



# CS241 Coursework 2021-2022

Released on November 5, 2021


Deadline: December 7, 2021 @ 12 noon

## 1. The Task

For this coursework, you will implement a **basic intrusion detection system**. This will test your understanding of **TCP/IP protocols** (networks) and **threading** (OS) as well as your **ability to develop a non-trivial program in C**. The coursework contributes **20%** to your total marks in the module.

You have been provided with an application skeleton that is able to intercept (sniff) packets at a specific interface and print them to the screen. The code uses the **libpcap** library to receive packets and strips the outer-most layer of the packet. The goal of this coursework is to extend the skeleton to detect potentially malicious traffic in high-throughput networks. The key deliverables of this coursework and their associated weightings are as follows.

- Extend the skeleton to efficiently intercept internet packets and navigate through the packet headers. (~20%)
- Detect specific malicious activities (e.g., SYN attacks, ARP cache Poisoning attack, Blacklisted URL deection) and show them on the terminal as specified later. (~25%)  
(relative weights: SYN attack detection 50%, ARP poisoning attack detection 25%, Blacklisted URL detection 25%)
- **Implement a threading strategy** to allow efficient detection of the attacks when there is high traffic. (~25%)
- Write a report no more than 1000 words in length (excluding references) explaining the critical design decisions and testing of your solution. The report should be short, mainly containing a description of your threading strategy and implementation, a justification for your choice of the threading model, and how you have tested your solution. (~20%)
- The final ~10% is awarded for code quality and adherence to relevant software engineering principles.

You must base your solution on the [skeleton](#)  provided and it must be written entirely in the C programming language.

Your solution must compile and run without errors, warnings, memory leaks, and without crashing (e.g., seg faults) on **the DCS system**. If this is not the case, then marks will be lost.

You should **only consider IPv4** - there are no additional marks available for IPv6 functionality.

You may choose to use appropriate academic or industrial literature, which **must be referenced appropriately** in your report. When writing an academic report, you should not write in first person (i.e., **Don't write "I did this, I did that, etc."** rather use *"This was done.."* or *"We did this.."*).

## 2. Step-by-Step setup instructions

This tutorial (for setting up the VM and running the skeleton) assumes that (1) you are already logged into the DCS system, (2) have downloaded and extracted the [skeleton code](#), and (3) have the skeleton in a directory as follows

```
~/cs241/skeleton
```

If you haven't done the steps above, please do so before following the tutorial.

You will not be able to run the skeleton directly in the DCS system as it requires you to have root permissions to access the network interfaces. Since you don't have root permissions in the DCS system, you will first need to create a virtual machine (VM) where you do have root permissions. Once the VM is created, you can run the skeleton within the VM. The following steps will guide you through the process of creating and setting up the virtual machine (VM) and running the skeleton.

**Step 1 (Create the VM):** Run the following command in the terminal

```
/courses/cs241/coursework-multi-test2021 &
```

This will create the VM and boot it. It also creates a copy-on-write (COW) file to automatically store the changes that you make to the VM. The output of the command looks something like

```
Creating COW file for courseworkFormatting '/dcs/acad/u1872683/cs241-qemu/cs241vm_d11.cow', f
done.
copying SSH key...
SSH server will start on port 3187, to connect use:
    ssh -p 3187 root@localhost

VNC server will start on port 7187, to connect use ( password= Xaepu9re )
    vncviewer localhost:7187
Running VM...
```

These details give you the information required for logging into the VM using SSH and VNC.

**Step 2 (Log into the VM):** You can now log into the VM from the terminal using ssh by issuing the following command

```
ssh -p 3187 root@localhost
```

Please note that you should use the port number shown in the output of the previous step (in this case shown above it is 3187).

This command will take you to the command prompt of the VM you have just created. Remember that you have root access to this VM. This is crucial for this coursework as it allows you to access the different network interfaces in the VM. You can see the list of available network interfaces in the VM by issuing the following command

```
ifconfig
```

You should see two interfaces named 'eth0' and 'lo' in the VM. The interface 'eth0' is the external interface and is used to access the internet. The other interface 'lo' is the loopback interface.

**Step 3 (Create a working directory):** Type the following command to create a working directory called 'cs241' in the home folder of the VM

```
mkdir cs241
```

This will be your working directory within the VM.

**Step 4 (Mount the directory containing the skeleton):** Now, mount the '~/cs241' directory in the DCS system into the '~/cs241' directory of the VM using the following command

```
sshfs u2037230@remote-30.dcs.warwick.ac.uk:cs241 cs241
```

This command will mount the '~/cs241' directory from your DCS system to the '~/cs241' in the VM. Hence, the skeleton should now be available from the VM. From this point onwards, any changes you make to the cs241 directory in the DCS system will be automatically available to the working directory of the VM and vice versa.

**Step 5 (Building the skeleton):** Change to the folder '~/cs241/skeleton/src' and run

```
make
```

to build the project. This will create a new folder '~/cs241/skeleton/build' where the main executable file 'idsniff' will be stored.

**Step 6 (Running the skeleton):** To run the skeleton from the 'src' directory issue the following command

```
../build/idsniff -i <interface>
```

Replace <interface> with the name of the interface (either eth0 or lo) at which you wish to capture packets. This will start a packet capturing session at the specified interface. If no interface is specified, i.e., if the -i option is not used, then the program will assume the default interface name eth0 for the interface. The program will continue to run unless you kill it by pressing Cntrl+c .

The skeleton code will not show any captured packets unless you run it with the -v option and set the verbose flag to 1 as shown below

```
../build/idsniff -i <interface> -v 1
```

If the verbose option is used, then the program will print the contents of all packets captured at the interface. This option may be useful for debugging purposes.

The setup process is now complete. If you do not use the VM for a long time, it will shutdown automatically. You can re-boot the VM and log into it again following steps 1 and 2 above. The changes you make to the VM are not lost when the VM shuts down. So when you restart it, you should still see the folder cs241 within the VM. However, you still need to mount the '~/cs241' directory from the DCS system to the VM (Step 4) everytime the VM is restarted.

## Re-creating the VM

If you need to re-create your virtual machine for whatever reason, you can do so by killing the VM process (if it is running) and removing the COW file. To kill the VM process, run

```
ps -aux | grep $USER | grep kvm
```

This will print the pid of the kvm process running on your account. You can kill these process using

```
kill <pid>
```

with <pid> replaced by the actual pid of the kvm process. Then delete the VM by using

```
rm -rf ~/cs241-qemu/
```

You can then follow Steps 1-6 above again to set up a fresh copy of the VM.

Note that the skeleton code only starts a packet capturing session on the specified interface. But it doesn't filter any suspicious packet. Your job will be to extend the skeleton to detect some suspicious packets (as specified below) and prepare a summary of all suspicious activities when the program is killed using `Cntrl+c`.

Your solution should not require the installation of any additional packages or libraries. However, if you would like to use any additional applications to help you with this coursework, you are welcome to install them. Many common Debian applications can be installed by using `apt-get` as follows

```
apt-get install <package_name>
```

## Working Remotely

For remotely accessing the DCS system please follow the instructions given [here](#).

## 3. Study the Skeleton

The coursework skeleton consists of several files, each with a specific purpose:

### 3.1 Makefile

As this project spans multiple files we have provided a makefile to automate the build process for you. To compile your solution you should change into the `src` directory and then run the `make` command. This will build the application binary `../build/idsniff`. *Your solution should not require changes to this file.*

### 3.2 main.c Do not change this file

This is the application's entry point. It contains logic to parse command line arguments, allowing you to set the verbose flag and specify the network interface to sniff. It also calls the `sniff()` function which starts a packet capturing session on the specified interface. *Your solution should not require changes to this file.*

### 3.3 sniff.c

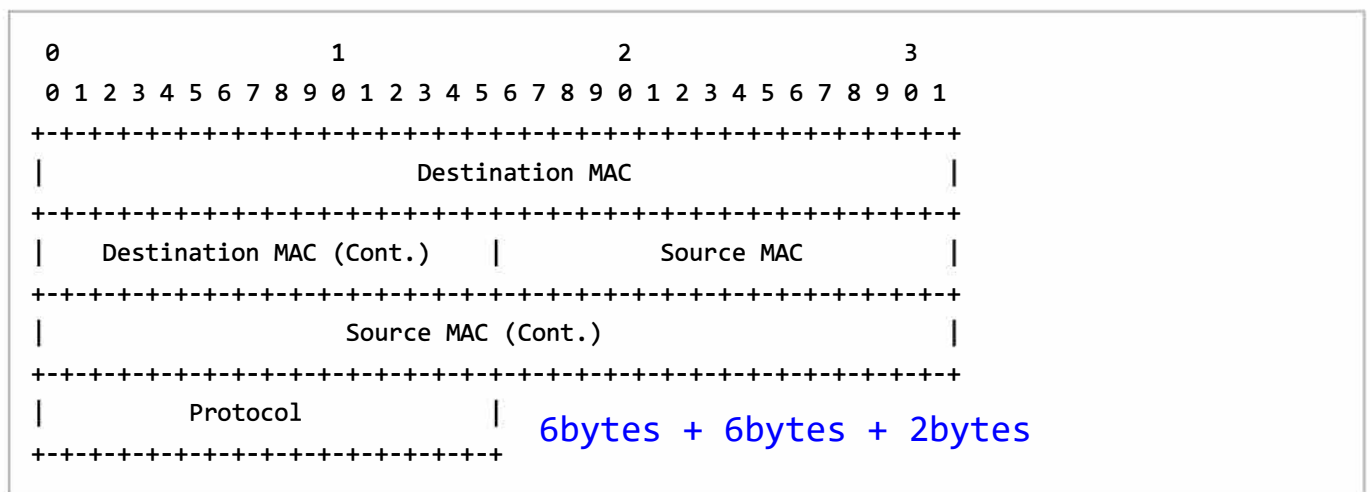
This file contains the `sniff()` function which starts a packet capturing session at the specified network interface using the pcap library functions `pcap_open_live()` and `pcap_next()`. You should understand how these pcap library functions work by looking at their man pages. You can also look at the nice tutorial on pcap given [here](#). Note that the skeleton captures one packet at a time using `pcap_next()` and passes the packet on to the

dispatch() function. In the final solution, instead of capturing packets one at a time using pcap\_next(), you should use pcap\_loop() which will continue to capture packets until a specific condition is met. This simplifies the implementation and also makes packet capturing more efficient.

The function `dump()` is used to print the packet data in ASCII when verbose mode is enabled ( `-v 1` ). **You should study this function carefully** as it demonstrates how to parse a packet header. The idea is to use a structure pointer that points to the beginning of the packet header you want to parse. The fields within the structure should exactly match the fields within the packet header. Aligning the beginning of the structure to the beginning of the packet header, therefore, will automatically extract the required fields of the packet header into the corresponding fields of the structure pointer. This idea is explained in more detail below:

Suppose we want to parse the Ethernet header of a packet (as is done in the `dump()` function). Recall from the lectures that the Ethernet header is outermost header of a packet and has a fixed length of 14 bytes.

Furthermore, it has the following format (note one tick mark represents one bit, meaning there are 32 bits = 4 bytes per complete line) :



Due to C's contiguous memory layout guarantees, the fields of a struct can be arranged to match the fields within a captured packet. We can therefore define a struct which maps to this format, or use the one provided in `<netinet/if_ether.h>`: (You can see its contents by issuing the command `cat /usr/include/netinet/if_ether.h`)

```
// if_ether.h excerpt

#define ETH_ALEN 6 /* Octets (bytes) in one ethernet addr */
#define ETH_HLEN 14 /* Total octets in header */

struct ether_header {
    u_char ether_dhost[6];
    u_char ether_shost[6];
    u_short ether_type;
};
```

It is easy to see how the struct maps to the ethernet format - 6 bytes for the destination address, 6 for the source address and 2 bytes for the protocol.

Once we have this struct defined we can use it to read values directly from the packet data:

```
struct ether_header * eth_header = (struct ether_header *) data; //make the structure pointer
printf("\nType: %hu\n", eth_header->ether_type);
```

The code above shows you how to parse the outermost layer of the packet and access members of the ethernet header. You will need similar logic to parse the IP and TCP headers. This will be required to detect 'malicious packets' (to be defined later). The structures of the IP and TCP headers can be found [here](#) and [here](#), respectively. There are built-in data structures such as `struct tcphdr` (defined in 'netinet/tcp.h') to parse the TCP header and `struct iphdr` and `struct ip` (defined in 'netinet/ip.h') to parse the IP header. To see the fields within these structures, look into these header files located in your system at "/usr/include/netinet/ip.h" and "/usr/include/netinet/tcp.h" using the `cat` command.

An important point to remember when parsing multi-byte fields within packet headers is the use of network and host byte orders. In order for different machines to communicate, a standard ordering of bytes for multi-byte data types (e.g short and int) must be observed. This is because some machines place the most significant byte first (big-endian) while others place the least significant byte first (little-endian). To allow such diverse machines to communicate over the internet, the internet always uses the standard network byte order which happens to be big-endian. With this in mind, when multi-byte values are read from a socket they must be converted from network byte order, to the host byte order (the byte-order that the host uses). Since `ether_type` is a multi-byte value (a short), it will need to be converted before being printed or used in any comparisons. To do this, the function `ntohs` can be used as follows.

```
#include <netinet/in.h>
...
unsigned short ethernet_type = ntohs(eth_header->ether_type);
```

### 分析数据包，识别恶意数据包。对于接口捕获的每个包都要调用analyse() 3.4 analysis.c

This file is where you should put code to analyse packets and identify malicious packets. The `analyse()` function should be called for each packet captured at the interface. Currently the function does nothing. Your code should analyse each packet to determine if the packet is malicious (according to the specification given below).

### 3.5 dispatch.c

This file is where you should put code to `parallelise your system`. At the moment, the `dispatch()` simply calls `analyse()` in `analysis.c`. This sequential, single-threaded behaviour should be replaced by code to distribute work over multiple threads.

## 4. Specification of malicious packets

In this project, you will detect the possibility of three different network attacks. They correspond to three different types of internet packet. In this section, we specify the network attacks and the corresponding packets your code should detect. The output of your program should look something as shown below when the code is killed using `Cntrl+c`.

```

../build/idsniff invoked. Settings:
    Interface: eth0

    Verbose: 0
SUCCESS! Opened eth0 for capture
=====
Blacklisted URL violation detected
Source IP address: 10.0.2.15
Destination IP address: 212.58.237.249
=====
=====
Blacklisted URL violation detected
Source IP address: 10.0.2.15
Destination IP address: 142.250.200.3
=====
=====
Blacklisted URL violation detected
Source IP address: 10.0.2.15
Destination IP address: 142.250.200.3
=====
^C
Intrusion Detection Report:
4 SYN packets detected from 1 different IPs (syn attack)
2 ARP responses (cache poisoning)
3 URL Blacklist violations

```

## 4.1 SYN Flooding Attack

This attack is achieved when a server listening on a TCP socket is flooded with TCP SYN packets (packets whose SYN bit is set to 1 and all other flag bits are set to 0). For each received SYN packet, the server opens a TCP connection, allocates some resources, replies with a SYN-ACK packet and then waits for an ACK from the sender. However, the malicious sender does not send the ACK. This creates a half-open TCP connection at the server which occupies some resources. As the attacker sends many such SYN packets, the server resources get used up and as a result, legitimate connection requests are dropped. This is a form of **denial-of-service attack**. In most cases, the attacker generates the SYN packets from spoofed IP addresses. Spoofed IP addresses are **randomly generated** and do not correspond to the attacker's real IP address. They are used to hide the attacker's own identity.

Your job is to count

1. the **total number of SYN packets sniffed**,
2. and the number of them **having a unique source IP address**

and finally, print a report (as shown above) containing this information.

**Hint:** To implement the above, you may need to use a dynamically growing array to store the source IP addresses of the incoming SYN packets. This array can be processed upon receiving the Cntrl+C signal to get the number of unique IP addresses stored in the array.

**Testing your code:** To test your code, you can send SYN packets to your loopback (lo) interface or localhost using `hping3` on one terminal window while the sniffer is listening to the lo interface in another terminal window. You can issue the following command

```
hping3 -c 100 -d 120 -S -w 64 -p 80 -i u100 --rand-source localhost
```

This will send 100 packets ("-c 100" option) of a size of 120 bytes ("-d 120" option) each with the SYN Flag ("-S" option) enabled, with a TCP window size of 64 (-w 64) to port 80 (-p 80) of localhost at an interval of 100 microseconds between two consecutive packets ("-i u100" option). With the "--rand-source" option, the source IP addresses are randomly generated. The output of the sniffer program should look something as shown below when killed using Cntrl+c .

```
#../build/idsniff -i lo
../build/idsniff invoked. Settings:
    Interface: lo
    Verbose: 0
SUCCESS! Opened lo for capture
^C
Intrusion Detection Report:
100 SYN packets detected from 100 different IPs (syn attack)
0 ARP responses (cache poisoning)
0 URL Blacklist violations
```

## 4.2 ARP Cache Poisoning

The Address Resolution Protocol (ARP) is used by systems to construct a mapping between network layer (Media Access Control) and link layer (Internet Protocol) addresses. Consider a simple scenario: two systems share a network - *dcs\_laptop* has IP address 192.168.1.68 and is trying to communicate with *broadband\_router* at 192.168.1.1. To achieve this, *dcs\_laptop* broadcasts an ARP request asking for the MAC address of the node at 192.168.1.1. When *broadband\_router* sees this message it responds with its MAC address. *dcs\_laptop* will cache this address for future use and then use it to establish a connection.

The ARP protocol has a serious flaw in that it performs no validation. An attacker can craft a malicious ARP packet which tricks the router into associating the ip address of *dcs\_laptop* with the attacker's own MAC address. This means all traffic bound for *dcs\_laptop* will be redirected to the attacker, potentially exposing sensitive data or allowing for man-in-the-middle attacks. To make matters worse, ARP allows unsolicited responses, meaning *dcs\_laptop* does not even have to send out a request - an attacker can simply broadcast a message informing all nodes to send *dcs\_laptop* traffic to their machine.

Although ARP messages can be legitimate, the use of caching means they should be very rare. A burst of unsolicited ARP responses is a strong indication that an attacker has penetrated a network and is trying to take it over. You should add code which detect **all** ARP responses.

**Hint:** there is an ether\_arp struct defined in `netinet/if_ether.h`

**Testing your code:** You have been provided with a python script which you may find useful when testing your code. The `arp-poison.py` script can be found in the `test` directory. This can be run with the following command

```
python3 arp-poison.py
```

This should send a suspicious arp response packet to the loopback interface. The sniffer should be able to detect this packet if it is listening on the loopback interface (-i lo). The output of the sniffer should look something like the following:



```
../build/idsniff invoked. Settings:
    Interface: lo
    Verbose: 0
SUCCESS! Opened lo for capture
^C
Intrusion Detection Report:
0 SYN packets detected from 0 IP (syn attack)
1 ARP responses (cache poisoning)
0 URL Blacklist violations
```

## 4.3 Blacklisted URLs

Intrusion detection systems typically watch traffic originating from the network they protect in addition to attacks coming from outside. This can allow them to detect the presence of a virus trying to connect back to a control server for example, or perhaps monitor any attempts to smuggle sensitive information to the outside world. For this exercise, we have identified `www.google.co.uk` and `www.bbc.com` as suspicious domains that we wish to monitor. Specifically, we wish to be alerted when we see HTTP traffic being **sent** to these domains.

You should first filter TCP packets that are sent to port 80 (i.e. the HTTP port). A subset of these packets will be HTTP requests to the blacklisted domains. The contents of such a malicious HTTP request may look something like (note that the escape characters will not be displayed if printed):

```
GET / HTTP/1.0\r\n
User-Agent: wget/1.11.4\r\n
Accept: */*\r\n
Host: www.google.co.uk\r\n
Connection: Keep-Alive\r\n
```

Your code should look for the appropriate string within the HTTP header to detect if the HTTP request is malicious.

**Testing your code:** One way to test your code for blacklisted URL detection is to use the `wget` command. On one terminal window you can run your sniffer code on the `eth0` interface and in another terminal window run the following commands (one at a time)

```
wget www.google.co.uk
wget www.bbc.com
```

The above commands will send two HTTP requests to the blacklisted domains. The output of the sniffer upon pressing `Cntrl+c` should look something as shown below:

```
# ../build/idsniff -i eth0
../build/idsniff invoked. Settings:

    Interface: eth0
    Verbose: 0
SUCCESS! Opened eth0 for capture
=====
Blacklisted URL violation detected
Source IP address: 10.0.2.15
Destination IP address: 142.250.200.3
=====
=====
Blacklisted URL violation detected
Source IP address: 10.0.2.15
Destination IP address: 212.58.237.249
=====
^C
Intrusion Detection Report:
3 SYN packets detected from 1 different IPs (syn attack)
1 ARP responses (cache poisoning)
2 URL Blacklist violations
```

Note that your program should output the source and destination IP addresses of each malicious HTTP request as soon as it is detected (even before the program is killed using `Cntrl+c` ). The total number of blacklist violations should be printed only after killing the program (as shown above).

## 4.4 A note on PCAP filters

PCAP exposes a domain specific language which allows you to specify packet filters. **This feature should not be used to complete the coursework. You must implement the logic to access and process packet headers manually.** A central aim of this coursework is for you to become familiar with the network stack; marks will be deducted from solutions that rely on external parsing or filtering logic.

## 5. Multithreading

Intrusion detection systems often monitor the traffic between the global internet and large corporate or government networks. As such they typically have deal with massive traffic volumes. In order to allow your system to handle high data rates you should make your code multi-threaded. There are several strategies you could choose to adopt to achieve this. Two common approaches are outlined below. Whatever approach you choose to implement you must remember to justify your decision in your report. For this work we will focus on POSIX threads you were introduced to in lab 3. In order to use POSIX threads, the `lpthread` linker flag must be added to the project makefile like so (should be done already):

```
LD_FLAGS := -lpthread -lpcap
```

### 4.3.1 One Thread per X Model

This approach to threading creates a new thread for each unit of work to be done (in our case our X is each packet to process) and is probably still the most common approach to threading. This model is sometimes called the Apache model after the Apache webserver which by default gives each client connection a dedicated

thread. The strength of this model can be found in its simplicity and low overhead when dealing with constant light loads as no threads are kept idle. The downside to this approach is that it scales poorly under heavy or bursty load.

### 4.3.2 Threadpool Model

This approach creates a fixed number of threads on startup (typically one or two per processor core). When a packet arrives it is added to a work queue. The threads then try to take work from this queue, blocking when it becomes empty. The strength of this approach is that it deals better in bursty or heavy traffic scenarios as it removes the need to create threads dynamically and limits the number of threads active at any given time, avoiding thrashing. Its weakness stems from the added implementation complexity.

Your threading code should mainly be placed in `dispatch.c` and `sniff.c`. Whichever model you choose you should deal with thread creation and work allocation here. You may find that you also have to make minor modifications to `analysis.c` to make your code threadsafe. In particular, you should be careful when storing intrusion records or dealing with dynamic arrays to avoid any lost updates or race conditions.

## 5. Submission

The deadline for submission is **December 7, 2021 @ 12pm (Noon)**. Normal late submission penalties will apply. You are expected to submit your solution via Tabula.

Please submit a single zip file containing

- The source files which make up your application (please include all source code (both `.c` and `.h`), even though some will be unchanged, so we can easily compile your code).
- A PDF file called "**Report.pdf**" containing your report.

If you have unsuccessfully attempted a multi-threaded version, you may submit your non-working version in a clearly labelled subdirectory of the zip file.

## 6. Useful References

- [RFC 791 - Internet Protocol](#)
- [RFC 792 - Internet Control Message Protocol](#)
- [RFC 793 - Transmission Control Protocol](#)
- [IEEE 802.3 - Ethernet](#)
- [RFC 2616 - Hypertext Transfer Protocol](#)
- [SYN flooding attacks](#)
- [Linux signals](#)
- [Useful resource for PCAP](#)

## 7. Module Tutors

Please contact the TAs of the module for any questions regarding the coursework.[remote-30.dcs.warwick.ac.uk](mailto:remote-30.dcs.warwick.ac.uk)

Department of Computer Science,  
University of Warwick, CV4 7AL  
E-mail: [comp-sci@dcs.warwick.ac.uk](mailto:comp-sci@dcs.warwick.ac.uk),  
Telephone: [+44 \(0\)24 7652 3193](tel:+4412476523193)

[DCS Intranet](#)

---

Page contact: Arpan Mukhopadhyay

Last revised: Thu 18 Nov 2021

Powered by Sitebuilder © MMXXI [Terms](#) [Privacy](#) [Cookies](#) [Accessibility](#)

## [Coronavirus \(Covid-19\): Latest updates and information](#)