CITS 3004 Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

Lab 9: Key Logging and Ransomware

Task 1: Key Loggers

Written by Jin Hong

An example code `keylog.py` (the one you have seen in the lecture) is provided for you. You have to first install `pynput` library. Make sure that you are installing this for Python 3 (continuing from last week's lab would install the library for Python 3.6. check before proceeding).

```
# sudo pip install pynput
```

If you get an error, you will likely have to install Python dev (replace `python3` with appropriate version).

```
# sudo apt-get install python3-dev
```

1A) USING KEY LOGGER

Read the `README` file, which provides the brief overview of how to use the provided `keylog.py` code. You can try opening various text editors and web browsers to see if the key logger is able to capture the keys pressed in various environment.

1B) MODIFYING KEY LOGGER

The key logger currently outputs the captured key presses onto the terminal only. Your task is to do the following:

1. Record the keys pressed, and store them onto a file using the current date (i.e., YEAR-MONTH-DAY.txt).
2. When the user starts the machine, it should run automatically on the start-up (optional).

You can also setup a script to automatically send the key logged data to a remote computer. However, we will not cover this in the lab (you can explore this on your own time).

NOTE: it is illegal to distribute key loggers in many countries, including Australia.

Answer Q1 ~ Q2 on LMS

CITS 3004 Cybersecurity

Task 2: Ransomware

Written by Jin Hong

For this task, ensure that you follow the instructions carefully, otherwise you may damage your VM (or computer). Ensure to create backup for your files.

You are provided with the Python code `ransomware.py` and the image file to test with (`original.jpg`). First, read the instructions written in the code. The code will work with Python 3. To run the code, you need the `pycryptodome` package installed to use the Crypto library.

```
# sudo pip install pycryptodome
```

Note: make sure that your pip command is linked to Python 3.

Currently, it will try to lock any extensions you specify inside the `/home/seed/testing` folder. You may change the working folder by editing the path in the main function.

Currently, the encryption used is AES_CBC. The key is set to 16 bytes (i.e., 16 characters) long, and the same for the initial vector (i.e., 16 bytes long) as follows.

```
# HARDCODED_KEY = b'veryweakpassword'
# INITIALVECTOR = b'1111111111111111'
```

Once you specified the extensions to encrypt and the folder location, we are ready to test the code. In the terminal, we can execute (assuming `python` command is linked to Python 3):

Encryption:

```
# python ransomware.py
```

Decryption:

```
# python ransomware.py -d
```

2A) TESTING THE RANSOMWARE

Place the image file original.jpg inside the folder you are conducting the ransomware encryption. Try encryption and then decryption to check that you can get the original message back.

Try putting files with different extensions and do the same as above. Try to specify a few extensions, and you should notice that files with unspecified extensions are not altered.

2B) MODIFYING THE ENCRYPTION MODE

Instead of the CBC mode, your task is to modify the code to use the CTR (counter) mode. Look up the pycryptodome manual online to see how to implement this.

Answer Q3 ~ Q7 on LMS

CITS 3004 Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

Task 3: Reverse Engineering Malware

Written by Alex Brown

WARNING: Do not complete this lab task on your host machine and only use the SEED Ubuntu VM! This lab task requires investigating ransomware that can encrypt your files.

Note: You will need to use the SEED Ubuntu VM for this lab task.

Malware analysts and security researchers use reverse engineering to investigate how malware function and if possible, try find a method to reverse the damage it causes.

One of the most interesting stories about reverse engineering is the story about the ransomware WannaCry. WannaCry propagated across the internet using the EternalBlue exploit, that was developed by the NSA and leaked by an anonymous hacker group called the Shadow Brokers. It was devastating computers across the world, until Marcus Hitchins reverse engineered the ransomware. Marcus found an un-registered domain within the malware and decided to register the domain. Consequently, he inadvertently found the kill switch for the ransomware, stopping one of the largest cyber-attacks known to this day.

In this lab, we will be using Ghidra to reverse engineer a newly discovered ransomware called `free_bitcoin`, specifically designed to target SEED Ubuntu VM users. It was reported that a victim tried to get free bitcoin by running the program, but instead encrypted everything in the working directory. We will try and reverse engineer the malware to retrieve the encryption key used to encrypt the victim's files.

Ideally, we would use a reverse engineering suite of tools such as Ghidra, but since it requires 4 GB of RAM we will use `strings`, `objdump` and `gdb-peda` to reverse engineer the ransomware. If you are interested and can allocate sufficient memory to your VM, you can download Ghidra from https://ghidra-sre.org/ and follow the installation guide here https://ghidra-sre.org/InstallationGuide.html.

3A) THE STRINGS COMMAND

We will begin our analysis of the ransomware by running the `strings` command on the binary. Below is a picture of the output of strings being piped into `grep` to highlight some key library functions and hardcoded strings that were found inside the ransomware.

```
[08/02/20]seed@VM:~/.../analysis$ strings free_bitcoin | grep "EVP\|1234567890abcdef"
EVP_DecryptFinal_ex
EVP_aes_128_cbc
EVP_DecryptInit_ex
EVP_EncryptFinal_ex
EVP_CIPHER_CTX_init
EVP_DecryptUpdate
EVP_EncryptInit_ex
EVP_EncryptUpdate
1234567890abcdef
EVP_CIPHER_CTX_init@@OPENSSL_1.0.0
EVP_DecryptInit_ex@@OPENSSL_1.0.0
EVP_EncryptFinal_ex@@OPENSSL_1.0.0
EVP_EncryptUpdate@@OPENSSL_1.0.0
EVP_aes_128_cbc@@OPENSSL_1.0.0
EVP_DecryptFinal_ex@@OPENSSL_1.0.0
EVP_DecryptUpdate@@OPENSSL_1.0.0
EVP_EncryptInit_ex@@OPENSSL_1.0.0
```

Figure 1: Strings output shows OpenSSL Crypto Library functions and an interesting string of characters.

The above screenshot (Figure 1), shows that the malware uses the OpenSSL Crypto library that we investigated back in Lab 2. The ransomware uses AES 128-bit encryption using the CBC mode, which means that the key used to encrypt the files is 128 bits (16 bytes) long.

The other interesting detail is the string "`1234567890abcdef`" inside the program, which could be the key since it is 16 bytes long. The other possibilities are that the key is created from characters in this string, or it is just there to throw off our investigation. We will just take a note of it for now.

3B) ANALYSING ASSEMBLY CODE

Next, we will investigate the ransomware further by having a look at the assembly code by using the following command.

```
# objdump -d free_bitcoin
```

Ignoring the included functions from libraries, we find that the malware has the functions `main`, `encrypt_file`, `decrypt_file` and `gen_key`. Let us take a closer look at the `gen_key` function since this is most likely where the key is created to be used for encryption. Below is the assembly code of this function.

```
# objdump -d free_bitcoin

...
080489eb <gen_key>:
 80489eb: 55                      push   %ebp
 80489ec: 89 e5                   mov    %esp,%ebp
 80489ee: 83 ec 18                sub    $0x18,%esp
 80489f1: 83 ec 0c                sub    $0xc,%esp
 80489f4: 68 d2 04 00 00          push   $0x4d2
 80489f9: e8 52 fd ff ff          call   8048750 <srand@plt>
 80489fe: 83 c4 10                add    $0x10,%esp
 8048a01: 83 7d 0c 00             cmpl   $0x0,0xc(%ebp)
 8048a05: 74 4a                   je     8048a51 <gen_key+0x66>
 8048a07: 83 6d 0c 01             subl   $0x1,0xc(%ebp)
 8048a0b: c7 45 f0 00 00 00 00    movl   $0x0,-0x10(%ebp)
 8048a12: eb 35                   jmp    8048a49 <gen_key+0x5e>
 8048a14: e8 87 fe ff ff          call   80488a0 <rand@plt>
 8048a19: 89 c2                   mov    %eax,%edx
 8048a1b: 89 d0                   mov    %edx,%eax
 8048a1d: c1 f8 1f                sar    $0x1f,%eax
 8048a20: c1 e8 1c                shr    $0x1c,%eax
 8048a23: 01 c2                   add    %eax,%edx
 8048a25: 83 e2 0f                and    $0xf,%edx
 8048a28: 29 c2                   sub    %eax,%edx
 8048a2a: 89 d0                   mov    %edx,%eax
 8048a2c: 89 45 f4                mov    %eax,-0xc(%ebp)
 8048a2f: 8b 55 f0                mov    -0x10(%ebp),%edx
 8048a32: 8b 45 08                mov    0x8(%ebp),%eax
 8048a35: 01 d0                   add    %edx,%eax
 8048a37: 8b 55 f4                mov    -0xc(%ebp),%edx
 8048a3a: 81 c2 70 91 04 08       add    $0x8049170,%edx
 8048a40: 0f b6 12                movzbl (%edx),%edx
 8048a43: 88 10                   mov    %dl,(%eax)
 8048a45: 83 45 f0 01             addl   $0x1,-0x10(%ebp)
 8048a49: 8b 45 f0                mov    -0x10(%ebp),%eax
 8048a4c: 3b 45 0c                cmp    0xc(%ebp),%eax
 8048a4f: 72 c3                   jb     8048a14 <gen_key+0x29>
 8048a51: 8b 55 08                mov    0x8(%ebp),%edx
 8048a54: 8b 45 0c                mov    0xc(%ebp),%eax
 8048a57: 01 d0                   add    %edx,%eax
 8048a59: c6 00 00                movb   $0x0,(%eax)
 8048a5c: 90                      nop
 8048a5d: c9                      leave
 8048a5e: c3                      ret

...
```

Figure 2: Assembly code for the gen_key function.

Of interest is that the function calls `srand`, which is the C function for setting the seed for the random number generator, which is shown in bold in Figure 2. To try and figure out what is the value of the seed, we will compile our own test program and compare the assembly code. We have provided you the test code inside the file `srand_test.c`. The test code uses the value of 16 (0x10 in hexadecimal) to set the seed, so we will look for where this value is in the assembly code.

```
# objdump -d srand_test
...
0804840b <main>:
 804840b: 8d 4c 24 04              lea    0x4(%esp),%ecx
 804840f: 83 e4 f0                 and    $0xfffffff0,%esp
 8048412: ff 71 fc                 pushl  -0x4(%ecx)
 8048415: 55                       push   %ebp
 8048416: 89 e5                    mov    %esp,%ebp
 8048418: 51                       push   %ecx
 8048419: 83 ec 04                 sub    $0x4,%esp
 804841c: 83 ec 0c                 sub    $0xc,%esp
 804841f: 6a 10                    push   $0x10
 8048421: e8 ba fe ff ff           call   80482e0 <srand@plt>
 8048426: 83 c4 10                 add    $0x10,%esp
 8048429: b8 00 00 00 00           mov    $0x0,%eax
 804842e: 8b 4d fc                 mov    -0x4(%ebp),%ecx
 8048431: c9                       leave
 8048432: 8d 61 fc                 lea    -0x4(%ecx),%esp
 8048435: c3                       ret
 8048436: 66 90                    xchg   %ax,%ax
 8048438: 66 90                    xchg   %ax,%ax
 804843a: 66 90                    xchg   %ax,%ax
 804843c: 66 90                    xchg   %ax,%ax
 804843e: 66 90                    xchg   %ax,%ax
...
```

Figure 3: Our seed value of 16 (0x10 in hex) is pushed to the stack before calling srand.

We can see that our seed value of `0x10` is pushed onto the stack directly before the program calls `srand`. Comparing this procedure to the assembly code in Figure 2, we can see that just before the `srand` call at machine instruction address of `0x80489f4` in `gen_key` the hexadecimal value of `0x4d2` is pushed to the stack. This means that in `gen_key`, the seed is set to 1234, which is `0x4d2` in decimal format.

CITS 3004 Cybersecurity

3C) USING GDB-PEDA TO ANALYSE RANSOMWARE STATE

Finally, we will use `gdb-peda` to execute the ransomware to piece together all the information we have gathered so far. `Gdb-peda` is an abbreviation of Python Exploit Development Assistance for `gdb`, where `gdb` is a tool originally used for debugging C programs. `Gdb-peda` comes pre-installed on the SEED Ubuntu VM, so we do not have to worry about any installation.

Below we list some useful commands for inside the `gdb-peda` shell to help you reverse engineer the ransomware.

```
gdb-peda$ info func # Prints out all the functions inside of the program.

gdb-peda$ disas <function name> # Print the assembly code and machine instruction number of a function.

gdb-peda$ b *<machine instruction address> # Pauses the programs execution at the machine instruction address and prints the programs state.

gdb-peda$ x/2x $esp # Prints the first 2*4=8 bytes from the start of the stack ($esp)

gdb-peda$ r # Starts the programs execution from the very start.

gdb-peda$ c # Continue the programs execution to the next breakpoint or until completion.

gdb-peda$ si # Execute the next machine instruction and then print the state of the program.
```

For a list of more commands to use `gdb`, take a look at https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf.

Since the ransomware is poorly designed and only encrypts the files in the working directory, we will create a test folder to execute the malware from. Ideally, if you are doing real malware analysis you would want to completely isolate it inside a separate VM before executing it. However, for our purposes running it from inside an isolated folder should be sufficient since it only encrypts files inside the working directory.

You can use the commands below to prepare your test folder and start `gdb-peda`.

```
# mkdir test
# cp free_bitcoin test/
# cd test/
# chmod 500 free_bitcoin
# gdb ./free_bitcoin
```

We will begin our analysis by getting the machine instruction for when the function `rand` is called and set a breakpoint at that instruction so we can analyse the state of the program. We will also set another breakpoint directly after `gen_key` returns to the function `encrypt_file`, so that we can pause the programs execution before any files are encrypted. Below are the commands with snippets to help you set up the breakpoints before starting the program.

```
gdb-peda$ disas gen_key

Dump of assembler code for function gen_key:
   0x080489eb <+0>: push   ebp
   0x080489ec <+1>: mov    ebp,esp
   0x080489ee <+3>: sub    esp,0x18
   0x080489f1 <+6>: sub    esp,0xc
   0x080489f4 <+9>: push   0x4d2
   0x080489f9 <+14>:     call   0x8048750 <srand@plt>
   0x080489fe <+19>:     add    esp,0x10
   0x08048a01 <+22>:     cmp    DWORD PTR [ebp+0xc],0x0
   0x08048a05 <+26>:     je     0x8048a51 <gen_key+102>
   0x08048a07 <+28>:     sub    DWORD PTR [ebp+0xc],0x1
   0x08048a0b <+32>:     mov    DWORD PTR [ebp-0x10],0x0
   0x08048a12 <+39>:     jmp    0x8048a49 <gen_key+94>
   0x08048a14 <+41>:     call   0x80488a0 <rand@plt>
...

gdb-peda$ b *0x08048a14

Breakpoint 1 at 0x8048a14
```

```
gdb-peda$ disas encrypt_file

Dump of assembler code for function encrypt_file:
   0x08048ca1 <+0>: push   ebp
   0x08048ca2 <+1>: mov    ebp,esp
   0x08048ca4 <+3>: sub    esp,0xe8
   0x08048caa <+9>: mov    eax,DWORD PTR [ebp+0x8]
   0x08048cad <+12>:     mov    DWORD PTR [ebp-0xdc],eax
   0x08048cb3 <+18>:     mov    eax,gs:0x14
   0x08048cb9 <+24>:     mov    DWORD PTR [ebp-0xc],eax
   0x08048cbc <+27>:     xor    eax,eax
   0x08048cbe <+29>:     mov    DWORD PTR [ebp-0x2d],0x0
   0x08048cc5 <+36>:     mov    DWORD PTR [ebp-0x29],0x0
   0x08048ccc <+43>:     mov    DWORD PTR [ebp-0x25],0x0
   0x08048cd3 <+50>:     mov    DWORD PTR [ebp-0x21],0x0
   0x08048cda <+57>:     sub    esp,0x8
   0x08048cdd <+60>:     push   0x11
   0x08048cdf <+62>:     lea    eax,[ebp-0x1d]
   0x08048ce2 <+65>:     push   eax
   0x08048ce3 <+66>:     call   0x80489eb <gen_key>
   0x08048ce8 <+71>:     add    esp,0x10
...

gdb-peda$ b *0x08048ce8

Breakpoint 2 at 0x8048ce8
```

We will start running the program to see the state of the registers and stack at each time the `rand` function is called.

```
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0x1c8f220e
EDX: 0x65 ('e')
ESI: 0xb7d30000 --> 0x1b1db0
EDI: 0xb7d30000 --> 0x1b1db0
EBP: 0xbfffea38 --> 0xbfffeb38 --> 0xbfffebf8 --> 0x0
ESP: 0xbfffea20 --> 0x0
EIP: 0x8048a14 (<gen_key+41>:    call   0x80488a0 <rand@plt>)
EFLAGS: 0x200283 (CARRY parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048a07 <gen_key+28>:      sub    DWORD PTR [ebp+0xc],0x1
   0x8048a0b <gen_key+32>:      mov    DWORD PTR [ebp-0x10],0x0
   0x8048a12 <gen_key+39>:      jmp    0x8048a49 <gen_key+94>
=> 0x8048a14 <gen_key+41>:      call   0x80488a0 <rand@plt>
   0x8048a19 <gen_key+46>:      mov    edx,eax
   0x8048a1b <gen_key+48>:      mov    eax,edx
   0x8048a1d <gen_key+50>:      sar    eax,0x1f
   0x8048a20 <gen_key+53>:      shr    eax,0x1c
No argument
[------------------------------------stack-------------------------------------]
0000| 0xbfffea20 --> 0x0
0004| 0xbfffea24 --> 0x5b ('[')
0008| 0xbfffea28 --> 0x1
0012| 0xbfffea2c --> 0xe
0016| 0xbfffea30 --> 0xb7fff000 --> 0x23f3c
0020| 0xbfffea34 --> 0xb7fff918 --> 0x0
0024| 0xbfffea38 --> 0xbfffeb38 --> 0xbfffebf8 --> 0x0
0028| 0xbfffea3c --> 0x8048ce8 (<encrypt_file+71>:      add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x08048a14 in gen_key ()
gdb-peda$
```

Figure 4: After reaching the rand function a second time, the character 'e' is saved to the EDX register.

Figure 4 above shows the state of the program after reaching the `rand` function a second time (continuing the execution of the program once). This snapshot of the program's state tells us two important things about how the key is generated.

Firstly, the key is generated inside a loop since when the program continued after reaching the first breakpoint it paused at the same breakpoint a second time, instead of reaching the breakpoint in `encrypt_file`.

The second observation is that the character 'e' is stored inside the `EDX` register, which is highlighted in Figure 4. This can mean that 'e' is the result of some operations following the first `rand` call, and is most likely the first character of the encryption key.

To investigate this further, we will now set a breakpoint after the `rand` call at the machine instruction at the address of `0x8048a14` and step through the program's execution by machine instruction (using the `si` command) until we find something interesting in the registers or the stack.

```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb1c --> 0x804914e (<stat+30>:       add    esp,0x18)
EBX: 0x0
ECX: 0x1bbffa83
EDX: 0x8049173 ("4567890abcdef")
ESI: 0xb7d30000 --> 0x1b1db0
EDI: 0xb7d30000 --> 0x1b1db0
EBP: 0xbfffea38 --> 0xbfffeb38 --> 0xbfffebf8 --> 0x0
ESP: 0xbfffea20 --> 0x0
EIP: 0x8048a40 (<gen_key+85>:   movzx  edx,BYTE PTR [edx])
EFLAGS: 0x200202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8048a35 <gen_key+74>:      add    eax,edx
   0x8048a37 <gen_key+76>:      mov    edx,DWORD PTR [ebp-0xc]
   0x8048a3a <gen_key+79>:      add    edx,0x8049170
=> 0x8048a40 <gen_key+85>:      movzx  edx,BYTE PTR [edx]
   0x8048a43 <gen_key+88>:      mov    BYTE PTR [eax],dl
   0x8048a45 <gen_key+90>:      add    DWORD PTR [ebp-0x10],0x1
   0x8048a49 <gen_key+94>:      mov    eax,DWORD PTR [ebp-0x10]
   0x8048a4c <gen_key+97>:      cmp    eax,DWORD PTR [ebp+0xc]
[------------------------------------stack-------------------------------------]
0000| 0xbfffea20 --> 0x0
0004| 0xbfffea24 --> 0x5b ('[')
0008| 0xbfffea28 --> 0x1
0012| 0xbfffea2c --> 0x3
0016| 0xbfffea30 --> 0xb7fff000 --> 0x23f3c
0020| 0xbfffea34 --> 0xb7fff918 --> 0x0
0024| 0xbfffea38 --> 0xbfffeb38 --> 0xbfffebf8 --> 0x0
0028| 0xbfffea3c --> 0x8048ce8 (<encrypt_file+71>:      add    esp,0x10)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08048a40 in gen_key ()
gdb-peda$
```

Figure 5: The register EDX points to a section of the string that we found earlier on during this lab task.

Figure 5 shows us the state of the ransomware after the assembly line `add edx,0x8049170`, which is the machine instruction at the address of `0x8048a3a`. We can see the `EDX` register now holds a pointer to a snippet of the string "`1234567890abcdef`" that we found earlier using `strings`. If we take a closer at what is stored at line `0x8049170` we find that the above string is saved there. The command below tries to print the data stored at the address `0x8049170` as a string in C.

```
gdb-peda$ x/1s 0x8049170
0x8049170:      "1234567890abcdef"
```

CITS 3004 Cybersecurity

Stepping forward by one machine instruction and we see that the ransomware gets the character at the start of snippet that was stored in the `EDX` register, which is the character '4'.

3D) PUTTING IT ALL TOGETHER TO GET THE KEY

With all the information that we gathered about how the ransomware generates its encryption key, we can now make an educated guess of the algorithm.

1. Sets the seed to 1234.
2. Using the `rand` number generator, select a character from the string "`1234567890abcdef`".
3. Repeat step 2 until a key with a length of 16 bytes has been generated for the `aes-128-cbc` encryption.

We can now do some experiments and check if we have found the right algorithm by setting the random seed to 1234, and use `rand` to select a character from the string above, such that the first character that is picked is 'e' then '4'.

It is now your task to figure out the complete key used for encryption by the ransomware.

3E) REVERSE ENGINEERING NEW RANSOMWARE

A new variant of the `free_bitcoin` ransomware has been found called `free_ethereum`. It is believed that they are both made by the same author and use the same algorithm for generating the encryption key. However, for some reason the key generated in `free_bitcoin` is not the same key as the one in `free_ethereum`.

Can you reverse engineer `free_ethereum` and get the encryption key used?

Answer Q8 ~ Q13 on LMS