HY255 Εργαστήριο Λογισμικού – L17

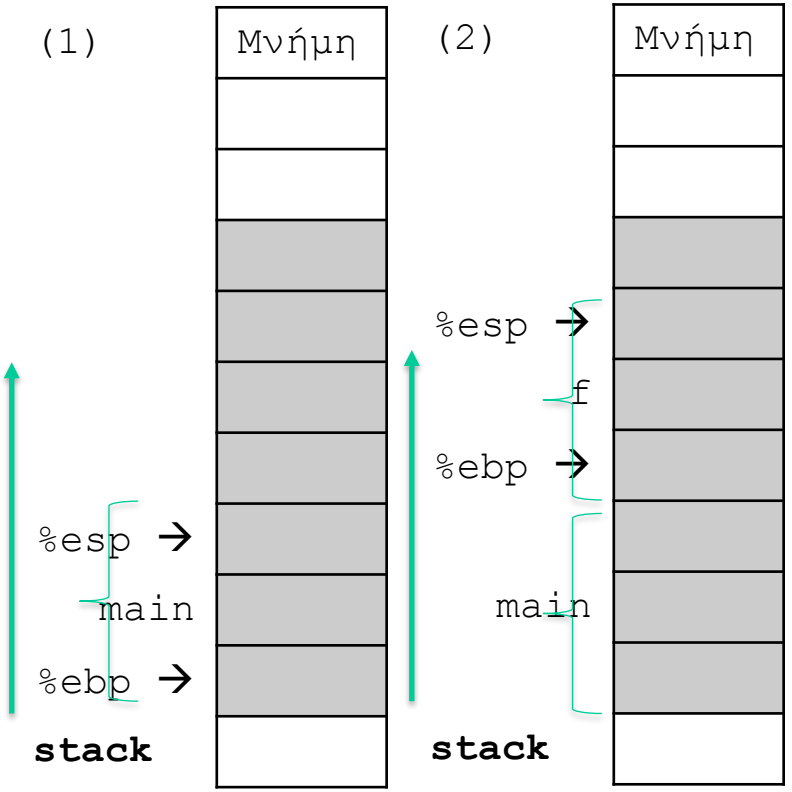
x86 stack

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

Στοίβα

- Τι ακριβώς συμβαίνει στη στοίβα καθώς καλούμε ή επιστρέφουμε από συναρτήσεις;
- Οι καταχωρητές %esp, %ebp χρησιμοποιούνται για την λειτουργία της στοίβας
  - Ο %ebp δείχνει στην βάση (bottom) του stack frame της τρέχουσας συνάρτησης
  - Ο %esp δείχνει στην κορυφή (top) του stack frame (x86: lowest used address)
  - Η τιμή τους προσαρμόζεται αυτόματα από τον κώδικα που παράγει ο compiler
- Η στοίβα μεγαλώνει από μεγάλες προς μικρές διευθύνσεις
  - Είναι σύμβαση αν η στοίβα μεγαλώνει προς μεγαλύτερες ή μικρότερες διευθύνσεις. Για πρακτικούς λόγους στα πιο πολλά συστήματα μεγαλώνει προς μικρές διευθύνσεις
  - Γενικά (αλλά όχι απαραίτητα) ζωγραφίζουμε τη στοίβα με τις μεγάλες διευθύνσεις "κάτω" και τις μικρές "πάνω",
  - Σε κάθε περίπτωση, προσέχουμε προς τα που μεγαλώνει η στοίβα και αν η προσαρμογή των %esp, %ebp χρειάζεται πρόσθεση ή αφαίρεση
- Ο %esp γενικά μπορεί να δείχνει στην χαμηλότερη ελεύθερη ή χρησιμοποιημένη θέση – αυτό είναι σύμβαση του κάθε συστήματος
  - Στον x86 δείχνει στην χαμηλότερη θέση που χρησιμοποιείται από το τρέχον stack frame

```
int f(int x, int y){
(2) ←
return x;
}
int main(void){
        (1) ←
        return f(1,2);
}
```



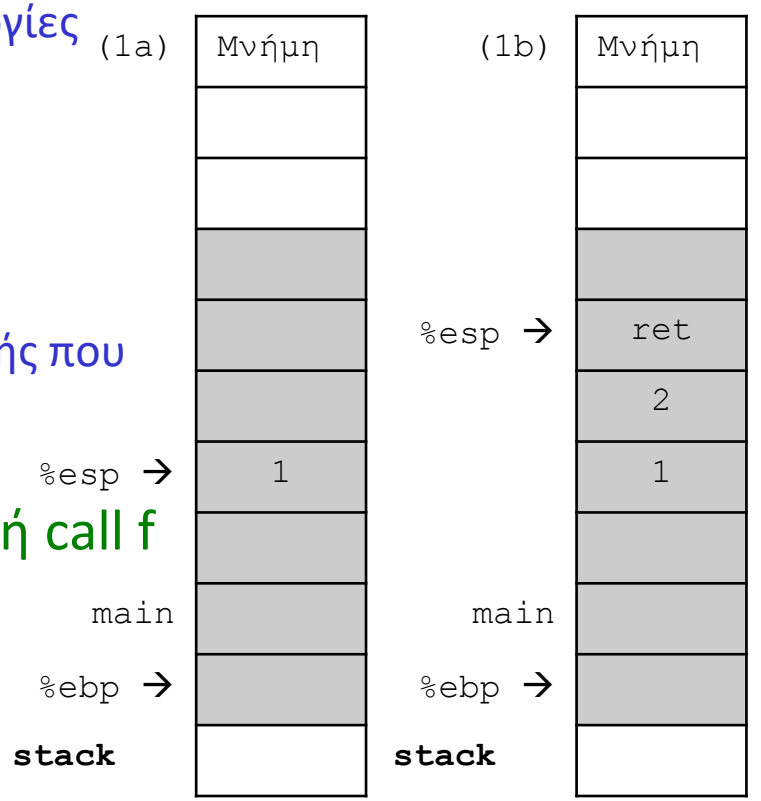
Κύρια στάδια στην λειτουργία της στοίβας

- Η στοίβα περνά από τέσσερα “ενδιαφέροντα” σημεία κατά την εκτέλεση μια συνάρτησης
  - (1) Πριν την κλήση της συνάρτησης, στο σημείο που γίνεται η κλήση της συνάρτησης
  - (2) Στην αρχή της συνάρτησης που κλήθηκε, αμέσως μετά την κλήση
  - (3) Στο τέλος της συνάρτησης που κλήθηκε, αμέσως πριν την επιστροφής της
  - (4) Μετά την επιστροφή της συνάρτησης, στο σημείο του κώδικα που έγινε η κλήση της συνάρτησης

1. Η main καλεί την f

- Η main κάνει τις εξής λειτουργίες
- (a) βάζει τα args της συνάρτησης στη στοίβα
  - Πρέπει πρώτα να πάρουμε μια θέση στη στοίβα
    - subl $4, %esp # αφού προς τα «πάνω» είναι πιο μικρές διευθύνσεις
  - Αντιγράφουμε τις παραμέτρους της f στην στοίβα
    - movl $1, (%esp)
  - Αυτές οι δύο εντολές έχουν μόνο ”άγνωστο” την τιμή ($1)
  - Ο x86 έχει μια εντολή που κάνει αυτές τις δύο λειτουργίες
    - pushl $1
  - Άρα τελικά η main κάνει
    - push $1
    - push $2
- (b) βάζει την return διεύθυνση στη στοίβα
  - Η διεύθυνση επιστροφής είναι η διεύθυνση της εντολής που ακολουθεί την call
- (c) αλλαγή ροής στο κώδικα της f
- Τα βήματα (b),(c) γίνονται όταν καλείται η εντολή call f
- Οπότε συνολικά έχουμε
  - push $1
  - push $2
  - call f

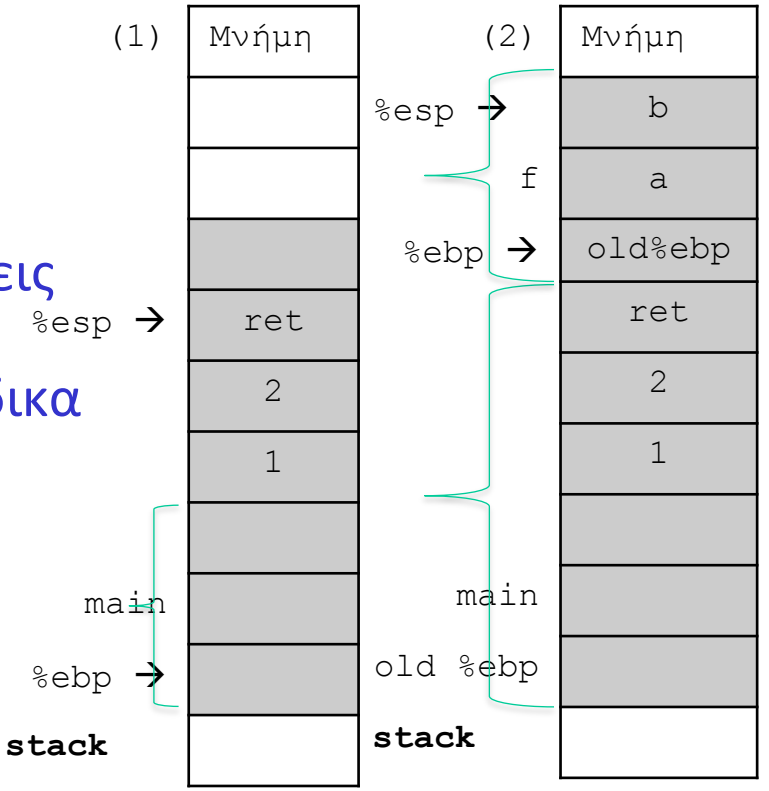
```
int f(int x, int y){
 int a,b;
 return x;
}
int main(void){
        int t;
        t = f(1,2); ← (1)
        return t;
}
```



2. Στην f πριν αρχίσει η εκτέλεση του κώδικα της

- Η f κάνει τις εξής λειτουργίες
- (a) κρατάει την παλιά τιμή του %epb
  - pushl %ebp
- (b) προσαρμόζει τον %ebp στην νέα του τιμή να δείχνει στη βάση του νέου stack frame, άρα
  - movl %esp, %ebp
- (c) δεσμεύει χώρο για όσες τοπικές μεταβλητές χρειάζεται η συνάρτηση
  - subl SIZE, %esp
  - Αλλάζει άραγε το SIZE σε διαφορετικές κλήσεις μια συνάρτησης; ΟΧΙ. Γιατί;
  - Γνωρίζουμε το SIZE όταν παράγουμε τον κώδικα της κάθε συνάρτησης; ΝΑΙ. Γιατί;
- Οπότε συνολικά έχουμε
  - pushl %ebp
  - movl %esp, %ebp
  - subl $SIZE, %esp

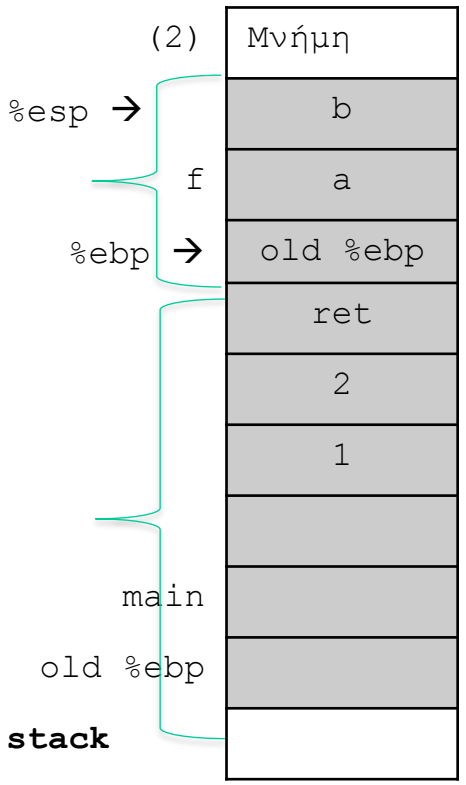
```
int f(int x, int y){
 int a, b;← (2)
 return x;
}
int main(void){
        int t;
        t = f(1,2);
        return t;
}
```



Εκτέλεση του κώδικα της f

- Στο σημείο αυτό εκτελείται ο κώδικας της f
- Όσο εκτελείται, μπορεί να προσπελαύνει τις παραμέτρους και τοπικές μεταβλητές ως εξής
  - Input params σχετικά με τον ebp: %ebp+8, %ebp+12, …
  - Local vars σχετικά με τον esp: %esp, %esp+4, …
- Οι global μεταβλητές έχουν απόλυτες διευθύνσεις μνήμης

```
int f(int x, int y){
int a, b;
…                    ← (2)
return x;
}
int main(void){
        int t;
        t = f(1,2);
        return t;
}
```



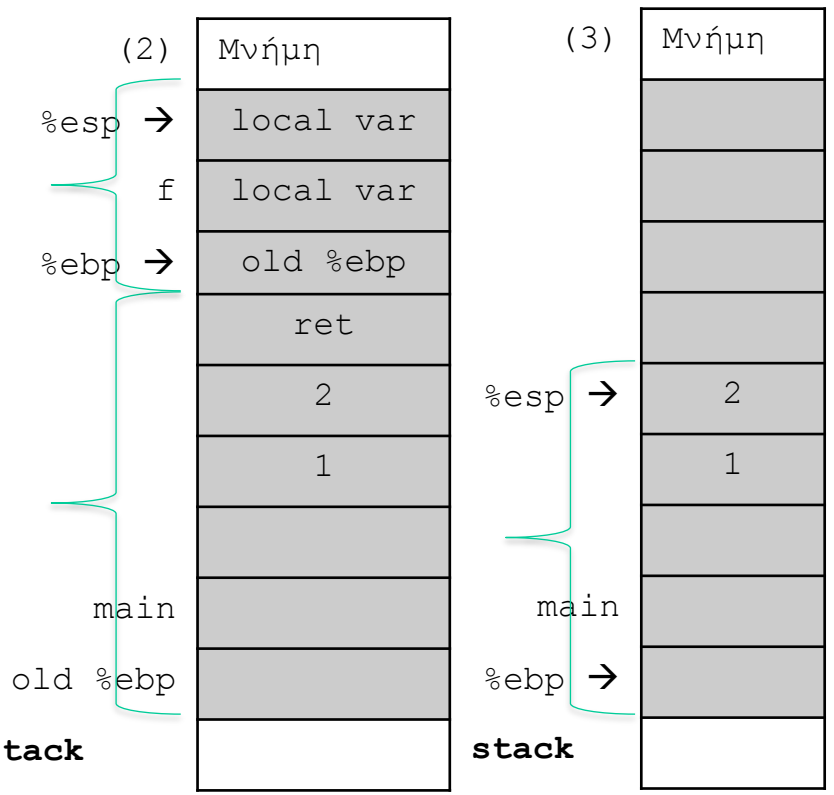
3. Στο τέλος της f πριν επιστρέψει στην main

- Η f κάνει τις εξής λειτουργίες
- (a) ελευθερώνει τον χώρο των τοπικών μεταβλητών στη στοίβα, άρα
  - addl SIZE, %esp
  - Αν παρατηρήσουμε αυτό είναι πάντα το ίδιο με το να δώσουμε στο %esp την τιμή του %ebp, άρα ισοδύναμα μπορούμε να γράψουμε
  - movl %ebp, %esp
  - Που έχει το πλεονέκτημα ότι δεν χρησιμποιεί το $SIZE που αλλάζει από συνάρτηση σε συνάρτηση
- (b) Επαναφέρει την παλιά τιμή του ebp (και ελευθερώνει την αντίστοιχη θέση στη στοίβα)
  - Ο %esp σε αυτό το σημείο δείχνει στην παλιά τιμή του %ebp, όπως έχει αποθηκευτεί στη στοίβα, άρα
  - movl (%esp), %ebp
  - addl $4, %esp
  - Αν παρατηρήσουμε αυτές τις δύο εντολές, ο μόνος «άγνωστος» είναι ο destination καταχωρητής %ebp, που θα αποθηκευτεί ότι βγάζουμε από την στοίβα
  - Ο x86 έχει μια ειδική εντολή για αυτό το σκοπό που κάνει αυτές τις δύο λειτουργίες μαζί, popl destination, άρα μπορούμε να γράψουμε
  - popl %ebp
- (c) Επιστρέφει στην διεύθυνση επιστροφής, που είναι αποθηκευμένη στη στοίβα, και ελευθερώνει την αντίστοιχη θέση της στοίβας
  - Αυτές οι δύο λειτουργίες γίνονται με μια ειδική εντολή ret σε ένα βήμα (δεν μπορούν να γίνουν χωριστά), άρα η συνάρτηση εκτελεί απλά την εντολή ret
  - Αν η συνάρτηση θέλει να επιστρέψει μια τιμή στην συνάρτηση που την κάλεσε, τότε πρέπει να βάλει αυτή την τιμή στον καταχωρητή %eax
- Οπότε συνολικά έχουμε

  (if returning a value) movl ..., %eax
  movl %ebp, %esp
  popl %ebp
  ret
  - Οι δύο εντολές
    - movl %ebp, %esp
    - popl %ebp
  - είναι πάντοτε οι ίδιες, οπότε ο x86 έχει μια νέα εντολή που κάνει αυτές τις δύο λειτουργίες την leave, άρα μπορούμε να γράψουμε και

  (if returning a value) movl ..., %eax
  leave
  ret

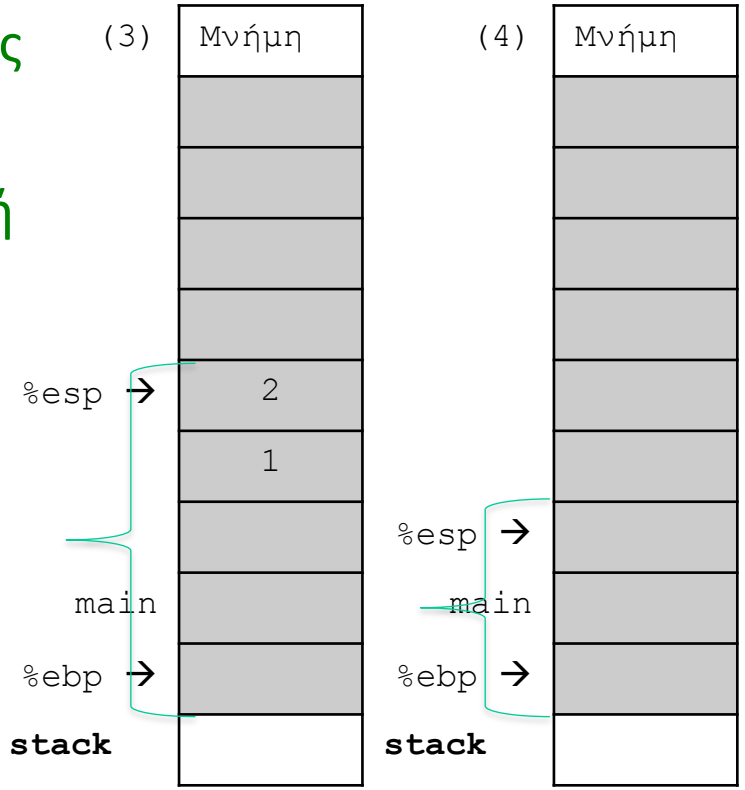
```
int f(int x, int y){
 int a,b;
 return x; ← (3)
}
int main(void){
        int t;
        t = f(1,2);
        return t;
}
```



4. Στην main μετά την επιστροφή της f

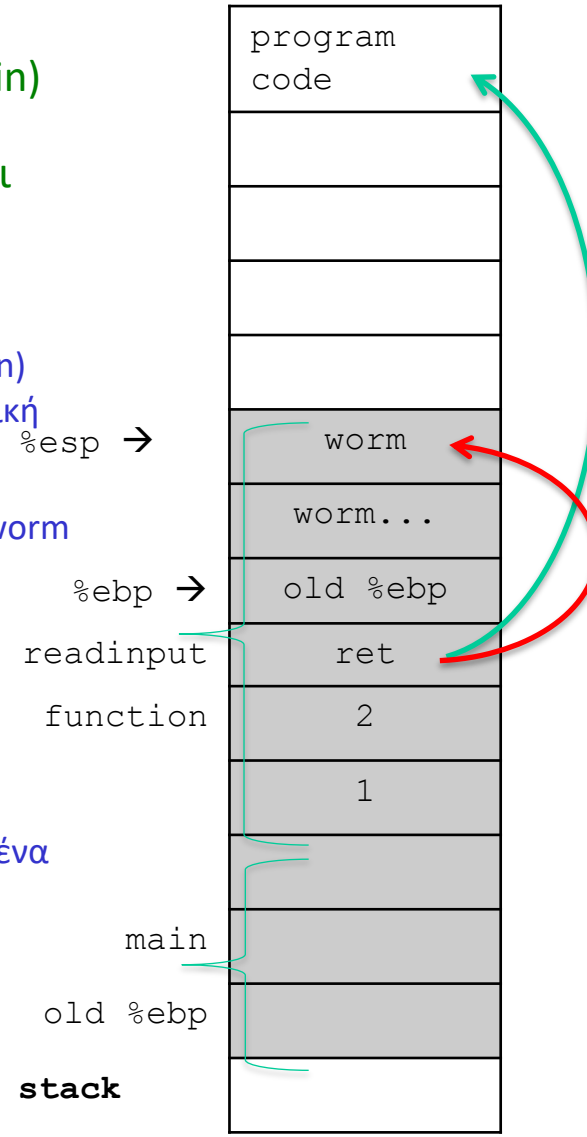
- Η main κάνει τις εξής λειτουργίες
- (a) ελευθερώνει τον χώρο των παραμέτρων στη στοίβα
  - Αυτό μπορεί να γίνει με το να προσθέσει το μέγεθος των παραμέτρων στον %esp
  - addl $PSIZE, %espf
  - Ή με το να κάνει pop για κάθε παράμετρο σε κάποιο καταχωρητή που δεν χρησιμποιείται, π.χ.
  - popl %ecx
- Σε αυτό το σημείο η στοίβα επιστρέφει ακριβώς στο σημείο που ήταν πριν κληθεί η f και πλέον δεν υπάρχει κανένα «σημάδι» στη στοίβα ότι εκτελέστηκε η f, εκτός από την ενδεχόμενη τιμή επιστροφής στον %eax και όποιο άλλο side-effect μπορεί να έχει η f
- Η όποια επόμενη κλήση συνάρτησης από την main θα γίνει με τον ίδιο τρόπο
- Αν η f καλούσε άλλες συναρτήσεις, οι κλήσεις αυτές θα γινόταν με τον ίδιο ακριβώς τρόπο

```
int f(int x, int y){
 int a,b;
 return x;
}
int main(void){
        int t;
        t = f(1,2);
                        ←
        (3)
        return t;
}
```



Άσκηση 6: Buffer Overrun Attacks (Morris Worm)

- Το πρώτο internet worm, in 1988, by Robert T. Morris
- Ένα μικρό πρόγραμμα που διέδιδε/μετέδιδε τον εαυτό του και εκτελούνταν αυτόματα
- Έβρισκε το επόμενο σύστημα από γειτονικά συστήματα στο δίκτυο
- Δεν έκανε κάτι «καταστροφικό», απλά ένα loop που κατανάλωνε cpu (spin) – και αυτό ήταν πρόβλημα βέβαια
- Το αποτέλεσμα ήταν ότι σε λίγα λεπτά όλα τα συστήματα έκαναν spin και δεν μπορούσαν να κάνουν άλλη χρήσιμη δουλειά
- Πως διέδιδε τον εαυτό του από σύστημα σε σύστημα και κατάφερε να εκτελείται;
  - Χρησιμοποιούσε ένα πρόγραμμα που υπήρχε σε όλα τα συστήματα (finger daemon)
  - Του έστελνε τον εκτελέσιμο κώδικα του worm και το έβαζε στη στοίβα, σε μια τοπική μεταβλητή
  - Μετά, κατάφερνε να αλλάξει την διεύθυνση επιστροφή στη στοίβα και αντί να συνεχίσει να εκτελείται το αρχικό πρόγραμμα, εκτελούνταν πλέον ο κώδικας του worm
  - Και επαναλάμβανε το ίδιο με ένα άλλο γειτονικό σύστημα
- Τα δύο βασικά σημεία είναι
  - (1) Να ετοιμάσει κανείς και να στείλει έτοιμο εκτελέσιμο κώδικα στην μνήμη ενός άλλου προγράμματος (που διαβάζει κάποιο input)
  - (2) Να "αναγκάσει" το αρχικό πρόγραμμα να αρχίσει να εκτελεί τον νέο κώδικα
- Trivia
  - Ο finger daemon έτρεχε τότε σε όλα τα συστήματα (Linux), έπερνε ένα μήνυμα με ένα login name και απαντούσε αν αυτός ο χρήστης είναι logged in εκείνη τη στιγμή
  - Δεν ήταν προσεκτικά γραμμένο κάποιο κομμάτι του κώδικά του – δεν έλεγχε το μέγεθος του login που διάβαζε από το δίκτυο
  - Αυτό ήταν γνωστό με κάποιο τρόπο
  - Το worm ήταν η πρώτη προσπάθεια που έδειξε τι είναι εφικτό να γίνει



Reading

- Chapter 4 and Appendix B of Programming from the Ground Up, Jonathan Bartlett 2004