μνήμης του τερματισμένου προγράμματος μπορούμε να αποδώσουμε έναν αρχικό δείκτη κρισιμότητας στο σφάλμα που το προκάλεσε.

Το πιο δημοφιλές εργαλείο αυτού του είδους είναι το !exploitable [38] το οποίο ταξινομεί την κρισιμότητα ενός σφάλματος αναλύοντας το στιγμιότυπο μνήμης του τερματισμένου προγράμματος. Οι πιθανές τιμές κρισιμότητας που αποδίδει σε ένα σφάλμα είναι οι εξής:

- NOT EXPLOITABLE
- PROBABLY NOT EXPLOITABLE
- PROBABLY EXPOITABLE
- EXPLOITABLE
- UNKNOWN

Ο δείκτης κρισιμότητας ενός σφάλματος μας δίνει μόνο μια αρχική εκτίμηση για το εάν το σφάλμα μπορεί να γίνει εκμεταλλεύσιμο ή όχι. Η τελική διαπίστωση μπορεί να γίνει μόνο από κάποιον ερευνητή.

2.8.3 Τρόπος Αποθήκευσης Αποτελεσμάτων

2.8.3.1 Αποθήκευση Σε Αρχεία

Ο πιο κοινός τρόπος αποθήκευσης των αποτελεσμάτων είναι μέσω αρχείων. Τα αρχεία αποτελεσμάτων μπορούν να αποθηκεύονται είτε τοπικά είτε να αποστέλλονται σε κάποιο κεντρικό διακομιστή μέσω FTP, HTTP, SSH ή άλλου πρωτοκόλλου.

2.8.3.2 Αποθήκευση Σε Βάση Δεδομένων

Η αποθήκευση των αποτελεσμάτων σε βάση δεδομένων αποτελεί μια ελκυστική λύση, ειδικότερα στις περιπτώσεις όπου εκτελούνται περισσότεροι του ενός fuzzer και παράγεται μεγάλος αριθμός αποτελεσμάτων.

Οι βάσεις δεδομένων είναι σχεδιασμένες ώστε να λύνουν τα περισσότερα προβλήματα που παρουσιάζονται κατά την αποθήκευση δεδομένων που προέρχονται από πολλούς παράλληλα εκτελούμενους fuzzers και ως εκ τούτου η χρήση τους προτιμάτε στις περιπτώσεις που το fuzzing εφαρμόζεται κατανεμημένα.

2.9 Προετοιμασία Fuzzing

Έχοντας αναλύσει την διαδικασία του fuzzing και τα περαιτέρω στάδια της, σε αυτό το υποκεφάλαιο εξετάζονται οι ενέργειες που είναι προαπαιτούμενες για την αποτελεσματική χρήση της μεθόδου και λαμβάνουν μέρος προτού εκκινήσει η διαδικασία.

2.9.1 Ανίχνευση Των Εισόδων Της Εφαρμογής

Προτού ξεκινήσει η διαδικασία του fuzzing μιας εφαρμογής χρειάζεται να ταυτοποιηθούν οι είσοδοι από τις οποίες η εφαρμογή δέχεται δεδομένα.

Σε αυτό το στάδιο, εφόσον είναι δυνατόν, εντοπίζεται και ο τύπος των δεδομένων που εισέρχονται στην εφαρμογή.

Στην περίπτωση της εύρεσης ευπαθειών μας ενδιαφέρουν μόνο οι είσοδοι της εφαρμογής που μπορούν να χρησιμοποιηθούν ως attack vectors [39].

2.9.2 Επιλογή Μεθόδου Δημιουργίας Test Cases

Επόμενο βήμα είναι η επιλογή της μεθόδου δημιουργίας των test-cases που θα αποστέλλονται στις εισόδους της εφαρμογής.

Οι παράγοντες που επηρεάζουν την απόφαση αυτή είναι ο τύπος των δεδομένων που λαμβάνει η εφαρμογή, η γνώση που έχουμε για την δομή των δεδομένων την δεδομένη χρονική στιγμή καθώς και ο χρόνος που διαθέτουμε για την επιλογή αυτή.

2.9.3 Δημιουργία Corpus

Στην περίπτωση όπου αποφασιστεί η χρήση αλγόριθμων μετάλλαξης δεδομένων είναι απαραίτητο να συλλέξουμε τα αρχικά δεδομένα στα οποία θα επιδρούν οι αλγόριθμοι. Το σύνολο των αρχικών δεδομένων που παρέχουμε στον fuzzer αναφέρεται ως corpus.

Η συλλογή των δεδομένων μπορεί να προέρχεται από οπουδήποτε. Η πιο συνήθης πηγή δεδομένων είναι το web, όπου μπορούμε να κατεβάσουμε χειρωνακτικά έναν αριθμό δεδομένων είτε να υλοποιήσουμε ένα web crawler που θα κατεβάσει τα δεδομένα για εμάς.

Ένας επιπλέον τρόπος επέκτασης του corpus είναι η αποθήκευση των δεδομένων που παράγονται κατά την χρήση fuzzers που κάνουν χρήση γενετικών αλγορίθμων όπως για παράδειγμα ο Choronzon και ο AFL. Κατά αυτό τον τρόπο η ποιότητα των δεδομένων είναι ως ένα βαθμό εγγυημένη καθώς αυτού του είδους οι fuzzers παράγουν δεδομένα προσπαθώντας να μεγιστοποιήσουν την κάλυψη κώδικα.

Μια σημαντική μεταβλητή κατά την συλλογή του corpus είναι το μέγεθος του κάθε test-case. Είναι προτιμότερο ένα test-case να είναι όσο το δυνατόν μικρότερο σε μέγεθος ώστε οποιοσδήποτε αλγόριθμος μετάλλαξης να έχει όσο το δυνατόν μεγαλύτερη επίδραση πάνω στην δομή των δεδομένων με όσο το δυνατόν λιγότερες αλλαγές.

Επιπλέον, τα μικρότερα σε μέγεθος test-case επιβαρύνουν σε μικρότερο βαθμό την ταχύτητα της διαδικασίας του fuzzing ,καθώς η προσπέλαση και η επεξεργασία τους από την εφαρμογή είναι γρηγορότερη.

Υπάρχουν διαθέσιμα εργαλεία που αυτοματοποιούν την διαδικασία ελαχιστοποίησης του μεγέθους των test-case χρησιμοποιώντας μεθόδους instrumentation ώστε να βρίσκουν το μικρότερο απαιτούμενο μέγεθος που προκαλεί την μεγαλύτερη κάλυψη κώδικα κατά την επεξεργασία των δεδομένων. Παράδειγμα τέτοιου είδους εργαλείου είναι το afl-tmin [40].

Τέλος, στην περίπτωση που έχουμε συλλέξει μεγάλο αριθμό δεδομένων είναι θετικό να καταλήξουμε στο ελάχιστο υποσύνολο των δεδομένων που οδηγεί στην μέγιστη κάλυψη κώδικα.

Αυτό το υποσύνολο μπορεί να βρεθεί αυτόματα χρησιμοποιώντας εργαλεία όπως το Minset [41] του Peach3 ή το afl-cmin utility [42] του AFL.

```
>peachminset -s pinsamples -m minset -t traces bin\pngcheck.exe %%s


] Peach 3 -- Minset

] Copyright (c) Deja vu Security


[*] Running both trace and coverage analysis

[*] Running trace analysis on 15 samples...

...

[*] Finished

[*] Running coverage analysis...

[-]   3 files were selected from a total of 15.

[*] Copying over selected files...

...

[*] Finished
```

Κείμενο 2: Παράδειγμα χρήσης του εργαλείου Peach3 Minset

2.9.4 Περιγραφή Δομής Δεδομένων

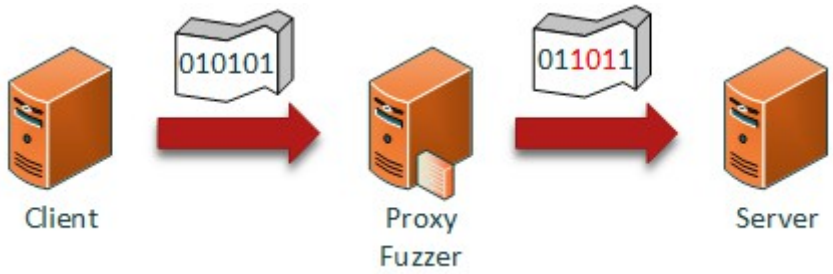
Εάν αποφασίσουμε πως τα test-case θα παράγονται μέσω έξυπνων αλγορίθμων μετάλλαξης ή μέσω παραγωγής δεδομένων τότε ίσως χρειάζεται να δημιουργήσουμε την κατάλληλη περιγραφή των δεδομένων.

Εξαρτώμενη από την πολυπλοκότητα και την ύπαρξη ή μη του specification των δεδομένων, η διαδικασία αυτή μπορεί να αποδειχθεί ιδιαιτέρως χρονοβόρα.

2.9.5 Proxy Based Fuzzing

Στις περιπτώσεις όπου τα δεδομένα που θέλουμε να μεταλλάξουμε είναι πακέτα δικτύου τότε μας δίνεται η δυνατότητα να συλλέγουμε και να μεταλλάσσουμε τα δεδομένα δυναμικά χρησιμοποιώντας τον fuzzer μας ως proxy.

Κατά αυτό τον τρόπο τοποθετούμε τον fuzzer μας ανάμεσα στα συνδιαλεγόμενα μέλη μεταλλάσσοντας τα πακέτα της μεταξύ τους επικοινωνίας.



Εικόνα 8: Εικόνα Proxy Fuzzer

Αξίζει να σημειωθεί πως η χρήση proxy based fuzzing δυσκολεύει τον εντοπισμό τερματικής συνθήκης επεξεργασίας των δεδομένων, ιδιαίτερα στις περιπτώσεις όπου το πρωτόκολλο επικοινωνίας είναι stateless(πχ. UDP), και στις περιπτώσεις όπου η εφαρμογή είναι multi-threaded.

2.9.6 Επιλογή Εργαλείων Instrumentation

Η χρήση εργαλείων instrumentation είναι πάντα επιθυμητή καθώς βοηθά στον εντοπισμό σφαλμάτων. Κατά την προετοιμασία του fuzzing χρειάζεται να προσδιοριστούν τα προαπαιτούμενα για την χρήση του εκάστοτε εργαλείου.

Συνήθη παραδείγματα αυτών των προαπαιτούμενων είναι η ύπαρξη του πηγαίου κώδικα της εφαρμογής που εξετάζεται ή η χρήση ενός συγκεκριμένου διαχειριστή μνήμης σωρού.

2.9.7 Επιλογή Εργαλείων Επεξεργασίας Των Αποτελεσμάτων

Προτού εκκινήσει η διαδικασία του fuzzing χρειάζεται να αποφασιστεί ποια εργαλεία επεξεργασίας των αποτελεσμάτων θα χρησιμοποιηθούν.

Η χρήση εργαλείων υπολογισμού της μοναδικότητας και της κρισιμότητας των σφαλμάτων που εντοπίζονται συνίσταται σε όλες τις περιπτώσεις.

Επιπλέον εργαλεία αυτού του είδους είναι τα εργαλεία αυτόματης εύρεσης των ελάχιστων απαιτούμενων διαφορών μεταξύ ενός trigger και του seed από το οποίο

προήλθε, και τα εργαλεία που μεταλλάσσουν εκ νέου ένα trigger ώστε να διαπιστωθεί εάν οι μεταλλαγμένες τιμές επηρεάζουν περαιτέρω τις τιμές των καταχωρητών του επεξεργαστή στο στιγμιότυπο μνήμης του τερματισμένου προγράμματος.

2.9.8 Επιλογή Τρόπου Αποθήκευσης Αποτελεσμάτων

Τέλος επιλέγεται ο τρόπος με τον οποίο θα αποθηκεύονται τα πιθανά αποτελέσματα. Η κύρια μεταβλητή της απόφασης αυτής είναι ο αριθμός των fuzzers που επιθυμούμε να εκτελεστούν και το είδος των αποτελεσμάτων που επιθυμούμε να συλλέξουμε.

2.9.9 Επιπλέον Επιλογές

2.9.9.1 Fuzzing Βιβλιοθηκών Λογισμικού

Στην περίπτωση που έχουν εντοπιστεί οι βιβλιοθήκες λογισμικού που χρησιμοποιεί η εφαρμογή υπάρχει η δυνατότητα να εκτελεστεί fuzzing απ΄ ευθείας σε αυτές.

Το γεγονός πως υπάρχει πρόσβαση στον πηγαίο κώδικα των βιβλιοθηκών επιτρέπει την χρήση των περισσοτέρων εργαλείων instrumentation καθώς και τον εύκολο εντοπισμό της συνθήκης τερματισμού. Για το fuzzing μιας βιβλιοθήκης απαιτείται η συγγραφή ενός, συνήθως μικρού σε έκταση, προγράμματος που θα χρησιμοποιεί την βιβλιοθήκη.

Ένα επιπλέον θετικό στοιχείο του fuzzing βιβλιοθηκών είναι το γεγονός πως τα σφάλματα που πιθανώς θα βρεθούν αφορούν όλες τις εφαρμογές που χρησιμοποιούν την βιβλιοθήκη και όχι μόνο την εφαρμογή που εξετάζεται.

2.9.9.2 Έλεγχοι Ακεραιότητας

Στην περίπτωση όπου διαθέτουμε τον πηγαίο κώδικα της εφαρμογής είναι θεμιτό να αφαιρέσουμε τους ελέγχους ακεραιότητας δεδομένων από τον κώδικα. Κατά αυτό τον τρόπο δεν απαιτείται η διόρθωση των πεδίων ακεραιότητας στο στάδιο δημιουργίας test-case, βελτιώνοντας την ταχύτητα της διαδικασίας.

2.9.9.3 Διόρθωση Σφαλμάτων Της Εφαρμογής (Patching)

Στις περιπτώσεις εντοπισμού ενός σφάλματος που προκαλείτε νωρίς κατά την επεξεργασία των δεδομένων είναι θεμιτή η διόρθωση του και η μετέπειτα επανεκκίνηση της διαδικασίας του fuzzing χρησιμοποιώντας την διορθωμένη έκδοση της εφαρμογής.

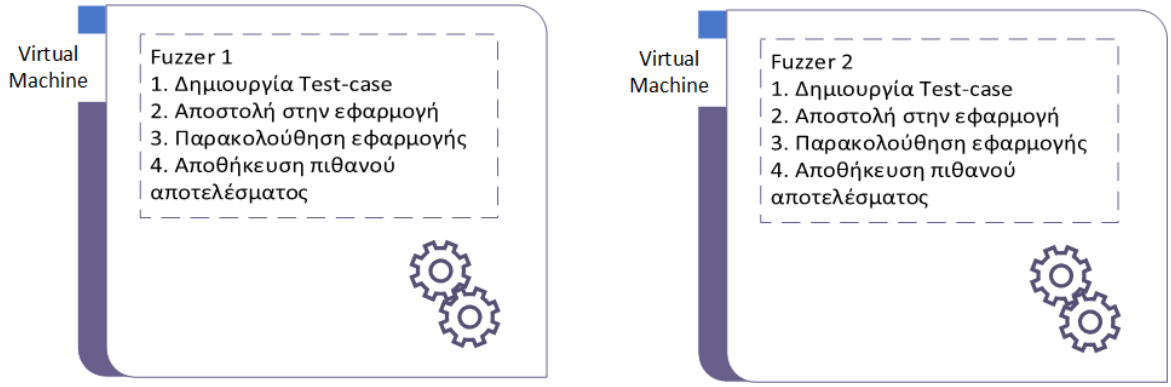
Κατά αυτό τον τρόπο είμαστε σίγουροι ότι η διαδικασία του fuzzing δεν περιορίζεται στην εύρεση μόνο αυτού του σφάλματος καθώς θα εξεταστεί και ο περαιτέρω κώδικας.

2.10 Κατανεμημένο Fuzzing

Η κατανεμημένη χρήση της μεθόδου του fuzzing είναι ο πιο άμεσος τρόπος βελτίωσης της αποτελεσματικότητας της. Αυτό συμβαίνει καθώς όσες περισσότερες επαναλήψεις τις διαδικασίας του fuzzing εκτελούνται τόσο αυξάνονται οι πιθανότητες να βρεθούν αποτελέσματα.

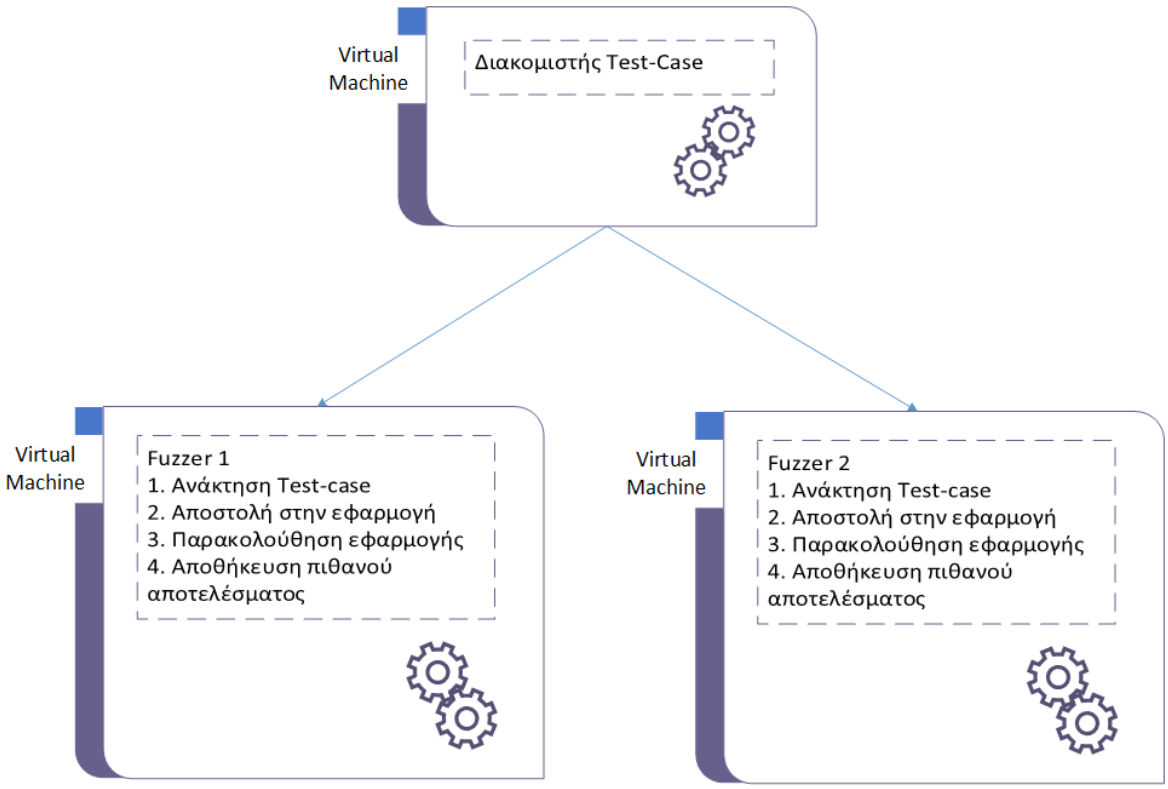
Η υλοποίηση του κατανεμημένου fuzzing εξαρτάται σε μεγάλο βαθμό από τον σχεδιασμό του fuzzer που πρόκειται να εκτελεστεί.

Η πιο απλή περίπτωση κατανεμημένου fuzzing είναι η εκτέλεση αυτοτελών fuzzer σε πολλά μηχανήματα παράλληλα. Με την έννοια αυτοτελής εννοούμε πως ο κάθε fuzzer εμπεριέχει και εκτελεί κάθε στάδιο της διαδικασίας τοπικά στο μηχάνημα που εκτελείτε. Η περίπτωση αυτή παρουσιάζεται σχηματικά παρακάτω.



Εικόνα 9: Παράλληλα εκτελούμενοι fuzzers

Εναλλακτικά ένας fuzzer μπορεί να έχει σχεδιαστεί ώστε διαφορετικά στάδια της διαδικασίας να εκτελούνται σε διαφορετικά μηχανήματα. Για παράδειγμα ένας fuzzer μπορεί να χωρίσει το στάδιο της δημιουργίας test-case και το στάδιο εκτέλεσης της εφαρμογής σε διαφορετικά μηχανήματα όπως εμφανίζεται παρακάτω.



Εικόνα 10: Παράδειγμα σεναρίου κατανεμημένου fuzzing

Ανεξαρτήτως προσέγγισης, για την κατανεμημένη χρήση ενός fuzzer είναι απολύτως απαραίτητο η εκτέλεση του να είναι τελείως αυτοματοποιημένη.

2.11 Επίλογος

Σε αυτό το κεφάλαιο εξετάστηκαν η διαδικασία της μεθόδου του fuzzing και τα επιμέρους στάδια της, οι ενέργειες που αφορούν την προετοιμασία που χρειάζεται να γίνει πριν την χρήση της καθώς και τα πιθανά σενάρια χρήσης κατανεμημένου fuzzing.

Τα ζητήματα που εξετάστηκαν σε αυτό το κεφάλαιο συνοψίζουν την συνολική εικόνα της μεθόδου του fuzzing, ενώ ταυτόχρονα υπογραμμίζουν τις βάσεις πάνω στις οποίες υλοποιήθηκε η υποδομή κατανεμημένου fuzzing που εξετάζεται στο επόμενο κεφάλαιο.

3 Κατανεμημένο Fuzzing: Σχεδιασμός και υλοποίηση

3.1 Εισαγωγή

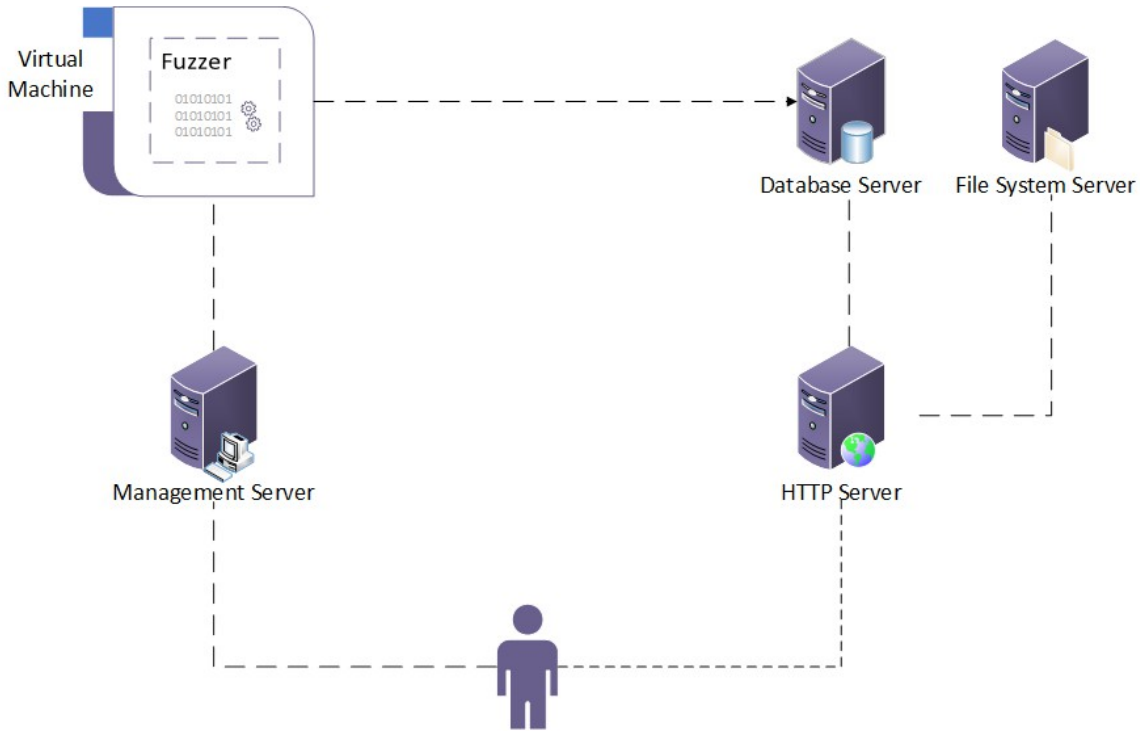
Στο παρόν κεφάλαιο παρουσιάζονται ο σχεδιασμός και η υλοποίηση της υποδομής που δημιουργήθηκε.

Η υποδομή σχεδιάστηκε ώστε να ικανοποιεί τους παρακάτω στόχους:

- Να βοηθά στην υλοποίηση fuzzer που είναι κατάλληλοι για κατανεμημένη εκτέλεση, μέσω ενός fuzzing framework.
- Να υποστηρίζει ήδη υπάρχοντες fuzzers.
- Να επιτρέπει την παράλληλη εκτέλεση ενός fuzzer σε πολλά μηχανήματα.
- Να αποθηκεύει όλα τα αποτελέσματα σε ένα κεντρικό σημείο ώστε να διευκολύνεται η αναζήτηση και διαχείριση τους.

3.2 Αρχιτεκτονική

Η αρχιτεκτονική της υποδομής παρουσιάζεται στο παρακάτω σχήμα.



3.3 Διαχείριση Του Υλικού Της Υποδομής

Τα φυσικά μηχανήματα της υποδομής χρησιμοποιούνται αποκλειστικά για να φιλοξενούν και να εκτελούν εικονικές μηχανές. Οι εικονικές μηχανές είναι το περιβάλλον μέσα στο οποίο εκτελούνται οι fuzzers.

Η αναφορά του τρόπου διαχείρισης των φυσικών μηχανημάτων είναι εκτός των ορίων της πτυχιακής.

3.4 Διαχείριση Εικονικών Μηχανών

Οι εικονικές μηχανές ελέγχονται από ένα κεντρικό διακομιστή διαχείρισης έτσι ώστε η διαχείριση τους να είναι εύκολη και πρωτίστως αυτοματοποιήσιμη.

Είναι εξαιρετικά σημαντικό η διαχείριση των εικονικών μηχανών να μπορεί να γίνει προγραμματιστικά καθώς υπάρχουν σενάρια fuzzing όπου ο fuzzer χρειάζεται να διαχειρίζεται την εικονική μηχανή μέσα στην οποία εκτελείται ώστε να αυτοματοποιείτε η διαδικασία.

Ένα παράδειγμα αυτού του είδους σεναρίου είναι το fuzzing εφαρμογών που εκτελούνται στον πυρήνα του λειτουργικού συστήματος. Ο απρόσμενος τερματισμός της λειτουργίας μιας εφαρμογής αυτού του είδους απαιτεί την επανεκκίνηση ολόκληρου του συστήματος ώστε να συνεχιστεί η διαδικασία του fuzzing.

Για την επίτευξη αυτού του στόχου, η διαχείριση των εικονικών μηχανών στην πρώτη έκδοση της υποδομής υλοποιείται κάνοντας χρήση του VirtualBox API [43].

3.4.1 Στιγμιότυπα Εικονικών Μηχανών

Για κάθε ένα λειτουργικό σύστημα που θέλουμε να υποστηρίζει η υποδομή συντηρείτε ένα στιγμιότυπο εικονικής μηχανής. Κάθε στιγμιότυπο εικονικής μηχανής περιλαμβάνει όλα τα εργαλεία και τις εφαρμογές που απαιτούνται για την εκκίνησή της διαδικασίας fuzzing σε αυτό.

Η χρήση καλών πρακτικών οργάνωσης των στιγμιοτύπων επιτρέπει την εύκολη και γρήγορη υλοποίηση deployment scripts όπως εξετάζεται στο κεφάλαιο 3.7.

Μετά την ολοκλήρωση της διαδικασίας του fuzzing σε μια εικονική μηχανή επαναφέρεται το αρχικό στιγμιότυπο της.

3.5 Βάση Δεδομένων - Κεντρικοποιημένη Αναφορά

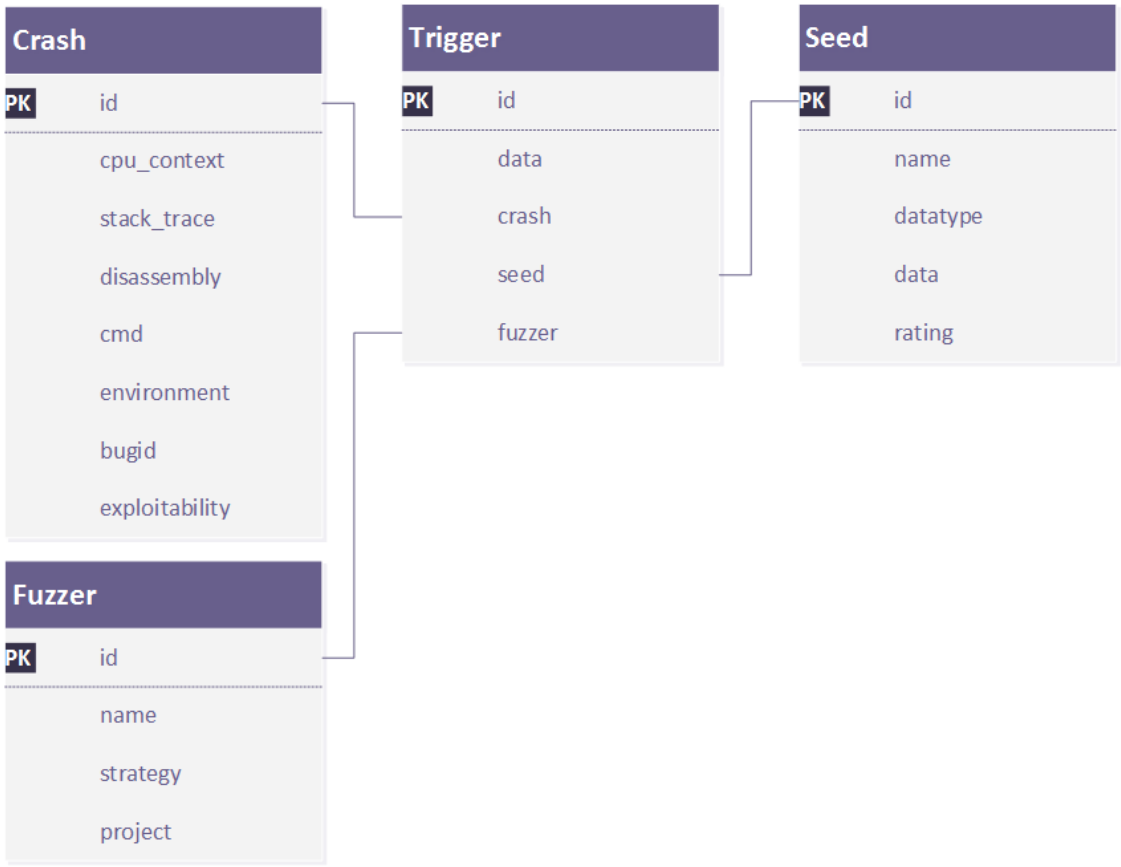
Η κεντρικοποιημένη αναφορά υλοποιείται μέσω της χρήσης μιας κεντρικής απομακρυσμένης βάσης δεδομένων. Η απομακρυσμένη βάση δεδομένων αποθηκεύει πληροφορίες για τα σφάλματα που βρίσκονται, τα test-cases που τα προκαλούν καθώς και και τα seeds που διαθέτουμε ανά τύπο.

Στην περίπτωση που τα δεδομένα ξεπερνούν ένα ορισμένο μέγεθος αποθηκεύονται στον απομακρυσμένο διακομιστή αρχείων και η βάση δεδομένων αποθηκεύει την τοποθεσία τους.

Η βάση δεδομένων που επιλέχθηκε για την υλοποίηση της υποδομής είναι η MySQL ωστόσο, όπως παρουσιάζεται στο κεφάλαιο 3.5.2, λόγο του τρόπου υλοποίησης της επικοινωνίας fuzzer – βάσης δεδομένων θα μπορούσε να είναι οποιαδήποτε άλλη.

3.6 Σχήμα Βάσης Δεδομένων

Παρακάτω παρουσιάζεται το σχήμα της βάσης δεδομένων που χρησιμοποιείτε.



Εικόνα 11: Σχήμα βάσης δεδομένων

Το σχήμα της βάσης δεδομένων αντανακλά τα όσα αναφέρθηκαν στο κεφάλαιο 2.8.

3.6.1 Επικοινωνία Fuzzer - Βάσης Δεδομένων

Ο κώδικας επικοινωνίας των fuzzers με την βάση δεδομένων υλοποιήθηκε με την χρήση της Python βιβλιοθήκης SQLAlchemy [44], η οποία μας επιτρέπει την χρήση του αντικειμενοσχεσιακού μοντέλου βάσης δεδομένων .

Η SQLAlchemy επιτρέπει την υποστήριξη διαφορετικών βάσεων δεδομένων από την υποδομή, χωρίς να χρειάζεται η συγγραφή διαφορετικού κώδικα για κάθε βάση δεδομένων.

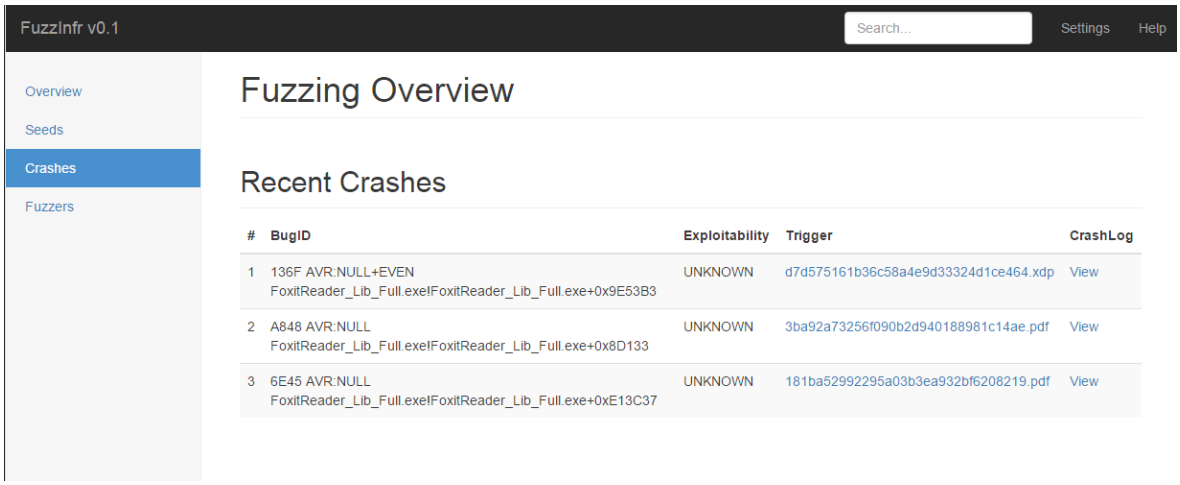
Υλοποιώντας την επικοινωνία fuzzer - βάσης δεδομένων κατά αυτό τον τρόπο η υποδομή μπορεί να υποστηρίξει τις εξής βάσεις δεδομένων χωρίς περαιτέρω τροποποιήσεις:

- Firebird
- Microsoft SQL Server
- MySQL
- Oracle
- PostgreSQL
- SQLite
- Sybase

3.7 Προβολή Των Αποτελεσμάτων Μέσω HTTP/HTML

Η προβολή των αποτελεσμάτων υλοποιείται μέσω ενός web application που επικοινωνεί με την βάση δεδομένων και παρουσιάζει τα αποτελέσματα σε HTML μορφή.

Για την δημιουργία του web application χρησιμοποιήθηκε το Python WSGI framework Bottle [45] σε συνδυασμό με το front-end framework Bootstrap [46].



Εικόνα 12: Προβολή αποτελεσμάτων μέσω HTTP και HTML

3.8 Fuzzing Deployment

Το deployment ενός fuzzer σε διαφορετικά μηχανήματα επιτυγχάνεται με την χρήση ενός deployment script.

Αυτό το script εκτελεί όλες τις απαραίτητες ενέργειες ώστε να είναι εγκατεστημένο και έτοιμο προς εκτέλεση το λογισμικό που πρόκειται να δοκιμαστεί καθώς και ο fuzzer που θα υλοποιεί την διαδικασία fuzzing.

Εφόσον όλες οι ιδεατές μηχανές που θα φιλοξενούν τους fuzzers προέρχονται από το ίδιο στιγμιότυπο, ένα deployment script μπορεί να εκτελεστεί παράλληλα σε όσες μηχανές χρειάζεται.

Το deployment script συνήθως γράφεται στην scripting γλώσσα που υποστηρίζει το εκάστοτε λειτουργικό. Για παράδειγμα ένα deployment script για Linux μπορεί να υλοποιηθεί σε Bash ενώ για Windows σε Powershell.

Για την συγκεκριμένη υποδομή μπορεί να γίνει χρήση και της γλώσσας Python, καθώς βρίσκεται εγκατεστημένη σε όλα τα στιγμιότυπα των εικονικών μηχανών.

Ένα παράδειγμα deployment script παρουσιάζεται στο κεφάλαιο 4.2.6.

3.9 Fuzzing Framework

Για να είναι η υποδομή όσο το δυνατόν γενικότερη, καλύπτοντας τα περισσότερα σενάρια που δύναται να παρουσιαστούν, δημιουργήθηκε ένα fuzzing framework.

Σκοπός του framework είναι να επιβοηθά την εύκολη και γρήγορη ανάπτυξη κατανεμημένων fuzzers.

3.9.1 Αλγόριθμοι Μετάλλαξης

Για τους σκοπούς της υποδομής υλοποιήθηκαν οι εξής αλγόριθμοι μετάλλαξης δεδομένων:

- mutators/bytemut.py (Random Byte Mutation)
- mutators/bitmut.py (Random Bit Mutation)
- mutators/swapmut.py (Random Swap Bytes)
- mutators/radamsa.py (Radamsa Wrapper)

Επιπλέον δημιουργήθηκε η βιβλιοθήκη bintemplates.py η οποία επιτρέπει την δημιουργία έξυπνων αλγορίθμων μετάλλαξης χρησιμοποιώντας την περιγραφή των δεδομένων με τρόπο παρόμοιο αυτού του Peach3.

```
Header:
 WORD magic
Block:
 DWORD start
 DWORD length
 DWORD crc
 BYTE *content
FileData:
 Header header;
 Block blocks[4];
```

Οι αλγόριθμοι μετάλλαξης μπορούν να χρησιμοποιηθούν για την συγγραφή μιας μιας εφαρμογής διακομιστή που θα επιστρέφει test-cases μέσω HTTP. Αυτό επιτρέπει την κεντρικοποιημένη διανομή test-cases στους fuzzers που εκτελούνται. Μια απόπειρα υλοποίησης αυτής της προσέγγισης μπορεί να βρεθεί στο αρχείο mutators/rmut.py.

3.9.2 Αλγόριθμοι Παρακολούθησης Εκτέλεσης Εφαρμογής

Για την εκτέλεση και παρακολούθηση μιας εφαρμογής δημιουργήθηκαν οι εξής δύο αλγόριθμοι που καλύπτουν τις πρώτες δυο περιπτώσεις παρακολούθησης, όπως αυτές αναφέρθηκαν στο κεφάλαιο 2.7.2.

- runners/simple_runner.py
- runners/imenu_context_runner.py

Οι αλγόριθμοι αυτοί καλύπτουν μόνο τις περιπτώσεις εκτέλεσης εφαρμογών Windows. Ο πρώτος αλγόριθμος αφορά την εκτέλεση εφαρμογών από την γραμμή εντολών ενώ ο δεύτερος την εκτέλεση εφαρμογών χρησιμοποιώντας το γραφικό μενού των Windows.

Η επικοινωνία των αλγορίθμων με το Windows Debug API υλοποιήθηκε κάνοντας χρήσης της βιβλιοθήκης WinAppDbg [31].

3.9.3 Αλγόριθμοι Αναφοράς Αποτελεσμάτων

Για την αποθήκευση αποτελεσμάτων δημιουργήθηκαν οι εξής αλγόριθμοι:

- handlers/local_store.py
- handlers/remote_report.py

Ο πρώτος αφορά την τοπική αποθήκευση των αποτελεσμάτων ενώ ο δεύτερος αποστέλλει τα αποτελέσματα στην κεντρική βάση δεδομένων.

3.9.4 Χρήση Ήδη Υπαρχόντων Fuzzers

Όπως αναφέρθηκε στην εισαγωγή του κεφαλαίου, ένας από τους στόχους της υποδομής είναι η υποστήριξη ήδη υπαρχόντων fuzzers.

Για την επίτευξη αυτού του στόχου είναι απαραίτητη η συγγραφή κώδικα που θα μετατρέπει τα αποτελέσματα του fuzzer σε μορφή κατάλληλη προς αποθήκευση στην κεντρική βάση δεδομένων. Η μετατροπή αυτή μπορεί να επιτευχθεί εύκολα κάνοντας χρήση των αλγορίθμων αναφοράς αποτελεσμάτων της υποδομής.

Για την πρώτη έκδοση της υποδομής υλοποιήθηκε ο αλγόριθμος fuzzers/foe2_wrapper.py ο οποίος επιτρέπει την χρήση του fuzzer FOE2.

3.10 Επίλογος

Σε αυτό το κεφάλαιο παρουσιάστηκε ο σχεδιασμός και η υλοποίηση της υποδομής κατανεμημένου fuzzing που δημιουργήθηκε.

Αναλυτικότερα, παρουσιάστηκε η αρχιτεκτονική της υποδομής αναφέροντας πληροφορίες σχετικά με την διαχείριση των εικονικών μηχανών της και τον τρόπο οργάνωσης της κεντρικής βάσης δεδομένων.

Έπειτα παρουσιάστηκαν τα μέλη του fuzzing framework που υλοποιήθηκε. Το fuzzing framework συμβάλει στην δημιουργία fuzzer κατάλληλων προς κατανεμημένη χρήση.

Στη συνέχεια παρουσιάζεται ένα πρακτικό παράδειγμα χρήσης της μεθόδου κατανεμημένου fuzzing μέσω της υποδομής.

4 Παράδειγμα Fuzzing: Foxit Reader

4.1 Εισαγωγή

Σε αυτό το κεφάλαιο εξετάζεται η εφαρμογή Foxit Reader v7.2.8.1124 χρησιμοποιώντας την υποδομή για την εκτέλεση κατανεμημένου fuzzing.

Η εφαρμογή εκτελέστηκε στο λειτουργικό σύστημα Windows 7 για επεξεργαστές x86.

Για τους σκοπούς αυτού του κεφαλαίου δημιουργήθηκε ένας απλός fuzzer αρχείων για Windows ο οποίος υλοποιεί την διαδικασία και αποθηκεύει τα πιθανά αποτελέσματα στην κεντρική βάση δεδομένων.

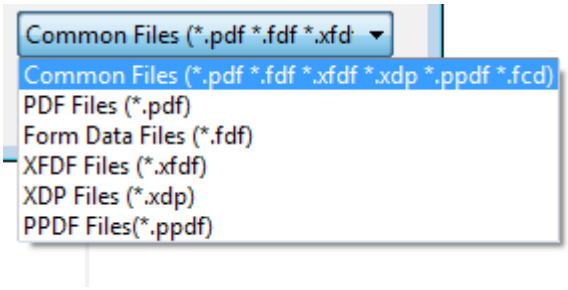
Τα σφάλματα που βρέθηκαν αναφέρθηκαν εγκαίρως στην εταιρία ανάπτυξης του λογισμικού και παρουσιάζονται στο τέλος του κεφαλαίου.

4.2 Περιγραφή Της Διαδικασίας Που Ακολουθήθηκε

Σε αυτό το σημείο περιγράφεται η διαδικασία που ακολουθήθηκε, αντιστοιχίζοντας τα στάδια που αναφέρθηκαν στο κεφάλαιο 2.11 με το συγκεκριμένο σενάριο.

4.2.1 Ανίχνευση Των Εισόδων Της Εφαρμογής

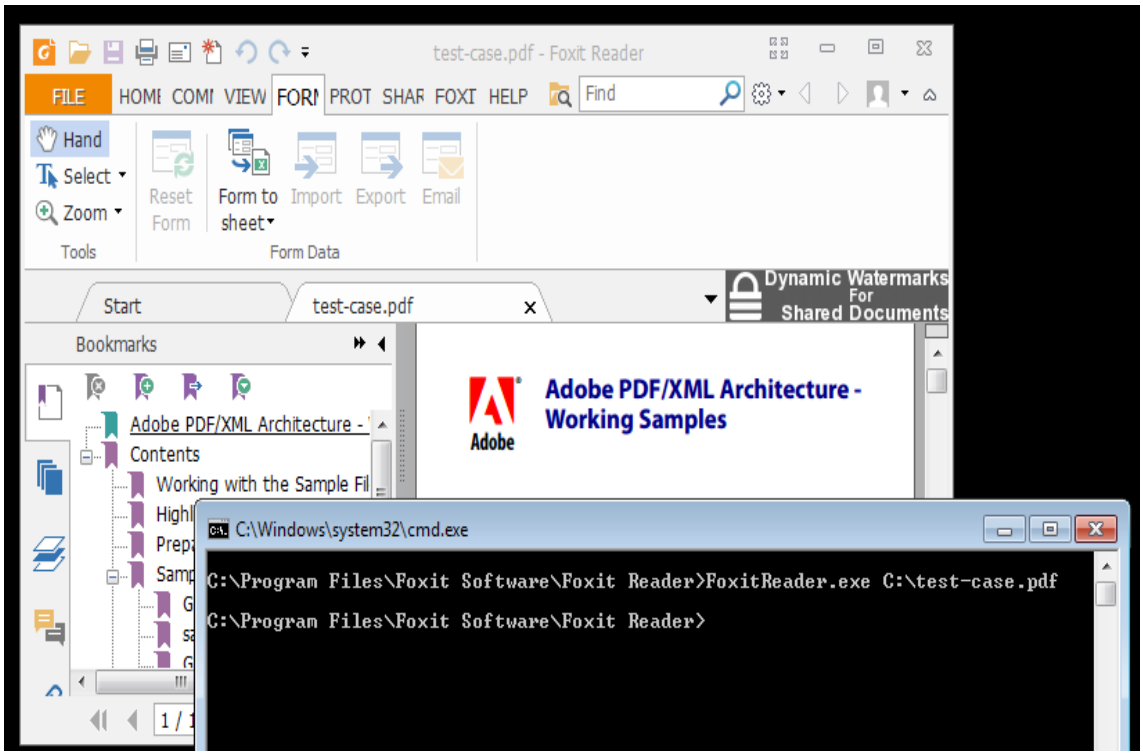
Το Foxit Reader, όντας μια εφαρμογή προβολής και επεξεργασίας αρχείων, δέχεται τα δεδομένα εισόδου της ως αρχεία. Μια γρήγορη εξέταση της επιλογής “Open File..” μας επιτρέπει να βρούμε το είδος των αρχείων που υποστηρίζει η εφαρμογή.



Εικόνα 13: Υποστηριζόμενοι τύποι αρχείων του Foxit Reader

Επόμενο βήμα είναι να εξετάσουμε εάν μπορούμε να στείλουμε αρχεία στην εφαρμογή είτε μέσω γραμμής εντολών είτε προγραμματιστικά.

Εκτελώντας την εφαρμογή από την γραμμή εντολών μπορούμε να διαπιστώσουμε πως μπορούμε να στείλουμε αρχεία σε αυτή δίνοντας τη τοποθεσία του αρχείου ως πρώτη παράμετρο.



Εικόνα 14: Αποστολή αρχείου στην εφαρμογή μέσω γραμμής εντολών

Το γεγονός αυτό επιτρέπει την αυτοματοποίηση της διαδικασίας αποστολής δεδομένων.

4.2.2 Επιλογή Μεθόδου Δημιουργίας Test Cases

Η μέθοδος που επιλέχθηκε για την δημιουργία test-case είναι η απλή μετάλλαξη δεδομένων. Η επιλογή αυτή έγινε σύμφωνα με το γεγονός πως η συγκεκριμένη μέθοδος συνδυάζει τον μικρό χρόνο που απαιτείται για να υλοποιηθεί με αρκετές καλές πιθανότητες εύρεσης αποτελεσμάτων.

Διαθέτοντας τέσσερις εικονικές μηχανές μπορούμε να εκτελέσουμε τέσσερις διαφορετικούς αλγόριθμους απλής μετάλλαξης παράλληλα, αυξάνοντας έτσι τα διαφορετικά test-case που παράγονται. Οι αλγόριθμοι που χρησιμοποιήθηκαν είναι οι bytemut.py, bitmut.py, swap.py και radamsa.py.

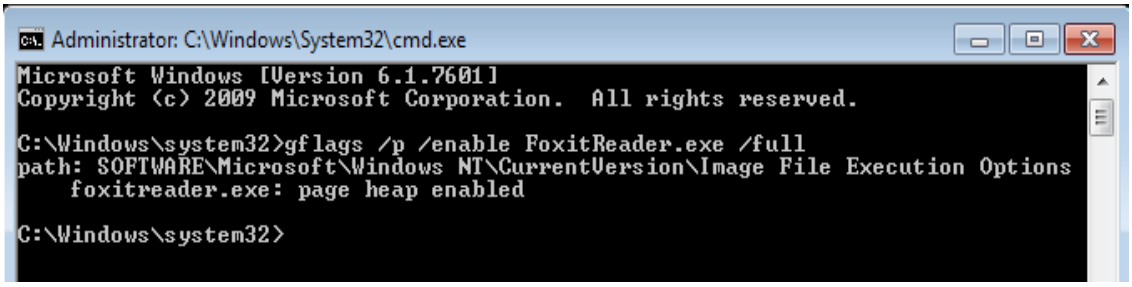
Για την χρήση της μεθόδου απλής μετάλλαξης δεδομένων χρειάζεται να συλλέξουμε ένα coprus αρχείων pdf, fdf, xfdf, xdp, ppdf και fcd.

Η συλλογή αρχείων έγινε αναζητώντας αρχεία στο web.

4.2.3 Επιλογή Εργαλείων Instrumentation

Το γεγονός πως δεν έχουμε πρόσβαση στον πηγαίο κώδικα της εφαρμογής αποτρέπει την χρήση των περισσοτέρων instrumentation εργαλείων.

Καθώς όμως ο Foxit Reader χρησιμοποιεί τον default διαχειριστή σωρού των Windows(Low Fragmentation Heap) χρησιμοποιήσαμε το PageHeap για τον έγκαιρο εντοπισμό σφαλμάτων μνήμης σωρού.



Εικόνα 15: Ενεργοποίηση του εργαλείου PageHeap για την διεργασία FoxitReader.exe

4.2.4 Επιλογή Τρόπου Αποθήκευσης Αποτελεσμάτων

Όλα τα αποτελέσματα αποθηκεύονται κεντρικά στην απομακρυσμένη βάση δεδομένων της υποδομής.

4.2.5 Επιλογή/Δημιουργία Fuzzer

Για το συγκεκριμένο σενάριο θα μπορούσε να χρησιμοποιηθεί ένας ήδη έτοιμος fuzzer αρχείων, όπως για παράδειγμα ο FOE2. Καθώς όμως η διαδικασία του fuzzing σε αυτή την περίπτωση εκτελείτε για τους σκοπούς παρούσας πτυχιακής εργασίας, δημιουργήθηκε ένας καινούργιος fuzzer αρχείων χρησιμοποιώντας τους αλγόριθμους του fuzzing framework που περιγράφηκε στο κεφάλαιο 3.

Ο fuzzer αυτός μπορεί να βρεθεί στον φάκελο fuzzers/sff.py στο πρακτικό μέρος της εργασίας.

4.2.6 Deployment Script

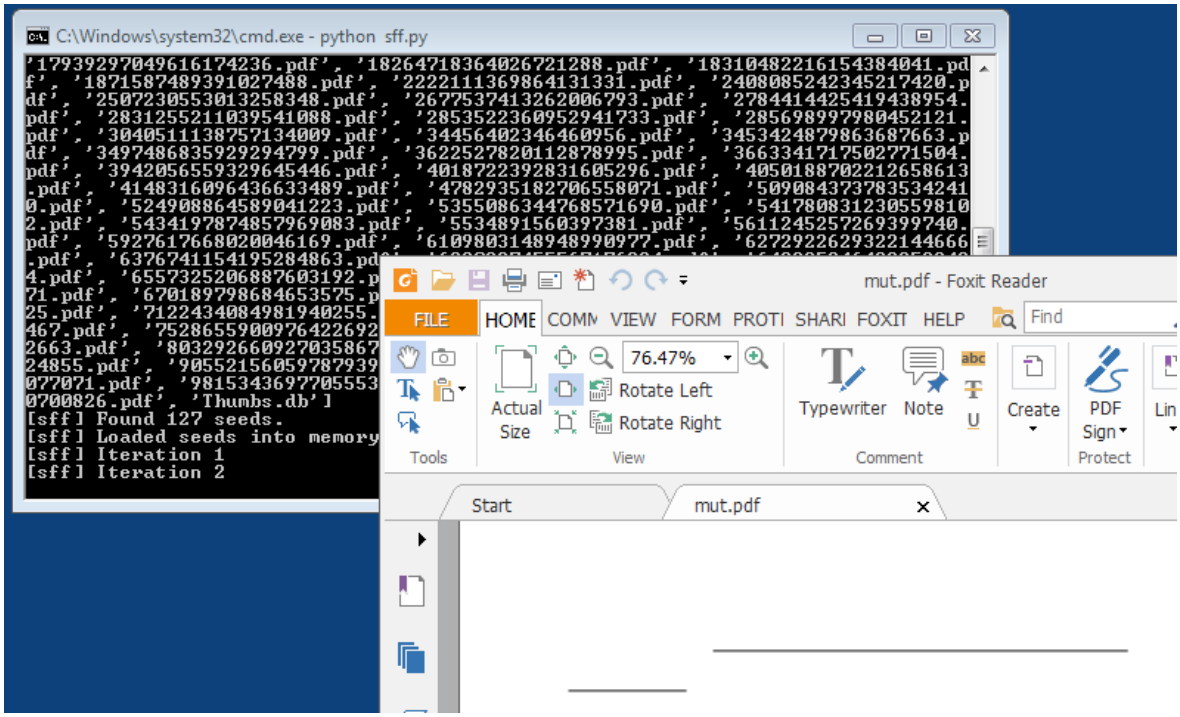
Για την εγκατάσταση του Foxit Reader στις εικονικές μηχανές όπου επρόκειτο να εκτελεστεί η διαδικασία του fuzzing χρησιμοποιήθηκε το παρακάτω deployment script γραμμένο σε Powershell.

```
$url = @"
http://cdn01.foxitsoftware.com/pub/foxit/reader/desktop/win/
7.x/7.3/en_us/FoxitReader73_enu_Setup_Prom.exe"@

$path = "foxit_install.exe"


Write-Host "[*] Downloading Foxit Reader."

(New-Object System.Net.WebClient).DownloadFile($url, $path)

Write-Host "[*] Installing Foxit Reader."

.\foxit_install.exe /SP- /VERYSILENT /SUPPRESSMSGBOXES
```

4.3 Αποτελέσματα

Σε αυτό το σημείο παρουσιάζονται τα αποτελέσματα που βρέθηκαν κατά το fuzzing της εφαρμογής FoxitReader.

Για κάθε αποτέλεσμα παρουσιάζονται ο κώδικας μηχανής και οι τιμές των καταχωρητών που εξήχθησαν από το στιγμιότυπο μνήμης καθώς και η περιληπτική ανάλυση του σφάλματος που το προκάλεσε.



Εικόνα 16: Εκτέλεση του fuzzer sff.py

4.3.1 Crash 1

```
Registers:

eax=0b843dba ebx=ffffffff ecx=00000000 edx=01b96100
esi=07af6aa8 edi=00000000

...

Code disassembly:

   01B79A00 |      55 | push ebp

   01B79A01 |    8bec | mov ebp, esp

 * 01B79A03 | 8b410c | mov eax, [ecx+0xc]        (A)

...

Stack trace:

FoxitReader_Lib_Full.exe + 0x9E53B3 (13 in id)

FoxitReader_Lib_Full.exe + 0x9F681A (6F in id)   (B)

...
```

Το πρώτο crash προκαλείται κατά την επεξεργασία του αρχείου d7d575161b36c58a4e9d33324d1ce464.xdp το οποίο προήλθε από την μετάλλαξη του αρχείου tsp301.xdp χρησιμοποιώντας τον αλγόριθμο μετάλλαξης bytemut.py.

Εξετάζοντας τον κώδικα μηχανής στον οποίο σταμάτησε η εκτέλεση του προγράμματος μπορούμε να διαπιστώσουμε πως πρόκειται για σφάλμα χρήσης κενού δείκτη. Η αναφορά σε μνήμη στο σημείο (A) γίνεται χρησιμοποιώντας τον καταχωρητή ECX, η τιμή του οποίου είναι μηδέν.

Επιπλέον, οι δύο εντολές που προηγούνται την εντολή (A) αλλάζουν την θέση του καταχωρητή EBP, ο οποίος στην αρχιτεκτονική x86 δείχνει πάντοτε στο πλαίσιο στοίβας της τρέχων συνάρτησης. Το γεγονός αυτό υποδηλώνει πως βρισκόμαστε στην αρχή μιας συνάρτησης [47].

Τέλος το γεγονός πως ο καταχωρητής ECX χρησιμοποιείται στην αρχή της συνάρτησης χωρίς να έχει πρώτα αρχικοποιηθεί η τιμή του υποδηλώνει πως, κατά πάσα πιθανότητα, η τιμή του ECX περιλαμβάνει τον δείκτη _this του αντικειμένου C++ που ανήκει η συγκεκριμένη συνάρτηση [48].

Η ανάλυση αυτού του αποτελέσματος πρέπει να συνεχιστεί εξετάζοντας τον κώδικα που καλεί την συνάρτηση που μόλις εξετάσαμε.

Η συνάρτηση στην οποία ανήκει αυτός ο κώδικας μπορεί να βρεθεί στο στιγμιότυπο στοίβας (B).

4.3.2 Crash 2

```
Registers:

eax=00000000 ebx=ffffffff ecx=0189c56d edx=000000d2
esi=07750798 edi=0774fd44

...

Code disassembly:

   014C9082 | e804383d00 | call foxitreader!start+0x4faaa

 * 014C9087 |     0fb608 | movzx ecx, byte [eax]          (Γ)

   014C908A |         53 | push ebx

   014C908B |     83f905 | cmp ecx, 0x5

   014C908E |       7572 | jnz foxitreader!0xe9102

...

Stack trace:

FoxitReader_Lib_Full.exe + 0x8D133 (A8 in id)

FoxitReader_Lib_Full.exe + 0x19F022 (48 in id)

...
```

Το δεύτερο crash προκαλείται κατά την επεξεργασία του αρχείου 3ba92a73256f090b2d940188981c14ae.pdf το οποίο προήλθε από την μετάλλαξη του αρχείου tsp301.pdf χρησιμοποιώντας τον αλγόριθμο μετάλλαξης radamsa.py.

Εξετάζοντας το κώδικα μηχανής αυτού του στιγμιότυπου διαπιστώνουμε πως το σφάλμα που προκάλεσε το συγκεκριμένο crash είναι ένα σφάλμα αναφοράς κενού δείκτη το οποίο προέρχεται από τον ελλειπή έλεγχο της τιμής επιστροφής της συνάρτησης foxitreader!start+0x4faaa .

Αναλυτικότερα, σύμφωνα με το calling convention του x86 η τιμή επιστροφής μιας συνάρτησης αποθηκεύεται στον καταχωρητή EAX [49] . Η εντολή στο σημείο (Γ) εκτελείται αμέσως μετά την επιστροφή της συνάρτησης foxitreader!start+0x4faaa και συνεπώς ο EAX περιέχει την τιμή επιστροφής της.

Στην συνέχεια, η τιμή επιστροφής της συνάρτησης χρησιμοποιείται ως δείκτης αναφοράς μνήμης χωρίς να έχει γίνει πρωτύτερα κάποιος έλεγχος. Αυτό οδηγεί στον απροσδόκητο τερματισμό του προγράμματος στις περιπτώσεις όπου η συνάρτηση foxitreader!start+0x4faaa επιστρέφει τιμή που δεν αντιστοιχεί σε θέση μνήμης. Για την συνέχεια της ανάλυσης αυτού του σφάλματος χρειάζεται να εξετάσουμε τον κώδικα της συνάρτησης foxitreader!start+0x4faaa.

4.3.3 Crash 3

```
Registers:

eax=00000000 ebx=2f1ccfd8 ecx=0184c56d edx=000003d4
esi=00000002 edi=2eef0fe4

...

Code disassembly:

   021911a1 | e8ddb96bff | call    FoxitReader+0x4bcb83   (Δ)

   021911a6 |         46 | inc     esi

 * 021911a7 |     803802 | cmp     byte ptr [eax],2       (E)

   021911aa |       754c | jne     FoxitReader!0xe011f8

...

Stack trace:

FoxitReader_Lib_Full.exe + 0xE13C37 (6E in id)

FoxitReader_Lib_Full.exe + 0xE14982 (45 in id)

...
```

Το τρίτο crash προκαλείται κατά την επεξεργασία του αρχείου 181ba52992295a03b3ea932bf6208219.pdf το οποίο προήλθε από την μετάλλαξη του αρχείου tsp301.pdf χρησιμοποιώντας τον αλγόριθμο μετάλλαξης bytemut.py.

Το σφάλμα που προκάλεσε το συγκεκριμένο crash είναι εφάμιλλο με το σφάλμα που εξετάσαμε στην προηγούμενη περίπτωση. Η εντολή στο σημείο (Ε) αναφέρεται στην θέση μνήμης που δείχνει ο καταχωρητής EAX. Η τιμή του EAX έχει τεθεί στο σημείο (Δ) ως την τιμή επιστροφής της συνάρτησης `FoxitReader+0x4bcb83`.

Στην περίπτωση αυτή η συνάρτηση επέστρεψε μηδέν, οδηγώντας στο σφάλμα αναφοράς κενής μνήμης στο σημείο (Ε). Για την συνέχεια της ανάλυσης αυτού του σφάλματος χρειάζεται να εξεταστεί η συνάρτηση `FoxitReader+0x4bcb83`.

4.4 Επίλογος

Σε αυτό το κεφάλαιο παρουσιάστηκε ένα πρακτικό παράδειγμα εφαρμογής της μεθόδου του κατανεμημένου fuzzing κάνοντας χρήση της υποδομής που δημιουργήθηκε.

Τα στάδια της προετοιμασίας του fuzzing που αναφέρθηκαν θεωρητικά στο κεφάλαιο 2.11 αντιστοιχίστηκαν με πραγματικά παραδείγματα ενώ τα στάδια της διαδικασίας του fuzzing που παρουσιάζονται στο κεφάλαιο 2.5 υλοποιήθηκαν δημιουργώντας τον fuzzer sff.py, έναν fuzzer εφαρμογών Windows.

Στην συνέχεια παρουσιάστηκαν τα αποτελέσματα που βρέθηκαν κατά την εφαρμογή της μεθόδου για την εξέταση της εφαρμογής Foxit Reader.

Επενδύοντας ελάχιστο χρόνο και κόπο, χωρίς να εξετάσουμε καθόλου τον κώδικα του λογισμικού ή την δομή των δεδομένων που αυτό επεξεργάζεται, κατορθώσαμε να εντοπίσουμε 3 σφάλματα σε ένα ήδη καλά δοκιμασμένο λογισμικό.

5 Συμπεράσματα

Σκοπός της παρούσας πτυχιακής εργασίας ήταν ο σχεδιασμός και η δημιουργία μιας υποδομής κατανεμημένου fuzzing. Μελετώντας την διαδικασία του fuzzing και τα επιμέρους στάδια της καταλήξαμε στα εξής συμπεράσματα:

- Για την εκκίνηση του κατανεμημένου fuzzing μιας εφαρμογής απαιτείται η πλήρης αυτοματοποίηση της διαδικασίας.
- Η αυτοματοποίηση της διαδικασίας του fuzzing διαφέρει από σενάριο σε σενάριο. Δεν υπάρχει μια γενική λύση που καλύπτει όλες τις περιπτώσεις.
- Για την χρήση κατανεμημένου fuzzing είναι απαραίτητη η αυτοματοποίηση της προετοιμασίας κάθε μηχάνηματος που λαμβάνει μέρος στη διαδικασία.
- Ανεξαρτήτως σεναρίου, το στάδιο της αποθήκευσης των αποτελεσμάτων μπορεί να γίνει κεντρικοποιημένα. Η κεντρικοποιημένη συλλογή αποτελεσμάτων βοηθά στην μετέπειτα αναζήτηση και διαχείριση τους και είναι πάντα επιθυμητή.
- Η χρήση κατανεμημένου fuzzing αναμένεται να παράγει μεγάλο όγκο αποτελεσμάτων. Η χρήση εργαλείων instrumentation και εργαλείων υπολογισμού της μοναδικότητας ενός αποτελέσματος βοηθά στην αποφυγή αποθήκευσης φαινομενικά διαφορετικών αποτελεσμάτων που επί της ουσίας προέρχονται από το ίδιο σφάλμα.

Η υποδομή που δημιουργήθηκε επιχειρεί να λύσει τα πρώτα δύο προβλήματα υλοποιώντας ένα fuzzing framework. Κατά αυτό τον τρόπο βοηθά στην ανάπτυξη fuzzer που είναι κατάλληλοι για κατανεμημένη χρήση. Παράλληλα δίνειται η δυνατότητα συγγραφής εργαλείων που ενσωματώνουν ήδη υπάρχοντες fuzzers στην υποδομή.

Στο τρίτο πρόβλημα προτείνεται η λύση συγγραφής deployment scripts σε συνδυασμό με την χρήση καλών πρακτικών οργάνωσης της υποδομής εικονικών μηχανών που διαθέτεται.

Για την λύση του τέταρτου προβλήματος δημιουργήθηκε μια βάση δεδομένων, στην οποία αποθηκεύονται όλα τα αποτελέσματα κεντρικά.

Η υποδομή δημιουργήθηκε και χρησιμοποιείται στις εγκαταστάσεις της εταιρίας CENSUS στην Θεσσαλονίκη.

6 Μελλοντικές Επεκτάσεις

Η υποδομή βρίσκεται ακόμη στην πρώτη έκδοση της και ως εκ τούτου επιδέχεται πολλές βελτιώσεις.

Αρχικά, όσο ανεβαίνει ο αριθμός των εικονικών μηχανών της υποδομής τόσο πιο απαιτητική γίνεται η διαχείριση τους. Για αυτό τον σκοπό η διαχείριση των εικονικών μηχανών στο μέλλον σχεδιάζεται να γίνει μέσω μιας ολοκληρωμένης λύσης cloud computing, όπως για παράδειγμα το OpenNebula [50].

Η περαιτέρω επέκταση της υποδομής αφορά κυρίως την εμπλούστευση του fuzzing framework με περισσότερους αλγορίθμους και εργαλεία. Όσο μεγαλύτερο είναι το framework τόσες περισσότερες περιπτώσεις καλύπτει η υποδομή.

Τέλος, ένα ζήτημα που δεν εξετάστηκε στην παρούσα εργασία είναι το ζήτημα της ασφάλειας της υποδομής. Για τις μελλοντικές εκδόσεις σχεδιάζεται η υποστήριξη της κρυπτογράφησης των δεδομένων που διαχειρίζεται η υποδομή καθώς και η αυθεντικοποίηση μεταξύ των μελών της.

Βιβλιογραφία

[1]: R. Shirey, Internet Security Glossary,https://tools.ietf.org/html/rfc2828,Ημερομηνία τελευταίας προσπέλασης:2016-1

[2]: Microsoft,Definition of a Security Vulnerability,https://msdn.microsoft.com/en-us/library/cc751383.aspx,Ημερομηνία τελευταίας προσπέλασης:2016-01

[3]: Peter Stavroulakis, Mark Stamp, Handbook of Information and Communication Security, 2010

[4]: Wikipedia,Systems development life cycle,https://en.wikipedia.org/wiki/Systems_development_life_cycle,Ημερομηνία τελευταίας προσπέλασης:2016-01

[5]: Aleph One,Smashing The Stack For Fun And Profit,http://phrack.org/issues/49/14.html,Ημερομηνία τελευταίας προσπέλασης:2016-01

[6]: Tavis Ormandy,The poisoned NUL byte, 2014 edition,http://googleprojectzero.blogspot.gr/2014/08/the-poisoned-nul-byte-2014-edition.html,Ημερομηνία τελευταίας προσπέλασης:2016-01

[7]: Team Teso,Exploiting Format String Vulnerabilities,https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf,Ημερομηνία τελευταίας προσπέλασης:2016-

[8]: OpenGrok,OpenGrok,https://opengrok.github.io/OpenGrok/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[9]: Scitools,Scitools Understand,https://scitools.com/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[10]: Coverity,Coverity,https://www.coverity.com/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[11]: Fabian Yamaguchi,Joern,https://github.com/fabsx00/joern,Ημερομηνία τελευταίας προσπέλασης:2016-01

[12]: Hex-Rays,IDA Pro,https://www.hex-rays.com/products/ida/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[13]: radare2,radare2,http://www.radare.org/r/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[14]: Hopper,Hopper,http://www.hopperapp.com/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[15]: Zynamics,BinNavi,http://www.zynamics.com/binnavi.html,Ημερομηνία τελευταίας προσπέλασης:2016-01

[16]: Patrice Godefroid, Michael Y.Levin, David Molnar, SAGE: Whitebox Fuzzing for Security Testing, 2012

[17]: Klee,Klee,https://klee.github.io/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[18]: Miasm2,Miasm2,https://github.com/cea-sec/miasm,Ημερομηνία τελευταίας προσπέλασης:2016-01

[19]: SemTrax,Introducing SemTrax,https://vimeo.com/113971293,Ημερομηνία τελευταίας προσπέλασης:2016-01

[20]: Microsoft,Security Development Lifecycle,https://www.microsoft.com/en-us/sdl/process/verification.aspx,Ημερομηνία τελευταίας προσπέλασης:2016-01

[21]: Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, David Brumley, Optimizing Seed Selection for fuzzing, 2014

[22]: Charlie Miller,Babysitting an Army of Monkeys,https://fuzzinginfo.files.wordpress.com/2012/05/cmiller-csw-2010.pdf,Ημερομηνία τελευταίας προσπέλασης:2016-01

[23]: Peach3,Peach3,http://www.peachfuzzer.com/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[24]: Radamsa,Radamsa,https://github.com/aoh/radamsa,Ημερομηνία τελευταίας προσπέλασης:2016-

[25]: J.C. Huang, Program Instrumentation and Software Testing, 1978

[26]: Zisis Sialveras, Nikolaos Naziridis,Introducing Choronzon: An approach at knowledge-based evolutionary fuzzing,http://census-labs.com/media/choronzon-zeronights-2015.pdf,Ημερομηνία τελευταίας προσπέλασης:2016-01

[27]: Michał Zalewski,American Fuzzy Lop,http://lcamtuf.coredump.cx/afl/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[28]: Microsoft,The Debugging Application Programming Interface,https://msdn.microsoft.com/en-us/library/ms809754.aspx,Ημερομηνία τελευταίας προσπέλασης:2016-01

[29]: Linux, Linux Programmer's Manual ,http://man7.org/linux/man-pages/man2/wait.2.html,Ημερομηνία τελευταίας προσπέλασης:-

[30]: Linux,Linux man page,http://linux.die.net/man/2/ptrace,Ημερομηνία τελευταίας προσπέλασης:2016-01

[31]: Mario Vilas,WinAppDbg 1.5,http://winappdbg.sourceforge.net/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[32]: CERT,Failure Observation Engine,https://www.cert.org/vulnerability-analysis/tools/foe.cfm?,Ημερομηνία τελευταίας προσπέλασης:2016-1

[33]: Google,Sanitizers,https://github.com/google/sanitizers,Ημερομηνία τελευταίας προσπέλασης:2016-01

[34]: Valgrind,Valgrind,http://valgrind.org/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[35]: Microsoft,PageHeap,https://msdn.microsoft.com/en-us/library/windows/hardware/ff549561(v=vs.85).aspx,Ημερομηνία τελευταίας προσπέλασης:2016-01

[36]: Christopher J. Madsen,vbindiff,http://www.cjmweb.net/vbindiff/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[37]: Skylined,BugId,https://github.com/SkyLined/BugId,Ημερομηνία τελευταίας προσπέλασης:2016-01

[38]: Microsoft Security Engineering Center ,!exploitable Crash Analyzer,https://msecdbg.codeplex.com/,Ημερομηνία τελευταίας προσπέλασης:-

[39]: OWASP,Attack Surface Analysis Cheat Sheet,https://www.owasp.org/index.php?title=Attack_Surface_Analysis_Cheat_Sheet&oldid=156006,Ημερομηνία τελευταίας προσπέλασης:-01

[40]: Michal Zalewski,afl-tmin,https://groups.google.com/forum/#!msg/afl-users/eWb2PgjLnUo/8AKqadYzSBoJ,Ημερομηνία τελευταίας προσπέλασης:2016-01

[41]: Peach3,Minimum Set,http://community.peachfuzzer.com/v3/minset.html,Ημερομηνία τελευταίας προσπέλασης:2016-01

[42]: Michal Zalewski,afl-cmin,https://github.com/mirrorer/afl/blob/master/afl-cmin,Ημερομηνία τελευταίας προσπέλασης:2016-01

[43]: Oracle,VirtualBox Main API,https://www.virtualbox.org/sdkref/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[44]: SqlAlchemy,SqlAlchemy,http://www.sqlalchemy.org/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[45]: Bottle,Bottle: Python Web Framework,http://bottlepy.org/docs/dev/index.html,Ημερομηνία τελευταίας προσπέλασης:2016-01

[46]: Bootstrap,Bootstrap,https://getbootstrap.com/,Ημερομηνία τελευταίας προσπέλασης:2016-01

[47]: Jonathan de Boyne Pollard,The gen on function perilogues.,http://homepage.ntlworld.com./jonathan.deboynepollard/FGA/function-perilogues.html,Ημερομηνία τελευταίας προσπέλασης:2016-01

[48]: Microsoft,__thiscall,https://msdn.microsoft.com/en-us/library/ek8tkfbw.aspx,Ημερομηνία τελευταίας προσπέλασης:2016-01

[48]: Wikibooks,x86 Disassembly/Calling Conventions,https://en.wikibooks.org/wiki/X86_Disassembly/Calling_Conventions,Ημερομηνία τελευταίας προσπέλασης:2016-01

[49]: OpenNebula Project,OpenNebula,http://opennebula.org/,Ημερομηνία τελευταίας προσπέλασης:-