: μ
:
http://arch.icte.uowm.gr

PyEmu: Ένας πολλαπλών-χρήσεων προγραμματιζόμενος IA-32 εξομοιωτής

Cody Pierce

TippingPoint DVLabs
cpierce@tippingpoint.com

1 Εισαγωγή

Οι εξομοιωτές υπάρχουν από τότε που τα σύγχρονα υπολογιστικά συστήματα εξομοιώνονται. Το 1965 κυκλοφόρησε το πρώτο σύστημα υπολογιστή το οποίο βασίζεται εξ ολοκλήρου σε ολοκληρωμένα κυκλώματα[1]. Με αυτό δημιούργησαν έναν εξομοιωτή με σκοπό την ενίσχυση της έκδοσής του. Στις εποχή μας εξομοιωτές εμφανίζονται σε όλα τα είδη εφαρμογών. Αυτές οι εφαρμογές ποικίλλουν από ολοκληρωμένα εικονικά μηχανήματα έως συστήματα στοάς. Στην εργασία αυτή θα δούμε πώς ο κόσμος της εξομοίωσης σχετίζεται και βοηθά την αντίστροφη μηχανική πειθαρχία.

Όταν βλέπει κανείς την εξομοίωση στη σύγχρονη επιστήμη των υπολογιστών, είναι δυνατόν να αναλύσει ό,τι γίνεται αντιληπτό σε δύο κύριες μεθόδους εξομοίωσης αυτή των λειτουργικών συστημάτων και της εξομοίωσης οδηγιών.

1.1 Εξομοίωση συστήματος

Η εξομοίωση συστημάτων είναι μία πολύ ελκυστική μέθοδος για να γίνει η πλήρης αναπαραγωγή της λειτουργίας ενός ολοκληρωμένου συστήματος. Αυτό περιλαμβάνει όχι μόνο την εξομοίωση ενός επεξεργαστή και μίας μνήμης, αλλά επίσης και των περιφερειακών συσκευών του συστήματος.

Το πιο σημαντικό στοιχείο που διαφοροποιεί την εξομοίωση λειτουργικών συστημάτων από την εξομοίωση εντολών είναι η περιφερειακή εξομοίωση. Δεδομένου ότι ο στόχος της εξομοίωσης συστήματος είναι να παρέχει ένα πλήρες περιβάλλον για το βασικό λογισμικό, όπως είναι ένα λειτουργικό σύστημα για να εγκατασταθεί ο εξομοιωτής πρέπει να διεκπεραιώνονται τα αιτήματα για κάρτες γραφικών, ελεγκτών δίσκων, συσκευές δικτύου, καθώς επίσης και για το παρεχόμενο BIOS.

Ένα καλό παράδειγμα αυτού του τύπου εξομοίωσης είναι ο εξομοιωτής bochs[2] IA-32. Παρέχει στο χρήστη τη δυνατότητα να εγκαθιστά ξένα λειτουργικά συστήματα σε έναν εικονικό δίσκο διαχειριζόμενο από bochs. Όπως αναφέρθηκε προηγουμένως, αυτού του είδους πλήρους συστήματος εξομοίωσης θα δράσει όπως ένας φυσικός υπολογιστής, παρέχοντας είσοδο και έξοδο πληκτρολογίου/ποντικιού, καθώς και άλλες συσκευές.

1.2 Εντολές Εξομοίωσης

Η δεύτερη μορφή εξομοίωσης είναι αυτή που θα μπορούσαν να ειπωθούν ως εντολές εξομοίωσης. Υπό την έννοια αυτή οι εξομοιωτές εντολών χειρίζονται μόνο τα καθήκοντα μετάφρασης της συμπεριφοράς της CPU σε ισοδύναμους λογικής και μνήμης υπολογισμούς . Αυτό το είδος εξομοίωσης είναι πιο κατάλληλο για συγκεκριμένη χρήση και θα είναι το επίκεντρο αυτής της εργασίας.

Η εξομοίωση εντολών μπορεί να φανεί περιοριστική με την πρώτη ματιά. Ωστόσο, είναι προσαρμοσμένη για να εξυπηρετεί το ρόλο ενός εργαλείου, σε αντίθεση με ένα σύστημα εξομοίωσης που δουλεύει ως μια εφαρμογή.
Το πλεονέκτημα αυτής της προσέγγισης είναι η διαφάνεια και η ευελιξία. Ενώ διατηρείται ο σκοπός βασικός, επιτρέπει στο χρήστη να καθορίσει τι εξομοιώνεται με μεγαλύτερο έλεγχο.

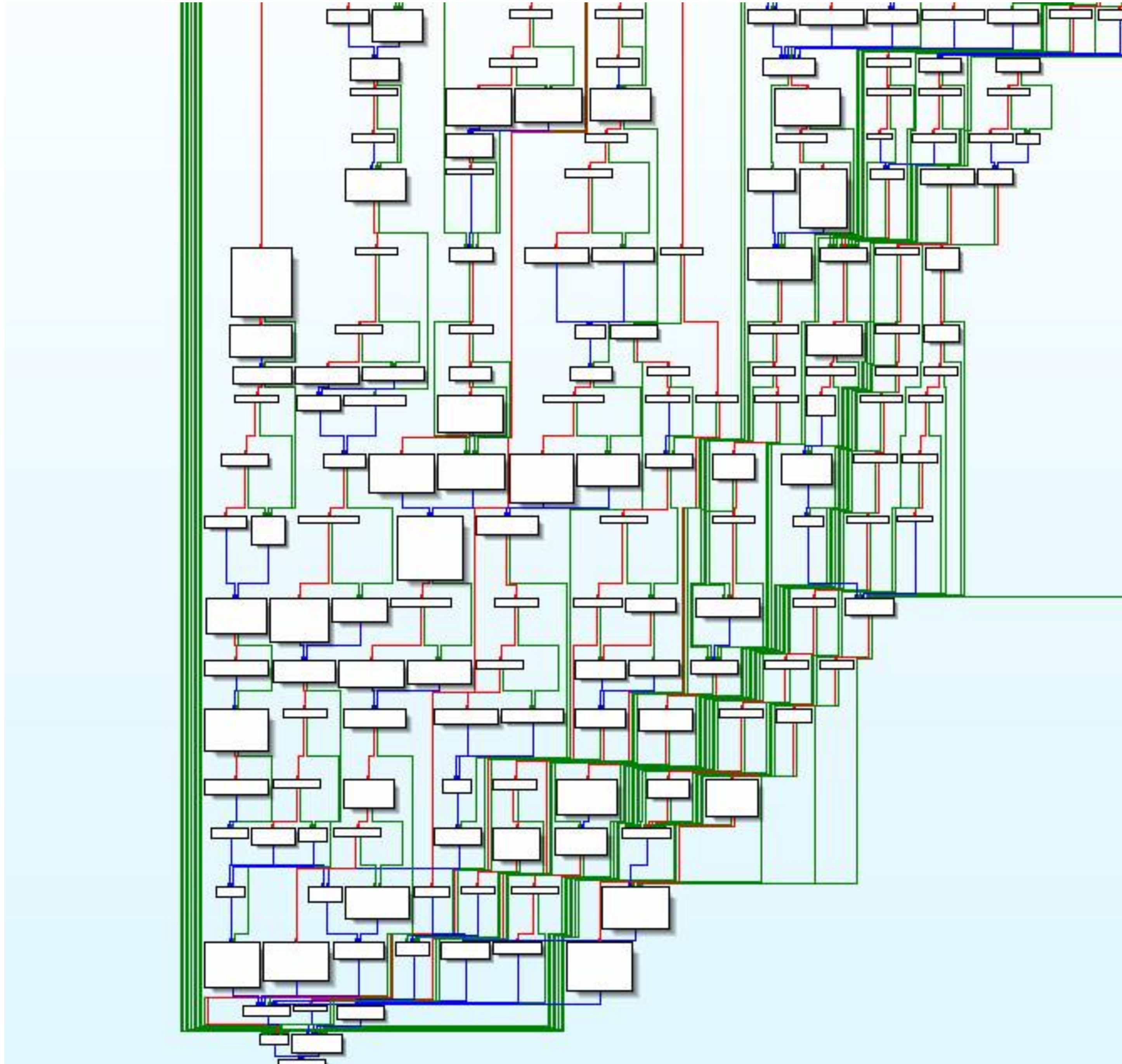
2 Εξομοίωση, όπως εφαρμόζεται σε αντίστροφη μηχανική

Δεδομένου ότι το επίκεντρο της παρούσας εργασίας είναι η εξομοίωση, όπως εφαρμόζεται στην αντίστροφη μηχανική, πρέπει κανείς να εξετάσει την τρέχουσα κατάσταση των πραγμάτων και των εφαρμογών της τεχνολογίας αυτής. Η κατάσταση της αντίστροφης μηχανικής γίνεται όλο και πιο περίπλοκη. Καθώς η εφαρμογή συνεχίζει να εξελίσσεται και να αποκτά περισσότερες δυνατότητες, ο απαραίτητος χρόνος για να κατανοήθει μια εφαρμογή μέσω αντίστροφης μηχανικής αυξάνει σε μεγάλο βαθμό. Αυτές οι πολυπλοκότητες συχνά οδηγούν στην απογοήτευση και την απελπισία για κάποιον που προσπαθεί να κατανοήσει τις δράσεις σε επίπεδο συναρμολόγησης ενός προγράμματος σε στατική αποσυναρμολόγηση.

2.1 Πολύπλοκες διαδρομές κώδικα

Ένα συχνό ανυπέρβλητο έργο, όταν γίνεται αναστροφή λογισμικού, είναι οι πολύπλοκες διαδρομές κώδικα. Οποιοδήποτε δυαδικό δεδομένο μπορεί να περιέχει χιλιάδες, δύσκολες στην κατανόηση και χρονοβόρες λειτουργίες. Είτε αυτό εμφανίζεται ως μία μεγάλη λειτουργία, είτε σαν εκατοντάδες λειτουργίες, το πρόβλημα παραμένει. Η κατανόηση του κώδικα διαδρομής είναι απαραίτητη για τη συνολική κατανόηση της λογικής ενός προγράμματος. Ως εκ τούτου μπορεί να είμαστε σε θέση να χρησιμοποιήσουμε την εξομοίωση για να αποκρυπτογραφήσουμε κρυφούς κόμβους.

Θεωρήστε το παρακάτω παράδειγμα ως μία περίπλοκη διαδρομή κώδικα όπως αυτή εμφανίζεται στο IDA[3]



Για να αναστραφεί στατικά η λειτουργία αυτής της ενιαίας συνάρτησης σε δυαδικό σύστημα θα χρειαστεί ένα μεγάλο χρονικό διάστημα. Αντ' αυτού τα ορίσματα της συνάρτησης

μπορεί να προσδιοριστούν και η συμπεριφορά του κώδικα να προσομοιωθεί. Τα αποτελέσματα μπορούν στη συνέχεια να χρησιμοποιηθούν για να καθορισθούν οι τροποποιήσεις και η λογική που λαμβάνεται υπόψη κατά την υλοποίησης, σύμφωνα με αυτές τις πληροφορίες. Ενώ αυτό μπορεί να φαίνεται σαν μια υπεραπλούστευση ενός περίπλοκου προβλήματος, θα φανεί ότι ο PyEmu μπορεί εύκολα να πετύχει αυτά μέσω διαφόρων μεθόδων.

2.2 Διφορούμενος Κώδικας

Ένα άλλο παράδειγμα στο οποίο θα αναφερθούμε σύντομα σε αυτό είναι ένα φαινομενικά διφορούμενο τμήμα κώδικα που εμποδίζει τη διαδικασία της αντίστροφης μηχανικής. Αυτή είναι μια συνήθης ανεπιθύμητη ενέργεια της στατικής ανάλυσης ενός προγράμματος αλλά δεν αποτελεί πρόβλημα κατά την άμεση ανάλυση με τη χρήση ενός προγράμματος εντοπισμού σφαλμάτων. Ωστόσο, εάν υπάρχει η επιθυμία μεταφοράς μέσω μιας απόλυτα στατικής μεθόδου ανάλυσης, χωρίς να χρειαστεί να καταφύγουμε στη χρήση προγραμμάτων αποσφαλμάτωσης, το πρόβλημα του διφορούμενου κώδικα θα πρέπει να αντιμετωπιστεί..

Παράδειγμα διφορούμενου τμήματος κώδικα

```
    loc_3056C46F:
018 push    ebx
01C call    sub_30025460
018 push    [ebp+11ECh+var_1174]
01C lea     eax, [ebp+11ECh+var_828]
01C push    esi
020 push    eax
024 call    sub_30566839
018 push    ebx
01C call    sub_30025460
018 push    dword ptr [edi+0Ch]
01C call    sub_30025460
018 lea     eax, [ebp+11ECh+var_828]
018 push    eax
01C call    sub_30565A22
018 push    ebx
01C call    sub_30025460
018 push    dword ptr [edi+0Ch]
01C call    sub_30025460
018 cmp     dword_3087B628, esi
018 push    2
01C pop     ebx
018 jnz     short loc_3056C4C6
```

Με μία πρώτη ματιά αυτό το απόσπασμα κώδικα ενός βασικού τμήματος κώδικα δεν δίνει στην πραγματικότητα πολλές πληροφορίες. Ακόμα και αν διατηρήσουμε την υπόλοιπη συνάρτηση ανέπαφη, το εν λόγω τμήμα κώδικα έχει 7 κλάδους, διάφορες τοπικές μεταβλητές, και αυτό που φαίνεται είναι ένα αντικείμενο ή κάποιου είδους δομή. Επί του παρόντος δεν υπάρχουν μέσα για να βοηθήσουν έναν ερευνητή στην οργάνωση και κατανόηση αυτού του

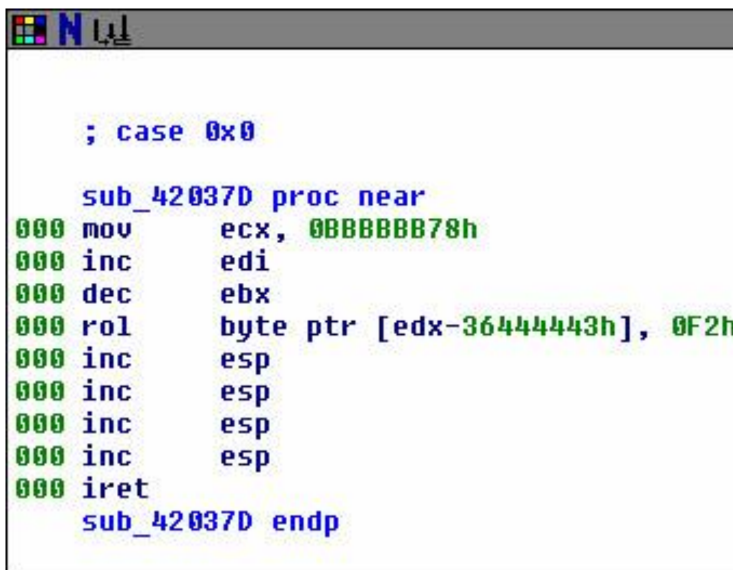
βασικού τμήματος κώδικα. Σε αυτή την περίπτωση ένας εξομοιωτής δημιουργίας σεναρίων θα βοηθήσει σημαντικά, κάνοντας την αντίστροφη μηχανική διαδικασία πιο αποτελεσματική.

2.3 Συσκότιση Κώδικα

Αν και δεν είναι κατ 'ανάγκην γνωστό στις υπηρεσίες παραγωγής, ο κώδικας συσκότισης κερδίζει σημαντικό έδαφος σε εταιρείες που προσπαθούν να προστατεύσουν την πνευματική τους ιδιοκτησία. Με την εμφάνιση και τον πολλαπλασιασμό της αντίστροφης μηχανικής ως μέσο για την απόκτηση πλεονεκτήματος έναντι ενός ανταγωνιστή, πολλές φορές μια εταιρεία μπορεί να προσθέσει απροσπέλαστα τμήματα, για να αποτρέψει αυτό και να προστατέψει τα μυστικά της.

Ο τεχνικές συσκότισης κώδικα ποικίλλουν από παραπλανητικές μεθόδους συναρμολόγησης, εξαπάτησης αποσυναρμολόγησης και εντοπισμού σφαλμάτων, έως παράδοση συναρτήσεων σε εφαρμογή με σκοπό την εξαπάτηση μιας πιθανούς αντίστροφης μηχανικής. Δεδομένου ότι αυτό γίνεται κοινός τόπος για όλα τα λογισμικά, πρέπει κανείς να έχει ένα μέσο για τη γρήγορη μείωση της πολυπλοκότητας, όπως αυτή προκύπτει λόγω της σημασίας αυτών των πραγμάτων.

Ένα απλό παράδειγμα της συσκότισης



Το παραπάνω παράδειγμα δείχνει μια πιθανή προσπάθεια ενός οποιουδήποτε θεατή να ανατρέψει ό,τι πραγματικά μπορεί να συμβαίνει. Θα μπορούσε, επίσης, να είναι μια προσπάθεια να αποτραπεί ο αποσυναρμογητής από τη σωστή ανάλυση σε δυαδικό επίπεδο. Σε αυτήν την περίπτωση μπορεί κανείς να χρησιμοποιήσει έναν εξομοιωτή για να τρέξει όλες τις διαδρομές κώδικα που οδηγούν σε αυτή τη συνάρτηση και διατηρούν τις αξίες, όπως αυτές τροποποιήθηκαν κατά τη διάρκεια της εκτέλεσης.
Αυτό μπορεί ενδεχομένως να επιταχύνει τη διαδικασία καθορισμού για το πώς χρησιμοποιήθηκαν οι τιμές , ή εάν έχουν καμία σημασία σε όλα αυτά.

2.4 Χρόνος

Ο χρόνος είναι ο πιο πολύτιμος και αξιοποιήσιμος από τους πόρους που σχετίζονται με την αντίστροφη μηχανική.
Προχωρώντας στο πεδίο του χρόνου, πρέπει πάντοτε να περιλαμβάνεται η μείωση του χρόνου που χρειάζεται για να εξεταστούν πλήρως τα κομμάτια ενός δυαδικού κώδικα και να φτάσει στο μυθικό 100% του στόχου ως προς την κάλυψη του κώδικα. Αυτό θα επιτευχθεί με ένα συνδυασμό σεναρίων και εργαλείων που βοηθούν εστιάζοντας στο εγχειρίδιο ανάλυσης.

Είναι δύσκολο να ποσοτικοποιηθεί ο χρόνος που μπορεί να χρειαστεί για να κατανοήσει κανείς πλήρως έναν δεδομένο δυαδικό κώδικα.
Πολλοί παράγοντες πρέπει να λαμβάνονται υπόψη κατά τον καθορισμό του τρόπου με τον οποίο μπορεί να σπαταληθεί πολύς χρόνος. Το μέγεθος, οι επιπλοκές, η επάρκεια, και η οργάνωση όλων παίζουν σημαντικό ρόλο στη χρονική εξίσωση όσον αφορά την αντίστροφη μηχανική. Το ακόλουθο παράδειγμα είναι ένα στιγμιότυπο από ένα σημαντικό κομμάτι λογισμικού και των συναρτήσεων του.

Functions window

Function name	Segment	Start	Length	R	F	L	S	B	T	=
sub_303F5508	.text	303F5508	00001467	R	.	.	.	B	.	.
sub_305690CE	.text	305690CE	0000147C	R	.	.	.	B	T	.
sub_3068AF6D	.text	3068AF6D	00001489	R	.	.	.	B	.	.
sub_3056BD48	.text	3056BD48	00001498	R	.	.	.	B	T	.
sub_30573FC6	.text	30573FC6	0000150B	R	.	.	.	B	.	.
sub_303AE2C4	.text	303AE2C4	0000153C	R	.	.	.	B	T	.
sub_3034C00E	.text	3034C00E	00001610	R	.	.	.	B	.	.
sub_3056A725	.text	3056A725	00001623	R	.	.	.	B	.	.
sub_30823F95	.text	30823F95	00001659	R	.	.	.	.	.	.
sub_30533567	.text	30533567	000016A9	R	.	.	.	B	.	.
sub_3080EA3A	.text	3080EA3A	000016E3	R	.	.	.	B	.	.
sub_3046F6D3	.text	3046F6D3	000016F8	R	.	.	.	B	.	.
sub_30653511	.text	30653511	000017CB	R	.	.	.	B	.	.
sub_303F0500	.text	303F0500	00001AF1	R	.	.	.	B	.	.
sub_3058284D	.text	3058284D	00001CF3	R	.	.	.	B	.	.
sub_305B4154	.text	305B4154	00001E0F	R	.	.	.	B	T	.
sub_3045EB4D	.text	3045EB4D	00001E63	R	.	.	.	B	T	.
sub_304306E5	.text	304306E5	000021B7	R	.	.	.	B	.	.
sub_30118D16	.text	30118D16	00002214	R	.	.	.	B	T	.
sub_3058CCE5	.text	3058CCE5	0000225E	R	.	.	.	B	T	.
sub_301F8A44	.text	301F8A44	0000288B	R	.	.	.	.	T	.
sub_304A9CAD	.text	304A9CAD	000028FE	R	.	.	.	B	.	.
sub_304D98F9	.text	304D98F9	00002BB9	R	.	.	.	B	.	.
sub_304D5098	.text	304D5098	00002BE7	R	.	.	.	B	.	.
sub_30635807	.text	30635807	00003281	R	.	.	.	B	.	.
sub_30845A57	.text	30845A57	00004A87	R	.	.	.	B	.	.

Line 27754 of 27754

Όπως μπορούμε να δούμε αυτός ο δυαδικός κώδικας έχει 27754 συναρτήσεις. Παρατηρήστε το μήκος των ταξινομημένων συναρτήσεων. Σε αυτό το παράδειγμα βλέπουμε συναρτήσεις μήκους 0x4A87 (19079) bytes!
Υποθέτοντας ένα εξειδικευμένο λογισμικό αντίστροφης μηχανικής θα διαρκέσει περίπου 10 λεπτά ανά συνάρτηση, για ένα φιλόδοξο χρονοδιάγραμμα an ambitious time frame, (αυτό αγνοεί το γεγονός ότι **950** από τις συναρτήσεις ξεπερνούν αρκετά τα 1024 bytes) ο χρόνος που θα χρειαστεί για να αντιστρέψει το λογισμικό είναι

((27754 * 10) / 60) / 24 = 193 μέρες

Αν υποθέσουμε ότι θα χρειαστούν 10 λεπτά ανά συνάρτηση γεγονός παράλογο, αλλά ακόμη και με τον υπεράνθρωπο στο τιμόνι οπισθοπορείας θα του έπαιρνε 193 ημέρες, ώστε να κατανοήσει πλήρως το 100% του τμήματος του λογισμικού. Όπως μπορεί να δει κανείς ξεκάθαρα, η μείωση του χρόνου με σκοπό την κατανόηση των συναρτήσεων αποτελεί σημαντική προτεραιότητα. Η εξομοίωση είναι μια τεχνική που μπορεί να βοηθήσει σε μεγάλο

βαθμό σε αυτόν τον τομέα.

2.5 Τρέχοντα **Εργαλεία**

Η τρέχουσα λίστα των διαθέσιμων εργαλείων για την αντίστροφη μηχανική και τα εργαλεία που στηρίζονται κυρίως στην Python μεγαλώνει καθημερινά. Με εργαλεία που έχουν αναπτυχθεί με επαγγελματισμό όπως το BinNavi[4], τα έργα κοινοτήτων ανοιχτού κώδικα, όπως το PaiMei[5] και τα σενάρια, καθώς επίσης άλλες επιπρόσθετες λειτουργίες που δημιουργήθηκαν μέσω κοινοτικής συνεισφοράς για το IDA Pro,δεν υπάρχει καμία έλλειψη επιλογών που βοηθούν στα προαναφερθέντα προβλήματα. Ωστόσο, σήμερα υπάρχει μόνο ένας εξομοιωτής που σχετίζεται με την αντίστροφη μηχανική. Ο IDA Pro plugin idax86emu[6] του Chris Eagle επιτρέπει σε έναν χρήστη να προσθέτει τιμές σε μια στοίβα, να αλλάζει και να παρακολουθεί τα μητρώα, ακόμα και να εξομοιώσει βιβλιοθήκες που καλεί. Αν και αυτή είναι μια πολύ καλή πρόσθετη λειτουργία και είναι προφανή τα οφέλη της, οδηγεί στην έλλειψη ευελιξίας και επεκτασιμότητας. Οι πρόσθετες λειτουργίες που είναι γραμμένες σε C, όπως για παράδειγμα όλες οι πρόσθετες λειτουργίες του IDA , μπορεί να είναι μια ευλογία ή μια κατάρα. Αναμφισβήτητα, μπορεί κανείς να ασκήσει δυναμικό έλεγχο, να παρακολουθεί ή να τροποποιεί τιμές σχετικές με την υλοποίηση με εγγενή ταχύτητα και την ευκολία μιας γλώσσας προγραμματισμού. Δεν επιτρέπει απλά εύκολη επέκταση αλλά και την αληθινή ενσωμάτωση της ροής εργασιών.

3 Η Αρχιτεκτονική PyEmu

Προτού προχωρήσουμε σε λεπτομέρειες της αρχιτεκτονικής που σχετίζονται με τον PyEmu, θα πρέπει πρώτα να εξετάσουμε γιατί επιλέχθηκε η Python ως μια γλώσσα προγραμματισμού. Προφανώς, δεν είναι κοινή πρακτική να εξομοιώνεται κώδικας χαμηλού επιπέδου σε γλώσσα υψηλού επιπέδου. Δεδομένου ότι η συναρμολόγηση χαμηλού επιπέδου λειτουργεί σύμφωνα με την απλή βασική υπολογιστική λογική θεωρείται ότι θα είναι απλό να εξομοιωθεί σε μια γλώσσα όπως η Python. Επίσης, ένας αλλά καθοριστικός παράγοντας για την επιλογή της γλώσσας ήταν οι τρέχουσες εξελίξεις σε άλλα εργαλεία της Python.

Πολλοί άνθρωποι απολαμβάνουν τη χρήση της Python και συνεπώς έχουν δημιουργήσει εργαλεία γύρω από αυτή για να βοηθήσουν στα καθήκοντα της όσον αφορά στην αντίστροφη μηχανική. Το IDAPython [7] υπάρχει για να επιτρέπει την πρόσβαση σεναρίων στη γλώσσα προγραμματισμού(IDC) IDA Pro καθώς επίσης επιπρόσθετων εργαλειών SDK. Αυτό από μόνο του δίνει αμέτρητες επιλογές, μία είναι η κατασκευή των πρόσθετων βιβλιοθηκών στην κορυφή της γλώσσας. Μία από αυτές τις βιβλιοθήκες είναι PIDA [8], μια βιβλιοθήκη αφαίρεσης για γρήγορη πρόσβαση σε δομικές πληροφορίες σχετικά με την τρέχουσα δυαδική αποσυναρμολόγηση στην IDA.

Εκτός από την IDA, υπάρχουν και άλλα εργαλεία για τη ζωντανή ανάλυση και δυαδική επεξεργασία. Η Pydbg[9] είναι μια βιβλιοθήκη της Python που τυλίγει την μητρική win32 debugging API που επιτρέπει μια έρευνα εφαρμογής ευέλικτων σεναρίων για τον έλεγχο σφαλμάτων που συμπεριλαμβάνονται κατά την εκτέλεση, την πρόσβαση στη μνήμη, καθώς και σε πληροφορίες για πλαίσια όπως είναι τα μητρώα.. Η Pefile[10] είναι άλλη μια βιβλιοθήκη για

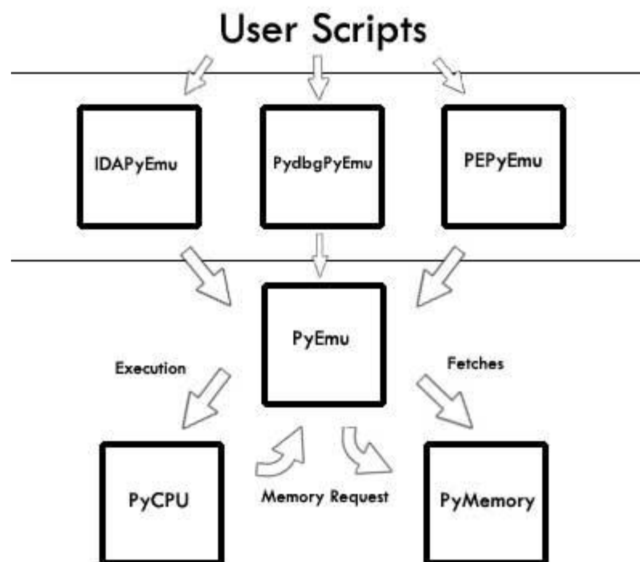
την επεξεργασία εκτελέσιμων PE αρχείων σε Python. Αυτή η βιβλιοθήκη επιτρέπει την μεταγλώττιση των σημαντικών πληροφοριών που αφορούν σε ένα εκτελέσιμο για την αποσυναρμολόγηση συμπεριλαμβανομένων των εισαγωγών, του κώδικα και των τμημάτων δεδομένων, καθώς επίσης των διευθύνσεων των σημείων εισόδου. Τέλος, υπάρχει η pydasm [11], η οποία είναι μια διεπαφή της Python για τον αποσυναρμολογητή της βιβλιοθήκης libdasm[12]. Η Pydasm μπορεί αυθαίρετα να χειρίζεται την αποσυναρμολόγηση των οδηγιών και επιτρέπει σε έναν εξομοιωτή να είναι ακόμη πιο ευέλικτος σε λειτουργία.

3.1 Επισκόπηση

Η αρχιτεκτονική PyEmu λειτουργεί παρέχοντας στον χρήστη έναν ευέλικτο και αφηρημένο API με τη μορφή της κατηγορίας PyEmu. Η κατηγορία αυτή θα χειριστεί την εκτέλεση των οδηγιών, την ανάκτηση της μνήμη και κάθε πληροφορία που έχει ζητηθεί από τους χρήστες. Η αρχιτεκτονική PyEmu χωρίζεται σε τρεις κατηγορίες, συμπεριλαμβανομένων των PyCPU, PyMemory, και φυσικά PyEmu. Με το διαχωρισμό αυτών των πτυχών ενός εξομοιωτή, μπορούμε να παρέχουμε τον έλεγχο για το πώς λειτουργεί κάθε υποσύστημα. Αυτή η δυνατότητα είναι η ουσία της PyEmu. Ως χρήστης, η ικανότητα να ελέγχει τις θέσεις μνήμης, την εκτέλεση των εντολών και την εκτέλεση μέσω καθαρών μεθόδων είναι απίστευτα ευέλικτη.

Όταν ο PyEmu είναι επιφορτισμένος με την εκτέλεση, δίνει οδηγίες στην PyCPU να εκτελέσει μια απλή εντολή. Η PyCPU θα ζητήσει τη μνήμη για την εν λόγω οδηγία από τον PyEMU ο οποίος στη συνέχεια θα διαβιβάσει την αίτηση στην PyMemory. Το αντίθετο συμβαίνει όταν η αίτηση επιστρέφεται.

Αυτό μπορεί να φαίνεται μη-διαισθητικό, αλλά, επειδή ο χρήστης έχει τη δυνατότητα να ελέγχει όλες τις πτυχές αυτής της διαδικασίας μέσω του PyEmu, αυτή απαιτείται. Η αλληλεπίδραση αποδεικνύεται παρακάτω.



Αυτός είναι ο πυρήνας του τρόπου λειτουργίας του πακέτου PyEmu. Σε γενικές γραμμές, ο χρήστης θα πρέπει να διασυνδέεται μόνο με τις κλάσεις του PyEmu, όλες οι χρήσιμες πληροφορίες εκτίθεται μέσω δημόσιων μεθόδων κατά τη δημιουργία εμφανίσεων του PyEmu από την παραγόμενη κλάση. Δεν υπάρχει τίποτα να πω, εάν κάποιος δε θέλει να δημιουργήσει νέες κλάσεις και νέα στρώματα αφαίρεσης.

3.1 PyCPU

Η PyCPU είναι η καρδιά του εξομοιωτή PyEmu. Η PyCPU χειρίζεται όλες τις λογικές εντολές, εκτέλεσης και των σχετικών με την εκτέλεση καθηκόντων του επεξεργαστή. Η δουλειά της κεντρικής μονάδας επεξεργασίας είναι να εκτελεί μια δεδομένη οδηγία που βασίζεται αυστηρά στις προδιαγραφές αναφοράς της Intel[13]. Όπως και με κάθε κομμάτι της αρχιτεκτονικής PyEmu, ο κώδικας CPU προσπαθεί να χειριστεί αυτόνομα όλες αυτές τις απαραίτητες λειτουργίες.

Η πιο βασική δράση είναι η εκτέλεση μιας εντολής. Για την ανάκτηση αυτής την εντολής έχουμε πρόσβαση στη μνήμη του τρέχοντα δείκτη διεύθυνσης της εντολής και την παράδοση της μνήμης στο pydasm. Το Pydasm μας επιτρέπει να αποκωδικοποιούμε σωστά την επιθυμητή οδηγία σε κώδικα εντολών και τελεστές. Αυτή η απλή λειτουργία είναι η ουσία της PyEmu. Επιτρέποντας ο εξομοιωτής την αυθαίρετη αποκωδικοποίηση των οδηγιών, βοηθά στην εξυπηρέτηση διαφόρων συναρτήσεων σε όποιο περιβάλλον θέλουμε, συμπεριλαμβανομένων των ζώντων αναλύσεων μέσω Pydbg ή στατικών αναλύσεων μέσω του IDA Pro.

```
def execute(self):
     # Save our old instruction pointer
     oldeip = self.EIP

     # Fetch raw instruction from memory
     rawinstruction = self.get_memory(self.EIP, 32)
     if not rawinstruction:
         print "[!] Problem fetching raw bytes from 0x%08x" %
(self.EIP)

     return False

     # Decode instruction from raw returning a pydasm.instruction
     instruction = pydasm.get_instruction(rawinstruction,
pydasm.MODE_32)
     if not instruction:
       print "[!] Problem decoding instruction"

        return False

     pyinstruction = PyInstruction(instruction)


     As can be seen once we have grabbed the next instruction we can then
call into our proper mnemonic handler.
```

```
        pyinstruction.mnemonic = pyinstruction.mnemonic.split()[0]

        if pyinstruction.mnemonic in self.supported_instructions:
            if not
self.supported_instructions[pyinstruction.mnemonic](pyinstruction):
                return False
        else:
            print "[!] Unsupported instruction %s" %
 pyinstruction.mnemonic
           return False

        if self.EIP == oldeip:
              #print "[*] Updating eip from 0x%08x -> 0x%08x" %
 (self.EIP, self.EIP + pyinstruction.length)
             self.EIP += pyinstruction.length

        return True
```

Ο PyEmu χρησιμοποιεί τεχνικές βελτίωσης της μνήμης, έτσι ώστε να μπορεί να χειριστεί καθαρά ομάδες κώδικα εντολών μέσα σε ένα μνημονικό. Δεδομένου ότι τα μνημονικά έχουν πολλαπλούς τρόπους λειτουργίας αυτό επιτρέπει στους προγραμματιστές την επέκταση της PyCPU, έτσι ώστε εύκολα να προσθέτουν εύκολα περισσότερα, αν χρειάζεται και να εφαρμόζουν μνημονικά και κώδικες εντολών στο επίπεδο αγκίστρωσης. Σήμερα ο PyEmu υποστηρίζει 100 + Intel IA-32 οδηγίες. Ενώ κάποιος μπορεί να σημειώσει ότι οι προδιαγραφές της Intel περιλαμβάνουν πάνω από 400 μνημονικά, σημαντικό για να γίνει κατανοητό ότι στις περισσότερες περιπτώσεις (είναι περίπου 50) χρησιμοποιήθηκαν σε εφαρμογές πραγματικού κόσμου.

Μόλις η PyCPU έχει τα κατάλληλα μνημονικά, τα θέτει σε συναρτήσεις που χειρίζονται τη συγκεκριμένη εντολή. Ο PyEmu επιδιώκει να είναι εύκολο να επεκταθεί το μεγαλύτερο μέρος της μνήμης και του κώδικα εντολών χειρισμού που έχουν προ-δημιουργηθεί και επιτρέπουν την ομοιόμορφη εμφάνιση και τη λειτουργία όλων των οδηγιών.

```
 def CMP(self, instruction):
        op1 = instruction.op1
        op2 = instruction.op2
       op3 = instruction.op3

        so = instruction.operand_so()
        ao = instruction.address_so()

        op1value = ""
        op2value = ""
        op3value = ""
        op1valuederef = None
        op2valuederef = None

        #38 /r CMP r/m8,r8 Compare r8 with r/m8
        if instruction.opcode == 0x38:

              size = 1
```

```
            if op1.type == pydasm.OPERAND_TYPE_REGISTER:
               op1value = self.get_register(op1.reg, size)
               op2value = self.get_register(op2.reg, size)

               # Do logic
               result = op1value - op2value

               self.set_arithmetic_flags(op1value, op2value,
 result, size)

            elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
               op1value = self.get_memory_address(instruction,
 1, size)
               op2value = self.get_register(op2.reg, size)

               # Do logic
               op1valuederef = self.get_memory(op1value, size)

               result = op1valuederef - op2value

               self.set_arithmetic_flags(op1valuederef,
 op2value, result, size)

         #39 /r CMP r/m16,r16 Compare r16 with r/m16
         #39 /r CMP r/m32,r32 Compare r32 with r/m32
         elif instruction.opcode == 0x39:

               if so:
                  size = 2
               else:
                  size = 4

               if op1.type == pydasm.OPERAND_TYPE_REGISTER:
                  op1value = self.get_register(op1.reg, size)
                 op2value = self.get_register(op2.reg, size)

                  # Do logic
                  result = op1value - op2value

                 self.set_arithmetic_flags(op1value, op2value,
result, size)

            elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
             op1value = self.get_memory_address(instruction,
1, size)
             op2value = self.get_register(op2.reg, size)

              # Do logic
              op1valuederef = self.get_memory(op1value, size)

             result = op1valuederef - op2value

              self.set_arithmetic_flags(op1valuederef,
op2value, result, size)

      #3B /r CMP r16,r/m16 Compare r/m16 with r16
```

```
    #3B /r CMP r32,r/m32 Compare r/m32 with r32
    elif instruction.opcode == 0x3b:

            if so:
                size = 2
            else:
                size = 4

            op1value = self.get_register(op1.reg, size)

            if op2.type == pydasm.OPERAND_TYPE_REGISTER:
                op2value = self.get_register(op2.reg, size)

                # Do logic
                result = op1value - op2value

                self.set_arithmetic_flags(op1value, op2value, result, size)

            elif op2.type == pydasm.OPERAND_TYPE_MEMORY:
                op2value = self.get_memory_address(instruction,
2, size)

            # Do logic
            op2valuederef = self.get_memory(op2value, size)

            result = op1value - op2valuederef

            self.set_arithmetic_flags(op1value,
op2valuederef, result, size)

    #3C ib CMP AL, imm8 Compare imm8 with AL
    elif instruction.opcode == 0x3c:

            size = 1

            op1value = self.get_register(0, size)
            op2value = op2.immediate & self.get_mask(size)

            # Do logic
            result = op1value - op2value

            self.set_arithmetic_flags(op1value, op2value, result,
size)

    #3D id CMP EAX, imm32 Compare imm32 with EAX
    #3D iw CMP AX, imm16 Compare imm16 with AX
    elif instruction.opcode == 0x3d:

            if so:
               size = 2
            else:
               size = 4

            op1value = self.get_register(0, size)
            op2value = op2.immediate & self.get_mask(size)

            # Do logic
```

```
            result = op1value - op2value

            self.set_arithmetic_flags(op1value, op2value, result,
size)

     #81 /7 id CMP r/m32,imm32 Compare imm32 with r/m32
     #81 /7 iw CMP r/m16, imm16 Compare imm16 with r/m16
     elif instruction.opcode == 0x81 and instruction.extindex
== 0x7:

     if so:
        size = 2
     else:
        size = 4

     if op1.type == pydasm.OPERAND_TYPE_REGISTER:
         op1value = self.get_register(op1.reg, size)
         op2value = op2.immediate & self.get_mask(size)

         # Do logic
         result = op1value - op2value

         self.set_arithmetic_flags(op1value, op2value,
result, size)

      elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
         op1value = self.get_memory_address(instruction,
1, size)
         op2value = op2.immediate & self.get_mask(size)

         # Do logic
         op1valuederef = self.get_memory(op1value, size)

         result = op1valuederef - op2value

         self.set_arithmetic_flags(op1valuederef,
op2value, result, size)


     #83 /7 ib CMP r/m16,imm8 Compare imm8 with r/m16
     #83 /7 ib CMP r/m32,imm8 Compare imm8 with r/m32
     elif instruction.opcode == 0x83 and instruction.extindex
== 0x7:

     if so:
        size = 2
     else:
        size = 4

     if op1.type == pydasm.OPERAND_TYPE_REGISTER:
         op1value = self.get_register(op1.reg, size)
         op2value = op2.immediate & self.get_mask(size)

         # Do logic
         result = op1value - op2value

           self.set_arithmetic_flags(op1value, op2value,
```

```
result, size)

      elif op1.type == pydasm.OPERAND_TYPE_MEMORY:
          op1value = self.get_memory_address(instruction,
1, size)
          op2value = op2.immediate & self.get_mask(size)

          # Do logic
          op1valuederef = self.get_memory(op1value, size)

          result = op1valuederef - op2value

          self.set_arithmetic_flags(op1valuederef,
op2value, result, size)

       else:
           return False

       return True
```

Αυτή η λεπτομερής εισαγωγή της εντολής CMP θα πρέπει να είναι πολύ ευανάγνωστη και κατανοητή. Όταν η PyCPU εκτελεί την μνημονική λειτουργία, στη συνέχεια, καθορίζει ποιος κώδικας εντολών ζητάται, και με βάση τις προδιαγραφές της Intel προσδιορίζεται το είδος του τελεστή με τον οποίο καταπιανόμαστε. Αυτό εκτελεί την λογική του κώδικα εντολών και θέτει τα κατάλληλα μητρώα, τη μνήμη, και τις αντίστοιχες σημαίες CPU, αν είναι απαραίτητο.

Η PyCPU βασίζεται σε μια μυριάδα βοηθητικών συναρτήσεων για καθαρή ανάγνωση τιμών μητρώου και διευθύνσεων μνήμης. Στην παρακάτω περίληψη εξηγείται η πιο κοινή από τις βοηθητικές λειτουργίες που επαναλαμβάνεται σε όλο τον πηγαίο κώδικα.της Η PyCPU στηρίζεται σε μεγάλο βαθμό στο μεταβλητό μέγεθος που καθορίζει πόσο μεγάλη θα είναι η μνήμη, ή τα μητρώα μνήμης. Αυτό το μέγεθος τότε δίνεται σε όλες τις λειτουργίες αρωγός σε μια προσπάθεια να κάνει τα πράγματα καθολικά.

- get_register(register, size)

Όπως προκύπτει, αυτό θα ανακτήσει το ζητούμενο μητρώο με βάση το όνομα ή το δείκτη και ορθά θα επιστρέψει την αξία της μάσκας σύμφωνα με την παράμετρο μεγέθους. Αν χρησιμοποιηθεί αυτό θα προκαλέσει, επίσης, ένα στοιχείο χειρισμού συμβάντος εάν ορίζεται για το μητρώο.

- set_register(register, value, size)

Παρόμοια με την get_register αυτή η συνάρτηση θα θέσει κάποια αξία στο υπάρχον μητρώο.

- get_memory_address(instruction, operand_index, size)

Αυτή η συνάρτηση θα υπολογίσει τη διεύθυνση ενός τελεστή ο οποίος ζητάται. Με βάση την οδηγία ModRM / SIB byte και τον κώδικα εντολών, θα επιστρέψει τη διεύθυνση που

χρησιμοποιείται στον τελεστή.

- get_memory(address, size)

Αυτή θα επιστρέψει την αξία των σημείων διευθύνσεως στη μνήμη. Όσον αφορά στην προσπέλαση της μνήμης, θα γίνει λεπτομερής ανάλυση παρακάτω.

- set_memory(address, value, size)

Ρυθμίζει τη διεύθυνση που ζητήθηκε ως προς την αξία και το μέγεθος.

- set_arithmetic_flags(operand_1_value, operand_2_value, result, size)

Μια βολική συνάρτηση για τον καθορισμό των κατάλληλων σημαιών CPU για μια αριθμητική πράξη. Αυτές οι σημαίες περιλαμβάνουν τα CF, OF, SF, PF, και ZF .

- set_shift_flags(result, size)

Παρόμοια με την set_arithmetic_flags εκτός του ότι ενημερώνει μόνο τις σημαίες που χρησιμοποιούνται σε bitwise λειτουργίες ολισθησης. Αυτές οι σημαίες περιλαμβάνουν τα SF, PF, και ZF.

Η κλάση CPU δε θα πρέπει να χρησιμοποιείται άμεσα. Οι κατηγορίες του εξομοιωτή θα διεκπεραιώνουν όλες τις κλήσεις για την εκτέλεση και τα αιτήματα της μνήμης. Αν ένας χρήστης θέλει να επεκτείνει τις υποστηριζόμενες οδηγίες ή τη συμπεριφορά, τώρα μπορεί να το κάνει.

3.2 PyMemory

Το δεύτερο κομμάτι του παζλ του PyEmu είναι ο χειριστής μνήμης. Η PyMemory είναι υπεύθυνη για το χειρισμό τυχόν ανάκτησης ή αποθήκευσης στις θέσεις μνήμης συμπεριλαμβανομένου και του κώδικα. Αυτή η κατηγορία είναι πολύ βασική για το σχεδιασμό, δεδομένου ότι βασίζεται σε μεγάλο βαθμό στις συναρτήσεις που παράγονται από το χρήστη που κάνουν ανάκτηση της άγνωστης μνήμης. Λειτουργεί, κρατώντας μια κρυφή μνήμη του ήδη ανακτημένων σελίδων μνήμης σε τοπικό επίπεδο. Αν μια σελίδα δεν υπάρχει στην κρυφή μνήμη, θα καλέσει την υπερφορτωμένη μέθοδο get_page(). Η get_page () θα χειριστεί την αίτηση, όπως ο χρήστης την έχει ορίσει.

```
def get_memory(self, address, size):
      page = address & 0xfffff000
      offset = address & 0x00000fff

      # Check our cache if not fetch
      if page in self.pages:
```

```
        # Return from our cache
        rawbytes = ""
        for x in xrange(0, size):
            rawbytes += self.pages[page][offset+x]

        if size == 1:
           return struct.unpack("<B", rawbytes)[0]
        elif size == 2:
            return struct.unpack("<H", rawbytes)[0]
        elif size == 4:
           return struct.unpack("<L", rawbytes)[0]
        else:
           return rawbytes
    else:
        # We need to fetch this
        if not self.get_page(page):
            print "[!] Problem getting page"
            return False
        else:
            rawbytes = ""
            for x in xrange(0, size):
                rawbytes += self.pages[page][offset+x]

            if size == 1:
                return struct.unpack("<B", rawbytes)[0]
            elif size == 2:
                return struct.unpack("<H", rawbytes)[0]
            elif size == 4:
                return struct.unpack("<L", rawbytes)[0]
            else:
                return rawbytes

    return False
```

Όταν στην ζωντανή ανάλυση μέσω Pydbg η μέθοδος get_memory () λειτουργεί, όπως φαίνεται παρακάτω.

```
def get_page(self, page):
      try:
      mempage = self.dbg.read_process_memory(page,
 self.PAGESIZE)
      except:
            print "[!] Couldnt read mem page @ 0x%08x" % page
            return False

      self.pages[page] = mempage

      return True
```

Αυτές οι μέθοδοι ανάκτησης και κρυφής μνήμης επιτρέπουν στην PyMem να ελέγχεται εύκολα από πού προέρχονται τα δεδομένα και πώς αυτά αποθηκεύονται. Διατηρεί το τοπικό αντίγραφο ξεχωριστά από τη μνήμη πραγματικών διαδικασιών. Μια καλή χρήση για αυτό θα

ήταν η απόρριψη όλων των σελίδων που χρησιμοποιούνται από τον εξομοιωτή ή η αποκατάσταση της μνήμη του εξομοιωτή πίσω στη διαδικασία αποσφαλμάτωσης.

Η ρύθμιση της μνήμης είναι πολύ παρόμοια σε λειτουργία

.

```
def set_memory(self, address, value, size):
      page = address & 0xfffff000
      offset = address & 0x00000fff

      if isinstance(value, int) or isinstance(value, long):
            if size == 1:
                  packedvalue = struct.pack("<B", int(value))
            elif size == 2:
                  packedvalue = struct.pack("<H", int(value))
            elif size == 4:
                  packedvalue = struct.pack("<L", int(value))
            else:
                  print "[!] Couldnt pack new value of size %d" %
(size)

                  return False
      elif isinstance(value, str):
            packedvalue = value[::-1]
      else:
            print "[!] Dont understand this value type %s" %
      type(value)

            return False

      # Check our page if not fetch
      if page in self.pages:
            newpage = self.pages[page][:offset]
            for x in xrange(0, size):
                  newpage += packedvalue[x]
            newpage += self.pages[page][offset + size:]

            self.pages[page] = newpage

            return True
      else:
            # We need to fetch this
            if not self.get_page(page):
                  print "[!] Problem getting page"
            return False
      else:
            newpage = self.pages[page][:offset]
            for x in xrange(0, size):
                  newpage += packedvalue[x]
            newpage += self.pages[page][offset + size:]

            self.pages[page] = newpage

            return True

return False
```

Ένα ισχυρό χαρακτηριστικό της PyMemory είναι η δυνατότητα να εφαρμόσει τη δική της διαχείριση μνήμης. Για παράδειγμα, αν κάποιο πρόγραμμα καλείται έτσι , γεμίζοντας αιτήσεις μνήμης με «A» μπορεί να γίνει σε πολύ λίγες γραμμές κώδικα. Για την εφαρμογή αυτή απλά επιβαρύνεται η γονική κλάση PyMemory και παρέχει μια μέθοδο την get_memory στην PyMemory.

```
class MyMemory(PyMemory):
    def __init__(self, fillchar="A"):
         self.fillchar = fillchar

         PyMemory.__init__(self)

    def get_page(self, page):
         try:
           mempage = self.fillchar * self.PAGESIZE
         except:
            print "[!] Couldnt read mem page @ 0x%08x" % page return False

         self.pages[page] = mempage

         return True
```

Η PyMemory δε θα πρέπει να χρειάζεται καμία εξωτερική τροποποίηση, εκτός του να υλοποιεί τις λειτουργίες ενός προσαρμοσμένου διαχειριστή. Η PyMemory είναι εξαιρετικά απλή, δεδομένου ότι λαμβάνει και θέτει μνήμη με βάση τον διαχειριστή που χρησιμοποιεί ο χρήστης. Όλα αυτά χειρίζονται, επίσης, εκ μέρους σας από την κλάση του εξομοιωτή.

3.3 PyEmu

Η PyEmu είναι η κύρια διασύνδεση μεταξύ της υποκείμενης CPU και των κλάσεων της Μνήμης και του χρήστη. Στις περισσότερες περιπτώσεις, ο χρήστης υπάγεται στη σχετική τάξη PyEmu και συνεργάζεται με τις παρεχόμενες μεθόδους του PyEmu. Αυτό το στρώμα αφαίρεσης παρέχει μια ταλαιπωρία ελεύθερης μεθόδου λειτουργίας κατά τη συγγραφή του σεναρίου, χρησιμοποιώντας τον PyEmu.

Οι μέθοδοι που εκτίθενται στην PyEmu αυξάνονται καθημερινά και περιλαμβάνουν λειτουργίες για την εκτέλεση, τα ερωτήματα, τη σύνδεση, την αποσφαλμάτωση, την απόρριψη και άλλους διάφορους τρόπους πρόσβασης και ελέγχου του εξομοιωτή της CPU. Ο παρακάτω κατάλογος περιλαμβάνει τις περισσότερες από τις σημαντικές μεθόδους για χρήση σε PyEmu σενάρια που θα αναλυθούν αργότερα.

Εκτέλεση:
- execute
- set_breakpoint

Τροποποίηση:
- set_register
- set_stack_argument
- get_stack_argument
- get_stack_argument
- get_stack_variable
- set_stack_variable
- get_memory
-
set_memory

Χειριστές:
- set_register_handler
- set_library_handler
- set_exception_handler
- set_instruction_handler
- set_opcode_handler
- set_memory_handler
- set_pc_handler
- set_memory_write_handler
- set_memory_read_handler
- set_memory_access_handler
- set_stack_write_handler
- set_stack_read_handler
- set_stack_access_handler
- set_heap_write_handler
- set_heap_read_handler
- set_heap_access_handler

Διάφορα:
- log
- debug
- dump_memory
- restore_context

Αυτή η βασική κατηγορία προορίζεται να κληρονομείται από μια πιο συγκεκριμένη κατηγορία εξομοιωτή. Για παράδειγμα, όταν εργάζεται στο IDA Pro, ένας χρήστης θα θέλει να χρησιμοποιηθεί η κλάση IDAPyEmu, καθώς παρέχει την πρόσθετη υποστήριξη που χρειάζεται κατά τη διάρκεια της εγκατάστασης. Αυτό δε συμβαίνει πάντα και ο χρήστης μπορεί να δημιουργήσει φυσικά και τη δική τους τάξη εξομοίωσης.

4 Using PyEmu

Η χρήση του PyEmu πρέπει να είναι φυσική και ευέλικτη. Προσπαθεί να παρέχει ένα λογικό στρώμα για την επίτευξη των στόχων που ο χρήστης μπορεί να χρειαστεί να λύσει μέσω της εξομοίωσης. Αυτό το κεφάλαιο θα καλύψει με περισσότερες λεπτομέρειες πώς μπορεί να

χρησιμοποιηθεί. Αυτό περιλαμβάνει τη δημιουργία των απαραίτητων αντικειμένων για την εξομοίωση, τη δημιουργία μεταβλητών, τη μνήμη, την εκτέλεση, και την καταγραφή.

4.1 Στιγμιότυπο

Το στιγμιότυπο ενός αντικειμένου PyEmu είναι το πρώτο βήμα για τη δημιουργία ενός σεναρίου PyEmu. Αυτό θα δημιουργήσει την αναγκαία κλάση αντικειμένων για ο,τιδήποτε άλλο έχει επιτευχθεί στον εξομοιωτή. Όταν το στιγμιότυπο του αντικειμένου βρεθεί, το πρώτο βήμα είναι να προσδιορίσετε ποιο θα είναι το περιβάλλον σας.

Υπάρχουν επί του παρόντος τρία περιβάλλοντα που παρέχονται στο πακέτο PyEmu. Αυτά παρουσιάζονται ως διεπαφές για τα IDA Pro, Pydbg, και ένα αυτόνομο αρχείο PE. Το Pro interface IDA επιτρέπει σε ένα χρήστη να εκτελέσει το σενάριο του, στο πλαίσιο του IDA μέσω του IDAPython και ρυθμίζει τις απαραίτητες τιμές των μεταβλητών, των ορισμάτων, και της μνήμης. Η διεπαφή Pydbg επιτρέπει στο χρήστη να γράφει ένα σενάριο pydbg με χρήση του εξομοιωτή απρόσκοπτα, ρωτώντας τη πραγματική μνήμη και το πλαίσιο επεξεργασίας πληροφοριών για θέματα, όπως οι τιμές των μητρώων και οι σημαίες. Το αρχείο PE αποτελεί μια αυτόνομη μέθοδο για τη χρήση ενός εξομοιωτή χωρίς IDA, σε ένα περιβάλλον στατικής ανάλυσης.

Αυτές όλες οι κατηγορίες κληρονομούν από τη βασική κλάση PyEmu και ονομάζονται περιγραφικά μετά το περιβάλλον τους. Στην περίπτωση του IDAPyEmu ένας χρήστης θα μπορούσε να δημιουργήσει το αντικείμενο κάπως έτσι:

```
from PyEmu import IDAPyEmu

emu = IDAPyEmu()
```

Από εδώ ο χρήστης μπορεί στη συνέχεια να έχει πρόσβαση σε όλες τις εκτεθειμένες μεθόδους σχετικές με τον έλεγχο του εξομοιωτή και συναφείς ιδιότητες. Η κλάση IDAPyEmu μπορεί να πάρει πολλά προαιρετικές επιχειρήματα και ορίζεται ως εξής:

```
class IDAPyEmu(PyEmu):
      def __init__(self, stack_base=0x0095f000, stack_size=0x1000,
heap_base=0x000a0000, heap_size=0x2000, frame_pointer=True):

        self.stack_base = stack_base
        self.stack_size = stack_size
        self.heap_base = heap_base
        self.heap_size = heap_size
        self.frame_pointer = frame_pointer

        PyEmu.__init__(self)

        self.setup_memory()

 def setup_memory(self):
        # Sets up memory of emulator
        self.memory = IDAMemory()
```

```
        # Do stack initialization
        self.memory.get_page(self.stack_base)
        self.cpu.set_register32("EBP", self.stack_base -
 self.stack_size / 2)
        self.cpu.set_register32("ESP",
 self.cpu.get_register32("EBP") - 4)

        return True
```

4.2 Εγκατάσταση

Η εγκατάσταση είναι απαραίτητη για τη συμπλήρωση και την οργάνωση των συναφών ιδιοτήτων του εξομοιωτή που θα πρέπει να εκτελεί όπως αναμένεται. Αυτό περιλαμβάνει τη φόρτωση του τμήματος κώδικα και το τμήμα των δεδομένων στη μνήμη.

Για το παράδειγμά μας IDAPyEmu θα κάνουμε το εξής: θα χρησιμοποιήσουμε την πρόσβαση του IDAPython στην IDC. .

```
textstart = SegByName(".text")
textend = SegEnd(textstart)

print "[*] Loading text section bytes into memory"

currenttext = textstart
while currenttext <= textend:
      emu.set_memory(currenttext, GetOriginalByte(currenttext),
size=1)
      currenttext += 1

print "[*] Text section loaded into memory"

datastart = SegByName(".data")
dataend = SegEnd(datastart)

print "[*] Loading data section bytes into memory"

currentdata = datastart
while currentdata <= dataend:
      emu.set_memory(currentdata, GetOriginalByte(currentdata),
size=1)
      currentdata += 1

print "[*] Data section loaded into memory"

emu.set_register("EIP", ScreenEA())
```

Αυτό θα συμπληρώσει τον κωδικό και το τμήμα των δεδομένων στις κατάλληλες βάσεις διευθύνσεων στην κλάση IDAMemory, και θα ρυθμίσετε την PyCPU του μητρώου EIP στην τρέχουσα διεύθυνση που έχει επιλεγεί στο πλαίσιο του IDA Pro. Μερικές γνωστές μέθοδοι που

χρησιμοποιούνται για να επιτευχθεί αυτό.

```
emu.set_register(register, value, name="")
```

Ο ορισμός μητρώο θα θέσει την ένδειξη μητρώου στην αξία που δηλώθηκε. Διαφορετικά από την κλάση PyCPU, μπορεί να ορίσει μόνο το μητρώο με βάση το όνομα. Το μέγεθος δεν είναι απαραίτητο, δεδομένου ότι αυτόματα θα καθορίζεται με βάση το όνομα μητρώου (δηλαδή EAX, AX, AH, AL). Το όρισμα λέξη-κλειδί είναι χρήσιμο να ορίσετε ένα όνομα στο μητρώο που μπορεί να έχει περισσότερο νόημα για τον χρήστη. Για παράδειγμα:

```
emu.set_register(“EAX”, 2, name=”counter”)
```

Θα ορίσει τον καταχωρητή ECX σε 2 και θα δημιουργήσει ένα όνομα "counter" γι 'αυτό. Το μητρώο αυτό μπορεί στη συνέχεια απλά να ανακτηθεί μέσω του ονόματός του, χρησιμοποιώντας get_register ("counter"). Ας ελπίσουμε ότι αυτό θα επιτρέψει μια αντίστροφη μηχανικά εύκολη για να οργανώσετε τις πληροφορίες τους.

```
emu.set_memory(address, value, size=1)
```

Θα ορίσει μνήμη, καθορίζοντας την τιμή στη κρυφή μνήμη του διαχειριστή μνήμης με προϋπόθεση την αξία. Ένα προαιρετικό όρισμα μέγεθος χρησιμοποιείται, επειδή στις περισσότερες περιπτώσεις ο PyEmu θα υπολογίσει αυτόματα το μέγεθος του ορίσματος. Αυτό είναι χρήσιμο για τον καθορισμό tastslike τιμών συμβολοσειράς του αυθαίρετου μήκους στη μνήμη.

```
emu.set_memory(0x41414141, ABCDEFGHIJKLMNOP)
```

Με αυτό το παράδειγμα θα ρυθμίσετε την διεύθυνση μνήμης 0x41414141 στο αλφαριθμητικό που παρέχεται, παράλληλα με τον αυτόματο υπολογισμό του μήκους του. Αυτό θα λειτουργήσει, επίσης, με τις τιμές του τύπου «long» και «int» οι οποίες είναι προαποφασισμένο να είναι μήκους 4 byte. Η συνάρτηση set_memory λειτουργία, τότε θα καλέσει τη συνάρτηση set_memory διαχείρισης μνήμης.

```
def set_memory(self, address, value, size=0):
<…>

if not self.memory.set_memory(address, value, size):
return False

return True
```

Αυτό το παράδειγμα, χρησιμοποιώντας τον IDAPyEmu, μπορεί να φαίνεται περίπλοκο με την πρώτη ματιά. Ωστόσο, όλοι προσπαθούμε να ολοκληρώσουμε την αρχικοποίηση της μνήμης και της CPU για χρήση, δεδομένου ότι έτσι θα ήταν αν είχε εκτελεστεί στο σύστημα. Επίσης, λέμε στην PyCPU ότι θέλουμε να εκτελεστεί από την τρέχουσα επιλεγμένη διεύθυνση στο παράθυρο αποσυναρμολόγησής μας.

4.3 Χειριστές

Οι διαχειριστές είναι ένα από τα μεγαλύτερα οφέλη της χρήσης του PyEmu. Ένας χειριστής επιτρέπει στο χρήστη να δημιουργήσει ορισμένα σημεία που πρέπει να καλέσουμε στον προσαρμοσμένο κώδικά τους. Αυτή η μέθοδος δίνει τον έλεγχο στο σενάριο του χρήστη και επιτρέπει σε αυτόν να λύσει μερικά από τα προβλήματα που αναφέρθηκαν προηγουμένως. Ο PyEmu παρέχει πολλούς χειριστές έξω από την εργαλειοθήκη, ενώ στην πραγματικότητα έχουν σχεδιαστεί με σκοπό την επέκταση.

Όλες οι χειριστές λειτουργούν, χρησιμοποιώντας δείκτες συναρτήσεων. Για να καλύψουν την κλήση ενός χρήστης πρέπει να ορίσουμε μια συνάρτηση, και να περάσει το όνομα της συνάρτησης στην παραγόμενη συνάρτηση χειρισμού – επανάκληση, όταν πληρούνται συγκεκριμένες προϋποθέσεις. Για παράδειγμα

```
def my_handler(emu):
      print "[*] Hit my handler @ %x" % emu.get_register("EIP")

      return True
```

Ένα τρέχον μειονέκτημα των χειριστών είναι ότι τα επιχειρήματα εξαρτώνται από το ποιοί χειριστές καθορίζονται. Στο μέλλον αυτό μπορεί να αλλάξει και να είναι πιο εύκολο μέσω μιας καθορισμένης δομής γεγονότων χειρισμού που περνά στο χρήστη μια προσδιορισμένη δομή επανάκλησης. Μια σημείωση είναι το γεγονός σε όλους τους χειριστές θα πρέπει να δοθεί ένα παράδειγμα της κατηγορίας PyEmu. Αυτό επιτρέπει στο σενάριο να έχει άμεση πρόσβαση στην CPU για τροποποίηση, ερώτηση, ή οποιαδήποτε άλλα καθήκοντα που πρέπει να ολοκληρωθούν.

Οι παρακάτω χειριστές περιλαμβάνουν το πακέτο PyEmu και τις αντίστοιχες μεθόδους που αναφέρονται παρακάτω.

4.3.1 Χειριστές εγγραφής

Οι χειριστές εγγραφής είναι όπως θα τους περιμέναμε. Εάν η ένδειξη του μητρώου έχει τροποποιηθεί, το σενάριο θα λάβει την ευκαιρία να δράσει, τόσο για την καταγραφή της αξίας, όσο και για τη μετατροπή του.

```
emu.set_register_handler("eax", my_register_handler)
```

Η παράμετρος που αντιπροσωπεύει τους καταχωρητές μιμείται τη μέθοδο set_register () και μπορεί να χρησιμοποιηθεί με βάση το όνομά της (δηλαδή EAX, AX, AL, AH) ή το όνομα με το οποίο αυτή έχει οριστεί από τον χρήστη (δηλαδή "counter"). Οι χειριστές εγγραφής είναι ισχυροί στον εντοπισμό τροποποίησης ενός γνωστού, ή στην εγγραφή σημαντικού καταχωρητή στον οποίο ο χρήστης θέλει να δώσει ιδιαίτερη προσοχή.

```
def my_register_handler(emu, register, value, type)
```

Ο ορισμός ενός χειριστή πραγματοποιείται με τη λήψη ενός αντικειμένου από τον εξομοιωτή ,την τιμή του μητρώου του , καθώς και τον τύπο του. Ο τύπος είναι μια συμβολοσειρά που υποδεικνύει τις εντολές "διάβασε" ή "γράψε" στον καταχωρητή.

4.3.2 Χειριστές βιβλιοθήκης

Οι χειριστές βιβλιοθήκης επιτρέπουν στο χρήστη να αντιληφθεί διαδικασία κλήσης βιβλιοθήκης πριν από την έναρξή της. Στον PyEmu, πολλά πρότυπα βιβλιοθήκης που καλούνται εξομοιώνονται με σκοπό να παρέχουν απρόσκοπτη εκτέλεση κατά την κλήση εισαγωγής τους. Ένας χειριστής μπορεί να χρησιμοποιηθεί με σκοπό να αλλάξει αυτή τη συμπεριφορά με τη έναρξη για διαδικασίες, όπως είναι ο έλεγχος μιας τοποθεσίας με τη κλήση της μεθόδου malloc().

```
emu.set_library_handler("malloc", my_library_handler)
```

Το όνομα της βιβλιοθήκης είναι το εξαγόμενο συμβολικό όνομα που δίνεται κατά την εισαγωγή. Αυτή η περίπτωσή δε μεταβάλλει το πρόγραμμα και επιτρέπει στο χρήστη να προσαρμόσει την εκτέλεσή του ακόμη περισσότερο.

```
def my_library_handler(emu, library, address)
```

Κατά τον ορισμό ενός χειριστή θα ληφθεί ένα αντικείμενο από τον εξομοιωτή, καθώς επίσης το όνομα εισαγωγής με το οποίο κλήθηκε και η διεύθυνση της σχετικής εισαγωγής.

4.3.3 Χειριστές εξαίρεσης

Οι χειριστές εξαίρεσης ενεργούν όπως ακριβώς θα περίμενε κανείς. Κάθε φορά που μια εξαίρεση τίθεται σε λειτουργία αυτοί καλούνται. Ένα προφανές παράδειγμα θα μπορούσε να καταπιάνεται με κάποια γενικά σφάλματα λόγω μη έγκυρης πρόσβασης στη μνήμη.

```
emu.set_exception_handler("GP", my_exception_handler)
```

Όπως συνέβη και παραπάνω το πρώτο όρισμα είναι ο κωδικός βλάβης της Intel και οδηγεί στην κλήση ενός χειριστή εξαίρεσης από την CPU.

```
def my_library_handler(emu, exception, address)
```

Ο ορισμός χειριστή αυτού του είδους πραγματοποιείται με τη λήψη ενός αντικείμενου εξομοίωσης, την κλήση του χειριστή εξαίρεσης, και τη διεύθυνση στην οποία παρουσιάστηκε σφάλμα.

4.3.4 Χειριστές οδηγιών

Ο σκοπός των χειριστών οδηγιών είναι να καταστεί δυνατή η αλίευση συγκεκριμένων μνημονικών, αφού αυτά ολοκληρωθούν. Συχνά, όταν εφαρμόζονται ορισμένες εντολές αντίστροφης μηχανικής, το έργο αυτών των χειριστών είναι ιδιαίτερα σημαντικό. Ένα καλό παράδειγμα εφαρμογής των χειριστών οδηγιών είναι η εντολή «CMP» που χρησιμοποιείται για αποφάσεις επιλογής υποπρογραμμάτων. Αν κάποιος θέλει να συνδέσει διαφορετικά υποπρογράμματα, μπορεί να το κάνει με τη χρήση της εντολής «CMP» και ό, τι είχε σχέση με αυτή.

```
emu.set_instruction_handler("cmp", my_instruction_handler)
```

Ένας τέτοιος χειριστής χρειάζεται μόνο το μνημονικό το οποίο πρέπει να παρακρατηθεί όπως και ο σχετικός δείκτης λειτουργίας του.

```
def my_cmp_handler(emu, mnemonic, op1, op2, op3)
```

Η συνάρτηση του χειριστή θα λάβει ένα αντικείμενο από τον εξομοιωτή, το μνημονικό, και τις αξίες όλων των πιθανών ακέραιων τελεστών τύπου dword.

4.3.5 Χειριστές κώδικα πράξεων

Οι χειριστές κώδικα πράξεων είναι ένα υποσύνολο των χειριστών εντολών. Αυτοί επιτρέπουν τον πιο λεπτομερή έλεγχο στην τοποθεσία που επιδιώκεται προσπέλαση. Χρεισιμοποιείστε τους μόνο αν θέλετε να κοινοποιείται όταν εκτελείται μια εντολή μνημονικού «CMP» , και μόνο στις περιπτώσεις σύγκρισης κατά μνήμη, όπως είναι η περίπτωση με το opcode 0x39.

```
emu.set_opcode_handler(0x39, my_opcode_handler)
```

Και πάλι η ρύθμιση των χειριστών γίνεται απλά, όπως θα ανέμενε κανείς.. Στην περίπτωση του κώδικα πράξεων multi-byte απλώς να αναφέρεσθε στους χειριστές ως μεταβλητές τύπου int ,δηλαδή ακέραιοι αριθμοί σταθερού μήκους (δηλαδή 0x0f9c).

```
def my_39_handler(emu, opcode, op1, op2, op3)
```

Η συνάρτηση χειριστή θα λάβει το αντικείμενο απο τον εξομοιωτή συνοδευόμενο με τον κωδικό εντολής και τις αξίες όλων των πιθανών τελεστών ως ακέραιοι αριθμοί τύπου dword.

4.3.6 Χειριστές μνήμης

Ένας χειριστής μνήμης παρέχεται με σκοπό να επιτρέπει σε ένα μέσο την καταγραφή κάθε πρόσβασης σε μια συγκεκριμένη διεύθυνση της μνήμης. Αυτός μπορεί να ε αντιστοιχεί είτε

σε μια εντολή ΄΄διάβασε΄΄ είτε σε μια εντολή ΄΄γράψε΄΄ και θα ενημερώνει σε μεγάλο βαθμό τον χρήστη για εντοπισμό απόπειρας προσπέλασης μνήμης σε μια γνωστή διεύθυνση του χρήστη.

```
emu.set_memory_handler(0x41424344, my_memory_handler)
```

Και πάλι παρέχουμε στο χειριστή τη διεύθυνση μέγεθους dword της διεύθυνσης μνήμης που μας ενδιαφέρει.

```
def my_memory_handler(emu, address, value, size, type)
```

Η συνάρτηση χειρισμού λαμβάνει ένα αντικείμενο του εξομοιωτή μαζί με τη διεύθυνση πρόσβασης, την αξία που διαβάζεται, ή γράφεται στη διεύθυνση και το μέγεθος της αίτησης. Επιπλέον, ορίζεται ένα αλφαριθμητικό το οποίο διαβάζεται ή γράφεται από τον χρήστη.

4.3.6 Χειριστής μετρητή προγράμματος

Ο χειριστής μετρητή προγράμματος χρησιμοποιείται για να ενεργοποιήσει μια επιστροφή κλήσης, όταν η εκτέλεση φτάσει σε μια συγκεκριμένη διεύθυνση μνήμης, επιτρέποντας στο χρήστη να δημιουργήσει σημεία σε δυαδικό κώδικα που να επιτρέπουν τον έλεγχο για τη μεταφορά στον πηγαίο κώδικα..

```
emu.set_pc_handler(0x45464748, my_pc_handler)
```

Συντάσσεται ακριβώς με τον ίδιο τρόπο όπως και οι υπόλοιποι χειριστές .

```
def my_memory_handler(emu, address)
```

Η συνάρτηση του μετρητή λαμβάνει, ως συνήθως, το αντικείμενο εξομοίωσης καθώς και την τιμή του μετρητή προγράμματος μητρώων (δηλαδή EIP).

4.3.7 Χειριστές μνήμης υψηλού επιπέδου

Οι χειριστές μνήμης υψηλού επιπέδου επιτρέπουν μόνο έναν χειριστή ανά δράση. Αυτό παρέχεται ως μια απλή διασύνδεση παρακολούθησης της πρόσβασης στη μνήμη. Αυτοί οι χειριστές παρακολουθούν το διάβασμα, την εκτύπωση, και τις επανακλήσεις πρόσβασης για κάθε μνήμη, κάθε στοίβα, ή τυχόν αιτήματα σωρό από την PyCPU.

```
emu.set_memory_write_handler(my_memory_write_handler)
emu.set_memory_read_handler(my_memory_read_handler)
emu.set_memory_access_handler(my_memory_access_handler)

emu.set_stack_write_handler(my_stack_write_handler)
emu.set_stack_read_handler(my_stack_read_handler)
emu.set_stack_access_handler(my_stack_access_handler)
```

```
emu.set_heap_write_handler(my_heap_write_handler)
emu.set_heap_read_handler(my_heap_read_handler)
emu.set_heap_access_handler(my_heap_access_handler)
```

Δεν υπάρχει επιλογή ώστε να καθορίσετε τη διεύθυνση του χειριστή. Αυτό υλοποιείται καλύτερα με τη μέθοδο set_memory_handler (). Και πάλι αυτοί οι χειριστές αποτελούν συναρτήσεις που προσφέρονται κυρίως για σκοπούς καταγραφής δεδομένων.

```
def my_memory_write_handler(emu, address)
def my_memory_read_handler(emu, address)
def my_memory_access_handler(emu, address, type)
```

Αυτές οι συναρτήσεις χειρισμού λαμβάνουν ένα αντικείμενο εξομοιωτή και στην περίπτωση της συγγραφής ή διαβάσματος ενός χειριστή προσπελαύνεται η αντίστοιχη διεύθυνση. Στην περίπτωση της προσπέλασης μνήμης ο τύπος που επιστρέφεται είναι μια συμβολοσειρά που προέρχεται από τη κλήση των διαδικασιών "διάβασε" ή "γράψε".

Οι χειριστές μνήμης υψηλού επιπέδου είναι απλοί στη χρήση τους και αποτελούν ένα εξαιρετικά ισχυρό εργαλείο στην πράξη. Ας ελπίσουμε ότι θα αναλύεται παραπάνω ο σκοπός τους με σαφήνεια και με τρόπο τέτοιο που να βοηθήσει στην ενίσχυση κάθε έργου που υλοποιείται με τον PyEmu.

4.4 Εκτέλεση

Εκτέλεση είναι ο τρόπος με τον οποίο υλοποιείται η όλη διαδικασία της εξομοίωσης κώδικα στα πλαίσια του PyEmu. Η βασική ιδέα της εκτέλεσης είναι απλή, θέλουμε τον εξομοιωτή μας να πάει από το σημείο του α, στο σημείο b.
Αυτό επιτυγχάνεται με διάφορους τρόπους. Η μέθοδος execute() είναι ο μόνος τρόπος για την προώθηση της CPU και ορίζεται ως

```
execute(self, steps=1, start=0x0, end=0x0)
```

Όλα τα ορίσματα είναι προαιρετικά. Να χρησιμοποιείται τη μέθοδο αυτή μόνη της, αυτό θα διευρύνει σύμφωνα με τον τρέχοντα μετρητή προγράμματος PyCPU. Στην περίπτωση της IDAPyEmu αυτή είναι η τρέχουσα θέση του δρομέα κατά την διαδικασία αποσυναρμολόγησης. Όλα τα προαιρετικά ορίσματα μπορούν να χρησιμοποιηθούν σε οποιοδήποτε συνδυασμό και λειτουργούν όπως ακριβώς αναμένεται. Κρατώντας έναν εσωτερικό μετρητή εξομοίωσης η διαδικασία τερματίζεται όταν ο αριθμός των βημάτων έχει επιτευχθεί. Η έναρξη μπορεί να καθοριστείμε σκοπό τη δημιουργία μιας διαφορετικής θέσης εξομοίωσης από ό,τι τρέχουσα θέση. Η εντολή "end" μας επιτρέπει να ορίσουμε το σημείο τερματισμού. Σημειώστε ότι η εντολή "end" μπορεί να μας παραπλανήσει όταν συναντάται σε μια σύνθετη συνάρτηση όπως πολλές φορές συμβαίνει η διεύθυνση κατά τύχη να είναι αδύνατο να επιτευχθεί σε περιπτώσεις όπου μια διακλάδωση ή μια κλήση δεν υλοποιείται.

Η μέθοδος της εκτέλεση είναι, και πρέπει να είναι, απλή. Δίνοντας βήματα , την έναρξη ,

και το τέλος μιας συνάρτησης ικανοποιείται το 99% των περιπτώσεων που μπορεί να χρησιμοποιηθεί η μέθοδος αυτή. Με το να προσθέσετε περισσότερες τέτοιες μεθόδους δεν θα συναντήσετε προβλήματα.

4.5 Τροποποίηση

Η δυνατότητα να τροποποιούνται και να αρχικοποιούνται τα δεδομένα σε έναν εξομοιωτή είναι ζωτικής σημασίας. Ο PyEmu προσπαθεί να παρέχει στο χρήστη τη δυνατότητα να καθορίζει τα δεδομένα σε διάφορες τοποθεσίες, και να οργανώνει τα δεδομένα, έτσι ώστε να έχουν νόημα κατά τη διάρκεια της αντίστροφης μηχανικής. Για τις περισσότερες περιπτώσεις, τα δεδομένα που ο χρήστης βλέπει μπορούν να χωριστούν σε 4 κατηγορίες: μητρώα, μεταβλητές στοίβας, ορίσματα στοίβας, και άλλες μνήμες. Οι τέσσερεις αυτές περιπτώσεις δεδομένων μπορεί να δημιουργήσουν μεγάλη σύγχυση. Αυτές οι τέσσερις κατηγορίες υποστηρίζονται από μεθόδους για τον καθορισμό των ορισμάτων και απόδοση τιμής σε αυτά.

```
emu.set_register("eax", 0x1234567, name="counter")
emu.get_register("eax")
emu.get_register("counter")
```

Οι κατηγορίες μητρώων έχουν αντιμετωπιστεί παραπάνω. Στην περίπτωση της ρύθμισης μητρώου θα πρέπει να δώσετε το όνομα της ομάδας καταχωρητών , να αρχικοποιήσετε δίνοντας τιμές στα μητρώα και ένα όνομα της επιλογής σας. Τέλος, βλέπουμε πώς μπορείτε να αποκτήσετε πρόσβαση σε αυτήν την τιμή με βάση το όνομα στο μέλλον. Με τη χρήση μιας εύκολης ετικέτας η πληροφορία παίρνει μια αναγνώσιμη από τον άνθρωπο μορφή.

```
emu.set_stack_variable(0x80, 0x12345678, name="var_80")
emu.get_stack_variable(0x80)
emu.get_stack_variable("var_80")
```

Αυτό μπορεί να προκαλέσει σύγχυση στην αρχή, δίνοντας μια αξία ως πρώτο όρισμα. Είναι απλά η μετατόπιση από το δείκτη στοίβας ή το πλαίσιο δείκτη σε περιπτώσεις όπου έχουμε ένα δείκτη πλαισίου. Αναγνωρίζεται εύκολα στο πλαίσιο του IDA ως ετικέτα της τοπικής μεταβλητής που μας αφορά. Σε ζωντανή ανάλυση, μπορεί να σταχυολογηθεί πέρνοντας μια μετατόπιση για τη διεύθυνση από το σχετικό μητρώο της στοίβας. Το "όνομα" του προαιρετικού ορίσματος επιτρέπει την καλύτερη οργάνωση των πληροφοριών.

```
emu.set_stack_argument(0x8,          0xaabbccdd,          name="arg_0")
emu.get_stack_argument(0x8)
emu.get_stack_argument("arg_0")
```

Παρόμοια σχεδόν σε κάθε τομέα της κατηγορίας μεθόδων stack_variable , τα ορίσματα της στοίβα λειτουργούν με τον ίδιο τρόπο, εκτός από αυτά που αποτελούν τις διευθύνσεις των επιχειρημάτων που ωθείται στη στοίβα πριν από το τρέχον πλαίσιο λειτουργίας.

```
emu.set_memory(0x12345678, "ABCDEFGHIJKLMNOP")
emu.set_memory(0x12345678, 0x12345678, size=2)
emu.get_memory(0x12345678, size=4)
```

Ο καθορισμός και η διαχείριση τμημάτων μνήμης είναι απλές διαδικασίες. . Παρέχοντας μια διεύθυνση και την αξία, η διεύθυνση της μνήμης θα πρέπει να οριστεί σε αυτή την τιμή. Το μέγεθος, πάλι, μπορεί στις περισσότερες περιπτώσεις να προσδιορίζεται αυτόματα, αλλά υποστηρίζεται ο καθορισμός μιας τιμής διαφορετικού μεγέθους. Η διαχείριση μνήμης λειτουργεί όπως αναμένεται με μια δοθείσα διεύθυνση που θα διακόψει την αναφορά στη διεύθυνση μνήμης και επιστρέφει την αξία. Σημειώστε ότι αν ζητήσετε μια σειρά μεγέθους > 4 τα δεδομένα αυτόματα θα επιστραφούν ως ένα αλφαριθμητικό.

5 Ο πραγματικός κόσμος

Τώρα που έχουμε κατανοήσει πλήρως το πώς τα πάντα λειτουργούν και μπορούν να χρησιμοποιηθούν, θα πρέπει να διερευνήσουμε μερικά πραγματικά παραδείγματα χρήσης του PyEmu. Ποικίλες εργασίες έχουν επιλεγεί για να αποδείξουν κάποιες προφανείς χρήσεις ενός εξομοιωτή με δυνατότητα δημιουργίας σεναρίων. Αυτό θα πρέπει, επίσης, να δώσει στον αναγνώστη ένα σημείο αναφοράς για το πώς μπορούν να εφαρμοστούν στον PyEmu ιδιαίτερα προβλήματα με σκοπό τη μεγιστοποίηση της αποτελεσματικότητας.

5.1 IDA Pro

Έχουμε ήδη συζητήσει για τη χρήση PyEmu λεπτομερώς. Η κύρια μέθοδος που εφαρμόζεται σήμερα χρησιμοποιεί IDAPython για την εκτέλεση σεναρίων του χρήστη. Στο σενάριό μας πρώτα χαρτογραφούμε το τμήμα κώδικα και το τμήμα των δεδομένων στη μνήμη. Αφού ολοκληρωθεί η διαδικασία αυτή, μπορούμε να εκτελέσουμε στον εξομοιωτή και λειτουργήσουν τα σενάρια όπως προβλέπεται. Το απόσπασμα που ακολουθεί θα εκτυπώσει κάθε εντολή εκτέλεσης σύμφωνα με τον εξομοιωτή.

```
emu = IDAPyEmu()

# Load .text and .data sections into memory
<…>

emu.set_register("EIP", ScreenEA())
emu.debug(2)

emu.execute(steps=5)
```

Και μετά την εκτέλεση

```
.text:00427E6B                                    ; ---------------------------------------
.text:00427E6B
.text:00427E6B                                    loc_427E6B:                              ; C
.text:00427E6B 8B 4D E8                                      mov     ecx, [ebp+var_18]
.text:00427E6E 89 4D D8                                      mov     [ebp+var_28], ecx
.text:00427E71 8D 55 E8                                      lea     edx, [ebp+var_18]
.text:00427E74 89 55 F4                                      mov     [ebp+var_C], edx
.text:00427E77 8D 45 D8                                      lea     eax, [ebp+var_28]
.text:00427E7A 83 E8 01                                      sub     eax, 1
.text:00427E7D 89 45 F0                                      mov     [ebp+arg_18], eax
.text:00427E80 C7 45 F8 04 00+                               mov     [ebp+var_8], 4
.text:00427E87
.text:00427E87                                    loc_427E87:                              ; C
.text:00427E87 8B 4D F8                                      mov     ecx, [ebp+var_8]
.text:00427E8A 83 E9 01                                      sub     ecx, 1
```

```
[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory
[*] Executing [0x427e6b][8b] mov ecx,[ebp-0x18]
[*] Executing [0x427e6e][89] mov [ebp-0x28],ecx
[*] Executing [0x427e71][8d] lea edx,[ebp-0x18]
[*] Executing [0x427e74][89] mov [ebp-0xc],edx
[*] Executing [0x427e77][8d] lea eax,[ebp-0x28]
[*] Executing [0x427e7a][83] sub eax,0x1
[*] Executing [0x427e7d][89] mov [ebp-0x10],eax
[*] Executing [0x427e80][c7] mov dword [ebp-0x8],0x4
[*] Done
```

5.2 Pydbg

Ο Pydbg μπορεί επίσης να είναι ένα μέσο μεταφοράς για τις εργασίες εξομοίωσης. Η επιλογή της ζωντανής ανάλυσης μπορεί να παράσχει ισχυρή ευελιξία στον καθορισμό των μονοπατιών κώδικα και της συμπεριφοράς των εφαρμογών. Η προσθήκη του εξομοιωτή θα πρέπει να είναι απρόσκοπτη. Σε αυτό το παράδειγμα ένα σενάριο θα δημιουργήσει το παράδειγμα pydbg και το αποδίδει σε μια διαδικασία που εκτελείται από τους χρήστες κατ' επιλογή. Μόλις συνδεθεί, ένα σημείο καμπής ορίζεται με σκοπό να υποδεικνύει τη διεύθυνση που θέλετε να ξεκινήσετε την εξομοίωση. Όταν η διεύθυνση πληγεί, δημιουργούμε ένα έναν βρόγχο επανάληψης, χρησιμοποιώντας τον εξομοιωτή ως κονσόλα επιθεώρησης των τιμών που έχουν οι καταχωρητές. Ένα απόσπασμα από το σενάριο μοιάζει με αυτό:

```
# Our pydbg initial break point handler
def handler_breakpoint(dbg):
      # Initial module bp we need to process entries
      If dbg.first_breakpoint:
          print "[*] First bp hit setting emu address @ 0%08x" %
dbg.emuaddress

          # Set up a custom break point handler for the emulator
          dbg.bp_set(dbg.emuaddress,
handler=handler_emu_breakpoint, restore=False)
```

```
            return DBG_CONTINUE

        print "[!] Unknown bp caught @ 0%08x" % dbg.exception_address
        return DBG_CONTINUE

# Do the emulation once the requested bp has been reached
def handler_emu_breakpoint(dbg):
        if dbg.exception_address != dbg.emuaddress:
            print "[!] Emulator handler caught unknown bp @ 0x%08x"%
(dbg.exception_address)

            return DBG_CONTINUE

        # Create a new emulator object passing a pydbg instance
        emu = PyDbgPyEmu(dbg)

        c = None
        while c != "x":
            emu.execute()
            emu.dump_regs()

            c = raw_input("emulator> ")

        return DBG_CONTINUE

procname = sys.argv[1]
emuaddress = sys.argv[2]

dbg.set_callback(EXCEPTION_BREAKPOINT, handler_breakpoint)

if not attach_target_proc(dbg, procname):
        print "[!] Couldnt load/attach to %s" % procname

        sys.exit(-1)

dbg.debug_event_loop()
```

Μόλις εκτελεστεί, η έξοδος μπορεί να μοιάζει ως:

```
C:>pydbgpyemu.py calc.exe 0x001001AF3

[*] Trying to attach to existing calc.exe
[*] Attaching to calc.exe (2516)
[*] First bp hit setting emu address @ 001001af3

[*] Executing [0x1001af3][6a] push byte 0xc
EAX: 0x0000002e
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f90c
ESI: 0x00000001
EDI: 0x0007f95c
EFLAGS: 0x244 [ ZF PF IF ]
```

```
EIP: 0x01001af5
emulator> t

[*] Executing [0x1001af5][58] pop eax
EAX: 0x0000000c
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f910
ESI: 0x00000001
EDI: 0x0007f95c
EFLAGS: 0x244 [ ZF PF IF ]
EIP: 0x01001af6
emulator> t

[*] Executing [0x1001af6][33] xor edi,edi
EAX: 0x0000000c
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f910
ESI: 0x00000001
EDI: 0x00000000
EFLAGS: 0x240 [ ZF IF ]
EIP: 0x01001af8
emulator> t

[*] Executing [0x1001af8][57] push edi
EAX: 0x0000000c
ECX: 0x00000000
EDX: 0x00000005
EBX: 0x010012a0
EBP: 0x0007f99c
ESP: 0x0007f90c
ESI: 0x00000001
EDI: 0x00000000
EFLAGS: 0x240 [ ZF IF ]
EIP: 0x01001af9
emulator> x

[*]Exiting the emulator.

C:>
```

Αυτό το απόσπασμα κώδικα είναι πολύ απλό και άμεσο. Με μια ενισχυμένη κονσόλα εξομοιωτή, ο συνδετικός κρίκος ανάμεσα στην ζωντανή εκτέλεση και τα παραδείγματα εκτέλεσης εξομοίωσης θα μπορούσε να υλοποιηθεί.

5.3 PE

Η μορφή του αρχείου PE περιέχει όλες τις απαραίτητες πληροφορίες για την εκτέλεση μιας εφαρμογής. Αυτό περιλαμβάνει διάφορα τμήματα κώδικα, τα δεδομένα και τις σχετικές διευθύνσεις τους από τη βάση των εικόνων. Δεδομένου ότι έχουμε πρόσβαση σε αυτές τις πληροφορίες και τη βιβλιοθήκη pefile python, μια γρήγορη υλοποίηση της κατηγορίας PEPyEmu έχει ολοκληρωθεί. Η κλάση αυτή σας επιτρέπει να γραφούν σενάρια, χωρίς την ανάγκη αποσυναρμολόγησης του IDA. Το σενάριο προς χρήση είναι απλό. Παίρνει ένα εκτελέσιμο όνομα και μια διεύθυνση, τα εξομοιώνει σε 10 βήματα

```
#!/usr/bin/env python

import os, sys, pefile

from PyEmu import PEPyEmu

exename = sys.argv[1]
address = int(sys.argv[2], 16)

emu = PEPyEmu(exename)
emu.debug(2)

emu.set_register("EIP", address)

emu.execute(steps=10)
```

Και η έξοδος του σεναρίου είναι

```
C:>pepyemu.py "examples.exe" 0x010022F9

[*] Image Base Addr: 0x01000000
[*] Code Base Addr: 0x01001000
[*] Data Base Addr: 0x01014000
[*] Entry Point Addr: 0x01012475

[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory

[*] Executing [0x10022f9][55] push ebp
[*] Executing [0x10022fa][8b] mov ebp,esp
[*] Executing [0x10022fc][81] sub esp,0x108
[*] Executing [0x1002302][53] push ebx
[*] Executing [0x1002303][56] push esi
[*] Executing [0x1002304][8b] mov esi,[ebp+0xc]
[*] Executing [0x1002307][8b] mov eax,[esi+0x10]
[*] Executing [0x100230a][57] push edi
[*] Executing [0x100230b][33] xor edi,edi
[*] Executing [0x100230d][89] mov [esi+eax*2+0x14],di

C:>
```

Αυτό το παράδειγμα καταδεικνύει την ευελιξία της PyEmu. Δεδομένου ότι το μόνο που

απαιτείται είναι ακατέργαστα byte οδηγιών, οι δυνατότητες εφαρμογής είναι πολυάριθμες. Αυτό μπορεί να επιτευχθεί, επειδή η PyEmu προσπαθεί να είναι όσο το δυνατόν περισσότερο αυτόνομη, όταν ασχολείται με εφαρμοζόμενες τάξεις PyEmu. Με τον τρόπο αυτό επιτρέπουμε στο χρήστη να έχει πλήρη έλεγχο πάνω σε ό,τι προσπαθεί να επιτύχει. Θα ήταν μάλιστα δυνατόν να δημιουργηθεί ένα NetPyEmu, αν αυτό είναι επιθυμητό.

5.4 Παρακολούθηση προσπέλασης μνήμης

Ο Καθορισμός του πότε τα δεδομένα διαβάζονται και γράφονται στη μνήμη είναι ζωτικής σημασίας για την κατανόηση του πώς λειτουργεί μια εφαρμογή. Μια συχνή ερώτηση κατά τον καθορισμό αυτό είναι: «Πότε και πού έχει πρόσβαση η μνήμη ». Για να λυθεί αυτό με PyEmu, μπορούμε να δημιουργήσουμε κάποιους υψηλότερου επίπεδου χειριστές πρόσβασης στη μνήμη. Αυτοί οι χειριστές θα επιστρέψουν τον έλεγχο, όταν κάτι τροποποιεί τη μνήμη διεργασιών. Το ακόλουθο παράδειγμα υλοποιείται στον IDA Pro.

```
from PyEmu import IDAPyEmu

def my_memory_access_handler(emu, address, value, size, type):

print "[*] Hit my_memory_access_handler %x: %s (%x, %x, %x,
%s)" % (emu.get_register("EIP"), emu.get_disasm(), address, value,
size, type)

    return True

# Our usual IDA setup mapping relevant sections
<…>

# Start the program counter at the current location in the
disassembly window
emu.set_register("EIP", ScreenEA())

# Set up our memory access handler
emu.set_memory_access_handler(my_memory_access_handler)

emu.execute(start=0x00427E6B, end=0x00427E8D)

print "[*] Done"
```

Και ακολουθεί η έξοδος

```
.text:00427E6B                                   ; ---------------------------------------------------------------
.text:00427E6B
.text:00427E6B                  loc_427E6B:                                     ; CODE XREF: sub_427
.text:00427E6B 8B 4D E8                          mov     ecx, [ebp+var_18]
.text:00427E6E 89 4D D8                          mov     [ebp+var_28], ecx
.text:00427E71 8D 55 E8                          lea     edx, [ebp+var_18]
.text:00427E74 89 55 F4                          mov     [ebp+var_C], edx
.text:00427E77 8D 45 D8                          lea     eax, [ebp+var_28]
.text:00427E7A 83 E8 01                          sub     eax, 1
.text:00427E7D 89 45 F0                          mov     [ebp+arg_18], eax
.text:00427E80 C7 45 F8 04 00+                   mov     [ebp+var_8], 4
.text:00427E87
.text:00427E87                  loc_427E87:                                     ; CODE XREF: sub_427
.text:00427E87 8B 4D F8                          mov     ecx, [ebp+var_8]
.text:00427E8A 83 E9 01                          sub     ecx, 1
.text:00427E8D 89 4D F8                          mov     [ebp+var_8], ecx
.text:00427E90 78 34                             js      short loc_427EC6
.text:00427E92 8B 55 F0                          mov     edx, [ebp+arg_18]
.text:00427E95 83 C2 01                          add     edx, 1
.text:00427E98 89 55 F0                          mov     [ebp+arg_18], edx
.text:00427E9B 8B 45 F0                          mov     eax, [ebp+arg_18]
.text:00427E9E 0F B6 08                          movzx   ecx, byte ptr [eax]
.text:00427EA1 89 4D FC                          mov     [ebp+our_dword_byte], ecx
.text:00427EA4

[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory
[*] Hit my_memory_access_handler 427e6b: mov ecx,[ebp-0x18] (95e7e8, 41414141, 4, read)
[*] Hit my_memory_access_handler 427e6e: mov [ebp-0x28],ecx (95e7d8, 41414141, 4, write)
[*] Hit my_memory_access_handler 427e74: mov [ebp-0xc],edx (95e7f4, 95e7e8, 4, write)
[*] Hit my_memory_access_handler 427e7d: mov [ebp-0x10],eax (95e7f0, 95e7d7, 4, write)
[*] Hit my_memory_access_handler 427e80: mov dword [ebp-0x8],0x4 (95e7f8, 4, 4, write)
[*] Hit my_memory_access_handler 427e87: mov ecx,[ebp-0x8] (95e7f8, 4, 4, read)
[*] Done
```

Η επίλυση ενός προβλήματος και η απάντηση σε ερωτήματα, όπως αυτό που έχει τεθεί παραπάνω, βοηθούν την την μηχανική αναστροφή στην επιτάχυνση στην επιτάχυνση της κατανόησης μιας συνάρτησης, ή ομάδας συναρτήσεων. Μπορούμε, επίσης, να δούμε τη χρήση της εκτέλεσης από την αρχή και το τέλος της διαδικασίας για τον γρήγορο προσδιορισμό των ορίων του εξομοιωτή.

5.5 Απαρίθμηση διαδρομών

Προηγουμένως αποδείξαμε μια εξαιρετικά πολύπλοκη συνάρτηση. Η συνάρτηση περιλαμβάνει εκατοντάδες διαδρομές κώδικα αποφάσεων και εμφανίζεται ως ιστός της αράχνης των διακλαδώσεων και των βρόχων. Για να αμβλυνθεί αυτό, μπορεί κανείς να χρησιμοποιήσει τον PyEmu, για να παρακολουθήσει αυτές τις διακλαδώσεις, τους όρους και τις τιμές που χρησιμοποιούνται κατά την απόφαση. Στην περίπτωση αυτή, συνδέοντας κάθε κλήση προς το μνημονικό «cmp» μας προσφέρεται μια απλή προβολή των συγκρίσεων που συμβαίνουν πριν από τη λήψη κάθε διακλάδωσης. Ενώ αυτό μπορεί να γίνει και με άλλους τρόπους, θα μπορούσαμε επίσης, να θέλουμε να παρέχουν συγκεκριμένες τιμές για να αλλάξουμε τη διαδρομή του κώδικα. Στον IDAPyEmu απλά θα δημιουργούσαμε ένα μνημονικό χειρισμού για την εντολή «cmp» και καταγράφαμε αξίες.

```
from PyEmu import IDAPyEmu

def my_cmp_handler(emu, address, op1, op2, op3):
    print "[*] Hit my_cmp_handler %x: %s (%x, %x)" %
```

```
(emu.get_register("EIP"), emu.get_disasm(), op1, op2)

      return True

# Start the program counter at the current location in the
disassembly window
emu.set_register("EIP", ScreenEA())

# This demonstrates setting local variables used in our
comparisons
emu.set_stack_variable(0x2c,          0x00000000,          name="var_2C")
emu.set_stack_variable(0x1d,          0x00000001,          name="var_1D")
emu.set_stack_variable(0x1e, 0x00000002, name="var_1E")

# Set up our memory access handler
emu.set_mnemonic_handler("cmp", my_cmp_handler)

emu.execute(start=0x00427E46, end=0x00427E6B)

print "[*] Done"
```

Αυτό το σενάριο θα οδηγούσε στο ακόλουθο

```
.text:00427E43 83 C4 08                    add     esp, 8
.text:00427E46 89 45 D4                    mov     [ebp+var_2C], eax
.text:00427E49 83 7D D4 00                 cmp     [ebp+var_2C], 0
.text:00427E4D 74 08                       jz      short loc_427E57
.text:00427E4F 8B 45 D4                    mov     eax, [ebp+var_2C]
.text:00427E52 E9 CA 00 00 00              jmp     loc_427F21
.text:00427E57                  ; ---------------------------------------
.text:00427E57
.text:00427E57                  loc_427E57:                            ; C
.text:00427E57 0F B6 55 E3                 movzx   edx, [ebp+var_1D]
.text:00427E5B 0F B6 45 E2                 movzx   eax, [ebp+var_1E]
.text:00427E5F 3B D0                       cmp     edx, eax
.text:00427E61 7C 08                       jl      short loc_427E6B
.text:00427E63 83 C8 FF                    or      eax, 0FFFFFFFFh
.text:00427E66 E9 B6 00 00 00              jmp     loc_427F21
.text:00427E6B                  ; ---------------------------------------
.text:00427E6B
.text:00427E6B                  loc_427E6B:                            ; C

[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory
[*] Hit my_cmp_handler 427e49: cmp dword [ebp-0x2c],0x0 (0, 0)
[*] Hit my_cmp_handler 427e5f: cmp edx,eax (0, 2)
[*] Done
```

5.6 Συνάρτηση επιστροφής στατιστικής τιμής

Οι συναρτήσεις που χρησιμοποιούνται συχνά για απλούς σκοπούς. Κάποιος μπορεί να έχει μια συνάρτηση υπολογισμού τιμών με βάση τα στοιχεία εισόδου. Αυτές μπορούν εύκολα να συγκεντρωθούν μέσω εξομοίωσης. Η ιδέα είναι να δημιουργηθεί μια λίστα των εισροών και να ανακτάται η τιμή επιστροφής άπαξ και να αποστέλλεται μέσω μιας συνάρτησης. Αυτό μπορεί να γίνει όσες φορές χρειάζεται για να προσδιορίσει το τι θα μπορούσε να είναι το αποτέλεσμα μιας συνάρτησης.

Το απλό αυτό παράδειγμα που θα γράψουμε θέτει τα ορίσματα της συνάρτησης και τα συνδέει έτσι ώστε, όταν η συνάρτηση τελειώνει, να μπορούμε να καταγράψουμε το αποτέλεσμα και αρχίσουμε ξανά.

```
from PyEmu import IDAPyEmu

def reset_stack(emu, value1, value2, value3):
      emu.set_stack_argument(0x8,         value1,         name="arg_0")
      emu.set_stack_argument(0xc,         value2,         name="arg_4")
      emu.set_stack_argument(0x10, value3, name="arg_8")

      return True
```

Αυτή η συνάρτηση θα επαναφέρει τις μεταβλητές στοίβας με σκοπό την επαναφορά των τιμών τους.

```
def my_ret_handler(emu, address):
      global count

      value1 = emu.get_stack_argument("arg_0")
      value2 = emu.get_stack_argument("arg_4")
      value3 = emu.get_stack_argument("arg_8")

      print "[*] Returning %x: %x, %x, %x = %x" % (address, value1,
value2, value3, emu.get_register("EAX"))

      reset_stack(emu, value1 + 1, value2 + 2, value3 + 3)
      emu.set_register("EIP", ScreenEA())

      count += 1


      return True
```

Ο χειριστής μας μνημονικού “ret” θα κληθεί κατά την επιστροφή. Όταν κληθεί, θα πάρουμε την τιμή των ορισμάτων της στοίβας και την τιμή επιστροφής της συνάρτησης για σκοπούς καταγραφής. Αφού καταγράψουμε τις ζητούμενες πληροφορίες, αυξάνουμε τις τιμές, μηδενίζουμε το μετρητή του προγράμματος και επαναλαμβάνουμε τη διαδικασία.

```
# Typical ida loading
<…>
```

```
# This sets our stack values for the function
reset_stack(emu, 0x00000000, 0x00000001, 0x00000002)

# Set up our memory access handler
emu.set_mnemonic_handler("ret", my_ret_handler)

count = 0
while count <= 10:
      if not emu.execute():
          break

print "[*] Done"
```

Μετά από 10 επαναλήψεις της συναρτήσεως έχει ολοκληρωθεί η διαδικασία και ο εξομοιωτής θα βγεί από την συνάρτηση. Και εδώ είναι το αποτέλεσμα.

```
.text:0041DC40 55                       push    ebp
.text:0041DC41 8B EC                    mov     ebp, esp
.text:0041DC43 8B 45 08                 mov     eax, [ebp+arg_0]
.text:0041DC46 23 45 0C                 and     eax, [ebp+arg_4]
.text:0041DC49 8B 4D 08                 mov     ecx, [ebp+arg_0]
.text:0041DC4C 23 4D 10                 and     ecx, [ebp+arg_8]
.text:0041DC4F 0B C1                    or      eax, ecx
.text:0041DC51 8B 55 0C                 mov     edx, [ebp+arg_4]
.text:0041DC54 23 55 10                 and     edx, [ebp+arg_8]
.text:0041DC57 0B C2                    or      eax, edx
.text:0041DC59 5D                       pop     ebp
.text:0041DC5A C3                       retn
.text:0041DC5A          sub_41DC40      endp
.text:0041DC5A
.text:0041DC5A          ; ---------------------------------------------
.text:0041DC5B CC CC CC CC CC           align 10h
.text:0041DC60
.text:0041DC60          ; ¦¦¦¦¦¦¦¦¦¦¦¦¦¦¦ S U B R O U T I N E ¦¦¦¦¦¦¦¦¦
.text:0041DC60
.text:0041DC60          ; Attributes: bp-based frame
```

```
[*] Loading text section bytes into memory
[*] Text section loaded into memory
[*] Loading data section bytes into memory
[*] Data section loaded into memory
[*] Returning 41dc5a: 0, 1, 2 = 0
[*] Returning 41dc5a: 1, 3, 5 = 1
[*] Returning 41dc5a: 2, 5, 8 = 0
[*] Returning 41dc5a: 3, 7, b = 3
[*] Returning 41dc5a: 4, 9, e = c
[*] Returning 41dc5a: 5, b, 11 = 1
[*] Returning 41dc5a: 6, d, 14 = 4
[*] Returning 41dc5a: 7, f, 17 = 7
[*] Returning 41dc5a: 8, 11, 1a = 18
[*] Returning 41dc5a: 9, 13, 1d = 19
[*] Returning 41dc5a: a, 15, 20 = 0
[*] Done
```

Όπως όλα μας τα παραδείγματα και αυτό μπορεί να αποδειχθεί χρήσιμο σε περιπτώσεις που οι συναρτήσεις μπορεί να επιστρέφουν σημαντικές αξίες που είναι άγνωστες. Εμείς στοχεύουμε στη μείωση του επενδυόμενου χρόνου σε κάθε συνάρτηση κατά την αντίστροφη μηχανική.

Έχουμε δει μερικές πραγματικές εφαρμογές και χρήσεις για τον PyEmu. Υπάρχουν

πολλές δυνατότητες κατά την αντίστροφη μηχανική και ελπίζουμε ότι αυτή η εργασία έχει αποδείξει κάποιες βασικές αρχές, ενώ παράλληλα εργαζόμαστε, για να δημιουργήσουμε πιο σύνθετες λύσεις για τις δικές σας ανάγκες

6 Περιορισμοί και μελλοντική εργασία

Προφανώς, υπάρχουν αρκετοί περιορισμοί στην τρέχουσα εργαλειοθήκης της αντίστροφης μηχανικής και του PyEmu. Υπάρχουν ακόμα πολλές χειροκίνητες αλληλεπιδράσεις και ρυθμίσεις, όταν χρησιμοποιείτε τον PyEmu. Η ρύθμιση των τιμών της μνήμης, η ενημέρωση των μεταβλητών της στοίβας και η βασική ανάγκη να έχουμε κάποια κατανόηση του προσομοιωμένου κώδικα αποτελούν μειονεκτήματα κάθε σύγχρονου εξομοιωτή. Δεδομένου ότι το εργαλείο ωριμάζει, ελπίζουμε ότι αυτά τα ζητήματα θα αντιμετωπιστούν. Είτε αυτό γίνεται μέσω προκαταρκτικής ανάλυσης, είτε με την εξαγωγή στατιστικών συμπερασμάτων, είτε μέσω της τεχνητής νοημοσύνης. Ανεξάρτητως αυτών προκειμένου να επιτευχθεί ο στόχος της μείωσης του επενδυόμενου χρόνου μας στην αντίστροφη μηχανική πρέπει να γίνοουν αυτές οι εξελίξεις.

Οι ελλείψεις περιφερειακών συσκευών εξομοίωσης μπορεί επίσης να έχουν αρνητικές συνέπειες στον PyEmu. Συχνά οι χρόνοι βαθιά πολύπλοκων διαδρομών κώδικα μπορεί να κάνουν μια προσπάθεια να αποκτήσει πρόσβαση σε έναν περιφερικό εξομοιωτή. Στην περίπτωση αυτή ο εξομοιωτής θα αναγκαστεί να αγνοήσει κάθε πρόσβαση και να συνεχίσει σαν να μην συνέβη τίποτα. Στο μέλλον οι περιπτώσεις αυτές μπορεί να διορθωθούν, έχοντας πιο ευφυείς απαντήσεις σε μη υποστηριζόμενες ενέργειες, όπως είναι η εξομοίωση μιας συσκευής εισόδου.

Το ενιαίο μεγαλύτερο μειονέκτημα για τον τρέχοντα εξομοιωτή PyEmu είναι η έλλειψη μιας πλήρους σειράς προσομοιωμένων βιβλιοθηκών και κλήσεων του συστήματος. Όλα τα προγράμματα θα εισάγουν πολλές εξωτερικές βιβλιοθήκες για χρήση κατά τη διάρκεια της εκτέλεσης. Για κλήση μιας βιβλιοθήκης, αυτό μπορεί να μην είναι μια μεγάλο πρόβλημα. Σε μελλοντικές εκδόσεις του, ο PyEmu θα φορτώσει τις ζητούμενες βιβλιοθήκες στη μνήμη και παρέχει πρόσβαση στις εξαγωγές της, όπως γίνεται συνήθως κατά την εκτέλεση. Ωστόσο, οι κλήσεις του συστήματος είναι αρκετά δύσκολο να εξομοιωθούν σε αυτό το επίπεδο. Παρά το γεγονός ότι ο PyEmu κάνει μια αξιοπρεπή δουλειά, προσπαθώντας να παράγει έναν κώδικα python χρησιμοποιήσιμο ισοδύναμα με μια υποδοχή, για παράδειγμα, πολλές άλλες ενέργειες μπορεί να αγνοηθούν. Ας ελπίσουμε ότι, μια αξιοπρεπής λύση για αυτό το πρόβλημα θα υλοποιηθεί πολύ σύντομα.

Στο μέλλον, ο PyEmu θα είναι πολύ πιο αυτοματοποιημένος, ή τουλάχιστον να έχει την αυτοματοποίηση να προστίθεται στη βάση για χρήση. Επίσης, οι καλύτερες βιβλιοθήκες και η κλήση συστήματος υποστήριξης θα θέσει τον εξομοιωτή σε ένα νέο επίπεδο. Με αυτό κατά νου, εξακολουθεί να λειτουργεί καλά και είναι μια έγκυρη λύση για αυτούς που καταπιάνονται κυριως με καθήκοντα της αντίστροφης μηχανικής.

7 Συμπέρασμα

Η εξομοίωση έχει παίξει καθοριστικό ρόλο στην προώθηση της επιστήμης των υπολογιστών από τα μέσα της δεκαετίας του 1960. Καθώς προχωρούμε στην προώθηση της αντίστροφης μηχανικής, πιστεύω πως είναι χρήσιμο να επιτραπούν οι εξομοιωτές, για να αποδειχθεί η χρησιμότητά τους στο πεδίο της επιστήμης των υπολογιστών. Με όλο και πιο σύνθετες εφαρμογές, υπό την παρουσία συσκότισης, και χρονικών περιορισμών πρέπει να δουλέψουμε πιο γρήγορα και πιο αποτελεσματικά.

Ο PyEmu σχεδιάστηκε με όλα αυτά κατά νου και το πιο σημαντικό να μπορεί να χρησιμοποιηθεί ευέλικτα με δυνατότητες εύκολης επέκτασης. Ο PyEmu προσπαθεί να λειτουργεί με ευχέρεια και όπως είναι αναμενόμενο, με τρόπο έτσι ώστε να μπορεί να ενσωματωθεί στα συνεχώς αυξανόμενα κουτιά εργαλειών της αντίστροφης μηχανικής. Ας ελπίσουμε ότι θα επιτυγχάνει όλα αυτά για τα οποία είναι σχεδιασμένος και ακόμα περισσότερα.

Παραπομπές

1. Η Ιστορία της Εξομοίωσης
http://www.zophar.net/articles/art_14-2.html

2. bochs: Έργο Ανοικτού Κώδικα του Εξομοιωτή IA-32
http://bochs.sourceforge.net/

3. Ο Απόσυναρμολογητής IDA Pro
http://datarescue.com/idabase/index.htm

4. SABRE Ασφάλεια BinNavi
http://www.sabre-security.com/products/binnavi.htm

5. PaiMei Πλαίσιο Αντίστροφης Μηχανικής
http://paimei.openrce.org/

6. O x86 εξομοιωτής πρόσθετων εφαρμογών για το IDAPro
http://ida-x86emu.sourceforge.net/

7. IDAPython
http://d-dome.net/idapython/

8. PIDA
http://paimei.openrce.org/

9. Pydbg
http://paimei.openrce.org/

10. pefile
http://dkbza.org/pefile.htm

11. pydasm
http://dkbza.org/pydasm.html

12. libdasm
http://www.nologin.org/main.pl?action=codeView&codeId=49&

13. Intel 64 και IA-32 Εγχειρίδιο Προγραμματιστών για το Λογισμικό αρχιτεκτονικής.
http://www.intel.com/design/processor/manuals/253666.pdf