CS155: Computer Security **Spring 2022**

Project 3: Networking

Due: Parts 1–3: May 27 11:59 PM PT
Part 4: June 3 11:59 PM PT

Introduction

This project is all about network security. You will both use existing software to examine remote machines and local traffic as well as play the part of a powerful network attacker. Parts one and two show you how a simple port scan can reveal a large amount of information about a remote server, as well as teach you how to use Wireshark to closely monitor and understand network traffic observable by your machine. Part three will focus on a dump of network traffic in a local network, and will teach you how to identify different types of anomalies. Finally, in part four, you will get to implement a DNS spoofer that hijacks a HTTP connection. This project will solidify your understanding of the mechanics and shortfalls of the DNS and HTTP protocols.

Go Language. This project will be implemented in Go, a programming language most of you have not used before. As such, part of this project will be spent learning how to use Go. Why? As you saw in Project 1, C and C++ are riddled with memory safety pitfalls, especially when it comes to writing network code. It's practically impossible to write perfectly secure C/C++ code—even experienced programmers make errors. For example, here are some vulnerabilities in Exim, a popular open source mail server uncovered by Qualys.

The security community has largely agreed that future systems need to be built in safe languages. Two notable languages have emerged: Go and Rust. Rust is a low-level language that achieves performance similar to C, but has an extreme learning curve. Go, is a much simpler language, and is typically used in less performance-critical environments.

If you haven't used Go before, the official A Tour of Go is a great place to start (in fact, we'll ask you to complete two sections for Part 0 below.) We've also included scaffolding for both parts that use Go (Parts 3 and 4) to give you a sense of how to approach building your solution.

Setup Instructions

1. You will need to install **nmap** locally for Part 1: follow the installation instructions here for Mac users, here for Windows users, and here for Linux users.
2. You will need to install **Wireshark** locally for Part 2: download the Stable Release for your corresponding operating system here.
3. You will need to install **Go** and **gopacket** locally for Part 3: first, 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 `go mod download` - this will automatically download gopacket.
4. Finally, you should already have Docker (used for Part 4) installed from Project 2. If not, take a look at the Project 2 handout for instructions on how to install it.

Part 0: Go Tutorial

If you haven't used Go before, we encourage you to first go through the Go tutorial before getting started with Parts 3 and 4. Specifically, complete the **Basics** and **Methods and Interfaces** sections of the "Tour of Go" tutorial: `https://tour.golang.org/list`. 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. There are no deliverables for this part of the project.

Part 1: Nmap Port Scanning

Port scanning is a method that can be used by an attacker to probe which ports are open on a given host, learning details about which software the server is running on publicly-addressable interfaces. With this information, an attacker gains a better understanding of where and how to attack the 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 (`https://nmap.org`; `https://en.wikipedia.org/wiki/Nmap`) to scan the server **scanme.nmap.org**. By doing so, you should be able to see the powerful information that a simple scan can reveal. In your scan, make sure to:

1. 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.**
2. Record the traffic with Wireshark (see part 2)
3. Use a TCP SYN scan. (Hint: Read the nmap man pages to find the appropriate flag to use.)
4. Enable OS detection, version detection, script scanning, and traceroute. (Hint: This is a single flag.)
5. Do a quick scan (-T4).
6. Scan all ports.

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. When you get the result, report the following about the target server (scanme.nmap.org) based on the results of the scan:

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?

Please answer all questions briefly; no response should take more than a few sentences.

Part 1 Deliverables

PortScanAnswers.txt: A text file containing your answers to questions 1–4 above. Make sure you follow the answer format in the file, since your responses are autograded. If you don't follow the format, you will lose points.

Part 2: Wireshark Packet Sniffing

Wireshark is a tool to monitor 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. Because of this it can be a valuable debugging tool for networking projects, as you will see in Part 4.

Use the Wireshark packet analyzer (https://www.wireshark.org/, https://en.wikipedia.org/wiki/Wireshark) to examine the traffic generated by nmap during the scan in Part 1. You will need to start Wireshark and record traffic on the interface nmap will use to scan before actually running the scan. When you get the result, 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?

Once again, please answer all questions briefly; no response should take more than three or four sentences.

Part 2 Deliverables

WiresharkAnswers.txt: A text file containing your answers to questions 1–4 above.

Part 3: Programmatic Packet Processing

In Part 2, you manually explored a network trace using Wireshark. Now, you will programmatically analyze a PCAP (Packet Capture) file to detect suspicious behavior. Specifically, you will be attempting to identify port scanning and ARP spoofing.

Port Scanning. In Part 1, you used `nmap` to find the open ports on a known host. Port scanning can also be used to find network hosts that have services listening on one or more target ports. It can be used offensively to locate vulnerable systems in preparation for an attack, or defensively for research or network administration. Since most hosts are not prepared to receive connections on any given port, typically, 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 is an attack that 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 initially only knows the IP address of Device B and needs to determine Device B's MAC address to populate the destination MAC address on the Ethernet frame. If Device A does not have this information, it broadcasts an ARP packet to all computers on the local network, asking which device is associated with the IP address for Device B. Normally, Device B would respond with an ARP reply message containing its MAC and IP addresses. Device A then caches this information before sending the packet.

Because ARP packets are not authenticated, any device can claim to have any IP address. Furthermore, most network devices automatically cache any ARP replies they receive, regardless of whether they were ever requested 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 system, thereby performing a man-in-the-middle or denial-of-service attack on other users on the network.

Your task is to develop a Go program that analyzes a PCAP file in order to detect possible SYN scans and ARP spoofing attacks. To do this, you will use gopacket, a library for packet manipulation and dissection, which you should have installed during setup earlier. You can find more information about gopacket at https://godoc.org/github.com/google/gopacket. In particular, you should examine the "layers" Go package to parse different networking layers: https://github.com/google/gopacket/tree/v1.1.19/layers.

Your program will take the path of the PCAP file to be analyzed as a command-line argument:

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

The printed output should begin with the line `Unauthorized SYN scanners:`, followed by the set of IP addresses (one per line) that sent more than 3 times as many SYN packets as the number of SYN+ACK packets they received and also sent more than 5 SYN packets in total. In calculating this, your program should silently ignore packets that are malformed or that are not using Ethernet, IP, and TCP. The line immediately following these IP addresses should print the line `Unauthorized ARP spoofers:`, followed by the set of MAC addresses (one per line) that send 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).

A large sample PCAP file captured from a real network can be downloaded at

https://cs155.stanford.edu/hw_and_proj/proj3/sample.pcap.gz (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
```

Part 3 Deliverables

detector.go: Write a program in Go that accomplishes the task specified above and submit it as `detector.go`. 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 takes a path to the PCAP file in the same manner and produces output identical to as above.

Part 4: Monster-in-the-Middle Attack

Now that you've analyzed previously captured malicious network traffic, it's time to write an exploit of your own. By doing so, you will reveal security issues associated with the lack of authentication in the ARP, DNS, and HTTP protocols and learn how to use powerful packet-manipulation libraries.

In this task, you will play the part of a network attacker who tricks a victim web browser to connect to the attacker's web server instead of the actual site that the victim wanted to visit. By doing so, the attacker can perform a monster-in-the-middle attack to forward the victim's requests to and from the site without being noticed while stealing confidential information along the way. To do so, you will have to spoof an ARP response to fool the user into using the attacker's device as a DNS server. Then, you need to 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. 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`.

Vulnerabilities

The vulnerabilities that you will be exploiting during this attack are the lack of authentication in DNS and ARP and the lack of encryption in plain HTTP. Since ARP is not authenticated, anyone on a local network is able to claim to have any IP address. This means an attacker can respond to ARP requests and pretend to be the local DNS server. Since DNS is not authenticated, the user has

no way of knowing they are not talking with the real DNS server. This lets the attacker respond to a DNS request with a false address, thereby tricking clients into connecting to the wrong IP address for the domain. Combined with the fact that HTTP is unencrypted and unauthenticated, any attacker can secretly intercept and even modify communication between two parties who think they are communicating with each other directly, in what is known as a meddler-in-the-middle attack. This attack demonstrates why HTTPS is so important today.

Network Topology

The topology of the network you're mounting your attack against is shown below. For this assignment, we will act as if the attacker, victim, and DNS resolver are all connected to each other. This situation can happen if the each computer is connected to the same unencrypted wifi network. This allows the attacker to sniff packets between the victim and DNS resolver as well as packets between the client public internet. This also allows the attacker to spoof packets.

Attack Requirements

Similar to Part 3, you will use Go and the gopacket library to manipulate packet headers in a quick and reliable manner. All code changes will be made to `mitm.go` in the `part4/` directory.

The requirements for your attack are as follows:

ARP Attack. When the client makes an ARP request for `10.38.8.2`, your attack must send a spoofed ARP response containing the attacker's MAC address. You must send this response in under one second so as to beat the real DNS server's response. You must 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. Your program must listen for HTTP requests, forward them to the bank server, and return the response from the server back to the client unchanged. In doing so, you must satisfy the following snooping and spoofing guidelines:

1. Whenever a client makes a `POST` request to the bank's `/login` endpoint, your attack must steal the user's credentials, which are the values of the username and password parameters in the body. It must log these by sending them to the `cs155.StealCredentials` function, which will print them to the console.

2. Whenever the client makes a `POST` request to the bank's `/transfer` endpoint, your program must change the `to` value to `Jason` when forwarding this request to the bank. When responding to the client, your program must reverse this change, so that the to parameter in the response contains the value that the client sent.

3. Whenever your program receives any request to the `/kill` endpoint, your program must exit immediately. This requirement has already been implemented for you in the starter code.

4. For all other endpoints, your program must forward traffic as-is without modification, just like a proxy server.

5. For any request made by the client, your program must 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-Cookie` header, your program must send the cookie name and value to the `cs155.StealClientCookie` or `cs155.StealServerCookie` functions, respectively.

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.)

- To build the images that will be running the backend HTTP and DNS servers, first run:

  ```
  $ 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.

- To test your `mitm.go` implementation, run:

  ```
  $ 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.

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

  ```
  $ bash stop_images.sh
  ```

If any of the above commands gives you trouble, try running `docker system prune`. This will clean up files related to previous instances, in case they are 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), do `docker system prune -a`.

Additional Information

- If your program is fully implemented, you should see output as described in `correct_mitm_output.txt`, with the exception that the lines referencing `tcpdump` or packets captured/received/dropped may not match exactly. Don't worry if you see extra output before the `STAGE 1/4` line or after the exit status line.
- For this assignment, assume the attacker controls the router between the victim and the resolver. Thus, there will be no race conditions between the attacker's response to the victim and that of the actual resolver. However, your attack must send a response in under one second. Note that this often would not be the case in reality.
- For your debugging convenience, we capture the network traffic from the `bash run_client.sh` run and output it to the `output/packetdump.pcap` file. You can open that pcap in Wireshark to see what network traffic occurred during the run. You gained some exposure to Wireshark in Part 2, and being able to see exactly what packets are being sent across the network is a powerful debugging tool.
- To help with troubleshooting, we made it so that the real DNS server never responds. Therefore, when 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.

- You may want to temporarily modify the DNS server so that it does respond and then observe what happens, in order to better understand the nature of the attack. If you do so, remember to re-run `bash start_images.sh` to re-build your changes, and **remember to comment out the functionality again** before submitting, because it will mess up your output otherwise.
- **Hint:** Think about the code that exists in the provided DNS and HTTP server and how that might relate to what the attacker is emulating in their attack.

Part 4 Deliverables

mitm.go: You should submit your standalone `mitm.go` source file. 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. 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.

Submitting

1. Make sure the deliverables mentioned in this writeup for part[1,2,3,4] are in their respective directories; misplaced deliverables may not receive credit. **Many items will be auto-graded, so be careful to follow the given formats exactly.**
2. Run the following command from the project root directory to generate the submission tarball submission.tar.gz: `make submission`.
3. Upload the submission tarball to Gradescope.

Acknowledgements

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