ΕΚΠΑ: Πληροφορικής & Τηλεπικοινωνιών
Ασφάλεια & Προστασία Πληροφοριακών Συστημάτων
ΥΣ13 ΕΑΡΙΝΟ 2017

Λαζαρίδης Νίκος, Μ1485

Project #2 : Buffer Overflows

Σε αυτή την άσκηση εκμεταλλευτήκαμε τρωτά (vulnerable) σημεία στον κώδικα του προγράμματος securelog, ώστε να αποκτήσουμε πρόσβαση διαχειριστή (super user) στον διακομιστή του προγράμματος. Η ευπάθεια που μας επέτρεψε να παρακάμψουμε την ασφάλεια του συστήματος λέγεται Υπερχείληση Πίνακα (Buffer Overflow).

Το πρόγραμμα securelog.c κάνει χρήση ανασφαλών συναρτήσεων, συγεκριμένα των sprintf και strcpy. Η συνάρτηση strcpy (ίσως το δημοφιλέστρερο παράδειγμα τέτοιας συνάρτηση), αλλά και η sprintf, αποθηκεύει τα δεδομένα στο δεξί της όρισμα στη θέση μνήμης που υποδεικνύεται από τον δείκτη στο αριστερό όρισμα. Ο δείκτης αυτός, buf στο πρόγραμμα μας, περιέχει τη συνάρτηση έναρξης του πίνακα - buffer μας, στον οποίο αποθηκεύονται όλα τα δεδομένα που εισάγουμε εμείς καθώς και άλλα και έπειτα εκτυπώνονται στο αρχείο secure.log.

Το αρχείο secure.log έχει δικαίωμα να το διαβάσει μόνο ο ιδιοκτήτης του t_niklazar, αλλά έχει τεθεί το setuid bit του. Όταν κάποιο εκτελέσιμο διαθέτει συγκεκριμένο bit, αυτό το καθιστά ιδιαίτερα ευάλωτο καθώς γίνεται πρωταρχικός στόχος επιθέσεων buffer overflow. Ένα τέτοιο πρόγραμμα εκτελείται με δικαιώματα, όχι του χρήστη που το εκτελεί - real user id (ruid), αλλά του ιδιοκτήτη. Ο ιδιοκτήτης έχει συνήθως περισσότερα δικαιώματα από αυτόν που το εκτελεί και μάλιστα πολλές φορές είναι και ο διαχειριστής του συστήματος, επομένως το ruid του χρήστη μεταπίπτει στο effective user id (euid).

Γνωρίζουμε ότι κάθε πρόγραμμα 2 που εκκινείται από ένα άλλο πρόγραμμα 1, τότε «τρέχει» με τα δικαιώματα του 1 (του γονέα). Το σκεπτικό μας είναι λοιπόν ότι εάν καταφέρουμε και αποκτήσουμε πρόσβαση στο πρόγραμμα securelog με το setuid bit τότε μπορούμε να εκτελέσουμε ένα shell /bin/sh ή /bin/bash του συστήματος με αυξημένα δικαιώματα. Έπειτα θα χρησιμοποιήσουμε αυτό το shell για να ξεκλειδώσουμε δικαιώματα πρόσβασης, γραψίματος και εκτέλεσης του αρχείο secure.log με αποτέλεσμα να δούμε τα περιεχόμενα του. Αυτό το κάναμε με την εντολή `chmod`.

Αρχικώς, έπρεπε να κατανοήσω ακριβώς το πρόγραμμα securelog.c. Διαπίστωσα ότι έπρεπε να μεταγλωττίσω, να εκτελέσω και να εκσφαλματώσω (debug) το πρόγραμμα, έχοντας πλήρη ελευθερία πειραματισμού ώστε να ανακαλύψω τα απόκρυφα τους και να κατανοήσω τη συμπεριφόρα του. Χρησιμοποιώντας την εντολή:

```
rsync -r -a -v -e "ssh -l niklazar" --delete /local/dir
bo.csec.gr:/remote/dir
```

έκανα συγχρόνισμο του απομακρυσμένου φακέλου με ένα κατάλογο στο σύστημα μου - Ubuntu 16.04 LTS x86_64. Θα βρείτε όλα τα αρχεία με τα οποία ασχολήθηκα τοπικά στον φάκελο "`local_files`" που σας παραδίδω. Επίσης κατέγραψα όλες τις κινήσεις μου, με την εντολή `script local_securelog_log.sh` , ώστε αν θέλετε να τις επαληθεύσετε. Δηλαδή στο αρχείο local_securelog_log.sh βρίσκεται ένα «ημερολόγιο» ολόκληρου του terminal session που διεξάχθηκε στον Η/Υ μου.

Παρομοίως στον φάκελο "`remote_files`" θα βρείτε όλα τα αρχεία στον server σας, τα οποία αποθήκευσα τη στιγμή που απέκτησα πρόσβαση στο σύστημα. Τα παραδίδω προς διευκόλυνση. Το

αρχείο κειμένου cmdHistory.txt έχει καταγράψει ένα αντίγραφο / transcript όλων των κινήσεων μου στον server. Αυτό το πέτυχα με την εντολή:

```
rsync -r -a -v -e "ssh -l niklazar" --delete
bo.csec.gr:/remote/dir /local/dir
```

η οποία συγχρονίζει το τοπικό directory με τον απομακρυσμένο.

Παρακάτω περιγράφω αναλυτικά πως υλοποίησα την επίθεση buffer overflow. Ο σκοπός μιας τέτοιας επίθεσης είναι να αντικαταστήσω τη διεύθυνση επιστροφής της τρωτής συνάρτησης με μία της επιλογής μου, στην οποία έχω τοποθετήσει ASCII κώδικα - shellcode, ο οποίος όταν εκτελεστεί θα μεταβάλλει την κανονική ροή του προγράμματος. Τη στιγμή όπου ένα byte αντιγράφεται στην διεύθυνση επιστροφής, στην οποία έχει αντιγράψει ο καταχωρητής RIP τα περιεχόμενα του - instruction pointer, συμβαίνει buffer overflow. Ο IP θα μεταπηδήσει στην νέα διεύθυνση μνήμης από την οποία θα συνεχίσει την εκτέλεση.

Για να επιτύχει η επίθεση συνήθως χρειάζεται να υπάρχει ένας πίνακας, ένας buffer στο πηγαίο πρόγραμμα μας. Ο buffer buff λαμβάνει κώδικα από το χρήστη και η συνάρτηση strcpy δεν ελέγχει για το μέγεθος του και τα περιεχόμενα του. Επιπλέον το πρόγραμμα μας έχει μεταγλωττιστεί χωρίς προστασία καναρινιών (canaries) και με τη μνήμη stack εκτελέσιμη, πράγματα τα οποία είναι σχεδόν αδύνατο να γίνονταν σε ένα πραγματικό πρόγραμμα (θα ήταν μεγάλη αφέλεια / ρίσκο). Χωρίς αυτές τις άμυνες το έργο μας γίνεται αρκετά πιο εύκολο.

Στην παρακάτω εικόνα απεικονίζω έναν αναλυτικό χάρτη της κύριας μνήμης με έμφαση στη μνήμη stack που μας αφορά. Δε θα εξηγήσω εδώ αναλυτικά τα χαρακατηριστικά της, τα θεωρώ δεδομένα. Αναφέρω μονάχα ότι η μνήμη stack μεγαλώνει προς μικρότερες διευθύνσεις μνήμης και όταν καλείται μια συνάρτηση στη στοίβα αποθηκεύονται τα παρακάτω με αύξουσα χρονική σειρά:

1. Ορίσματα,
2. Διεύθυνση επιστροφής,
3. Διεύθυνση βάσης πλαισίου (base pointer address) (RBP stored here),
4. Τοπικές μεταβλητές

Ο καταχωρητής της στοίβας RSP (Stack Pointer) δείχνει πάντα στη τελευταία λέξη που έχει αποθηκευτεί, επομένως πάντα έχει τιμή μικρότερη από τον RBP.

Σε υψηλό επίπεδο υπάρχουν 3 βήματα προσέγγισης μιας επίθεσης Buffer Overflow :

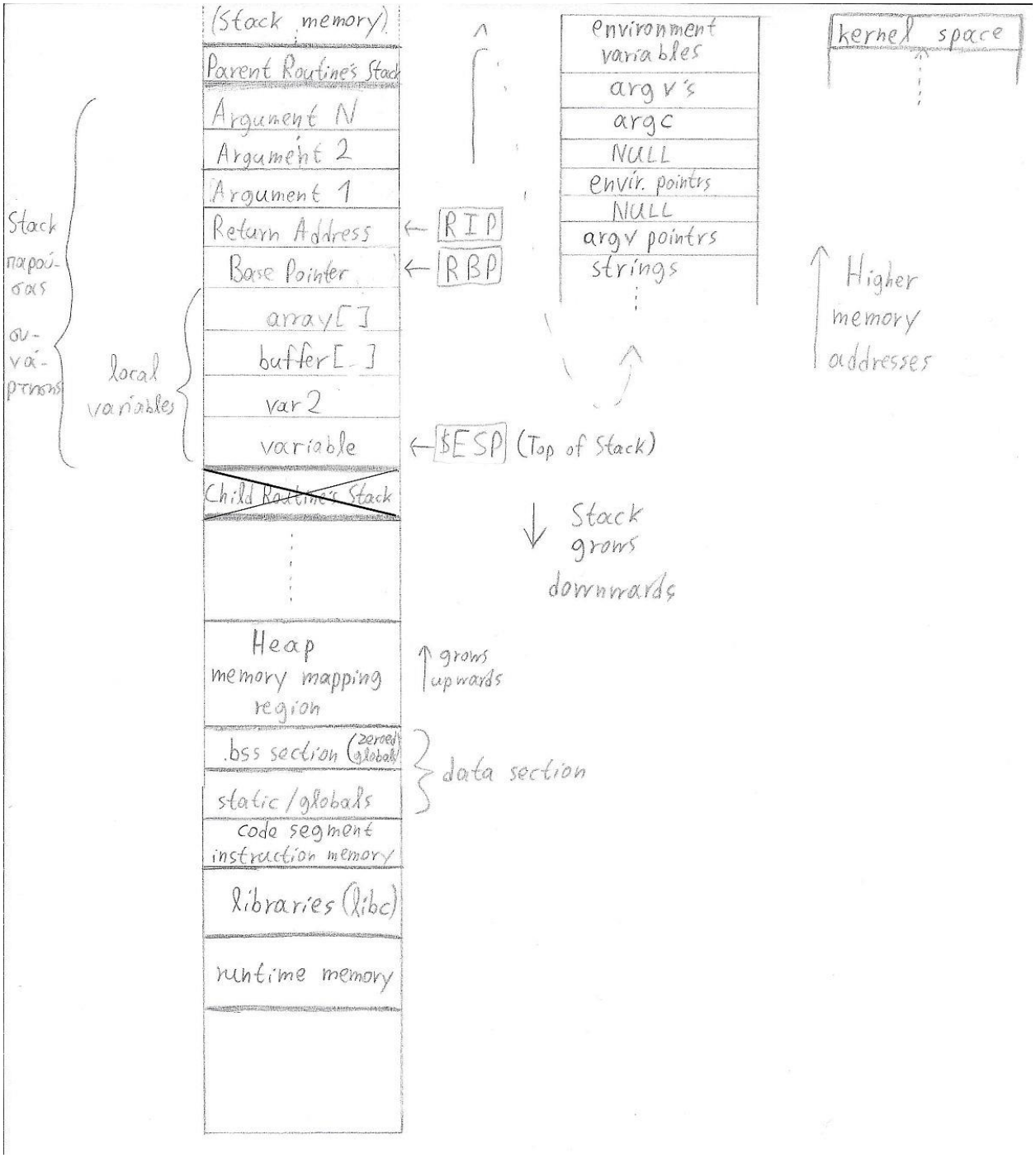
1. Το shellcode που θα παρεμβληθεί στο πρόγραμμα (injected) δημιουργείται,
2. Ερευνείται (με debuggers κυρίως - gdb) μια κατάλληλη τοποθεσία / θέση μνήμης για να εισαχθεί ο κώδικας,
3. Το πρόγραμμα μεταφέρει λειτουργία σε κάποιο άλλο όπως κωδικοποιείται από το shellcode.

Θεωρώ ότι το πιο επίπονο κομμάτι είναι να βρεθεί η σωστή διεύθυνση έναρξης των δεδομένων μου. Αυτή είναι η διεύθυνση του buffer. Χρησιμοποίησα κυρίως το πρόγραμμα gdb (GNU debugger) για να ανακαλύψω αυτή τη διεύθυνση και αναφέρω αμέσως μετά πως μηχανεύτηκα τα κόλπα. Αρχικά όμως παραθέτω εδώ καποιες εντολές οι οποίες με βοήθησαν για να φέρω εις πέρας με επιτυχία το έργο (reverse engineering, scripting commands & interpreters list) :

- strace ./[prog] : Traces system calls and signals
- ltrace ./[prog] : Intercepts and records the dynamic libraries called by the executed process
- readelf : Displays information about ELF files
- gdb : GNU Debugger allows you to see what is going on inside another program
- objdump : Displays information from object files. Use it as such
- objdump -d -M [architecture eg: intel] [programName]
- strings : Prints the strings of printable characters in files
- strip : Discards symbols from object files
- python : Scripting and OOP made easier

- env : run executable with no environment variables [env -i ./myProgram ..] or (gdb) unset env
- ldd : prints dynamic / shared libraries that are dependencies of the executable

Ελπίζω να μην υπάρχει πρόβλημα που αναφέρω μια σύντομη περιγραφή της κάθε εντολής στην αγγλική γλώσσα. (Έχουν γίνει copy-paste από προσωπικά μου αρχεία.)



- Με το εργαλείο objdump βλέπουμε τον κώδικα σε συμβολική γλώσσα και εστιάζουμε στη συνάρτηση (log_message) που περιέχει την ευπαθής κλήση στην strcpy. Λίγο πριν από αυτή την κλήση πρέπει να βρίσκεται μια εντολή όπως `lea rax,[rbp-0x1010]` η οποία κατανέμει επαρκής χώρο στη στοίβα για την αποθήκευση του buffer μας. Παρατηρούμε ότι συντηρούμε χώρο 0x1010 = 4,112 Bytes τη στιγμή που ο πίνακας buff έχει μέγεθος 4096.

Διαπιστώνουμε έπειτα ότι η διεύθυνση αυτή είναι η διεύθυνση του ESP (εάν δεν εισαχθούν και άλλοι μεγάλοι buffers στη στοίβα),

- Επιλέγουμε intel flavor για το disassembly, `set disassembly-flavor intel`
- Παύση εκτέλεσης (breakpoints) πριν από την ευπαθής συνάρτηση και επιθεώρηση της στοίβας. Κοιτάμε στη διεύθυνση του $esp και εκτυπώνουμε κάθε επόμενη διεύθυνση για να δούμε που βρίσκονται τα δεδομένα του buffer (με ENTER). Ψάχνουμε τη διεύθυνση έναρξης του shellcode μας,
- Αν είναι δυνατό εποπτεύουμε όλη τη στοίβα στα γρήγορα μέχρι ανερχόμενοι να φτάσουμε στον $rbp. Εκεί σταματάμε. Η διαφορά $rbp - $rsp είναι ο χώρος της stack στον οποίο μπορούμε να γράψουμε οτιδήποτα θέλουμε πριν φτάσουμε στη διεύθυνση επιστροφής (σε προηγούμενο frame), η οποία βρίσκεται αμέσως μετά από τον RBP. Ο καταχωρητής RIP είναι απεικονισμένος σε αυτή τη διεύθυνση.
- Με trial and error (και διαίσθηση) ψάχνουμε το ελάχιστο μέγεθος εισόδου με το οποίο θα τροφοδοτήσουμε το πρόγραμμα προτού συμβεί segmentation fault (πχ βάζουμε ως είσοδο πολλά ‘A’). Αυτό επιτυγχάνεται με την βοήθεια μιας scripting γλώσσας όπως η python, ή perl κλπ. Όταν βρούμε το μέγεθος στο οποίο δεν έχουμε segfault, τότε στην αμέσως μεγαλύτερη διεύθυνση (+8Bytes pointer size, εφόσον είμαστε σε 64bit αρχιτεκτονική) βρίσκεται η διεύθυνση επιστροφής. [Αυτό το σημείο είναι το πιο δύσκολο να ευρεθεί σωστά] Συμβαίνει segfault επειδή ο RIP μεταβαίνει σε περιοχή μνήμης στην οποία δεν έχει εξουσιοδότηση (πχ. στη θέση μνήμης 0x4141414141414141 - (char) A = 41h).
- Εφόσον βρούμε τη διεύθυνση επιστροφής αρκεί να την αντικαταστήσουμε με τη διεύθυνση κάπου μέσα στον buffer μας. Ανέφερα στρατηγικά «κάπου μέσα» δηλαδή περίπου κοντά αλλά πριν από τη διεύθυνση έναρξης του shellcode μας. Πριν από το shellcode τοποθετούμε εντολές no - operation (που είναι παρούσες στις περισσότερες αρχιτεκτονικές). Έτσι λοιπόν η CPU θα εκτελέσει αυτές τις μη-εντολές μέχρι να γλιστρήσει (nop sled / slide - εμπειρικός όρος που χρησιμοποιείται σε αυτό το context) στο shellcode το οποίο και τελικά θα εκτελέσει. Όταν εκτελεστεί με επιτυχία το shellcode, τότε καταφέραμε να ολοκληρώσουμε την επίθεση buffer overflow. Για «γέμισμα» των θέσεων από το πέρας του shellcode μέχρι τη διεύθυνση επιστροφής μπορούμε να τοποθετήσουμε τυχαίους χαρακτήρες, έστω πολλούς χαρακτήρες A. Τελικά η είσοδος θα είναι της μορφής:
  [NOP-SLED] + [SHELLCODE] + [Array of ‘A’s] + [RETURN ADDRESS]
  Ιδιαίτερα προσοχή χρειάζεται εδώ για να υπολογίσουμε σωστά το πλήθος χαρακτήρων που εισάγουμε να είναι ίσο με το ελάχιστο που χρειάζεται ώστε να παραχθεί segmentation fault (το οποίο έχει ήδη ανευρεθεί). Συγκεκριμένα έχουμε
    1. 46 Bytes προοιμίου προγράμματος - DATE & TIME - υποχρεωτικό
    2. 1000 Bytes nops (μπορούμε να βάλουμε περισσότερα εδώ και να προσαρμόσουμε αναλόγως το μέγεθος των άλλων τμημάτων)
    3. 48 Bytes shellcode
    4. 4112 - (46 + 1000 + 48) = 3018Bytes γεμίσματος με τυχαίο υλικό (A’s εδώ)
    5. + 6 bytes για τους 6 δεκαεξαδικούς χαρακτήρες της διεύθυνσης επιστροφής. Επιλέγουμε 6bytes και όχι 8, διότι στην αρχιτεκτονική x86_64 τα άνω 16 bits δε χρησιμοποιούνται (για x86 θα χρησιμοποιούσαμε +4 bytes για τη διεύθυνση επιστροφής)

Αναφέρω ότι το shellcode που μας δώσατε μου δημιουργούσε προβλήματα και γι’αυτό το αμέλησα. Χρησιμοποίησα ένα άλλο παρόμοιο, το οποίο διατίθεται στην ιστοσελίδα, ονόματι Linux/x86-64 - setuid(0) + execve(/bin/sh) 49 bytes by evil.xi4oyu (παρατίθεται στο αρχείο `x86_64_shellcode_evil.xi4oyu.txt`) το οποίο

1. Αντικαθιστά το real userid με το root id (0) (με αυτό το βήμα καλύπτω αποτελεσματικά τα ίχνη μου) και
2. Ανοίγει το shell /bin/sh με δικαιώματα φυσικά διαχειριστή εξαιτίας του setuid bit (που διαθέτει το πρόγραμμα του γονέα securelog, όπως προανέφερα).

Η διαδικασία παραγωγής αυτού του ASCII κώδικα είναι η εξής (σε Linux Debian τύπου σύστημα):

1. Έστω ότι το αρχείο mal.c περιέχει τον αρχικό C κώδικα που εκτελεί τα παραπάνω. Τότε εκτελούμε την παρακάτω εντολή για να παράγουμε τον assembly κώδικα στο αρχείο mal.s :
   `gcc mal.c -S mal.s`
2. Εκτελούμε τη παρακάτω εντολή για να παράγουμε τον αντικείμενο κώδικα (object code) στο αρχείο (mal.o) χρησιμοποιώντας τον Netwide Assembler - NASM:
   `nasm -f elf64 mal.asm`
3. Χρησιμοποιούμε το αρχείο objdump για αρχιτεκτονική intel x86_64 και elf64 εκτελέσιμο για να παράγουμε τον ASCII κώδικα, τον οποίο εξάγουμε στο αρχείο mal.ascii
   `objdump -M intel -d mal.o > mal.ascii`
4. Εξάγουμε τα bytes από το αρχείο mal.ascii και τα γράφουμε ένα ένα σε escaped μορφοποίηση, δηλαδή `\xfa\x5b\x48\x71` κ.ο.κ. Αυτή η αλληλουχία σε κώδικα ASCII είναι η τελική μορφή του shellcode μας.

Πρίν την εκτέλεση του αρχείου βεβαιωνόμαστε ότι έχουν απενεργοποιηθεί οι μηχανισμοί ASLR / DEP, οι οποίοι μεταβάλλουν συνεχώς τις διευθύνσεις έναρξης κρίσιμων memory segments για να αποτρέψουν τέτοιες ακριβώς επιθέσεις. Σε διανομές linux Debian αρκεί να εκτελέσουμε την ακόλουθη εντολή:

```
sudo echo 0 > /proc/sys/kernel/randomize_va_space
```

Επισημαίνω ότι υπάρχει μια μικρή διαφορά στις διευθύνσεις έναρξης των περιοχών μνήμης του εκτελέσιμου όταν αυτό τρέχει σε περιβάλλον GDB και χωρίς. Αυτό διότι ο GDB αποσπά κάποια environment variables κτλ. πριν την εκτέλεση. Επομένως με trial & error δοκιμάζουμε διάφορα inputs όσον αφορά τη διεύθυνση επιστροφής και βρίσκουμε μια κατάλληλη τέτοια διεύθυνση που να πέφτει πάνω σε nops του buffer μας. Αυτός είναι και ο λόγος που χρειαζόμαστε όσο το δυνατόν μεγαλύτερο nop-slide (και μεγαλύτερο buffer) για να έχουμε περισσότερες πιθανότητες να στοχεύσουμε στις nop εντολές. Διαπιστώσα ότι αυτή η διαφοροποίηση έχει μέγεθος περίπου 1104Bytes στο παρών πρόγραμμα. Συγκεκριμένα στο εκτελέσιμο που αντέγραψα στον υπολογιστή μου και είχα την ευχέρεια να πειραματιστώ χρησιμοποίησα ως return address την 0x7FFFFFFFC05C και χωρίς gdb τη διεύθυνση 0x7FFFFFFFFC4AC (C4AC - C05C = 1,104Bytes). Η τελική είσοδος στο πρόγραμμα (βρίσκονται και στο αρχείο `shellcode.txt` είναι η παρακάτω) :

Χωρίς GDB :

```
./securelog $(python -c 'print"1990-12-30T23:45:53_" + "\x90"*1000
+
"\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\
x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x5
7\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05" +
"A"*3026 + "\x5c\xc0\xff\xff\xff\x7f"')
```

Με GDB :

```
run $(python -c 'print"1990-12-30T23:45:53_" + "\x90"*1000 +
"\x48\x31\xff\xb0\x69\x0f\x05\x48\x31\xd2\x48\xbb\xff\x2f\x62\x69\
x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89\xe7\x48\x31\xc0\x50\x5
7\x48\x89\xe6\xb0\x3b\x0f\x05\x6a\x01\x5f\x6a\x3c\x58\x0f\x05" +
"A"*3026 + "\xac\xc4\xff\xff\xff\x7f"')
```

Μόνο όταν είχα επιτυχία με τα local αρχεία έτρεξα το εκτελέσιμο στον server (στο οποίο οι διευθύνσεις είναι ελαφρώς μετατοπισμένες) και, όπως θα διαπιστώσετε από το αρχείο secure.log)

με πολύ λίγες προσπάθειες trial & error βρήκα τη σωστή είσοδο (σωστή διεύθυνση επιστροφής) που έπρεπε να διοχετεύσω στο πρόγραμμα. Αφού άνοιξα root shell έτρεξα την εντολή :
`whoami`
και διαπίστωσα από την έξοδο του ότι είμαι πλέον συνδεδεμένος ως root. Έπειτα έτρεξα
`chmod 770 securelog.c`
για να αποκτήσω (niklazar) δικαιώματα στο αρχείο και να το διαβάσω.

Το αρχείο secure.log αρχικώς περιείχε τη παρακάτω γραμμή:
2017-05-23T16:06:17_00000_2017-05-19T15:56:52_One good turn deserves another. - - Gaius Petronius
και με κάθε εκτέλεση του αρχείου securelog αποθηκευόταν μια επιπλεόν γραμμή με την είσοδο που δέχεται το πρόγραμμα. Η συνάρτηση append_to_file() προσαρτά τα δεδομένα στο αρχείο.

ΥΣ. Ο mail server δεν με άφησε να παραδώσω τα shellcodes και τα αρχεία. Τα αναγνώριζε ως malware και γι'αυτό έπρεπε να τα τοποθετήσω σε συμπιεσμένα αρχεία προστατευμένα με κωδικό. Ο κωδικός είναι ο ΑΜ μου **M1485**.