# Reverse Engineering HW3

Tomer Bitan 322218611, Yosef Goren 211515606

## DRY:

1. A good way to set this hook would be to do a jump and replace hook. If poll writes to the space in memory to change the polling interval that means that area has write privileges and we can do it from our own hook. It could be rather simple to jump at the start of poll to our own code section added at the end of the file. All Our code section has to do is recreate everything in poll up to where it writes the polling rate. Then write our own value in and jump back to poll to the place after the write and after the replaced code.
   Another option is to inject a DLL to replace the one used to check the time. Our DLL can report a false time such that the next time interval will show only when we want the server to ping again.

2. Again a good way to do this hook would be to place a jump in the beginning of 'mine'.
   The jump will lead to a section of code that simply replaces the overwritten code then performs 'not' on the bits of the input and replaces the value in the same place in memory. We would also need to add another jump at the very end of the function to another code section of our own that will replace the overwritten code and 'not' the value of x again to restore the original value. This is to make sure that we don't mess up the rest of the functions using this space in memory. This way we don't need any additional memory space and we restore everything to the way it is before exiting. Since all we need to add is a single 'not' and 'mov' command we might be able to do this in the beginning without jumping if the function is set up for hooks.

3. To do this hook we can go the same way as before, placing a single jump to our own code in the beginning.
   In our code, we will recreate the overwritten code then check the arguments and jump back if they don't match. If they do we simply need to parse the full message and then encrypt it before mimicking the original function all the way until the placement of the message in the socket. This will prevent us from needing to replace the original sting and args which will be tricky now that they are encrypted.
   Since the message sending is most likely done with an external module DLL, a simpler way to do this hook if the structure of the code permits is to use DLL hooking to replace the actual socket send function with our own that encrypts the message before calling the original. This way we will have a lot less "restoring"

and code replication to do. Placing the DLL to replace the original function could be done as we saw in the workshop by calculating the offset and using an injector function to replace it.

4. Since both functions always appear one after the other, to hook parse and connect we only need to write a hook that will jump out of the start of the connect function to a code section of our choosing.
   In our code, we need to first create a new thread that shares memory with this process. Then we can run parse in that thread. We should already have the ability to calculate all the values we need to input to pars since we know they are not dependent on each other at all. Once we calculate the value and call parse in a separate thread we can add an exit for the thread's pid so it won't continue running after the function. Then in the current thread, we can either overwrite the return address to be the next step after parse. Or reproduce all the code in Connect including the overwritten code. So that in the end, we can jump to the place after parse.
   In both cases we will get code that executes connect in the original thread and parse in a new thread that stops after that function while the original thread skips the second call to parse and continues to run the program.

5. To hook this function we will need an IAT hook. We can create a section for our own code with a wrapper function inside. We will replace the address of calc in the IAT with the address of our function so then every time it is called our code will run. Now all our code needs to do is call the calc function with the same arguments and when it finishes log the value then finish. This will cause the original calc to remain unchained but our function will be able to run around it and return to the same location it's called from.

6.
   a. Here again an IAT hook will help us. If we replace the address of the solve function with code of our own in a new section then we can have our code call solve then multiply the result before returning to the caller. This after every call the value is doubled even when being called from itself.
   b. For the case of only multiplying at the last case, we will need a similar approach with an additional twist. We will create a wrapper function with iat hook as before but this time will make sure to keep track of a global variable somewhere in the stack that is increased every time before we call the real solve. Then after the real solve finishes we will check its value if it's 1 we will double the result. Then we decree the value by one. It's important to notice that we don't care if the number of nodes in a given layer it just one because we will eventually have one in the root and that is

the result we need to multiply. In both cases, we can replace the IAT hook with a jump to our code at the beginning of the function that also overwrites the return address to be our next piece of code. As long as we add correction the code is overwritten by the jump.

## **<u>WET:</u>**

*<u>Part 1:</u>*

- We know that we need to analyze ‘keygen.exe’ so that we can eventually create an executable that will execute it’s exact inverse operations.
- This is essentially an encryption & decryption scheme and we will refer to it as such from this point on.

- We start off by attempting to open the ‘keygen.exe’ file with IDA to perform static analysis.
    - we see the file is not too simple and that the section's name in the questions file is ‘dynamic analysis’, so we try a different direction.

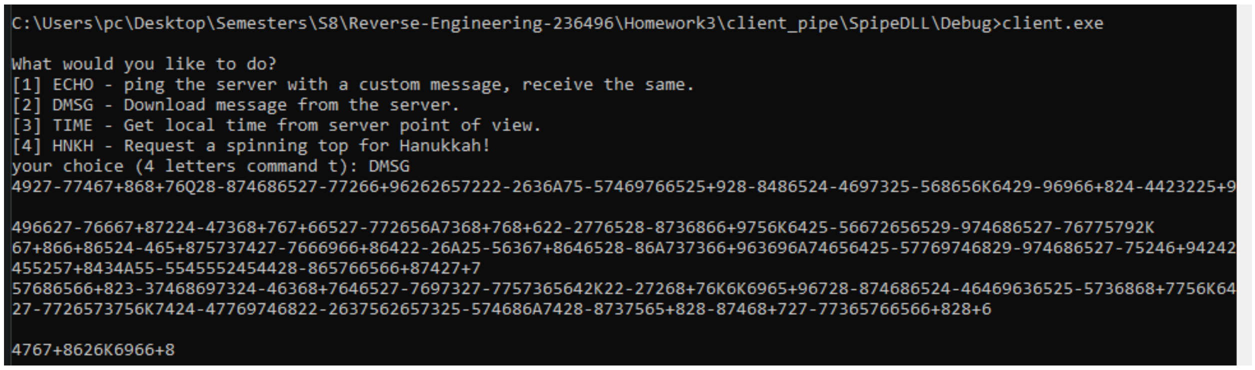
- We start off with black-box analysis by running the file with different outputs and seeing what it prints.
- We immediately notice that this encryption scheme works independently on each char.
- This is very good news for us since we don’t actually need to understand how the executable works - rather we need to create a decryption for it.
- To do so, we can implement a simple CPA table attack on this executable by running it on all possible chars and saving the output in a table; then changing the table such that the outputs now correspond to inputs and vice-versa.
    - This would typically be done by some simple Python script for creating the table.
    - we did this is a somewhat unique way, that would be easier to show than explain, so we have added a video of it and added it to the submission extras as ‘creating_kegen_table.mp4’.
- In any case, we now have a table to map each output of ‘keygen.exe’ to what it came from, so we make a simple cpp executable based on it which would decrypt a string given to it.
- Now that we have the decryption program - we simply run it on the key we have (and then after getting the new key) we manage to complete Part 1 and unlock ‘Tools’ using the decrypted password we have found.
- The password we were given to use as input was: ‘FZPF1FXY3YXUMCL1’
- And the reverse key we got from our code was : “27~2=2H8B8HsgVY=”

*Part2*:

- We start Part 2 by looking at the list of available links at the Tools section in the Catan website which we unlocked in the prior Part.

- Most interesting to us is the ‘client.exe’ file which we were instructed to ‘fix’ so it will output a readable output.
- Also, there is a link for a ‘secure pipe’. it claims to be the method in which the steam of data sent to the client is encrypted.

- We start off by downloading both of these files to our system and seeing what they do.
- We run ‘client.exe’ and give it the DSMG option as instructed and indeed we see something that looks like encrypted content:



```
C:\Users\pc\Desktop\Semesters\S8\Reverse-Engineering-236496\Homework3\client_pipe\SpipeDLL\Debug>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] HNKH - Request a spinning top for Hanukkah!
your choice (4 letters command t): DMSG
4927-77467+868+76Q28-874686527-77266+96262657222-2636A75-57469766525+928-8486524-4697325-568656K6429-96966+824-4423225+9

496627-76667+87224-47368+767+66527-772656A7368+768+622-2776528-8736866+9756K6425-56672656529-974686527-76775792K
67+866+86524-465+875737427-7666966+86422-26A25-56367+8646528-86A737366+963696A74656425-57769746829-974686527-75246+94242
455257+8434A55-5545552454428-865766566+87427+7
57686566+823-37468697324-46368+7646527-7697327-7757365642K22-27268+76K6K6965+96728-874686524-46469636525-5736868+7756K64
27-7726573756K7424-47769746822-2637562657325-574686A7428-8737565+828-87468+727-77365766566+828+6

4767+8626K6966+8
```

- Now we start to analyze the ‘secure_pipe.exe’ file and this time we start with black-box analysis, but we can’t get it to print anything to the screen with either the command line args or stdin - so we start running static analysis on it with IDA, and we quickly find that it uses the c++ builtin ‘fstream’ calls constructor on its first command line argument, which essentially means it opens that string as a file.
- So we create some arbitrary input file named ‘input.txt’ and run ‘secure_pipe.exe input.txt’, and this time it does output some gibberish that looks somewhat similar to the encryption from before.
- This is a good sign.
- So we start a sort of ‘grey-box’ analysis on the secure pipe by giving different inputs inside that input file and seeing what the output is.
- We quickly see that its output seems completely independent between different chars in the input - so it’s a good idea to implement a similar CPA table attack as in Part1.

- The problems start when we see the output of the program is not quite deterministic.

    - We see that the only types of output that seem to be changing are ones with ‘+’ or ‘-’ signs, so we try to do the same table attack - only this time we evaluate the two operands in the ‘+’ and ‘-’ expressions.
- Since the output is not deterministic and we see we want to implement this evaluation of expressions rather than the basic table attack - we decide to implement this in a Python script with will do all of these evaluations and create the decryption table (note that this was not our actual final solution…) and you can find this script in the ‘extras’ as ‘dec.py’.

- After some work on the script it seems to be close to working, only it seems to ‘mix up’ some of the letters: and we see that the table created by our script is ambiguous!
- This causes us to think maybe we need to try implement the decryption in a different way, so we go back to running static analysis on ‘secure_pipe.exe’.
- After a deeper look into it’s binary we can clearly see one subroutine which implements the encryption - so we know which function we need to reverse.
    - This function has a large loop - where each iteration processes the next character from the input strings and appends characters to the output string during the iteration.
    - We also see that the first thing done inside each iteration is separating the right and left nibbles of the current char.
    - Now we that at the start of the iteration - some chars corresponding to the left nibble are printed - then some chars corresponding to the right nibble are printed.
    - So we look at exactly what is printed for each nibble value and come up with the following rules for translating a token (outputted value in the ciphertext) back into the nibble value it came from:

| **Input Token** | **Source Nibble** |
|---|---|
| x-x | 0x0 |
| ‘A’ | 0x1 |
| ‘J’ | 0xA |
| ‘Q’ | 0xB |
| ‘K’ | 0xC |
| x+y | <x+y> |
| x (else) | <x-48> |

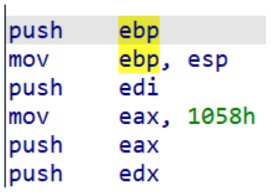
- The next thing is to be able to tokenize a cipher - meaning we get a long string as an input and we split it up into a list of tokens that can each be translated into a nibble according to the prior table.
- Indeed this is not a very complicated thing to implement since there are only two types of output patterns for any nibble in the encryption algorithm: either it generates 1 or 3 chars in the output, where in the case of 3 - it has to be either x-x or x+y.
- So, our tokenizer checks if the 3-char token is possible, and otherwise parses a single char as the next token.
- Now that we know how to split the cipher into tokens and how to translate each one back into its source nibble - we are left with a list of nibbles that represent the original input string to the encryption algorithm.
    - All that is left is to combine each pair in that list of nibbles into a byte which will be interpreted as a (ascii) char.
- After implementing all of this logic we are left with a working decryption algorithm.
- We use this decryption algorithm to try and decrypt the gibberish we got before with ‘client.exe DMSG’ and we see a sensible output.
- What is next for us is to take the decyption algorithm we have created and put it inside a file ‘dec.cpp’ which will be called from a hook to cause the ‘client.exe’ to decrypt its own output
- The easiest way to change the output would be to capture it right before this is printed and run a description function on it.
- So we want to hook the ‘puts’ function
    - We don't need to modify the function permanently so we don't need a static hook.
    - It seems that there is no room to inject code with a jump and replace the hook without adding an external section.
    - We will use IAT hook
- We started with the injector we saw in workshop 2. Then we modified it to run the subject file (the one we are injecting into) with additional arguments
- Since we are doing a IAT hook to create the DLL that will set the hook we need to take the one we saw in the workshop and change the name find the function offset and then change the hook content.
    - To find the offset of the function we find the symbol for ‘puts’ in the data section and note its address
    - Then we run the client.exe in debugger mode with IDA and find the address of the start of the module we are running. Subtracting this from the address of the symbol will give us its offset. This should stay consistent between runs letting us catch it every time.
    - During runtime we also find the module puts is imported from to input to the GetProcAddress.

- For the hook we simply split the message into lines and sent each line to the decryption program we mentioned before. Then concatenated the outputs and sent them to the real ‘puts’ function.
- The final message we got was
  “I took the robber captive. He is held in B2.
  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.“

*Part 3*

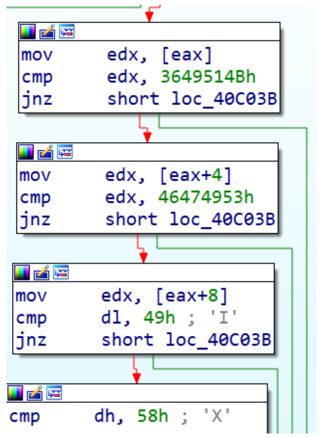
- Reading through the instruction we got at the end of the last section we see that we need to find some code associated with the event ROBBER_CAPTURED.
- We look at the exe file called codes. Running it indicates we need an old code and a code key.
- When looking through the code we find two Python queries that seem to indicate that the code_key is the event name which is the key to retrieving the code.
- However, the old_code seems to be compared to a value in a db so we can't see what it needs to be.
- The only use for the old code is to quit early if it doesn't match an existing code. So instead of finding a matching code, we can make sure that any code seems valid to the check.
- Since the check is done with strcmp we can try to replace it with our own version.
- To replace a function from an external library we need a DLL injection. The easiest way to do that is with an IAT hook.
- Like before we need a DLL, modifying the injector, and a hook.
- Modifying the injector took very little work. Only changing the DLL to find the arguments.
- To write the DLL we needed to:
  - Find the new offset. We did in the same way as before, viewing it in ida and finding the start of the module to calculate the difference
  - As well as replacing the name of the DLL being replaced and the signature..
  - Lastly, we had the DLL call the hook instead of the original strcmp
- Our hook was a lot simpler than in the previous section.
  - All the hook does here is check if we are in the strcmp with the name of the event ‘ROBBER_CAPTURED’. If so we return 0 otherwise 1.
  - This way when checking if the old code is “NO SUCH CODE” we will always get no and the program will continue.
  - And the check to see if the returned value from the old code is the same event key will always get a yes no matter what is actually returned.
- We Zip this code up in the same way as before with the DLL, the original codes.exe, and the injector renamed to toolfix.exe.
- We upload to the server and get the following code: ”KQI6SIGFIX”

- Since the message from the previous section says we need to get a 7 on the dice roll with this code we try it in the same format as in the previous HW with code-place to get: KQI6SIGFIX-B2. However, we don't get a 7.
- We understand that we might need to modify the existing dice.exe code to make sure it gives us a 7 when we give the code. However, we see we can't use the same style hook as before because the upload only accepts a single exe file.
- We look at the code of dice.exe to find a place to intercept the flow forcing a 7. Finding the function at the end of the main run that accepts the original argument of the program if it exists and the dice result.
    - This function seems to return two dice that sum up to this value.
    - We think if we can replace the value the dice decided before generating the summing dice we can check for our code and then change it to 7.
- We open dice.exe in CFF explorer to add a section at the end with execution privileges to write our hook in.
- After struggling to write the assembly in IDA we find a compiler online that shows the opcodes for assembly instructions and start rewriting our code in the hex view.
- The Idea for the code is to
    - reproduce the few instructions we replaced with the jump in the original code



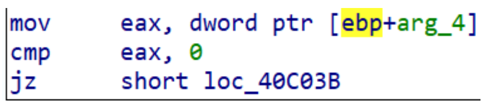
```
push    ebp
mov     ebp, esp
push    edi
mov     eax, 1058h
push    eax
push    edx
```

    - Then go over all chars in the input argument and compare them one by one to be the code we need to make sure the code functions the same in all other situations but still gives 7 for our code.

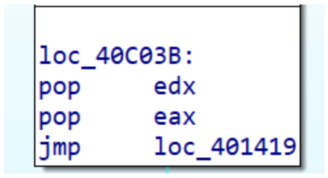


```
mov     edx, [eax]
cmp     edx, 3649514Bh
jnz     short loc_40C03B

mov     edx, [eax+4]
cmp     edx, 46474953h
jnz     short loc_40C03B

mov     edx, [eax+8]
cmp     dl, 49h ; 'I'
jnz     short loc_40C03B

cmp     dh, 58h ; 'X'
```

    - If we didn't receive an argument or got the wrong one we restore the registers we used and jump back to the next line in the code.

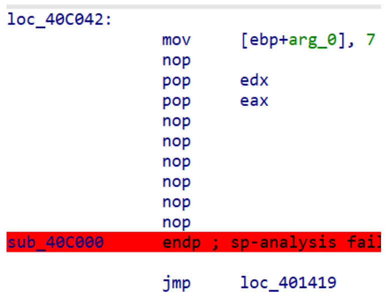


```
mov     eax, dword ptr [ebp+arg_4]
cmp     eax, 0
jz      short loc_40C03B
```



```
loc_40C03B:
pop     edx
pop     eax
jmp     loc_401419
```

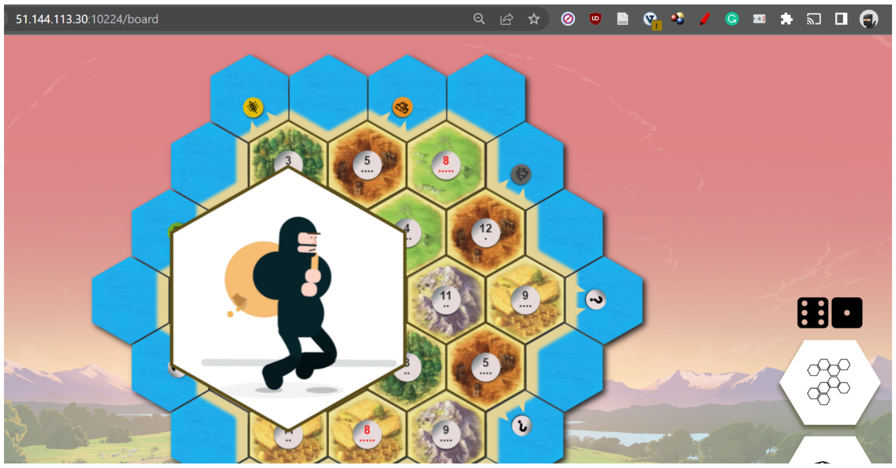
- ○ If we got our code we put 7 into the argument of the original function to mimic rolling a 7. Then we restore the registers we used and jump back to the original code to the next line after our jump. Before the calculation of the individual dice.



```
loc_40C042:
                mov     [ebp+arg_0], 7
                nop
                pop     edx
                pop     eax
                nop
                nop
                nop
                nop
                nop
                nop
sub_40C000      endp ; sp-analysis fai

                jmp     loc_401419
```

- ○ To calculate the jump we subtract the address of the destination from the jump-off point.

- Now that the dice returns a sum of seven every time we run with our code we upload to the site and FINALLY released the robber.


51.144.113.30:10224/board