REVERSE ENGINEERING AND ANALYSIS OF DIFFERENT C SCRIPTS

Task 3 in Malware Analysis workshop

Youmna Elmezayen

1- Reverse:

1) Hidden Functionality.c

Original script:

```c
#include <stdio.h>
#include <string.h>

void hidden_feature(char *input) {
    char key[] = {0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE};
    char secret[] = {0x7F, 0x49, 0x2F, 0x12, 0xA3, 0xDF, 0x4B};

    int correct = 1;
    for (int i = 0; i < sizeof(secret); i++) {
        if ((input[i] ^ key[i]) != secret[i]) {
            correct = 0;
            break;
        }
    }

    if (correct) {
        printf("Secret feature activated!\n");
    } else {
        printf("Access denied.\n");
    }
}

int main() {
    char input[20];
    printf("Enter command: ");
    fgets(input, sizeof(input), stdin);
    input[strcspn(input, "\n")] = 0;
    hidden_feature(input);
    return 0;
}
```

Reverse script:

```c
#include <stdio.h>
#include <string.h>

void Hidden_Feature_Unlock()
{
    char key[] = {0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE};
    char secret[] = {0x7F, 0x49, 0x2F, 0x12, 0xA3, 0xDF, 0x4B};

    char input[sizeof(secret)];

    for (int i = 0; i < sizeof(secret); i++)
    {
        input[i] = secret[i] ^ key[i];
    }

    printf("Input: ");
    for(int i = 0; i < sizeof(input); i++)
    {
        printf("%c", input[i]);
    }
}


int main()
{
    Hidden_Feature_Unlock();
    return 0;
}
```

Running the reverse script gives the needed input:

```
Input: m}yj9cò
Process returned 0 (0x0)   execution time : 8.370 s
Press any key to continue.
```

Using the reversed input in the original script:

```
Enter command: m}yj9cò
Secret feature activated!

Process returned 0 (0x0)   execution time : 24.897 s
Press any key to continue.
```

2) Encoded String Manipulation.c

Original script:

```c
#include <stdio.h>
#include <string.h>

void decode(char *str) {
    for (int i = 0; i < strlen(str); i++) {
        str[i] -= 3;
    }
}

int main() {
    char message[] = "Khoor#Zruog$";
    decode(message);
    printf("Decoded message: %s\n", message);
    return 0;
}
```

Reverse script:

```c
#include <stdio.h>
#include <string.h>

void encode(char *str) {
    for (int i = 0; i < strlen(str); i++) {
        str[i] += 3;
    }
}

int main() {
    char message[] = "Hello World!";
    encode(message);
    printf("Encoded message: %s\n", message);
    return 0;
}
```

Running the original script:

```
Decoded message: Hello World!

Process returned 0 (0x0)   execution time : 8.530 s
Press any key to continue.
```

Using the output to find its encoding:

```
Encoded message: Khoor#Zruog$

Process returned 0 (0x0)   execution time : 8.473 s
Press any key to continue.
```

3) Self-Modifying Code.c

Original script:

```c
#include <stdio.h>
#include <string.h>
#include <windows.h>

void self_modify(char *code, int len) {
    DWORD oldProtect;
    VirtualProtect(code, len, PAGE_EXECUTE_READWRITE, &oldProtect);
    for (int i = 0; i < len; i++) {
        code[i] ^= 0xFF;
    }
    VirtualProtect(code, len, oldProtect, &oldProtect);
}

int main() {
    char code[] = "\x55\x48\x89\xE5\xB8\x01\x00\x00\x00\x5D\xC3";
    self_modify(code, sizeof(code) - 1);
    printf("After modification: ");
    for (int i = 0; i < sizeof(code) - 1; i++) {
        printf("%02X ", (unsigned char)code[i]);
    }
    printf("\n");

    return 0;
}
```

Reverse script:

```c
#include <stdio.h>
#include <string.h>
#include <windows.h>

void self_modify(char *code, int len) {
    DWORD oldProtect;
    VirtualProtect(code, len, PAGE_EXECUTE_READWRITE, &oldProtect);
    for (int i = 0; i < len; i++) {
        code[i] ^= 0xFF;
    }
    VirtualProtect(code, len, oldProtect, &oldProtect);
}

int main()
{
    char code[] = "\xAA\xB7\x76\x1A\x47\xFE\xFF\xFF\xFF\xA2\x3C";
    self_modify(code, sizeof(code) - 1);
    printf("Before modification: ");
    for (int i = 0; i < sizeof(code) - 1; i++) {
        printf("%02X ", (unsigned char)code[i]);
    }
    printf("\n");
    return 0;
}
```

Running the original code to get code after modification:

```
After modification: AA B7 76 1A 47 FE FF FF FF A2 3C

Process returned 0 (0x0)   execution time : 8.800 s
Press any key to continue.
```

Running reverse code with modified code as input:

```
Before modification: 55 48 89 E5 B8 01 00 00 00 5D C3

Process returned 0 (0x0)   execution time : 8.297 s
Press any key to continue.
```

4) Obfuscated Math Operations.c

Original script:

```c
#include <stdio.h>

int obfuscated_math(int a, int b) {
    int result = 0;
    while (a != 0) {
        if (a & 1) {
            result ^= b;
        }
        b <<= 1;
        a >>= 1;
    }
    return result;
}

int main() {
    int x = 66, y = 41;
    printf("Result: %d\n", obfuscated_math(x, y));
    return 0;
}
```

In this code, we are trying to make a specific math operation with a bit of a work-around as a kind of obfuscation. If we look closely, we observe that in the obfuscated_math function, we check if a is odd. If it is, we XOR (^) b with result and store the answer in result. What happens regardless of if a is odd or even is that b is multiplied by 2 and a is divided by 2 until it becomes 0, in which case the loop will end. This means that if a was not odd at the beginning, it will surely be odd if divided enough by 2 (until it reaches 1). The thing to note here is that XOR could be thought of as an addition operation. Kind of.

The equation XOR follows is a ^ b = a + b – 2(a & b). This equation implies that if a & b is 0, then XOR is exactly equal to a + b.

Putting all this information together, we can deduce that this code is multiplying the inputs together if the (b * pow(2, count)) AND b = 0. If not, then (b * pow(2, count)) AND b is deducted from the product. Assuming count is the number of times a was divided by 2 before it became 1.

Example runs:

1- x = 66, y = 41

```
B after shift: 82
A after shift: 33
result xor: 82
B after shift: 164
A after shift: 16
B after shift: 328
A after shift: 8
B after shift: 656
A after shift: 4
B after shift: 1312
A after shift: 2
B after shift: 2624
A after shift: 1
result xor: 2578
B after shift: 5248
A after shift: 0
Result: 2578
```

2- x = 9, y = 12

```
result xor: 12
B after shift: 24
A after shift: 4
B after shift: 48
A after shift: 2
B after shift: 96
A after shift: 1
result xor: 108
B after shift: 192
A after shift: 0
Result: 108
```

3- x = 9, y = 9

```
result xor: 9
B after shift: 18
A after shift: 4
B after shift: 36
A after shift: 2
B after shift: 72
A after shift: 1
result xor: 65
B after shift: 144
A after shift: 0
Result: 65
```

Reverse script:

```c
#include <stdio.h>
#include <math.h>

int deobfuscated_math(int a, int b)
{
    int count = 0;
    int orig_a = a;
     while (a != 1)
     {
         a /= 2;
         count++;

     }
     a = orig_a;
     return (a * b) - 2 * ((int)(b * pow(2, count)) & (a % 2 == 0 ? b * 2 : b));
}

int main() {
     int x = 9, y = 9;
     printf("Result: %d\n", deobfuscated_math(x, y));
     return 0;
}
```

Example runs of reverse script:

1- x = 66, y = 41

```
Result: 2578

Process returned 0 (0x0)   execution time : 9.586 s
Press any key to continue.
```

2- x = 9, y = 12

```
Result: 108

Process returned 0 (0x0)   execution time : 6.909 s
Press any key to continue.
```

3- x = 9, y = 9

```
Result: 65

Process returned 0 (0x0)   execution time : 8.351 s
Press any key to continue.
```

5) Data Hiding in Executable.c

Original script:

```c
#include <stdio.h>
#include <string.h>

char hidden_data[] = {
    0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x00
};

void print_hidden_data() {
    printf("hidden: %d\n", hidden_data);
    char *data = (char *)((long)hidden_data ^ 0x55);
    printf("Hidden data: %d\n", data);
}

int main() {
    print_hidden_data();
    return 0;
}
```

Reverse script:

```c
#include <stdio.h>
#include <string.h>

void GetHiddenData(long fwOutput)
{
    long hiddenData = (fwOutput ^ 0x55);
    printf("Hidden data: %d\n", hiddenData);
}

int main()
{
    long fwOutput = 4206661;
    GetHiddenData(fwOutput);
    return 0;
}
```

Running the original script to get the hidden data:

```
Input: 4206608
Hidden data: 4206661

Process returned 0 (0x0)   execution time : 10.691 s
Press any key to continue.
```

Running the reverse script to get input:

```
Hidden data: 4206608

Process returned 0 (0x0)   execution time : 8.528 s
Press any key to continue.
```

2- Analysis:

1) Network Sniffer.c

A network sniffer is a program or tool that is used to capture traffic within a network. It captures packets and could display information about them that could be useful for analysis and sometimes to perform malicious activities like password stealing. They are also referred to as protocol analyzers.

The following C code is a basic network sniffer that we will attempt to analyze and explain in detail.

```
#include <stdio.h>
#include <pcap.h>
```

We will start by analyzing the libraries used. The stdio.h library is used to allow the program to handle input and output processes such as reading input from a user or printing something on the screen.

The pcap.h library is used for capturing network traffic. It is based on an API that facilitates packet sniffing by providing built-in functions and data types that help in the process.

Next, we will analyze the main function.

```
int main() {
    pcap_if_t *alldevs;
    pcap_if_t *d;
    pcap_t *adhandle;
    char errbuf[PCAP_ERRBUF_SIZE];
```

pcap_if_t * is a reference to a struct defined in the pcap library that typically holds the list of devices' interfaces that pcap can use to sniff the packets of. The code defines 2 variables, alldevs and d, of type pcap_if_t that will be used throughout the program.

Furthermore, the code defines a variable adhandle of type pcap_t that is used to handle an instance of an interface that is open for sniffing.

Lastly, the errbuf variable is used to store error messages that could result from specific function calls when something goes wrong. It is mainly used for debugging purposes. This buffer must be able to hold at least PCAP_ERRBUF_SIZE (currently 256) bytes.

```
if (pcap_findalldevs(&alldevs, errbuf) == -1) {
    printf("Error in pcap_findalldevs: %s\n", errbuf);
    return 1;
}
```

pcap_findalldevs(pcap_if_t **alldevsp, char *errbuff) is a built-in function used to get a list of available devices that can be opened with pcap_open_live() (as will be discussed shortly) and store the first element in the alldevsp variable. We will see in a moment how the code will access the entire list from the alldevsp reference. The function returns 0 upon success and -1 upon failure. When it fails it expects to store the error that occurred in the provided buffer errbuff.

Note that there may be network devices that cannot be opened with pcap_open_live() by the process calling pcap_findalldevs(), because, for example, that process might not have sufficient privileges to open them for capturing; if so, those devices will not appear on the list.

This if statement is checking if the function call was not successful, in which case it cannot access the returned devices' list, so it just prints out the error that occurred and exits the program. However, if the call was successful it will move on to the next code block.

```
for (d = alldevs; d != NULL; d = d->next) {
    printf("%d. %s\n", d->next, (d->description) ? d->description : "No description available");
}
```

The d variable defined earlier comes into play here. It is used as an iterator over the list of devices. Since, as mentioned, alldevs and d are of type pcap_if_t struct *, they have certain fields within that struct that can be accessed when needed. 2 fields are used in this snippet: next and

description. next is used to get the next device stored in the list as to iterate over all devices, and description is used to display a human-readable description of the device. So essentially, all this loop is doing is going over all devices and printing their descriptions, that is if they have

```
if ((adhandle = pcap_open_live(alldevs->name, 65536, 1, 1000, errbuf)) == NULL) {
    printf("Unable to open the adapter. %s is not supported by pcap\n", alldevs->name);
    return 1;
}
```

one. If one doesn't, "No description available" is printed instead.

This if statement is the start of capturing. It checks if the function call to pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *ebuf) failed execution, in which case it will return NULL and the program will print out the error and exit.
pcap_open_live(const char *device, int snaplen, int promisc, int to_ms, char *ebuf) is used to obtain a packet capture descriptor to look at packets on the network. device is a string that specifies the network device to open; in this case it is given the name of the first element in the devices' list. snaplen specifies the maximum number of bytes to capture. If this value is less than the size of a packet that is captured, only the first snaplen bytes of that packet will be captured and provided as packet data. A value of 65535 should be sufficient, on most if not all networks, to capture all the data available from the packet. promisc specifies if the interface is to be put into promiscuous mode. (Note that even if this parameter is false, the interface could well be in promiscuous mode for some other reason.) In promiscuous mode, a network device, such as an adapter on a host system, can intercept and read in its entirety each network packet that arrives. to_ms specifies the read timeout in milliseconds. The read timeout is used to arrange that the read not necessarily return immediately when a packet is seen, but that it waits for some amount of time to allow more packets to arrive and to read multiple packets from the OS kernel in one operation. errbuf is used to return error or warning text. It will be set to error text if pcap_open_live() fails and returns NULL.

```
pcap_loop(adhandle, 0, packet_handler, NULL);

pcap_freealldevs(alldevs);
return 0;
```

pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user) is the function used to collect a group of packets and handle them using the provided handler. It takes 4 arguments: a pcap_t reference that refers to the descriptor we got in the last code block, cnt that specifies if the loop should iterate indefinitely or not; if cnt was negative, the loop would run forever, pcap_handler which is the callback function that will handle the captured packets, and finally an unsigned char variable that contains the state of the capture session which was provided as NULL in this case. In summary, this loop takes the captured packets, iterates over each of them, and sends each one to the callback function to be handled accordingly. This callback function is user-defined and will be discussed shortly.

pcap_freealldevs(pcap_if_t alldevsp) is used to free the list allocated by pcap_findalldevs(). It takes the list to-be-freed as an argument.

```
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt_data) {
    printf("Packet captured: %d bytes\n", header->len);
}
```

We will finally look at the callback function mentioned earlier. This callback function should always follow a specific function signature. This signature is the following: funcname(u_char *user, const struct pcap_pkthdr *pkt_header, const u_char *pkt_data).

The user parameter refers to the same user parameter seen in the pcap_loop() function. pkt_header is the header associated by the capture driver to the packet. It is NOT a protocol header. pkt_data points to the data of the packet, including the protocol headers.

All this function does with the given packet is print out its length, which is provided as a field in the pcap_pkthdr struct.

2) Obfuscation with Control Flow Flattening.c

Obfuscation is the process of making code harder to follow or analyse. This could be done in several ways and for several reasons. Malicious actors often obfuscate their malicious code to make it harder for analysts to understand it and mitigate its damages.

One way to do obfuscation is with control-flow flattening. The main principle behind control-flow flattening involves transforming the original control structures, such as loops and conditionals, into an equivalent form that involves jumps and branches to various locations within the code. There could be different implementations of CFF, but the one we see the most is to transform the control flow structure into a switch-case based state machine.

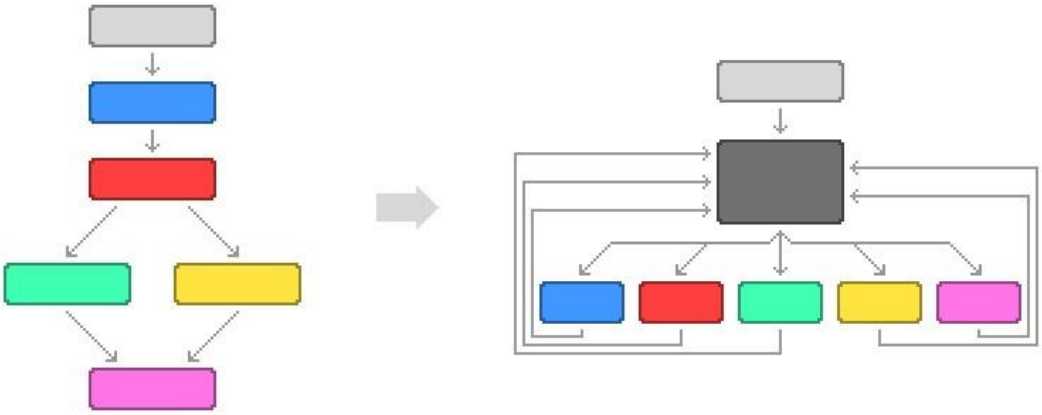
```c
int main() {
    obfuscated_code();
    return 0;
}
```

The main function of this code is simple enough. All it does is call on the obfuscated_code() function, which has the actual logic of the control-flow flattening.

```c
void obfuscated_code() {
    int x = 0;
    int y = 5;
    int state = 0;

    while (state < 4) {
        switch (state) {
            case 0:
                x = y + 2;
                state = 1;
                break;
            case 1:
                y = x - 3;
                state = 2;
                break;
            case 2:
                x = x + y;
                state = 3;
                break;
            case 3:
                printf("x: %d, y: %d\n", x, y);
                state = 4;
                break;
        }
    }
}
```

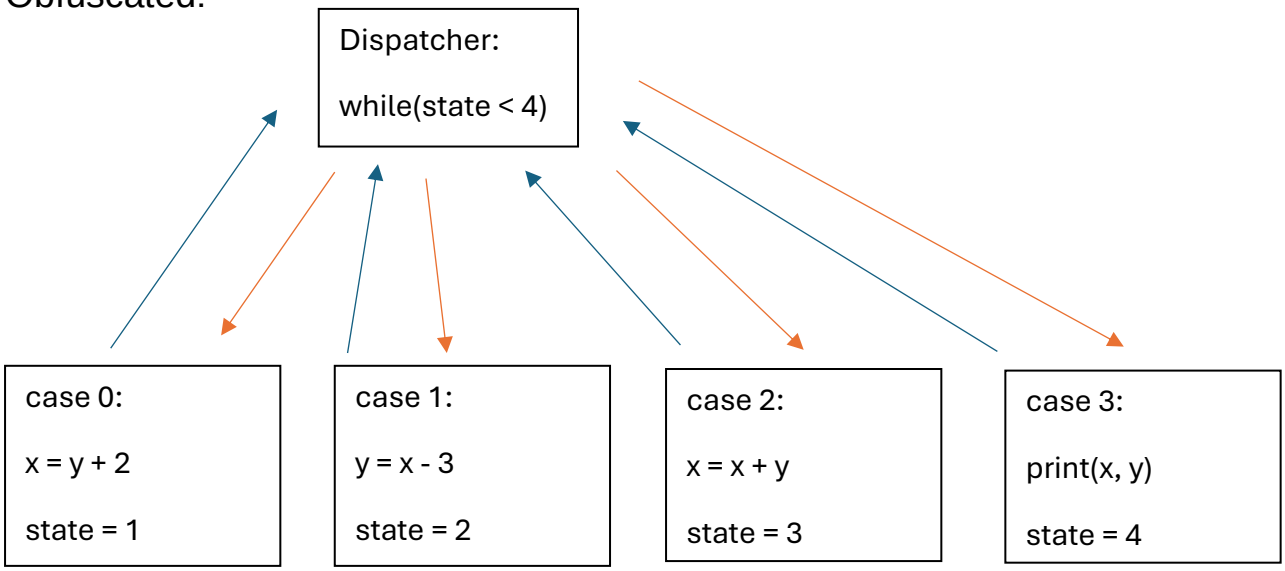
To understand this function, we will use a diagram that showcases the way CFF works.



During the normal flow, the code would reach some type of control structure, such as a conditional statement, and would go with one of 2 or more branches, like the diagram on the left. However, that is straightforward to understand if someone were to analyse this code. On the other hand, when CFF is applied, instead of going into the appropriate branch and continuing regularly with the code, the flow of a function is broken down into blocks. These blocks are placed in some sort of selective structure (like a switch statement) and the selection is wrapped in a loop which will act as a dispatcher that all the blocks will redirect to upon completion. This is demonstrated in the figure on the right, where the dark grey block is the dispatcher and each of the coloured blocks below are the blocks of code in the original function that reroute to the dispatcher when they are done executing. The dispatcher then repeats the process of selection between blocks based on a variable that updates in every block accordingly to achieve the original flow. This makes the flow more difficult to comprehend for the analyst but achieves the same result in the end.

To demonstrate this further, let us try to break down the given code snippet and see what it would like if it were not obfuscated.

Obfuscated:



Not obfuscated:

```
x = y + 2
y = x – 3
x = x + y
print(x, y)
```

3) Rootkit.c

A rootkit is a type of malware that tries to gain access to high-privilege data and operations that only a “root” user can access. It is also dangerous because it hides its existence or the existence of other malware in a system. They hide their presence by modifying the behaviour of core parts of an operating system.

This c code is a simple mock rootkit. Let’s analyse it.

```c
#include <stdio.h>
#include <windows.h>
#include <tlhelp32.h>
```

Firstly, we will look at the libraries used. windows.h is the primary C library for accessing the Win32 API. tlhelp32.h is a tool helper library used with windows.h to facilitate functionalities in windows.

```c
int main() {
    hide_process("notepad.exe");
    return 0;
}
```

The main function of this code is very simple; it only calls a hide_process().

```c
void hide_process(const char *process_name) {
    HANDLE hSnapshot;
    PROCESSENTRY32 pe32;
    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
```

hide_process(const char *process_name) is the function with the rootkit logic. It takes in a process name string as an argument. This process name is the name of the process that the rootkit will want to hide its existence (like a malware). A variable of type HANDLE is declared and will be used presently. A HANDLE is a pointer that points to an "object" that represents a resource - often an OS resource. It is often implemented as a void *.

Another variable of type PROCESSENTRY32 is declared. PROCESSENTRY32 is a struct in the windows.h library that is used to refer to an element of a list of processes that were running during a system snapshot.

Finally in this code snippet, we use the HANDLE variable we declared by giving it the resulting value of CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID). This function returns a snapshot of all processes as well as the heaps, modules, and threads used by these processes. It takes 2 arguments: a flag that determines the specifications of the snapshot (in this code, we gave it the TH32CS_SNAPMODULE32 flag which indicates the snapshot should include all processes in the system) and a process ID of the process that will be included in the snapshot (if given as 0, it means current process).

Note that the processID parameter is ignored when used with the TH32CS_SNAPMODULE32 flag. It will just capture all the processes.

```
if (hSnapshot == INVALID_HANDLE_VALUE) {
    printf("Error creating snapshot.\n");
    return;
}
```

If the CreateToolhelp32Snapshot() function succeeds, it returns an open handle to the specified snapshot. If it fails, it returns INVALID_HANDLE_VALUE. In this if statement, we're checking if the function failed, we print out a failure message and exit.

If it succeeded, we move to the next code snippet.

```
pe32.dwSize = sizeof(PROCESSENTRY32);

if (!Process32First(hSnapshot, &pe32)) {
    CloseHandle(hSnapshot);
    printf("Error getting first process.\n");
    return;
}
```

pe32.dwSize refers to the size of the struct. It should be initialized to sizeof(PROCESSENTRY32) before proceeding. If this was not done, calling Process32First() will fail. In the following if statement we check exactly that: if Process32First(HANDLE hSnapshot, LPPROCESSENTRY32 lppe) did not succeed, in which case the HANDLE for the current snapshot is closed, an error message is printed out, and the program is exited. Process32First() is used to get information on the first process in a system snapshot. It takes 2 arguments a HANDLE for the snapshot in question and a PROCESSENTRY32 struct reference (in this case, the address of pe32 is given). It returns true if the first entry of the process list has been copied to pe32 or false otherwise.

```
do {
    if (strcmp(pe32.szExeFile, process_name) == 0) {
        HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pe32.th32ProcessID);
        if (hProcess) {
            printf("Hiding process: %s (PID: %d)\n", pe32.szExeFile, pe32.th32ProcessID);
            TerminateProcess(hProcess, 0);
            CloseHandle(hProcess);
        }
    }
} while (Process32Next(hSnapshot, &pe32));

CloseHandle(hSnapshot);
```

This loop is where the hiding logic is done. szExeFile is a member of the PROCESSENTRY32 struct that holds the name of the executable file for the process. strcmp compares between pe32.szExeFile and the given process_name as a function parameter. If they are equal (strcmp returns 0), we found our process, which we will then attempt to hide.

We first get a handle on that process by using OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessId). This function takes 3 arguments: access to the process object (which was given as PROCESS_ALL_ACCESS which grants all possible access rights), a bool to specify if children processes of the current process will inherit this handle or not, and the identifier of the process to be opened.

OpenProcess() returns NULL if the call fails so this is what was checked in the if statement. If it’s not NULL, then a handle was returned, and we can proceed.

A hiding process message is printed with process name and ID. Then the process is terminated by calling TerminateProcess(). So we don't actually *hide* the process, we kill it. TerminateProcess(HANDLE hProcess, UINT uExitCode) takes 2 arguments: a handle of the process to be terminated, and the exit code to be used by the process and threads terminated as a result of this call.

Lastly, we call CloseHandle() to close the handle to that process. We keep looping over all processes by using the Process32Next() function. Process32Next() works just like Process32First() but instead deals with the next process of a system snapshot. It moves with respect to the current held process. It can be used to iterate over all processes in a snapshot and moving each of them to pe32 to process it. As long as there are processes left, the loop will keep iterating. As soon as all processes are checked, the loop is terminated, and the handle closed.

4) Steganography-Based Data Exfiltration.c

Steganography is the process of hiding confidential information like IP addresses or passwords in files such as images or music files. Steganography can be done in text, pictures, audios, videos, and many other types of files. Steganography can also be used with cryptography to increase the security.

Data exfiltration – also known as data extrusion or data exportation – is the act of stealing data and transferring it without authorization.

This code is a basic implementation of a steganography-based data exfiltration, which is using steganography to discretely steal data and transfer it within another legitimate file.

```
int main() {
    hide_data("image.bmp", "secret.txt");
    return 0;
}
```

Inside this main function, we call the hide_data() user-defined function to do the actual program logic.

```
void hide_data(const char *image, const char *data) {
    FILE *img = fopen(image, "rb+");
    FILE *dat = fopen(data, "rb");
    if (!img || !dat) {
        printf("Error opening files.\n");
        return;
    }
```

hide_data(const char *image, const char *data) takes 2 arguments: the name of the image file that we will embed our stolen data into and the name of the file of the actual data to be hidden. It then starts defining some variables, namely 2 file pointers for each file that we took as input. We open those files with fopen(). fopen(const char *file_name, const char *mode_of_operation) requires the name of the file to be opened and the mode with which it is opened. The mode depends on what operations will be done on the content of the file (ex: read, write, append, etc.). In this case, the image file is opened with a binary read/write, which basically means we read from or write bytes into the file's location. The data file is opened with a binary read. If fopen() failed to find the desired file, it returns NULL. So, the if condition in this case is checking if any of the files is NULL, and if so, it prints an error message out and exits the program.

```
fseek(img, 0, SEEK_END);
size_t img_size = ftell(img);
fseek(img, 0, SEEK_SET);
```

fseek(FILE *stream, long int offset, int whence) takes in a file stream, an offset, and a starting position (whence), and moves the pointer (or cursor) of the file from the starting position with offset number of steps. The starting position can be set as SEEK_SET which denotes the beginning of the file, SEEK_CUR which denotes the current location in the file, or SEEK_END which denotes the end of the file.

In the first line here, we start at the end of the file and stay there (offset of 0). We then use ftell() to *tell* us the current file position. ftell(FILE *stream) takes the file stream that we want to know the position of. If we

do these 2 lines together, we can essentially figure out the size of the image (in number of bytes), since we started at the end and we got that the position at the end of the file is whatever ftell() outputted, which is a long int denoting the number of bytes up to that position.

After getting the image size, we can reset the file position by using fseek() again, this time with the SEEK_SET as the starting position.

```
unsigned char *buffer = malloc(img_size);
fread(buffer, 1, img_size, img);

fseek(img, 0, SEEK_SET);
fwrite(buffer, 1, img_size, img);
```

In this code block, malloc(int size) is used to dynamically allocate memory to a block of data. In our example, we need the buffer to hold up to the image size measured in the previous step. fread(void *ptr, size_t size, size_t nmemb, FILE *stream) is used here to read the image bytes into the buffer. fread() takes in a location in which the read data would be stored, the size of each element to be read in bytes, the number of elements to be read, and finally the input stream to read from.

After reading all the image bytes into the buffer, we reset its location using fseek() as mentioned before. fwrite() is exactly like fread but the ptr is written into the stream instead of the opposite.

```
unsigned char ch;
while (fread(&ch, 1, 1, dat)) {
    fwrite(&ch, 1, 1, img);
}

free(buffer);
fclose(img);
fclose(dat);
```

Here, we define a char that we will use to carry a byte at a time from the data file to the image file. We do that in a while loop. The while loop keeps reading using fread() 1 byte from the data file, writing it into ch,

and checking if fread() returns 0 or not. If it returned 0, this means either its done reading or there is no data to read to begin with. If not, the loop body will execute. All the loop body does is fwrite() whatever is in ch into the image file. After the loop is done, we need to free the buffer that we allocated so we're not left with a memory leak, and we need to close both files.

5) Keylogger.c

A keylogger is a type of malware that logs any keystrokes that are done on the victim's device. Anything typed will be recorded and sent to the attacker to analyse and find useful confidential data. This code is an implementation of this idea with the use of hooks.

A hook is a mechanism by which an application can intercept events, such as messages, mouse actions, and keystrokes. A function that intercepts a particular type of event is known as a hook procedure.

```
int main() {
    HANDLE hThread = CreateThread(NULL, 0, KeyloggerThread, NULL, 0, NULL);
    if (hThread) {
        WaitForSingleObject(hThread, INFINITE);
        CloseHandle(hThread);
    }
    return 0;
}
```

We will start this program by defining a thread handle variable. This thread is created by the CreateThread() function. CreateThread() takes in 6 arguments, some of which are optional. The 1st argument is a flag to specify whether the thread is to be inherited by a child process. It is an optional argument so for now we place it as NULL. The 2nd argument is the initial size of the stack in bytes. If it's 0, the new thread uses the default size for the executable. The 3rd argument is a pointer to the function to be executed by the thread. In this case it's our user-defined KeyloggerThread function. The 4th argument is a pointer to a variable to be passed to the thread if needed. This argument is optional. The 5th argument is a flag that controls the creation of a thread. In our code, the flag is set to 0, which means the thread will run immediately after

creation. The final argument is an optional variable to carry the thread identifier. If it is NULL, the thread ID is not returned.

Next, the if statement checks if a handle was created successfully (not NULL). If it was, WaitForSingleObject() is called. This function is used to make the calling thread (in this case, the main thread) wait for the input thread (hThread) to finish executing or for the timeout to occur (the timeout in our case if INFINITE). After the main is done waiting for threads, it can close the thread handle and quit.

Next, we will discuss the KeyloggerThread function we mentioned earlier.

```
DWORD WINAPI KeyloggerThread(LPVOID lpParam) {
    HINSTANCE hInstance = GetModuleHandle(NULL);
    hHook = SetWindowsHookEx(WH_KEYBOARD_LL, KeyboardProc, hInstance, 0);
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    UnhookWindowsHookEx(hHook);
    return 0;
}
```

The WINAPI keyword in the function signature is used to define functions with the __stdcall calling convention. Most functions in the Windows API are declared using WINAPI. KeyloggerThread(LPVOID lpParam) takes in any variables passed in during the CreateThread() function call as input to a thread. If none were passed, this argument will be empty.

HINSTANCE is an instance's handle. It is considered the base address of the module in memory. Using GetModuleHandle(NULL) retrieves a module handle for the specified module. Giving it NULL will return a handle to the file used to create the calling process (.exe file). We use SetWindowsHookEx() to install the defined hook procedure into the hook we created earlier. It takes 4 arguments: the type of hook procedure to be installed (in our case, we used WH_KEYBOARD_LL which installs a hook procedure that monitors low-level keyboard input events which is what we want to monitor for keystrokes), a pointer to the hook procedure, a handle to the file containing the hook procedure, and a

thread identifier with which the hook procedure will be associated (if this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread).

In the while loop, we use GetMessage() to retrieve a message from the calling thread's message queue. GetMessage() blocks until a message is posted before returning. It takes a pointer to an MSG structure that receives message information from the thread's message queue, a handle to the window whose messages are to be retrieved (the window must belong to the current thread), the integer value of the lowest message value to be retrieved, and the integer value of the highest message value to be retrieved.

2 function calls are done the loop body: TranslateMessage() and DispatchMessage(). TranslateMessage() translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the GetMessage(). DispatchMessage() *dispatches* a message to a window procedure.

Finally, when the loop goes over all available messages, the current thread unhooks the procedure from the hook variable by using UnhookWindowsHookEx().

The last thing we need to discuss in this code is the hook procedure KeyboardProc().

```
LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam) {
    if (nCode == HC_ACTION) {
        if (wParam == WM_KEYDOWN) {
            KBDLLHOOKSTRUCT *p = (KBDLLHOOKSTRUCT *)lParam;
            FILE *file = fopen("keylog.txt", "a+");
            if (file) {
                fprintf(file, "%c", p->vkCode);
                fclose(file);
            }
        }
    }
    return CallNextHookEx(hHook, nCode, wParam, lParam);
}
```

This is the function that will handle the actual logging whenever a keyboard event occurs. This is guaranteed by the hook that will run this procedure as soon as the event takes place.

Both WPARAM and LPARAM are message parameters defined in the WinDef.h. The first argument (ncode) is a code the hook procedure uses to determine how to process the message. If it is given as HC_ACTION, the wParam and lParam parameters contain information about a keystroke message and should be handled accordingly.

We compare wParam with WM_KEYDOWN to check if a non-system key is pressed. A non-system key is a key that is pressed when the ALT key is not pressed. If that is the case, we define a variable of type KBDLLHOOKSTRUCT. This variable is a reference to a structure with information about a low-level keyboard input event.

Next, we open the file keylog.txt to append into it any new keys detected. To get the pressed key, we can rely on virtual codes, which are built-in values that maps every key on a standard keyboard to a numeric value. The code must be a value in the range 1 to 254. The KBDLLHOOKSTRUCT struct has a field called vkcode where in any event, the pressed key's virtual code is stored. So, we use this field to get the pressed key and store it in the keylog.txt file. Finally, we close the file and call CallNextHookEx(). CallNextHookEx() passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook information.

6) Function Pointer Obfuscation.c

A function pointer is a type of pointer that holds the address of a function. It could be used to provide a reference to a type of function without specifying a specific function. For instance, if we need a function that takes in one int argument and returns an int value, we can define a pointer to such a function in the following manner:

```
int (*func)(int)
```

This notation defines a function pointer to a function that returns an int with one int parameter without defining an actual function. This pointer can then be used to reference different functions of this behaviour throughout the program. The first int is for the return type, the () around the *func is used to denote that this is a pointer to a function and the (int) is the parameter list.

Function pointers can also be used for obfuscation. This technique is an example of what's called control-flow obfuscation. The idea is to use an indirect function call so that the function address must be computed first and then called. This will cause the function addresses to be more unpredictable and harder to follow during analysis.

Let's analyse how this is done in this simple c code.

```c
int main() {
    void (*func_ptr)();
    char choice;

    printf("Enter 1, 2, or 3: ");
    scanf(" %c", &choice);
```

Firstly, we define the function pointer as discussed earlier: void for the return type, (*func_ptr) to denote that it's a function pointer, and () for the parameter list, which in this case is empty. We also define a char choice to use for the user input. The program starts by printing on screen a prompt for the user to choose between 1, 2, or 3 to determine which function will be used. We get the user response by using scanf().

```c
if (choice == '1') {
    func_ptr = function1;
} else if (choice == '2') {
    func_ptr = function2;
} else if (choice == '3') {
    func_ptr = function3;
} else {
    printf("Invalid choice\n");
    return 1;
}

func_ptr();
```

Depending on the user input, we will choose the function that the pointer will be referencing (we will see the different functions shortly). If the user entered 1, func_ptr will point to the 1st function and so on. If they entered something else entirely, we print a usage error message and exit the program. After the program determines which function is meant to be called, it does the actual calling by using the function pointer and giving it all required parameters (in this case, none). -> func_ptr()

The 3 different functions are shown below.

```c
void function1() {
    printf("This is function 1\n");
}

void function2() {
    printf("This is function 2\n");
}

void function3() {
    printf("This is function 3\n");
}
```

Very simple, they all just print a message to show which function was called.

3- Applied Reverse Engineering: The Stack (summarized)

Credits to: Daax in https://revers.engineering/applied-re-the-stack/
Summarized by: Youmna Elmezayen

- What is the stack?

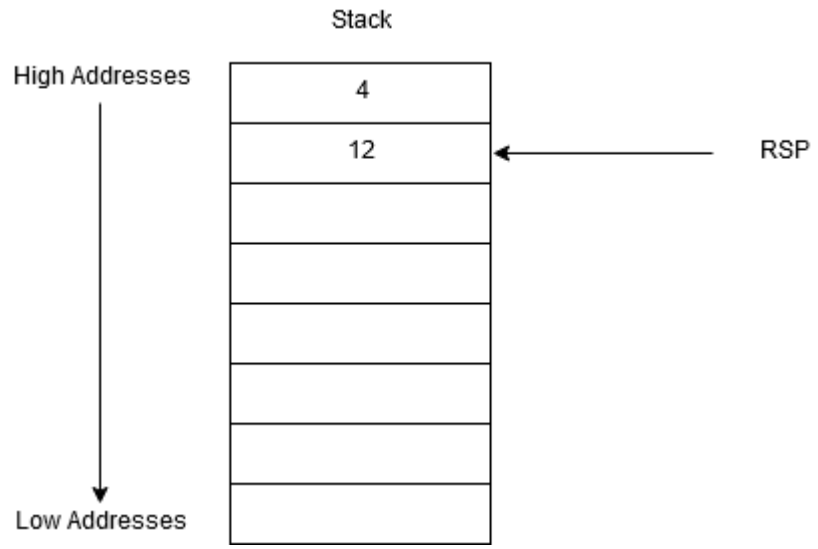
A stack is a part of the memory that is used in a linear format. It is accessed based on the last-in-first-out principle, which just means that the last element placed in the stack is the first element to be removed and/or processed in the stack.

The stack deals with certain operations based on the LIFO principle:

1- The only accessible item in the stack is the **top** element.
2- If something is to be added to the stack, it is **pushed** on top of the stack.
3- If something is to be removed from the stack, it is **popped** from the top of the stack.
4- The stack can be checked for **emptiness**.

- Stack Layout.

The stack is managed by 2 registers: SS and RSP. SS, a segment register, is the stack segment, which is what the processor references when trying to access the stack for any operation. RSP is the stack pointer, which ideally should always point to the top of the stack. This is because the stack grows downward (lower addresses) when data is added to it and shrinks upward (higher addresses) when data is removed from it. So essentially, it is anchored at the top, so the pointer is needed to identify what the top item in the stack is. Sometimes, by means of trickery (like obfuscation), the stack pointer is not pointing to the top of the stack but is used for some other generic purpose.



As shown in the figure above, the stack grew to the lower address. 4 was pushed first so it was the top of the stack momentarily and had RSP point to its location. Then, 12 was pushed and was the new top of the stack, which moved the RSP to its location instead.

The RSP is moved by the processor. Whenever a new item is pushed onto the stack, the processor decrements one memory location from the stack pointer for every item added. To access an element in the stack, we can either use an offset from RSP or pop from the stack. Let's discuss the offset method first.

Since the offset is taken from RSP, the top element (say 12 in our example) would have the offset of 0 from the RSP because the pointer is pointing right at it. But we must reference with respect to SS as mentioned to access that location. For instance: mov rbx, qword ptr **ss:[rsp]**

ss:[rsp] means that we access the element at RSP in the stack segment SS. What this code does is it copies whatever is stored in the location RSP + 0 of the stack segment into RBX. If we want to do the same for the other element (4), we give the offset between memory locations. Since the memory is byte-addressed, we add +8 to RSP to locate 4.

The second method is the pop method. This is more straightforward than the first method. One might be favoured over the other depending on the situation at hand. In the pop method, we do just that: pop items off the

stack, one at a time as needed. So, to do the same example we did before: pop rbx

This line would result in whatever is at the top of the stack be removed and placed into the given operand, namely RBX. If we need the second element as well, we just pop again.

As mentioned before, the RSP is decremented when adding items. On the other hand, when removing items, the RSP is incremented until it reaches the highest address in the stack frame given.

Note: A program or operating system can setup many stacks. The limitation is based on the maximum number of segments and available physical memory. However, only one stack is available at a time regardless of how many exist, and the current stack is the stack referenced by the SS register.

Next, we will discuss other instructions that operate on the stack apart from push and pop: call and ret.

- The call instruction.

Note: All segment registers are zero based except for the GS and FS segment registers. The FS and GS segment registers can still have non-zero base addresses because they may be used for critical operating system structures.

The call instruction has 4 formats or types: near call, far call, inter-privilege-level far call, and task switch. We will focus on the near call.

The near call is relative to the next instruction address. The difference between a near call and a far call is that the near call doesn't modify the code segment (CS) in use, meaning it doesn't exit the code frame that is currently executing. So, the code it is calling is *relatively* close. 64-bit operating systems use what's called the flat memory model where all memory can be accessed from the same segment, so a far call has no use in this case. The two sub-types of near calls are near relative calls (E8) and near absolute (FF). E8 and FF are their respective opcodes.

Relative means that the call target address will be relative to the address of the next instruction. When calculating the address, we should be aware of the endianness of the system we're dealing with. There are 2 types of endianness: little and big endian. Little endian places the least significant byte first, i.e. in the lower address. Big endian does the opposite. When taking an address in a 64-bit intel system, we should maintain the sign of the address, meaning that if it doesn't take all 64 bits, it should be sign-extended until it does. We can then use this address to calculate the relative target address we need.

Ex:

Since this has (E8) as its opcode, we can infer that it is a near relative call. So, to get the target address, given that we're dealing with a little-endian system, (FF A4 62 96) is the relative address we need to add to the next instruction address to get to the desired function. However, since this address starts with (F), that means it has a negative sign. Therefore, we need to sign extend it to fit 64 bits: (FF FF FF FF FF A4 62 96).

Another way to differentiate between a relative and an absolute call is that relative calls are generally direct calls, meaning they specify the target address (relatively) right there in the instruction. Absolute calls, on the other hand, are indirect calls, meaning they use a register or a memory location that holds the absolute target address.

Ex: call some_func -> direct: relative

call [rbx] -> indirect: absolute

- Call stack operations.

What happens when a call instruction is encountered, in a nutshell?

The RIP (instruction pointer) register is pushed onto the stack and the RSP decremented. We need to store the RIP somewhere so that when we jump to the target address, we know where to come back. This new stack value is used as the return-instruction pointer (as will be

mentioned shortly). The processor then uses the given address (relative or absolute) to jump to the desired target location. If the address was relative, it will be added to RIP, and if absolute, it will be used directly by storing it as the new value for RIP. After the callee function is done executing, we will need to get back to the caller code to continue the normal flow. To do this, we use the RET instruction at the end of the callee function.

- The return instruction.

The RET instruction is used by the callee function to return the program flow to the location after the call happened. This is done by popping the return address that was stored in the stack by the CALL instruction earlier and placing it back into RIP. The return instruction has a few different opcodes, most of the time in 64-bit targets we'll just see the C3 opcode. However, we may encounter C2 as well. Both operate in their own way. Of course, after popping the return address from the stack, the RSP should be incremented.

- Calling Conventions and the Microsoft ABI

A calling convention is the way a function should be used, i.e. the way it receives arguments, the way it returns its return values, the way it can be invoked, and the way it creates and manages its stack frames. It's the way the function call in the compiled language is converted into assembly. There are several calling conventions, but we will focus on fastcall. It's important to note that this calling convention is not standardized across all compilers, some may use different methods of passing arguments to the function or managing the stack and frames.

An application binary interface, or ABI, is an interface between a program and the platform it is being run on. ABI provides a set of conventions and details the program will need to know in order to work correctly with the system. These conventions include data types and their assigned sizes, object file format, and of course calling conventions. What we're going to be considering from the ABI is the layout of the stack frame for a function call, how arguments are passed,

and how stack cleanup is performed. We will focus on x64 ABI, which uses a four-register fastcall calling convention.

- Fast-call calling convention.

This convention uses four general-purpose registers to pass the arguments to the callee function. These registers are RCX, RDX, R8 and R9, in that order. So for instance, a function func(a, b) will have (a) passed in RCX and (b) in RDX in its assembly listing. For the return value, this convention uses RAX.

Note: The calling convention can also use these registers' 32-bit counterparts in case the arguments did not need to use the 64-bit ones.

These registers are typically used with int types. If we needed to pass floating point arguments or structs or anything else, we would need something else. Any argument that doesn't fit into a supported size 1, 2, 4, or 8 bytes must be passed by reference, this is because an argument is never split across multiple registers. All floating-point arguments and operations are done using the XMM registers. They are similar to the general-purpose registers we're used to. There are 16 of them and they are named by their indices (XMM0-XMM15).

Prior to executing a call instruction and as part of the convention, it is the job of the caller to allocate space on the stack for the callee to save the registers used to pass arguments. This space that's allocated by the caller is known as the shadow store. The space allocated is strictly the maximum size supported (8 bytes) times the number of registers used to pass arguments (4). This is seen in any assembly listing as (sub RSP, some_num) where some_num is the required space in the stack. Typically, some_num, if we used the maximum values mentioned above, would be 8 * 4 + 8 = 40. The +8 was for aligning the stack address with a 16-byte boundary. This means that the address of the top of the stack must be divisible by 16. Well 40 is not a multiple of 16… Yes but this 40 doesn't account for the 8 bytes of return address that will be added by default upon calling. So, the overall stack space allocated is 48, which is divisible by 16.

To reclaim this allocation when the function finishes execution we use (add rsp, some_num) to shrink the stack up to its original state prior to the call.

- Stack frames.

A stack frame is essentially just a collection of data that gets stored in the stack. In our context, we're talking about a call stack frame which represents a function call and its argument data. An important distinction is that the shadow store allocated is not part of the call stack frame. The call stack frame starts with the return address being pushed onto the stack first, then storage of the base pointer, and space for local variables is allocated. In some instances when a function is small enough and no locals are used we wind up not needing a stack frame.

What happens first is the processor pushes a general-purpose register onto the stack, RBP. This register is referred to as the base pointer, and its purpose is normally for use in stack frames and addressing local variables in a function. It's pushing this register onto the stack to preserve its value, most pushes you find preceding actual function code are used to preserve register values.

The code at the start of every function where the storage of RBP, local variables, and the shadow store occurs is called a function prologue. Any function that allocates stack space, calls other functions, and preserves registers then it will have a prologue. The epilogue is the sequence of instructions that cleanup any stack allocations and restore preserved registers prior to returning.

Inside the allocated stack frame, we can use RBP to index through the different locations to access different variables. Positive offsets from RBP access arguments passed on the stack. Negative offsets from RBP access local variables.

According to fastcall calling convention, if the number of arguments is greater than 4 the 5th argument and on is passed on the stack. Arguments not of size 1, 2, 4, or 8-bytes are passed by reference. To load these arguments in for processing, we can use LEA instruction (load effective address), which uses the argument's location in memory to

load it and use it in assembly. The destination is always a general-purpose register.