MP 5: Advanced Concepts

This project is split into two parts, with the first checkpoint due on **Dec 4** at **11:59pm** and the second checkpoint due on **Dec 13** at **11:59pm**. We strongly recommend that you get started early. We will run the autograder on Friday and Sunday for Checkpoint 1. Autograder will run on **Saturday** and **Tuesday** for Checkpoint 2 to give you feedback and once after the due date.

This is a group project; you SHOULD work in **teams of two** and if you are in teams of two, you MUST submit one project per team. If two different sets of answers are submitted, the one with lower total grade will be accepted. Please find a partner as soon as possible. If you have trouble forming a team, post on Piazza's partner search forum. Build your team such that at least one member of the team can run the required tools.

The code and other answers your group submits must be entirely your own work, and you are bound by the Student Code. You MAY consult with other students about the conceptualization of the project and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone outside your group. You MAY consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

Solutions MUST be submitted electronically in any one of the group member's SVN repository, following the filename and solution formats. The sample answers are provided in your svn repository. Failure to follow instructions and format your answers properly will result in losing your grades.

"There's an entire flight simulator hidden in every copy of Microsoft Excel 97."

– Bruce Schneier

Introduction

This project will introduce you to new attacks that have not been covered in the other programming assignments.

Objectives

- Understand offensive techniques in real-world attacks.

Guidelines

- You SHOULD work in a group of 2.
- Your answers may or may not be the same as your classmates'.
- All the necessary files to start the project are given under a folder called "mp5" in your SVN repository: `https://subversion.ews.illinois.edu/svn/fa17-cs461/NETID/mp5`
- We generated submission files for you to submit your answers in. You MUST submit your answers in the provided files; we will only grade what's there!
- Each submission file contains an example answer of the expected format. **Overwrite the example answers in the submission files with your answer!**

Read this First

This project asks you to perform attacks, with our permission, against target websites that we are providing for this purpose. Attempting the same kinds of attacks against other networks or websites without authorization is prohibited by law and university policies and may result in *fines, expulsion, and jail time*. **You MUST NOT attack any network without authorization!** There are also severe legal consequences for unauthorized interception of network data under the Electronic Communications Privacy Act and other statutes. Per the course ethics policy, you are required to respect the privacy and property rights of others at all times, *or else you will fail the course.* See "Ethics, Law, and University Policies" on the course website.

5.1 Checkpoint 1

5.1.1 Reverse Engineering Intro (5 points)

Security analysts and attackers both frequently perform reverse engineering of executable binary files to search for vulnerabilities and to understand program behaviors. In this section, you will reverse engineer a kernel module, provided as a binary file, from a hypothetical malware that we set up for this assignment.
You have been given a binary kernel module `knockd.ko` of the malware. In Checkpoint 2, you will search for a secret communication channel, i.e., a port knocking sequence, that the malware used to communicate with its creator. You will use `objdump` to analyze the binary file. The malware uses `netfilter`, a packet filtering framework implemented in Linux kernels, and `port knocking` technique to enable the secret channel. For Checkpoint 1, we will introduce you to some of these concepts and guide you through a bit of the process with a series of questions.

Reverse-engineer a kernel module. Examine the binary kernel module file, `knockd.ko`, using following command `objdump -xDsl knockd.ko`.
Look at disassembled code of each function and understand what each function does. Hint: A graphical disassembler like IDA, Hopper, or Radare might help here.
Note: We have left parts of this reverse-engineering MP a bit vague. In real-world reversing applications, you will have very limited information of your target binary.

Provide concise answers to the following questions.

1. Identify all the function names in the symbol table of the file.

   What to submit (1pt) Submit `5.1.1.1.txt` that contains: 1) section, and 2) the function names that start with `knockd`. Each pair `section,function name` is separated by a comma and is listed on a line.

2. Identify the netfilter hook function.

   What to submit (1pt) Submit `5.1.1.2.txt` that contains the name of the netfilter hook function.

3. What port does the netfilter hook function filter?

   What to submit (1pt) Submit `5.1.1.3.txt` that contains the port number.

4. What is the protocol for receiving the port knocks?

   What to submit (2pts) Submit `5.1.1.4.txt` that contains the `keyword` of the protocol, as defined in `RFC 1700` (Assigned Internet Protocol Numbers).

5.1.2 Object Deserialization Attack (5pts)

Serialization is the process of translating an object on memory of a running program into a data structure that can be stored, e.g., on a file on disk. The serialized object can be transmitted, e.g., via a network or read from disk, and later deserialized by another program to get a clone of the original object.
Attackers can modify the serialized object to include malicious code that will be executed when a target program deserializes the malicious object.
Python pickle: a case study of object deserialization attack. The `pickle` module in Python implements a protocol for serializing and de-serializing a Python object structure. You are given a target, which is a Python (version 2.7) script web server, running at `http://copland.crhc.illinois.edu:46152`. The target accepts and deserializes any pickle file at its endpoind `/upload`.

Your goal in Checkpoint 2 will be to craft a malicious pickle file that tricks the target into to loading the malicious pickle file and retrieving a `SECRET_KEY` variable stored in the target's Python script. You must name your pickle file `malicious.pickle`. The target only returns whether your pickle

file has been successfully loaded or there has been an error loading pickle file. You are only allowed to read the `SECRET_KEY`, do not delete anything nor attempt to stop the target's web server nor alter the key after reading it.

Hint: You should create a communication channel to exfiltrate the `SECRET_KEY` from the target to a server that you control.
Hint: Use `curl` to upload your malicious file
`curl -i -F afile=@malicious.pickle http://copland.crhc.illinois.edu:46152/upload`
Reference: `https://docs.python.org/2.7/library/pickle.html`
In Checkpoint 1, you only need to submit a valid pickle file.
In Checkpoint 2, you will submit a malicious pickle.

1. Can you create a valid pickle file?
   What to submit (5pts) Submit `5.1.2.1.pickle` that contains any valid pickle file that the target can load.

5.1.3 Double withdrawal attack (5pts)

A double withdrawal is an attack that withdraws money from a digital wallet multiple times, such that the wallet balance only decreases by the amount of one withdraw. For example, the available balance in a digital wallet is `10,000` and the attacker runs a double withdrawal attack, each time withdrawing `100`. After running the attack, the attacker ends up with `200`, while the new balance of the digital wallet is `9,900` (only decreases by `100`).

Race condition: a case study of double withdrawal attack. You are given a digital wallet server associated with your *netid* running at
`http://copland.crhc.illinois.edu:46153`. The wallet provides following features.
Register a wallet at `/register/<netid>`
Check available balance at `/balance/<netid>`
Withdraw an amount *x* at `/withdraw/<netid>?amount=x`
Unregister and reset available balance at `/unregister/<netid>`

Your goal is to run a double withdrawal attack against the digital wallet server. The attack is considered successful when you can withdraw more than the current available balance. In such case, the server will return a special token acknowledging successful completion of your attack. The token is unique and is associated with your netid.
For Checkpoint 1, you only need to register your wallet and withdraw exactly *1* from your account.

1. Register an account and withdraw an amount of `1` from your account
   What to submit (5pts) Submit `5.1.3.1.txt` that contains the receipt (the hash) of your withdrawal.

5.1.4 Password Cracking Intro (5pts)

In Checkpoint 2, you will be cracking passwords using a variety of tools and methods. In this section, we want to introduce you to the topic by brute force cracking some simple passwords. You should download and install the password cracking tool `hashcat`. If possible, we recommend installing it on a machine with a GPU. `hashcat` binaries for Windows and Linux and source code are all available at this URL: `https://hashcat.net/hashcat/`

Once you have installed hashcat, your task will be to crack the three SHA1 password hashes in `5.1.4.hashes`. The corresponding passwords are all six characters long, so you should be able to brute force them relatively quickly. You can read about how to perform brute force attacks with `hashcat` at this URL: `https://hashcat.net/wiki/doku.php?id=mask_attack`. Below, you'll find a command to get you started:

```
hashcat64 -m 100 -a 3 5.1.4.hashes ?a?a?a?a?a?a
```

Once you find the plaintext passwords, you should place them `5.1.4.txt`, with one password per line. The line number for each password should match the line number for its hash in `5.1.4.hashes`.

What to submit (5pts) You should submit the plaintext passwords in `5.1.4.txt`. One password per line. Each password should be on the same line number as its hash in `5.1.4.hashes`

5.2 Checkpoint 2

5.2.1 Port-knocking attack (cont.) (20pts)

Now, you will need to reverse engineer the application described in 5.1.1. Provide concise answers to the following questions.

1. How many ports need to be knocked before the filtered port is openned?

   What to submit (5pts) Submit `5.2.1.1.txt` that contains the number of ports need to be knocked.

2. What is the knocking sequence, i.e., the ports need to be knocked, before the filtered port is openned?

   What to submit (10pts) Verify your answer by loading the kernel module in a Virtual Machine, using following command `insmod knockd.ko` as `root`. Submit `5.2.1.2.txt` that contains the sequence of the ports need to be knocked, separated by a comma.

3. How long will the filtered port be openned after receiving the knocking sequence?

   What to submit (5pts) Submit `5.2.1.3.txt` that contains the time in seconds.

5.2.2 Object Deserialization Attack (cont.) (25pts)

Now you should craft a malicious pickle file to attack the server described in 5.1.2. Good luck!

1. What is your malicious pickle file that reads and extracts the `SECRET_KEY`?
   What to submit (10pts) Submit `5.2.2.1.pickle` that contains your malicious pickle file.

2. What is the `SECRET_KEY`?
   What to submit (15pts) Submit `5.2.2.2.txt` that contains the secret key.

5.2.3 Double withdrawal attack (cont.) (25pts)

As indicated in 5.1.3, there is a race condition vulnerability in our fake banking server. Your job is to cheat the server and withdraw some extra money. Find some way to exploit the vulnerability. If you're clever, you might not even need to write a script.

When your attack is successful, the server will return a special race condition token indicating that you've completed at least one double withdrawal. Submitting that token is all you need for this section. You found a way to create free money. What more could we ask of you?

1. What is the token?
   What to submit (25pts) Submit `5.2.3.1.txt` that contains the race condition token acknowledging successful completion of your attack.

5.2.4 Password Cracking (cont.) (25pts)

In this section, you have 10k SHA1 password hashes to crack in `5.2.4.hashes`. You will need to use a variety of methods to crack these passwords. Your grade for this section will be based on the proportion of the passwords that you manage to crack. You can use any tools you like, but we recommend `hashcat`, `johntheripper`, and `RainbowCrack`. You'll find links to these tools below.

- hashcat `https://hashcat.net/hashcat/`
- johntheripper `http://www.openwall.com/john/`
- RainbowCrack `http://project-rainbowcrack.com`

We have precomputed *partial* rainbow tables for all US keyboard typable passwords up to lenght 7. They are available at this URL: `https://uofi.box.com/v/rainbowtables`. These will work with `RainbowCrack`. Depending on your hardware, you might find that using rainbow tables takes more time than brute forcing with hashcat. The increases in CPU and GPU performance over the past decade have made rainbow tables less useful than they once were.
Daniel Miessler provides a github with large lists of known passwords from previous breaches available at this URL: `https://github.com/danielmiessler/SecLists/tree/master/Passwords`. These will be extremely useful for dictionary attacks and hybrid attacks with `hashcat`.
Hints:

- As you crack passwords, remove them from your list for more advanced and computationally intense cracking. Trying to crack passwords you've already cracked is a waste of time.
- Look for patterns in shorter passwords you crack. Longer passwords might follow similar patterns.
- GPU acceleration can dramatically speed up cracking. We've seen performance improvements of 100x and even 1,000x, depending on the appliation, attack method, and hardware.
- Read the documents for `hashcat`. You can highly customize your attacks. `https://hashcat.net/wiki/`
- Try dictionary attacks and variations on dictionary attacks.
- Some passwords might not be in your dictionary, but might be statistically similar to passwords. Both `hashcat` and `johntheripper` provide enhanced tools for Markov modeling. Training a model based on known passwords might be helpful. `http://www.openwall.com/john/` `https://hashcat.net/wiki/doku.php?id=statsprocessor`
- The three passwords you cracked in 5.1.4 are hints for some of the passwords this section.

Once you find the plaintext passwords, you should place them `5.2.4.txt`, one password per line. The line number for each password should match the line number for its hash in `5.2.4.hashes`.
What to submit (25pts) You should submit the plaintext passwords in `5.2.4.txt`. One password per line. Each password should be on the same line number as its hash in `5.2.4.hashes`

5.2.5 Extra Credit (15pts)

'Capture the Flag' contests are competitive events where players apply their security knowledge and creative thinking to hack their way through various challenges, to uncover the secret 'flag' (a string) which is then submitted for points. This checkpoint is a reverse engineering challenge (kind of) in the spirit of CTF challenges. You are required to find the flag (in the format $flag\{XXX\}$) hidden in the binary. Some basic automation is required; I suggest using python pwntools.
Challenge Read the text file `5.2.5.txt` provided with the binary `5.2.5`, and reverse engineer the binary `5.2.5` to uncover its secrets.

Provide concise answers to the following questions.

1. What is the first password?
   What to submit (2pts) Submit `5.2.5.1.txt` that contains the password.

2. What is the second password?
   What to submit (2pts) Submit `5.2.5.2.txt` that contains the password.

3. What is the third password?
   What to submit (2pts) Submit `5.2.5.3.txt` that contains the password.

4. What is the AES key?
   What to submit (4pts) Submit `5.2.5.4.txt` that contains the AES key, as a string (not hexpairs).

5. What is the flag?
   What to submit (5pts) Submit `5.2.5.5.txt` that contains only the flag, including the opening 'flag{' and closing curly brace.

All credits go to **Benjamin Lim** for creating this extra credit challenge.