CS155: Computer Security **Spring 2024**

Project #3: Networking

Due: Part 1-3: Friday May 31 11:59pm PT,
Part 4: Friday June 7th 11:59pm PT

Submit by Gradescope

Ready to learn about network security? In Parts 1 and 2, you will see how a simple port scan can reveal a large amount of information about a remote server. In Part 3, you will analyze a dump of network traffic on a local network and learn how to identify different types of network anomalies. Finally, in Part 4, you will implement a DNS spoofer that hijacks an HTTP connection!

This project will be implemented in Go. You likely haven't used Golang before - that is totally okay! Part 0, will ask you to complete two sections of Go's official tutorial, A Tour of Go. We've also included scaffolding for both parts that use Go (3 and 4).

As you saw in Project 1, C and C++ are riddled with memory safety pitfalls, making it practically impossible to write perfectly secure C/C++ code. The security community has largely agreed that future systems need to be built in safer languages. Two notable contenders have emerged: Go and Rust. Rust is a low-level language with performance similar to C, but has an extreme learning curve. Go is a much simpler language, often used in less performance-critical environments.

Contents

0.1 Setup Instructions

1. For Part 1, install nmap locally. Follow the installation instructions here for Mac users, here for Windows users, and here for Linux users.

2. For Part 2, install Wireshark locally. Download the Stable Release for your corresponding operating system here.

3. For Part 3, install Go and gopacket. You can install Go by following the instructions for your corresponding operating system here. Then, in your terminal, go to the Part3/ directory from the unzipped assignment (you should see the files detector.go and go.mod). From this directory, run the following command to automatically download gopacket:

   ```
   $ go mod download
   ```

4. For Part 4, install Docker. You probably already have this installed from project 2!. If not, take a look at the Project 2 handout for instructions.

0.2 Part 0: Go Tutorial

If you haven't used Go before, we encourage you to use the Go tutorial before Parts 3 and 4. Specifically, complete the Basics, Methods, and Interfaces sections of the "Tour of Go" tutorial. We also optionally recommend installing VSCode with the Go Extension enabled, which may make coding with Go easier. These tutorial sections cover:

- Packages, variables, and functions.
- Flow control statements: for, if, else, switch and defer
- More types: structs, slices, and maps.
- Methods and interfaces

The above tutorial will cover the essentials of what you need to know about Go to complete Parts 3 and 4.

Deliverables: None!

1 Part 1: Nmap Port Scanning

Port scanning is a way to probe which ports are open on a given host, and what software the server is running on any publicly-addressable interfaces. With this information, an attacker can gain a better understanding of where and how to attack a victim server. Port scanning takes advantage of conventions in TCP and ICMP that seek to provide a sender with (perhaps too much!) information on why their connection failed.

In this part, you will use the nmap tool to scan the server scanme.nmap.org.

In your scan, make sure to:

- Only scan scanme.nmap.org. Do not scan any other servers. You should only scan a server if you have explicit permission from the server operator to do so.
- Use a TCP SYN scan. (Hint: Read the nmap man pages to find the appropriate flag to use.)
- Enable OS detection, version detection, script scanning, and traceroute. (Hint: This is a single flag.)
- Do a quick scan (-T4).
- Scan all ports.
- Record the traffic with Wireshark (see part 2)

The Stanford network rate limits nmap scans, so if you are running the scan from within the Stanford network, it may take about 30 minutes to complete. Off the Stanford network, it should take between 1 and 5 minutes to complete. To prevent your computer from sleeping during the scan, you can run caffeinate from the terminal on Mac OS or disable sleep in settings for Windows/Linux.

Report the following about the target server (scanme.nmap.org):

1. What is the full command you used to run the port scan (including arguments)?
2. What is the IP address of scanme.nmap.org?
3. What ports are open on the target server? What applications are running on those ports? (For this part, you only need to report the service name printed by nmap.)
4. The target machine is also running a webserver. What webserver software and version is being used? What ports does it run on?

Deliverables: Answer the above questions (1-4) in `part1/PortScanAnswers.txt`. For full credit, please answer in the formats given in the file.

2 Part 2: Wireshark Packet Sniffing

Wireshark is a tool that monitors local network traffic. Wireshark has access to complete header information of all packets on a monitored interface and presents a helpful GUI for understanding the structure of different protocols. It can be a valuable debugging tool for networking projects, as you will see in Part 4.

Use Wireshark to examine the traffic generated by nmap during the scan in Part 1. Take a look at the Wireshark capture. Use Wireshark's filtering functionality to look at how nmap scans a single port. Report the following about the target server based on the results of the scan:

1. What does it mean for a port on scanme.nmap.org to be "closed?" More specifically, what is the TCP packet type, if any, the server gives in response to a SYN packet sent to port that is "closed?"

2. What does it mean for a port on scanme.nmap.org to be "filtered?" More specifically, what is the TCP packet type, if any, the server gives in response to a SYN packet sent to port that is "filtered?"

3. In addition to performing an HTTP GET request to the webserver, what other http request types does nmap send?

4. What TCP parameters does nmap alter to fingerprint the host's operating system?

Deliverables: Answer the above questions (1-4) in `part2/WiresharkAnswers.txt`. For full credit, please answer in the formats given in the file.

3 Part 3: Programmatic Packet Processing

In this part, you will programmatically analyze a PCAP (Packet Capture) file to detect some suspicious behavior - port scanning and ARP spoofing.

3.1 Background:

Port Scanning: Port scanning can find network hosts with services listening on one or more target ports. It can be used to locate vulnerable systems - either in preparation for an attack or research or network administration. Most hosts are not prepared to receive connections on any given port. As a result, during a port scan, a much smaller number of hosts will respond with SYN+ACK packets than originally received SYN packets. By observing this effect in a packet trace, you can identify hosts attempting a port scan.

ARP Spoofing: ARP spoofing exploits the Address Resolution Protocol (ARP), the protocol used to discover the MAC address associated with a given IP address within a network.
When Device A needs to send a packet to Device B on the network, it goes through the following steps:

1. Initially Device A only knows the IP address of Device B. But in order to populate the Ethernet frame of A's outgoing request, A needs to determine B's MAC address.
2. If Device A does not have this information, it broadcasts an ARP packet to all computers on the local network. This ARP request is just asking for the B's MAC address (more specifically the MAC address that corresponds with B's IP address).
3. Normally, B would respond with an ARP reply that contains its MAC and IP address.
4. Device A then caches this information and finally sends it initial packet to B.

However, ARP packets are not authenticated! Any device can claim to have any IP address. Additionally, most network devices automatically cache any ARP replies they receive - even if they never requested them in the first place!

In an ARP spoofing attack, the attacker repeatedly sends unsolicited replies claiming to control a certain address with the aim of intercepting data bound for another device. This results in a man-in-the-middle or denial-of-service attack on other users on the network.

3.2 Requirements

Your task is to develop a Go program that takes in a PCAP file and detects possible SYN scans and ARP spoofing attacks. You will use gopacket, a library for packet manipulation - including "layers" Go package to parse different networking layers.

You can run your program with:

```
$ go run detector.go sample.pcap
```

Unauthorized SYN scanners

You should report IP addresses that (1) sent more than 5 SYN packets in total and (2) sent 3 times as many SYN packets as the number of SYN+ACK packets they received. Silently ignore packets that are malformed or not using Ethernet, IP, and TCP.

Unauthorized ARP spoofers

You should report MAC addresses that sent more than 5 unsolicited ARP replies. Unsolicited ARP replies are those which contain a source IP and destination MAC address combination that does not correspond to a previous request (in other words, each request should correspond to at most one reply, and any extra replies are unsolicited).

Testing

A large sample PCAP file captured from a real network can be downloaded here (you'll need to uncompress the gzip file first). You can examine the packets manually by opening this file in Wireshark. For this input, your program's output should match the following, with the addresses in each section in any order:

```
Unauthorized SYN scanners:
128.3.23.2
128.3.23.5
128.3.23.117
128.3.23.150
128.3.23.158
Unauthorized ARP spoofers:
7c:d1:c3:94:9e:b8
14:4f:8a:ed:c2:5e
```

3.3 Deliverables

Submit detector.go to gradescope.

We will compile your program with Go 1.18. You can assume that gopacket is available, and you may use standard Go system libraries, but your program should otherwise be self-contained. It may not use any other third-party dependencies. We will grade your detector using a variety of different PCAP files. We provide a skeleton file that we encourage you to use, but you are not required to as long as your program in the same manner as above and produces the same output.

4 Part 4: Monster-in-the-Middle Attack

It's time to write an exploit of your own! And explore vulnerabilities arising from a lack of authentication in ARP, DNS, and HTTP protocols.

You will play the part of a network attacker who tricks a victim web browser into visiting your web server instead of the victim's intended site. This is called a monster-in-the-middle attack. You can secretly forward the victim's requests to and from their intended site while easedropping on their conversation. This will involve three steps:

1. Spoof an ARP response to fool the user into using the attacker's device as a DNS server.
    - As you learned in part 3, ARP is not authenticated. This means that anyone on a local network can claim to have any IP address. This will let y ou pretend to be the local DNS server - while the victim is none the wiser.
2. Send a spoofed DNS response to trick the user into associating the hostname fakebank.com with the attacker's IP address instead of its real address.
    - DNS is also not authenticated. The attacker can respond to the victim's DNS request for fakebank.com with a false IP address. This will trick the victim into connecting to the attacker's IP address instead of the real IP address for fakebank.com.
3. Once the DNS response has been successfully spoofed, you will accept a connection from the victim and forward all HTTP requests to and from the actual web server for fakebank.com.
    - HTTP is unencrypted and unauthenticated so a network easedropper can view and even modify any http communication.

4.1 Network Topology

For this assignment, the attacker, victim, and DNS resolver are all connected - they are on the same unencrypted wifi network.

- fakebank.com is the non-malicious website the victim is trying to visit. fakebank.com's real IP address is "10.38.8.3". In your code, you can access this by called `cs155.GetBankIP()`.
- You can get your (the attacker's) IP address by calling `cs155.GetLocalIP()` and MAC address by calling `cs155.GetLocalMAC()`. This is all in `network/network.go`.
- The "real" DNS server has IP address 10.38.8.2
- You may assume that requests originating from anyone else are from the client.

4.2 Attack Requirements

Similar to Part 3, you will use Go and the gopacket library to manipulate packet headers. All code changes will be made in `part4/mitm.go`. We'll walk you through the different parts with detailed "TODO" comments. You should not edit any other file (except maybe to debug).

ARP Attack:

- When the client makes an ARP request for the DNS server's IP address (10.38.8.2), send a spoofed ARP response containing the attacker's MAC address.
- You must send this response in under one second. While there aren't any actual race conditions for this assignment between the attacker's response to the victim and that of the actual DNS resolver, this requirement simulates beating the real DNS resolver.
- You should ignore ARP requests for other IP addresses

DNS Attack:

- When the client queries a DNS A record for fakebank.com, your attack must send a spoofed DNS response containing the attacker's IP address.
- You must ignore DNS queries for other names or record types.

HTTP Attack:

Follow this behavior when receiving a request to each endpoint:

- **/login:** Steal the user's credentials (username and password parameters in the body of the request). Send these to the cs155.StealCredentials function, which will print them to the console.
- **/transfer:** Change "`to`" key's value to Jason and then forward the request to the bank. When responding to the client, reverse this change, so that the to parameter in the response contains the value that the client sent.
- **/kill:** Exit immediately. We already did this for you!
- **other:** Forward traffic as-is without modification, just like a proxy server.
- **all:** Steal all cookies sent by the client or set by the server. That is, any time the client sends the Cookie request header, or the server responds with the Set-Cookieheader, your program must send the cookie name and value to the cs155.StealClientCookie or cs155.StealServerCookie functions, respectively.

4.3 Testing Your Attack

Similar to Project 2, we will use Docker. (Note: If you are on Linux, you will have to run the below commands with sudo privileges.)

1. Build the images running the backend HTTP and DNS servers:

   ```
   $ bash start_images.sh
   ```

   If you modify any of the files in the `network/` directory while debugging, you'll need to re-run this command.

2. Test your mitm.go implementation:

   ```
   $ bash run_client.sh
   ```

   You'll need to re-run this command anytime you make changes to the mitm.go file and want to see the updated output.

3. To stop your images once you're done with the project, run:

   ```
   $ bash stop_images.sh
   ```

Troubleshooting: Try running `docker system prune`. This will clean up files related to previous instances - they might be causing issues with the build process. Similarly, if you'd like to completely remove the unused images and containers from your machine (e.g. when you're done with the project and want to save space), run `docker system prune -a`

4.4 Grading

A fully correct submission will match the output in `correct_mitm_output.txt` with a couple exceptions. The lines referencing tcpdump or packets captured/received/dropped may not match exactly. Also don't worry if you see extra output before the STAGE 1/4 line or after the exit status line.

4.5 Debugging

My programming isn't working and I don't know why :/

Networking applications can be tough to debug! Luckily you already have experience using Wireshark from part 2! When you run `bash run client.sh`, we also capture the network traffic from the run and output to `output/packetdump.pcap`. You can open this pcap in Wireshark to examine what happened.

My program is crashing during login with a timeout/not found error

To make troubleshooting easier, the real DNS server never responds. If your program fails to send a valid DNS response, DNS resolution will fail, and the client will crash during login with an error such as i/o timeout or Answer to DNS question not found.

I'm not sure what the correct behavior is/ what to look for in Wireshark

You may want to temporarily modify the DNS server so that it does respond and then observe what happens. What does the pcap look like in wireshark? If you do, remember to re-run bash start images.sh to re-build your changes, and comment out the functionality again before submitting.

I'm making changes but nothing is changing when I run the attack

Check out "Testing your attack" above and make sure you're running the right commands to rebuild/rerun the project.

4.6 Deliverables:

1. Navigate to the project root directory
2. Call `make submission` to generate `submission.tar.gz`.
3. Submit to Gradescope

We will only look at mitm.go - so do not edit any other files. We will compile your program with Go 1.18. You may use gopacket and any standard Go system libraries, but your program should otherwise be self-contained. It may not have any other third-party dependencies. You may change the structure of mitm.go, but not any of the dependencies (i.e., anything in the cs155 library). You must call the cs155 functions— do not copy that code into your source file or build your own print functions. The autograder may use a different version of the library that provides the same interface.

5 Acknowledgements

This project incorporates project components from the University of Michigan and University of Illinois.