

REPORT

Dry:

- 1) We need to change the number which written in the global variable to be minimal as possible. We need to jmp at the start of the poll function to our hook function because there are many ret points along the poll function. If there are hot-patching with 5 nops we will hook and then go back to the rest of poll function. If there are no nops and we need to use the write and jump hook-method, we need to remember the instruction we overwritten and to write them back at the start of our hook. If we don't plan to remove the hook we can use a Boolean variable which tell us if we already changed the global value.
- 2) Because the "mine" function has many entry points at different addresses at the body of the function we need to hook it at the end of the function - we will overwrite a jmp to the hook function which will reverse the input which located in the stack ebp+4 then we will replace it with the reversed one. Now we will change the ret address located at ebp to :
(ret_address - 5) .
The minus 5 is because we want to preform the function again - "call" opcode is 5 bytes.
This way the value that returns is for input \bar{X} .
For the function to operate normally and won't repeat it self over and over we need a global flag variable to indicate if we already preformed the hook and we will check the variable's value at the start of the hook function. If the value is 1 its indicates that we want the function to end and not reverse again - we will change the global flag variable to be 0 and ret from mine function. Else it's means that this is the first time the hook is on.
- 3) We will jmp to hook function before using the handle to the socket - it means in the beginning of sendf function.
We will use the encryption function in the hook function on each argument on the stack - we can know how many arguments there are by looking at the format string which also in the stack. After traverse on all arguments we will use the socket's handle to connect and send the encrypted message.
If there were 5 nops we don't need to restore any instructions after doing the hook, else , we will need to restore the instruction which we overwrite.

- 4) We will preform the jmp at the start of the connect function .
In the hook function we will create a new thread (CreateThread) which will run on parse function located at : [ret_address] of connect function, and we will run the thread.
We need to change the ret address from connect function so we will add 5 to the value stored in the ret address at the stack - that way we will return to 1 instruction after the call to parse function.
If there were nops we won't restore instructions ,else we do at the start of the hook function and then create the thread which will run the parse function.
- 5) We will set a hook at the beginning of calc using DLL injection such that it jumps to the following function: first it restores the beginning of calc (we can set the page to ReadWriteExecute during DLL injection) by restoring the overridden bytes (constant bytes can be saved in the hooks body). Second, it overrides the return address of calc to point within our function and saves the old return address, then it'll jump back to calc, calc will perform it's normal code including checking it's binary code to assert it hasn't been tampered with and since the only change is on the stack it'll conclude that it's code isn't changed, next calc will return to a point within our hook since it's return address was changed. Now we will log calc return value (present in eax), overrides the beginning of calc with a jump to our hook, and finally return to the saved original return address. This way we successfully logged calc's return value and restored the hooks initial state so it'll be able to log the return value from the next call as well.
- 6) We will set a hook at the beginning of solve such that it jumps to the following function: first it performs commands overridden by the jump. Second, it overrides the return address of solve to point within our function and saves the old return address, then it'll jump back to solve. When solve returns we'll double it's return value (in eax) and return to the saved original return address. This works in case we wish to double the return value of every call to solve including recursive calls, in case we want to double the return value for the external call only we'll add an if statement in our hook that in case the return address is equal to solve's address we don't override the return

address like we did before and just jump back to solve, this way only the external call to solve will have its return value doubled.

Wet -

Part1 -

We opened IDA with keygen.exe program.

We tried to follow but it was massive.

We tried to understand what the code does to each char he gets.

So we entered the next string as our parameter:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdef  
ghijklmnopqrstuvwxyz{|}~
```

which represent all chars which can be used for password, from 0x21 to 0x7e.

This way we can map each char - we will use this information to create the reverse code.

for that to work we added a few breakpoints before the end of the program, and this was our output:

```
5>AdIG*8Ee}@BDRFkzwJZYgrsm1q\2|3Jj(KuX!':_v6aWf{y"C;+L<S4HO[0?$bh/MtoT9,  
N)&l=xUpV^7~n#ciQP
```

Now we know what keygen program does for each char , BUT there is a problem - 2 chars are missing - input has 95 chars BUT output has 93 chars.

After examine we found that keygen.exe doesn't do anything with the chars: 'e' and ''.

Meaning we can't get 2 chars: '%' and '' '.

Therefore we will add it manually - we decided that '%' provide 'e' AND '' ' will provide ''.

The output from keygen_reverse.exe which we created :

```
&21&7gk7Y1=VCQ2]
```

Part2 –

We downloaded the client.exe & securepipe.exe files from tool page.

Examine Client.exe in IDA.

```
PS D:\Documents\Technion\ReverseEngineering\HW3\Part2> .\client.exe

What would you like to do?
[1] ECHO - ping the server with a custom message, receive the same.
[2] DMSG - Download message from the server.
[3] TIME - Get local time from server point of view.
[4] HNHK - Request a spinning top for Hanukkah!
your choice (4 letters command t): DMSG
4927-77467+868+76Q24-474686525-57267+86262657226-6636A72-27469766527+728-8486523-3697324-468656K6426-66966+823-3423226+8
496627-76667+87227-77367+868+56526-672656A7367+865+924-4776528-8736866+9756K6424-46672656526-674686527-76775792K
67+868+66527-767+675737425-5666965+96426-66A22-26366+9646528-86A737366+963696A74656424-47769746826-674686527-75246+94242
455258+7434A57-7545552454423-365766567+77425+9
57686566+825-57468697327-76367+8646525-5697326-6757365642K26-67268+76K6K6967+76728-874686523-36469636524-4736868+7756K64
27-7726573756K7423-37769746827-7637562657327-774686A7423-3737566+726-67466+928-87365766565+925+9

4767+8626K6968+6

PS D:\Documents\Technion\ReverseEngineering\HW3\Part2> |
```

We can't preform a WriteOver&jump hook on encrypt function because it occurs on the server and then send to the client. We can influence on the client.

We found that after using 'send' function our client.exe will preform the 'recv' function to receive answer from the server.

'recv' function is an imported function from WS2_32.DLL dynamic library.

So we want to inject dll using IAT hooking.

After examine the encrypt function in securepipe.exe , address : .text 0x401410

We can understand that there are 13 cases to encrypt each char in the DMSG message.

Lets analyze what encrypt function does in the securepipe.exe:

It takes the string (the response from the server) and for each char do the next:

1. Separate the hex number which represent the char to 2 buffers.
(Example : 'A' = 0x41 then separate to 4 in first buffer1 and 1 to buffer2).
2. For each buffer starting with the one representing the higher bits check if it larger then 9 :
if $2 \leq \text{Byte} \leq 9$: print the number as it is. (at the code there is 'ja' command which not includes zero and not includes CF-carry)

else, **switch**:

```
if Byte == 'A'(10): print 'J'
if Byte == 'B'(11): print 'Q'
if Byte == 'C'(12): print 'K'
if Byte == 1 : print 'A'
```

* if Byte == 0 :

Rand a num = X

Do the next calculation: (psodocode) □

res = divide it by 31 and keep the sign.

$$res = \frac{res}{29}$$

$$X = X + res$$

$$X = X \& (0x7)$$

$$X = X - res$$

$$X = X + 2$$

$$ans = X$$

Then we will print □'□'

Then we will print ans again

overoll we will print : ' ans – ans'

if $13 \leq Btye \leq 15$:

X = rand a number.

calculate :

$$expr1 = X\%(18 - Byte) + (Byte - 9)$$

$$expr2 = Byte - expr1$$

Then print: ' expr1 + expr2'

Note: the sum of exp1 and exp2 is Byte according to the second expression so all we need to do is to sum them.

Now we understood how the cypher works.

We would like to reverse it and the code will be the body of the hook function.

Note: Because we don't have control on server side - we will hook the function which receive the encrypted message we

We will use the DLL template and the injector template and we will adjust it for our needs:

recv function address is at: 0040A2AC

The Image base of the IAT table address is at: 400000

Means the offset is : $40A2AC - 400000 = A2AC$

Syntax

```
C++  
  
int recv(  
    [in] SOCKET s,  
    [out] char *buf,  
    [in] int len,  
    [in] int flags  
);
```

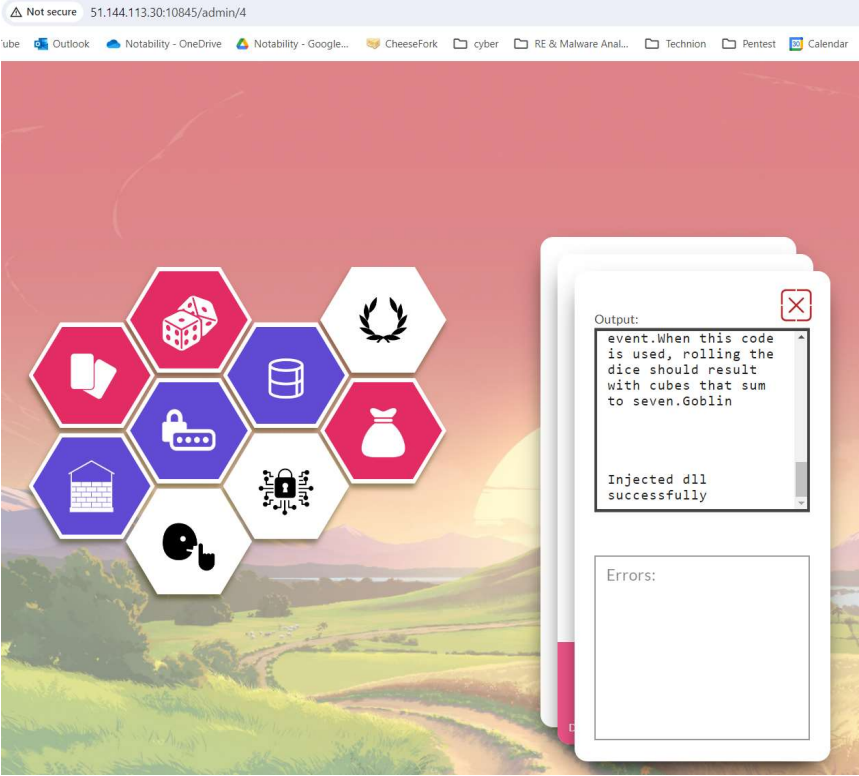
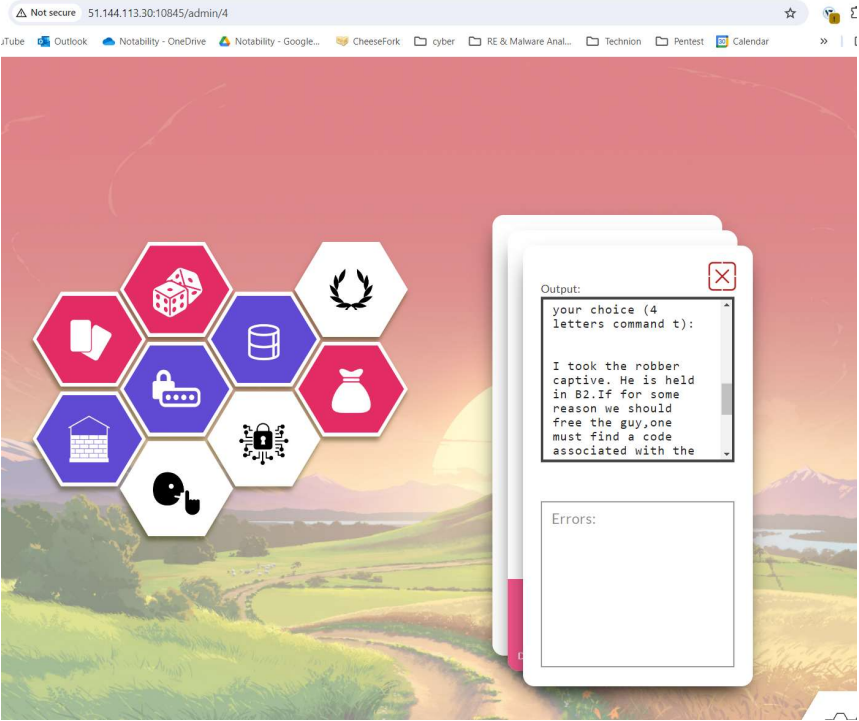
We wrote client.dll and ClientInjector build them together and created a ClientInjector.exe file. Now we will run the injector :

```
I took the robber captive. He is held in B3.If for some reason we should free the guy,one must find a code associated with  
the ROBBER_CAPTURED event.When this code is used, rolling the dice should result with cubes that sum to seven.Goblin
```

The decrypted answer:

“I took the robber captive. He is held in B3.If for some reason we should free the guy,one must find a code associated with the ROBBER_CAPTURED event.When this code is used, rolling the dice should result with cubes that sum to seven.Goblin “.

we will upload it to the website:



Part 3 –

codes.exe:

We examined the exe files - starting with codes.exe

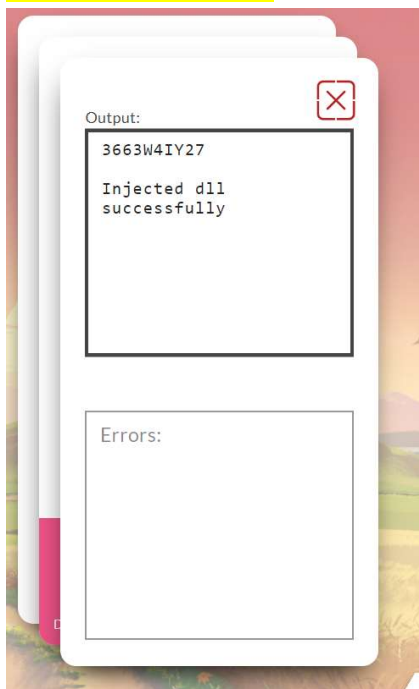
We need to find the event which ROBBER_CAPTURED - we can see that there is a query sent ,probably to database of the site , which will see if there is such CODE.

While dynamic analysis we understand that the second argument is checked first and then the first one - and we can see we must pass the 'strcmp' of those 2 arguments in functions sub_401410 and sub_401473 .

So we did IAT hook on strcmp which located in msvcrt.dll and counted the number of times it called. If it's the first time then return 1 to pass jne assembly command , and if it's the second time return 0 to pass jz assembly command.

This way the query is sent and we get the next message:

CODE: 3663W4IY27



*Note: The same template we used for part2 IAT hook didn't work for us in part3 which we also used IAT hook , so both parts the dll files are different a bit – it took us hours to figure the problem.

Dice.exe:

We examined the code and encountered:

At sub_401410 function :

0A7072696E7428436F64652E71756572792E66696C7465725F627928636F646520
3D2027257327292E666972737428292E6576656E742E6B657929

```
print(Code.query.filter_by(code = '%s').first().event.key)
```

At sub_4019F4 function :

there is a command on the stack:

707974686F6E202D63202266726F6D2064625F6D6F64656C7320696D706F727420
2A3B20257322

translate to:

```
python -c "from db_models import *; %s"
```

We understand that the code that provided for us in the codes.exe part is useful in dice.exe part.

We also noticed we must change some things in dice.exe for everything to work properly but we don't have a function which we can use from know dll.

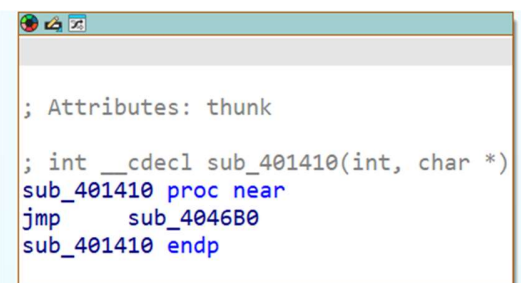
So we preformed a physical Hook. At function sub_401410 we need to jump to our hook compare OUR code to the code int str2 and to change the dices number that randomly chosen to 7 if its not 7.

We will over-write the first 5 instructions for jmp opcode and then we will recover the over written instructions at the start of the hook. Then we will continue to push our code and to compare it using _strcmp and change the dice to 7 .

The location of the hook will be at the end of the text section and we will append the hook binary to the dice binary and then compile it to be a new exe file dice_hook.exe

We need to patch the instruction of jmp opcode with the right address. After do so we can run it properly.

We upload dice_hook.exe to the website and it asking us where the robber is held. We know that it held in B2 from the message we got from the Golbin (part2).



```
; Attributes: thunk  
; int __cdecl sub_401410(int, char *)  
sub_401410 proc near  
jmp     sub_4046B0  
sub_401410 endp
```

Here's our simple implementation of the hook:

```
_hook:
    push    ebp
    mov     ebp, esp
    push    edi

    push    3732h
    push    59493457h
    push    33363633h
    push    esp            ; Str2

    mov     eax, [ebp + 12]
    push    eax            ; Str1

    call    strcmp
    add     esp, 14h

    cmp     eax, 0
    jnz     short loc_4046F5

    mov     [ebp + 8], 7

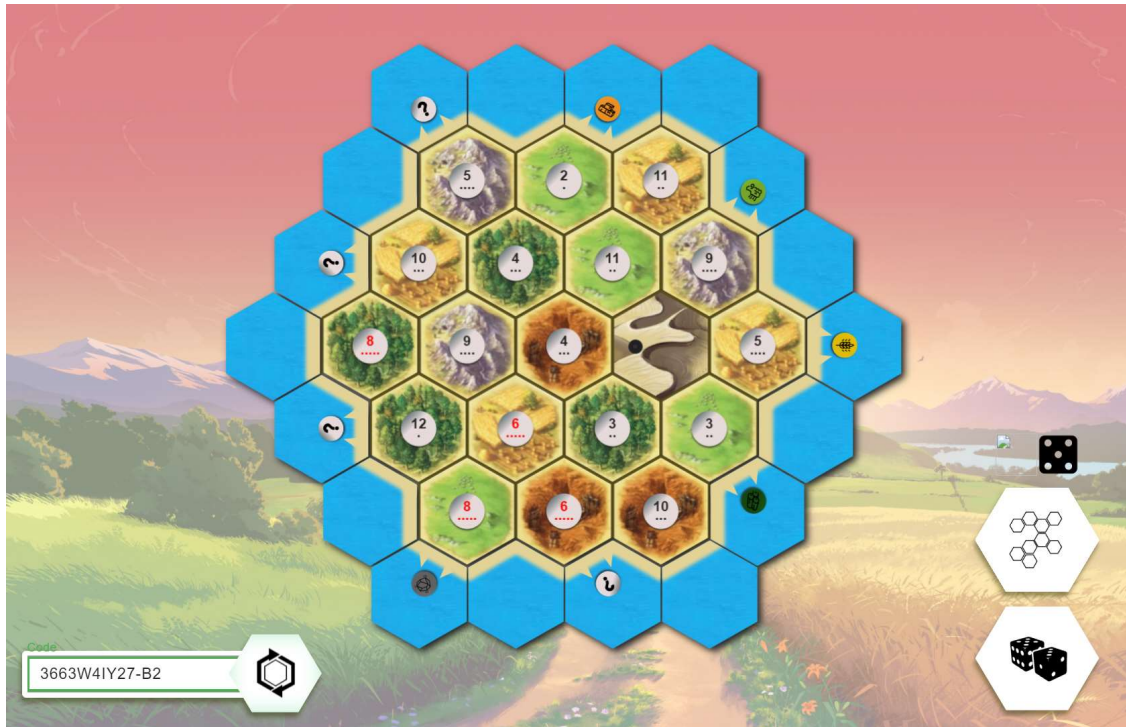
loc_4046F5:
    mov     eax, 1058h
    jmp     loc_401415 // back to sub_401410
```

Since the hook is relatively small we used CFF Explorer to extend the .text section of dice.exe by a little and then using IDA we located the end of the .text section then used it's Assemble feature to add the hook. After that we hooked it to function sub_401410 which handles dice rolls and thus successfully overridden dice rolls to our taste.

Note: while this method is relatively simple editing the code is extremely hard so we added a lot of nop operations in case we needed to edit the code we can override them.

```
push    ebp
mov     ebp, esp
push    edi
mov     eax, 1058h
nop
nop
nop
nop
nop
nop
push    3732h
push    59493457h
push    33363633h
push    esp            ; Str2
mov     eax, [ebp+arg_4]
push    eax            ; Str1
call    strcmp
nop
add     esp, 14h
nop
nop
nop
nop
```

Then we go to the board game it self we dice the cubes and also use the next code:



Result: (I don't know why we can see the second dice but it the same every time)

