Lab 5: Social and Reverse Engineering

Revised: March 5, 2019

Background. The goal of this lab is to familiarize students with how adversaries attack common mistakes humans make with respect to security. These include social engineering as well as attacking the common programming mistakes that leave software vulnerable to control hijacking and arbitrary code execution.

Task I. Security Questions

Many website use security questions as a mechanism for password reset/recovery. Like with all shared secret systems, security questions make the trade-off between security and usability. Specifically, security questions are designed to be more memorable than the password protecting the account, which leaves them prone to guessing. Sites try to make up for this "guessability" by increasing the number of questions a user has to answer in order to reset their password.

In general, a good security questions should be:

Safe: Not able to be guessed or researched
Memorable: easy for the user to remember
Simple: has a precise and consistent answer
Many: could have many possible answers

Unfortunately, many sites don't abide by these recommendations, and let users choose weak, and ultimately, easily guessable security questions. Indeed, security questions are particularly problematic for celebrities, whose private information and preferences may be more easily researchable. As an example, during the 2008 Presidential campaign, Sarah Palin's email was hacked via guessable security questions.

In this task, your job is to pick a celebrity or historical figure and see if you can find the answers to the following common security questions:

1. What was the make and model of your first car?
2. What was the name of your elementary / primary school?
3. In what city or town does your nearest sibling live?
4. What time of the day were you born? (hh:mm)
5. What is your pet's name?
6. In what year was your father born?
7. What is your favorite color?
8. What was your favorite place to visit as a child?

Please submit the name of your target subject plus answers to as many of the security questions as part of your lab write up.

Task II. Ransomware Attack!

Oh no! Your employer is experiencing a ransomware attack, leaving some very important files inaccessible. You've received this memo asking for your help:

> **TO:** Students of CPE321
>
> **FROM:** Someone in a crisis
>
> **SUBJECT:** URGENT: Under attack, need help!!!
>
> We are being attacked by some sort of ransomware! This is what I've figured out so far: This ransomware is a collection of executable C programs that seem to all work differently, and have encrypted some very important files that need to be recovered ASAP! Before you ask, no I didn't back the files up.
>
> To decrypt these files, each ransomware executable must be run with the correct input. This is where I got stuck.
>
> I have attached the following:
>
> - The encrypted files, one from each attack.
> - `attackX`, the executable C program that was used to perform attack X and prompts you to enter a password to decrypt the corresponding file when you run it with no arguments.

Your employer refuses to pay the ransom for the files because they believe that you have the skills to recover them. Your task is to use your reverse engineering skills to recover the files. Good luck!

Disclaimers

You should never run actual malware of any kind in an unsafe environment. In this lab nothing bad will happen, but other malware isn't so nice. Typically, if you want to execute malware in order to learn what it does and how it does it, you would do so on a virtual machine.

Additionally, reverse engineering (which is what you are going to do!) is considered illegal under certain circumstances. Read about it here:

https://www.eff.org/issues/coders/reverse-engineering-faq.

Attack 1

Encrypted file: `attack1_enc.jpg`

Executable file: `attack1`

A. Getting started.

You will be examining x86-64 assembly. A good guide to refresh your memory can be found here: https://web.stanford.edu/class/cs107/guide/x86-64.html.

Some tools that might be useful in this lab are:

- `gdb`

  The GNU debugger. A command line debugger that allows you to look at assembly code, trace through a program, examine memory and registers, set breakpoints, and more. A useful reference for gdb commands can be found here: http://csapp.cs.cmu.edu/2e/docs/gdbnotes-x86-64.pdf
- `objdump -t`

  This prints the symbol table of the given executable, which includes names of functions and global variables.
- `objdump -d`

  This disassembles the provided file, outputting the assembly code.

A walk-through has been provided for the first level to help you develop some strategies that can be used to analyze the ransomware. For subsequent attacks, you are on your own...but always feel free to ask for help if you get stuck!

B. Analyzing main.

We will start with `attack1`, the attack executable, and find out what it's doing. First, start `gdb` on the file by typing the command:

```
gdb attack1
```

To view the disassembled main function, use the gdb command:

```
(gdb) disas main
```

If you prefer, outside of `gdb` you can run the following to pipe the entire disassembled `attack1` executable into a text file to reference along the way:

```
 objdump -d attack1 > attack1.objdump
```

Now you have a nice chunk of assembly code to examine. Let's start by looking at this line (the hex values may be different for you, but it will be the first line in `main` with a comment off to the right):

```
mov 0x200788(%rip),%rax     # 0x601080 <encrypted>
```

What does this mean? Well, gdb gave us a helpful comment that gives the result of the memory calculation `0x200788(%rip)` (in red) and a hint (in blue) telling us that a variable `encrypted` is stored at that address.

A few lines down, we see that a call to `fopen` is made with `encrypted` as the first argument (recall that `%rdi` is the 1st argument in x86-64). This means that `encrypted` must be a string! You might be able to guess what this string is by its name, but we can check by examining the address of `encrypted`, which is a pointer to a character pointer:

```
(gdb) x/s *(char **)0x601080
```

So, it turns out that `encrypted` is the name of the encrypted file, and `main` opens it.

Below that, there is some similar-looking assembly code that appears to do something with `decrypted`. You can deduce what this does or repeat the process from above to find out.

C. Analyzing the decryption function.

Now that we are warmed up, let's find out how to decrypt this file. In `main`, find the call to `scanf`. In order to determine what kind of input it is expecting, we must inspect the contents of the first argument passed to `scanf` (recall `%rdi`/`%edi` contains the first argument in x86-64, and the first argument to `scanf` is a format string).

Let's do this a little differently than before and inspect the contents of the register. Add a breakpoint at the address where `scanf` is called. To find this address, look at your disassembled `main` and find the line where `scanf` is called, which will look something like this:

```
400990: e8 fb fc ff ff      callq 400690 <__isoc99_scanf@plt>
```

The value in red is the address we want to add a breakpoint at. So, add the breakpoint and run the executable:

```
(gdb) break *0x400990
```

```
(gdb) run
```

Execution will stop at the breakpoint you have set — where `scanf` is called. To examine the format string that is being passed to `scanf`:

```
(gdb) print (char *)$edi
```

From the output, see if you can identify the format of the input we need to provide.

After this input is read, notice that it is passed to a function called `decrypt` along with a pointer to the encrypted file and a pointer to the decrypted file. We can guess that this probably decrypts the encrypted file and writes the result to the decrypted file, so the input that was read via `scanf` is probably the key it will use to decrypt. So, let's have a look at the disassembled `decrypt` function (in your `objdump` file or by running `disas decrypt` in gdb).

Towards the end of `decrypt`, notice the `cmpl` and `jne` instructions. These indicate the presence of a loop. See if you can identify what the loop condition is. Then, see if you can answer the following:

1. How is the input we provided in `%edi` (the key) being used?
2. On each iteration, what is being written to the decrypted file (via `fputc`)?
3. What encryption scheme do you conclude is being used here?

Once you have answered these questions, you will know what kind of input we need to provide to `attack1`, and how this input is used to decrypt the encrypted file.

But, we still do not know what specific input to provide. Quit gdb and print out the symbol table, including all function names, with the following command:

```
objdump -t attack1
```

Notice from the output that there is an `encrypt` function in the program. Let's look at this next.

D. Analyzing the encryption function.

First, notice some familiar assembly code at the beginning of `main` that appears to open two files and pass the file pointers to `encrypt`. So, the encrypt function is getting two file pointers. One is probably the original, unencrypted file and the other is used to write the encrypted file to.

Similarly to above, disassemble the `encrypt` function in gdb or refer to it in the `objdump` file you created earlier.

In the disassembled `encrypt`, there is some more assembly code that should look familiar. (Hint: Look for something that resembles the disassembled `decrypt` function.) From this and your understanding of what the `decrypt` does, you should be able to find the key that you need to provide to `attack1` in order to decrypt the file.

Run `attack1`, enter the key that you found, and if you got it right you will be able to open the decrypted image `attack1_dec.jpg`!

Attack 2

Encrypted file: `attack2_enc.jpg`

Executable file: `attack2`

It looks like the attackers have stepped up their game, and started using a real cipher this time!

Hint:

- Analyzing this attack will be similar to what you did for Attack 1–use the methods you developed in Attack 1 to reverse engineer the executable and determine how the file is encrypted and decrypted (both functions exist in the binary).

- Further, consider all the things that go into an encryption function, and where they might be stored.

Attack 3

Encrypted file: `attack3_enc.jpg`

Executable file: `attack3`

To get started, take a look at the symbol table for this executable, focusing on the function names:

```
objdump -t attack3
```

It looks like the attackers removed the decryption algorithm from the executable this time, but the executable still has the encryption function, so all hope is not lost!

Hints:

- Again, look at the inputs to the encryption function–are they as you expect to encryption function? Maybe it's not encryption after all.
- It might be useful to run the executable to encrypt a small file of your own and follow along with gdb to determine how it gets encrypted; *i.e.* can you follow the operations that happen to each byte of the file?
- You will probably want to write a decryption program in order to decrypt the file. (Feel free to use any language!)

Deliverables

Please attempt to break all three attack programs. Write a brief report describing what you did to break each. Please include a brief explanation of your attacks in sufficient detail that it can be reproduced. Please use screenshots only if it is important for explaining your attack. Submit your completed write up to PolyLearn in PDF format.