Reverse Engineering Project Documentation

Team 06 (Shane, Humberto, Jared)

Discoveries and Assumptions:

- The first line of the encrypted file is the hashed password and the rest of the lines are the encrypted plaintext. Modifying the first line of the encrypted file results in a failed decryption.
- Removing characters from the rest of the file and the effects on decryption
    - Removing the first nine characters from the given encryption.txt results in the first word ‘malware’ being removed from the decrypted plaintext
    - Removing any subset of characters besides the first n characters results in a failure to decrypt the message
    - Appending subsets of characters to the end of the encrypted message results in garbage text at the end of the decrypted plaintext
- We initially assumed that the SHA256 hashing algorithm was used to hash the password and encrypt the plaintext; however, given the behavior of the decryption oracle it is clear that a form of block cipher was used to encrypt the plaintext and the hash function was only needed to hash the password.
- Using IDA to analyze EDTool_D.exe, several strings from Main_Students.cpp can be found that are used in the graph at function sub_434A20. These strings include:
    - “wb+”
    - “decrypted.txt”
    - “Error - Could not open output file.\n\n”
    - “Error - Input file too large.\n\n”
    - “Error - Password is too long or zero!\n”...
    - "Error - Could not allocate %d bytes of "...
    - "Error - Password not hashed correctly."...
- It appears that most, if not all of the code needed to rebuild Main_Students.cpp is located at sub_434A20 in IDA. It seems to contain the entirety of the encryption/decryption function and contains calls to multiple cryptographic functions that are not included in Main_Students.cpp.
- The three new functions are located at sub_434980, sub_4348c0, and sub434860
- After performing a static analysis and commenting through the disassembly in IDA, we drafted a pseudocode version of the decryption algorithm, and then we began decompiling the code with Ghidra.
- The decompiled C++ code confirmed that three cryptographic functions are defined outside of the encryptFile function that are called upon once per function, per character

in buffer (encrypted message/encrypted.txt) as well as being XOR'd with some specific bytes from pwdHash (var_134 or var_125) depending on the outcome of an AND operation.

- After stepping through the disassembly in IDA Pro and monitoring the values in the edx register during the XORs, we determined that var_125 and var_134 correlated with pwdHash[8] and pwdHash[22] respectively.
- We successfully cleaned up the raw decompiled code from Ghidra and began encrypting secret.txt with the block cipher algorithm. Sometimes the output encryption.txt looks extremely similar to the original encrypted message format but EDTool_D.exe is unable to decrypt the file.

Relevant Code:

Drafted Pseudocode

```
// Run *(buffer + index var) through crypto function 434980 and replace it with the return value

// if the result AND 4 equals 0 :

        // XOR the current buffer char with var_134 and replace it with the result

// else

        // XOR the current buffer char with var_125 and replace it with the result

// Run the current buffer char through crypto function 4348C0 and replace it with the return
value

// Run the current buffer char through crypto function 434860 and replace it with the return value

 // increment the index variable and continue the loop
```

Decompiled Code from Ghidra

```
i = 0;
while (i < fileSize - (passwordLength + 1))
{
        uVar1 = thunk_FUN_00434980(*(byte *)((int)buffer + i));


        /*uint __cdecl FUN_00434980(byte param_1)

        {
                int iVar1;
                undefined4 *puVar2;
                undefined4 local_f4[60];

                iVar1 = 0x3c;
        puVar2 = local_f4;
        while (iVar1 != 0) {
                iVar1 = iVar1 + -1;
                *puVar2 = 0xcccccccc;
                puVar2 = puVar2 + 1;
        }
        return (uint)(byte)((byte)(((uint)param_1 & 0x30) << 2) | (byte)((int)((uint)param_1 & 0xc0) >> 2)
                | (byte)(((uint)param_1 & 3) << 2) | (byte)((int)(uint)param_1 >> 2) & 3);
}

*/

*(undefined *)((int)buffer + i) = (char)uVar1;

if ((i & 4) == 0) {
        *(byte *)((int)buffer + i) = *(byte *)((int)buffer + i) ^ var_125; // pwdHsah[8]
}
else
{
        *(byte *)((int)buffer + i) = *(byte *)((int)buffer + i) ^ var_134; // pwdHash[22]
}


uVar1 = thunk_FUN_004348c0(*(byte *)((int)buffer + i), '\0');

/*
uint __cdecl FUN_004348c0(byte param_1, char param_2)

{
        int iVar1;
        undefined4 *puVar2;
        undefined4 local_dc[49];
        byte local_15;
        byte local_9;

        iVar1 = 0x36;
        puVar2 = local_dc;
        while (iVar1 != 0) {
                iVar1 = iVar1 + -1;
                *puVar2 = 0xcccccccc;
                puVar2 = puVar2 + 1;
        }
        if (param_2 == '\x01') {
                local_9 = (byte)((int)(uint)param_1 / 2) | param_1 << 7;
        }
        else {
                local_15 = param_1 & 0x80;
                if (local_15 != 0) {
                        local_15 = 1;
                }
                local_9 = param_1 << 1 | local_15;
        }
        return (uint)local_9;
}

*/
```

```
        *(undefined *)((int)buffer + i) = (char)uVar1;



        uVar1 = thunk_FUN_00434860(*(byte *)((int)buffer + i));

        /*
        uint __cdecl FUN_00434860(byte param_1)

        {
                int iVar1;
                undefined4 *puVar2;
                undefined4 local_dc[54];

                iVar1 = 0x36;
                puVar2 = local_dc;
                while (iVar1 != 0) {
                        iVar1 = iVar1 + -1;
                        *puVar2 = 0xcccccccc;
                        puVar2 = puVar2 + 1;
                }
                return (uint)(byte)(param_1 * '\x10' + (char)((int)(uint)param_1 >> 4));
        }
        */

        *(undefined *)((int)buffer + i) = (char)uVar1;
        i = i + 1;
}
```

Failures:

- Overall, our executable does encrypt the plaintext with a block cipher but unfortunately does not use the correct encryption algorithm. Thus, plaintext encrypted using our executable cannot be decrypted by EDTool_D, nor does the executable decrypt the messages it encrypts
- Jared: I attempted to encrypt the plaintext by using the initial password hash and the buffer from the input file. The variables ‘pwdHashIndx’ and ‘i’ being defined in the code template gave me the idea that somehow the small 32 byte hash was being reused in a loop until the entire plaintext was encrypted. I realized this was a red herring when I figured out that the decryption oracle was for a block cipher and not SHA256, and it became clear that if my theoretical solution was never going to work after multiple failed attempts to loop through buffer and putting it through the hash function, then somehow write it to ‘encryption.txt’.
- Humberto: I followed the code from ida pro using the breakpoints. I was able to follow all the functions the program ran. By finding encrypted.txt and other error messages printed to stderr i was able to find the functions that opened the txt file for decryption and the ones that printed to stderr the error messages. When I found what I believe was the decryption algorithm, I failed to translate that to c code since it had a lot of instructions. I

then decided to use ghidra. I was able to find the functions that opened the file for reading and for writing to a file. I also found the functions that had the decryption algorithm. Since all of this code was just decompile code from the assembly instructions it had many variables and instructions that I wasn't able to understand or find what they were.

- Shane: I attempted to find the value that the buffer was being XOR'd with directly through IDA Pro by examining the encryption function's assembly code. However, I was unable to determine the exact value of var_134 and var_125, and how they related to pwdHash. Thankfully, Jared was able to find the pwdHash values in the stack. Unfortunately, towards the end I ran into some form of error with my IDE that would return a memory issue whenever I ran our main file.

Contributions:

- Humberto Gonzalez: Located usage of relevant strings used from Main_Students.cpp in EDTool_D.exe using IDA Pro, produced the decompiled C++ code from EDTool_D.exe using Ghidra.
- Jared: Proofread the documentation, determined that a block cipher was being used to encrypt the plaintext, decoded the encrypted.txt password/message format, performed static analysis on the disassembly produced by IDA Pro and found the bulk of the relevant code that is also in Main_Students.cpp, commented through the disassembly and updated the variable offset names, produced pseudocode for the encryption/decryption algorithm.
- Shane: Proofread the documentation, helped to locate the code that mirrored encryptFile(...) from Main_Students.cpp using IDA Pro, translated and rewrote missing C++ code from Humberto's Ghidra decompilation.