Network Attacks

In this project, you will use low-level networking primitives to sniff raw network packets, maliciously terminate preexisting TCP streams, and inject attacker-controlled data into preexisting TCP streams. This project will demonstrate the relative ease with which these attacks can be launched.

More specifically, you will attack TCP streams that flow between two endpoints that reside on your local machine: one endpoint will be within a virtual machine, and the other endpoint will reside on the host machine that runs the VM. This setup is sufficient for you to launch real attacks without endangering the outside world.

> **Caution!**
>
> Obviously, **YOU SHOULD NOT LAUNCH THESE ATTACKS ON REAL NETWORKS!**

Project Setup

- Click on the provided GitHub Classroom assignment link, login via GitHub if necessary, and click "Accept assignment".
- Wait for the repository to be created, then create a fork.
- Enable Travis builds for the fork, and turn on "Auto cancel branch builds".
- Login to the VM.
- `cd` to your home directory and run `git clone <fork_url> network-attacks/` to clone your fork.
- Run `cd network-attacks/` to enter the project directory.
- Run `./pre_setup.sh` to download dependencies.

Refer to Project 0's writeup for elaboration on any of these steps.

> **Caution!**
>
> Before starting, remember the warning from Project 0:
>
> It is important that you **do not** modify the original repository created by GitHub Classroom (the one with `harvard-cs263` in the URL). Only ever modify the fork. This applies to **all** projects in this course.

> **Note**
>
> The project may work in other Linux environments; however, the course staff will only support the course VM, and you are solely responsible for making sure your code passes Travis tests.
>
> If you didn't follow the Project 0 setup instructions, some of the constants in the specification won't make sense for your system. In this case, you will need to replace the following:
>
> - `192.168.26.3` : The VM's IP on the hypervisor's host-only network
> - `192.168.26.1` : The host's IP on the hypervisor's host-only network
> - `eth0` : The VM's interface to the hypervisor's host-only network
> - `eth1` : The VM's interface with internet access (e.g. NAT).

Host Setup

Because this project involves your host machine as well, you will need to make sure its environment is suitable. Specifically:

- Install Python 2.7.
- Clone your fork to wherever you'd like.
- [32-bit Windows only] From the cloned directory, run `pip install curses-2.2-cp27-none-win32.whl`
- [64-bit Windows only] From the cloned directory, run `pip install curses-2.2-cp27-none-win_amd64.whl`

Specification

Caution!

For all projects, trying to modify or otherwise game the test cases will result in a grade of zero and academic dishonesty sanctions. Contact the course staff if you encounter issues with the tests.

Tip

For all projects, you may commit and push your changes at your leisure. Each push will trigger a remote test, which you can view on the Travis website.

Important

Throughout the project, you will probably need to change the `Makefile`. Although your `main()` functions should be in the specified `.c` files, you are free (and encouraged) to create other source and header files as you see fit -- be sure to modify your `Makefile` accordingly.

Important

Take a look at `sniffer.h` before continuing to read this document.

The file `sniffer.h` defines various data structures and constants that your code should use. For example, `sniffer.h::struct tcp_hdr` describes the various fields in a TCP header, e.g., the destination port, the source port, the sequence number, and so on.

Important

Read the entire document before you start coding.

At various points, the document will suggest ways to efficiently structure your code so that your overall coding effort is minimized. Please follow the suggestions!

Part 1: Sniffing Packets

Before you can inject packets into the network, you must learn how to inspect preexisting network traffic. So, your first job is to write a network sniffer. Your sniffer will be a simplified version of tools like WireShark.

Place the `main()` function for your sniffer in `sniffer.c`. The program should be invocable like this:

```
sniffer [dev_name]
```

Here, `dev_name` is an optional argument that specifies the network device that the sniffer should attach to. Examples of Linux network devices include `eth0` (which belongs to a machine's Ethernet card) and `lo` (the loopback device, which is useful for network tests in which a machine sends packets to itself). If this optional argument is not provided, the program should listen to the default Ethernet device.

The program should read and log each packet on the device according to the section Logging Format. Furthermore, it should handle signals according to the section Handling Signals.

Important

Note that all programs that you create will require root privileges to run, e.g.:

```
sudo ./sniffer lo
```

The reason is that reading and writing raw packets is a privileged operation.

Tip

All programs that you write for this project will require you to use libpcap to sniff packets. So, before you start working on `sniffer.c`, you should write a utility library which defines functions to open a new `pcap_t` handle, and close a preexisting `pcap_t` handle. When you open the handle, you should:

- Set the handle to promiscuous mode, so that your sniffer will receive all of the packets that are transmitted on the wire, not just the ones that are destined for your VM.
- Set the snapshot length to 64KB so that you can capture full packets, including all headers and all data.

For more details on how to open and close PCAP handles, see the official PCAP documentation.

Logging Format

You should use `pcap_next_ex()` (**not** `pcap_loop()`) to read a raw packet. **Be sure to handle all of the possible return values for** `pcap_next_ex()`:

- 1 means that a packet was successfully read.
- 0 means that libpcap's read timeout expired; this is not a hard error, so your sniffer code should just return to the beginning of the sniffing loop.
- -1 indicates a libpcap error which should cause your sniffer to terminate.
- -2 means that `pcap_breakloop()` has been called, meaning that your sniffer should exit its sniffing loop and gracefully shutdown. We discuss `pcap_breakloop()` in more detail in the section Handling Signals.

`sniffer.h` defines structs for various types of network headers. For each packet that your sniffer finds, your sniffer should log various pieces of information for each header:

Ethernet header: Log the source and destination Ethernet addresses. Log these values like this:

```
ETHERNET: src[02:63:de:ad:be:ef] dst[02:63:fe:ed:fa:ce]
```

IP header: Log the source and destination IP addresses; the length of the IP header; the length of the data (i.e., non-header) part of the IP packet; and the protocol (which will be either `sniffer.h::IP_ICMP`, `sniffer.h::IP_TCP`,

`sniffer.h::IP_UDP`, or another protocol which you can just log as "other"). Log these values like this:

```
IP: src[26.3.26.3] dst[3.26.3.26]
    ip_hdr_len[20] ip_data_len[24] Protocol: IP_TCP
```

Note the indent of the second line!

TCP header: If a packet contains TCP data, then your sniffer should log the source and destination port, the sequence and acknowledgment number for the packet, the length of the TCP header, the length of the data (i.e., non-header) part of the packet, and any flags that the TCP header has (e.g., `sniffer.h::TCP_SYN`). If the TCP segment contains any data, you should also log that data, writing one or more lines in which each line contains 16 characters of TCP data printed using the `"%c"` `printf()` modifier. For example, here's an example line of output that your sniffer might generate for the final message in the 3-way TCP handshake:

```
TCP: src_port[8181] dst_port[42870]
     seq_num[205568001] ack_num[3018600331]
     tcp_hdr_len[24] tcp_data_len[0] flags: SYN ACK
```

Note the indent of the second and third lines! Also note that, in this case, the TCP segment contained no data.

Here's the line that your sniffer might generate for the initial part of an HTTP request:

```
TCP: src_port[36696] dst_port[80]
     seq_num[1566988577] ack_num[19264002]
     tcp_hdr_len[20] tcp_data_len[138] flags: PUSH ACK
User-Agent: Wget
/1.17.1 (linux-g
nu)
Accept: */*

Accept-Encodin
g: identity
Hos
t: www.cnn.com

Connection: Keep
-Alive
```

In this case, the TCP segment **did** contain data. Note that data lines are unindented!

ICMP header: If a packet contain ICMP data, then your sniffer should log the type of the message (i.e., `sniffer.h::ICMP_ECHOREPLY`, `sniffer.h::ICMP_ECHO`, or another type which you can just log as "other"), the id of the message, and the sequence number of the message. Log these values like this:

```
ICMP: type[ICMP_ECHO] id[18572] seq[3]
```

UDP header: You do not need to log anything extra for UDP datagrams.

So, putting it all together, here's an example of what your sniffer might output for a pair of ICMP echo request/echo reply messages:

```
ETHERNET: src[08:00:27:16:b3:17] dst[5b:54:11:12:35:02]
IP: src[10.0.2.15] dst[192.168.26.3]
ip_hdr_len[20] ip_data_len[64] Protocol: IP_ICMP
ICMP: type[ICMP_ECHO] id[18617] seq[3]

ETHERNET: src[5a:54:11:12:35:02] dst[08:00:27:16:b3:17]
IP: src[192.168.26.3] dst[10.0.2.15]
```

```
ip_hdr_len[20] ip_data_len[64] Protocol: IP_ICMP
ICMP: type[ICMP_ECHOREPLY] id[18617] seq[3]
```

As another example, suppose that you issue the command `wget www.cnn.com`. That command will use the HTTP protocol to fetch the CNN homepage. Your sniffer will capture the 3-way TCP handshake, as well as the raw TCP data that represents the HTTP request and response. For example, the first five packets that your sniffer logs will look something like the following:

```
ETHERNET: src[08:00:27:16:b8:30] dst[52:54:00:12:35:02]
IP: src[10.0.2.15] dst[151.101.116.73]
    ip_hdr_len[20] ip_data_len[40] Protocol: IP_TCP
TCP: src_port[36696] dst_port[80]
     seq_num[1566988576] ack_num[0]
     tcp_hdr_len[40] tcp_data_len[0] flags: SYN

ETHERNET: src[52:54:00:12:35:02] dst[08:00:27:16:b8:30]
IP: src[151.101.116.73] dst[10.0.2.15]
    ip_hdr_len[20] ip_data_len[24] Protocol: IP_TCP
TCP: src_port[80] dst_port[36696]
     seq_num[19264001] ack_num[1566988577]
     tcp_hdr_len[24] tcp_data_len[0] flags: SYN ACK

ETHERNET: src[08:00:27:16:b8:30] dst[52:54:00:12:35:02]
IP: src[10.0.2.15] dst[151.101.116.73]
    ip_hdr_len[20] ip_data_len[20] Protocol: IP_TCP
TCP: src_port[36696] dst_port[80]
     seq_num[1566988577] ack_num[19264002]
     tcp_hdr_len[20] tcp_data_len[0] flags: ACK

ETHERNET: src[08:00:27:16:b8:30] dst[52:54:00:12:35:02]
IP: src[10.0.2.15] dst[151.101.116.73]
    ip_hdr_len[20] ip_data_len[158] Protocol: IP_TCP
TCP: src_port[36696] dst_port[80]
     seq_num[1566988577] ack_num[19264002]
     tcp_hdr_len[20] tcp_data_len[138] flags: PUSH ACK

User-Agent: Wget
/1.17.1 (linux-g
nu)
Accept: */*

Accept-Encodin
g: identity
Hos
t: www.cnn.com

Connection: Keep
-Alive
```

As you build your sniffer, remember to think about byte endianness! When you print a number inside a network header that was captured by libpcap, you'll often need to convert that number into the host byte order using a function like `ntohs()` or `ntohl()`. Functions like `inet_ntoa()` may also be useful when you need to print IP addresses in dotted quad notation like `127.0.0.1`. Be careful **not** to use `ether_ntoa()`, as this will fail to print the leading zero for each Ethernet address byte.

Many of you will be SSH'ing into the VM so that you can develop and test your code. By default, your scanner would log the TCP traffic that belongs to your SSH session! This SSH traffic would add a bunch of noise to your sniffer's output logs. So, `sniffer.c` should use `pcap_setfilter()` to ignore TCP traffic that involves port 22 (i.e., the SSH port).

> **Tip**

We recommend that you place your header extraction code and your logging code in two separate utility libraries. For example, the header extraction library would define functions that take a `u_char *` pointer to raw packet data and return pointers to various network headers. The logging code would define functions that take in a pointer to a network header and print the relevant parts of the header. By placing the header extraction and logging code in libraries, you make it easy for your attack programs to use the header extraction and logging functionality.

Handling Signals

Your sniffer program should handle the `SIGINT` and `SIGQUIT` signals gracefully. To do so, use `sigaction()` from the Linux `<signal.h>` header to register a signal handler for the signals. The signal handler should simply call `pcap_breakloop()`. `pcap_breakloop()` will cause `pcap_next_ex()` to return -2. In turn, this should cause your packet sniffing loop to exit, at which point you can gracefully close the `pcap_t` handle and deallocate any other resources that were created during the sniffing session.

Tip

You should put your code for signal handling into a separate library, so that it can be used by your attacks as well.

Testing Your Sniffer

First of all, `make sniffer` should successfully compile the program.

Try the following experiments to test your packet sniffer:

- Bind your sniffer to the `lo` interface, and then issue a `ping localhost` command. Your sniffer should log ICMP echo request and echo reply messages. Note that ping tools often use the PID of the ping process as the "id" field in the echo request.
- Bind your sniffer to the lo interface, and then run `./tcp-test.py <some_port_number>`. The program generates a localhost TCP server and a localhost TCP client, and then has the server send a bunch of `'x'` characters to the client. You should see the 3-way TCP handshake, the server sending `'x'` characters, the client acknowledging those characters, and then the TCP teardown sequence.
- Bind your sniffer to `eth1` and then issue a `ping www.google.com`. Your sniffer should log the request and response messages.
- Bind your sniffer to `eth1`, and then issue a `wget` command like `wget www.cnn.com -O /dev/null`. Your sniffer should log the 3-way TCP handshake, the HTTP request, the HTTP response, and then the TCP teardown sequence.

Finally, run `make test_sniffer`, which is not a comprehensive test but should verify basic functionality.

Part 2: Forced Disconnects via TCP RST

Now, you will implement a TCP RST attack to maliciously destroy a preexisting TCP stream. The attack will involve three parties: a web server, a web client, and the attacker. The web server will live on the host machine, whereas the web client and the attacker will live on the VM. This setup emulates a scenario in which the web client and the attacker reside on the same subnet, such that the attacker can sniff the web traffic that is sent by the client. The attacker's goal is to force a client HTTP request to fail by injecting TCP RST packets into the network.

The paper "Detecting Forged TCP Reset Packets" provides a nice overview of TCP RST attacks; the entire paper is interesting, but for the purposes of this project, Section 4 of the paper is the most relevant:

> The crucial field in a RST is its *sequence number*, which must be chosen correctly for the packet to be accepted by the destination. Per the RFC, when aborting a connection the sender should send an *in-sequence* RST, i.e., set the sequence number to the next available octet in sequence space if terminating an active connection.

So, at a high level, your attack should listen for **incoming** traffic from the server which has the TCP ACK flag set. Those packets represent HTTP response packets from the server. When your attack detects such packets, it should send an

outgoing TCP RST packet to the server.

Libnet

Now that you know how to sniff preexisting packets, you will learn how to use libnet to inject new packets into the network. Here are some tutorials on how to use libnet:

- "The Evolution of Libnet"
- "Libnet 1.1 tutorial"
- "Libpcap and Libnet"

Some of these tutorials are a bit out-of-date. Fortunately, the comments in the primary libnet header file are excellent -- see `/usr/include/libnet/libnet-functions.h`. That header file, and the rest of the libnet headers, should be treated as the canonical documentation for libnet.

libnet acts as a higher-level interface to an operating system's facilities for writing raw packets. For example, on Linux, libnet acts as a wrapper around the raw socket interface. Other operating systems define different interfaces for writing raw packets, so libnet acts as an abstraction layer which allows you to create portable code for injecting new packets into the network.

In libnet, a new packet is constructed by calling `libnet_build_XXX()` functions, where "XXX" is the name of a network layer like "tcp". For your attacks, you'll be constructing raw TCP packets, so you'll need to call `libnet_build_tcp()` and `libnet_build_ipv4()` to construct the necessary packet, and then `libnet_write()` to inject the packet into the network.

> **Important**
>
> You will **not** need to call `libnet_build_ethernet()`; by omitting that call, you instruct libnet to construct the appropriate Ethernet header for you. However, `libnet_build_tcp()` **must** be called before `libnet_build_ipv4()`, since libnet requires a new packet to be built from the top of the OSI stack.
>
> Also, you should call `libnet_clear_packet()` before you start building a new packet.

> **Tip**
>
> Like libpcap, libnet requires various incantations to create and destroy a libnet handle. You should create a utility library which provides a higher-level interface to those incantations. This library can be used by all of your attacks.

HTTP Client/Server

To run a web server on your host, simply use Python's built-in web server like this:

```
python2 -m SimpleHTTPServer 9263
```

Here, 9263 is the TCP port on which the server will listen for HTTP requests (feel free to change). The web server will look for requested files in the server's working directory. You should place a large file in that directory. For example, you can generate 32 MB of dummy file data like this:

```
python -c "print('x' * (2 ** 25))" > tmp.txt
```

You will launch your RST attack against an HTTP fetch of that file. Making the file large lowers the barrier to a RST attack, since the attacker has more opportunities to generate RST packets for ACK-bearing data packets that are sent by the server.

To run a web client within the VM, you can use the `wget` command like this:

```
wget -t 1 -O /dev/null -- http://192.168.26.1:9263/tmp.txt
```

Here, the `-t 1` means no retries, and the `-O /dev/null` means to throw away the received bytes.

Writing Your RST Attack

Place the `main()` function for your attack in `rst_http.c`. The program should be invocable like this:

```
rst_http server_port [dev_name]
```

Here, `server_port` is a required argument that specifies the TCP port of the victim server. `dev_name` should be handled in the same manner as in `sniffer`.

Make sure to do the following:

- At initialization time, register signal handlers for `SIGINT` and `SIGQUIT` so that `rst_http` will shut down gracefully.
- At initialization time, use a `pcap_setfilter()` so that libpcap will only deliver **incoming** TCP traffic whose **source** port is `server_port` and which has the TCP ACK flag set.

Then, when `rst_http` detects a packet on the device, it should inspect the headers in the packet, generate a RST packet with the appropriate sequence number (and other info) using libnet, and then inject that packet into the network to destroy the client/server HTTP connection.

Testing Your RST Attack

First of all, `make rst_http` should successfully compile the program.

Make sure the web server is running on the host (with a giant `tmp.txt` file), as described above.

From the VM, run:

```
sudo ./rst_http 9263 eth0
```

If the attack works, then running `wget` from the VM (as described above) will fail or hang, providing an error message like this:

```
$ wget -t 1 -O /dev/null -- http://192.168.26.1:9263/tmp.txt
--2000-01-01 00:02:63-- http://192.168.26.1:9263/tmp.txt
Connecting to 192.168.26.1:9263 ... connected.
HTTP request sent, awaiting response... 200 OK
Length: 33554433 (32M) [text/plain]
Saving to: ‘/dev/null’

/dev/null 0%[ ] 2.63K --.-KB/s in 0s

2000-01-01 00:02:63 (263 MB/s) - Read error at byte 2630/33554433 (Connection reset by peer). Giving up.
```

On the web server, you should see console output like this:

```
Exception happened during processing of request from ('192.168.26.3', 49263)
Traceback (most recent call last):
File "C:\Python27\lib\SocketServer.py", line 295, in _handle_request_noblock
self.process_request(request, client_address)
File "C:\Python27\lib\SocketServer.py", line 321, in process_request
self.finish_request(request, client_address)
```

```
File "C:\Python27\lib\SocketServer.py", line 334, in finish_request
self.RequestHandlerClass(request, client_address, self)
File "C:\Python27\lib\SocketServer.py", line 657, in __init__
self.finish()
File "C:\Python27\lib\SocketServer.py", line 716, in finish
self.wfile.close()
File "C:\Python27\lib\socket.py", line 283, in close
self.flush()
File "C:\Python27\lib\socket.py", line 307, in flush
self._sock.sendall(view[write_offset:write_offset+buffer_size])
error: [Errno 10054] An existing connection was forcibly closed by the remote host
```

Finally, run `make test_rst_http`, which is not a comprehensive test but should verify basic functionality.

> **Tip**
>
> To debug problems with your attack, you can use your sniffer to log the interactions between `rst_http`, the web client, and the web server.

Part 3: Telnet Hijacking via TCP Injection

In the final part of this project, you will inject new traffic into a preexisting TCP stream. In particular, you will inject traffic into a telnet connection. Telnet is a well-known, simple protocol which allows a client to send commands to a server over TCP. Telnet does not encrypt traffic; thus, telnet has become much less popular with the advent of more secure approaches like ssh. Nevertheless, because telnet is just a thin protocol layer atop TCP, network admins often use telnet clients as simple testing tools, e.g., to test if a particular server port is open, or to see whether an HTTP server is responding to commands.

If you're curious about the details of the telnet protocol, you can read these documents:

- "The Telnet Protocol"
- "TCP/IP Guide - Telnet Protocol"
- "IETF RFC 854"

However, for the purposes of this project, you can ignore the details of how a telnet client and telnet server negotiate session parameters at the beginning of a TCP connection. As we explain in the next section, you only need to focus on what happens once the parameters have been negotiated, and the client issues a command to the server.

Our Telnet Server

On your host machine, you can run the project's telnet server like this:

```
./telnet_server.py 8263
```

Inside your VM, you can connect to the server using this command:

```
telnet 192.168.26.1 8263
```

The telnet client will present you with a prompt. If you invoke the `echo` command, e.g.:

```
proj3 server> echo hello
```

then the telnet server will return a copy of the argument (which in this case is "hello"). If you enter the `boom` command, the telnet server will print `BOOM!` and then terminate. If you enter Control-D into your telnet client, the client will gracefully shut down its connection to the server, but the server will continue to run.

> **Note**
>
> The Windows implementation of the curses library has some quirks, so if you run the telnet server on a Windows host, the server may not be able to successfully handle the telnet user hitting the backspace key or the left-arrow key.

Using your sniffer, look at the packets that the client and server exchange in response to the user typing `echo hello` and `boom`. Make sure that you understand how sequence and acknowledgment numbers are being set, and what data is being placed in each TCP segment.

Writing Your Hijacking Attack

Place the `main()` function for your attack in `hijack_telnet.c`. The program should be invocable like this:

```
hijack_telnet server_name server_port [dev_name]
```

Here, `server_name` is a required argument that specifies the hostname or IP address of the victim server. `server_port` and `dev_name` should be handled in the same manner as in `rst_http`.

Your attack should listen only for telnet traffic involving the appropriate server hostname and port (via `pcap_setfilter()`). You may assume that, at any given time, there is at most one live telnet stream for your attack to sniff. As your attack encounters telnet packets, it should record the necessary information about the TCP headers in the stream.

Your attack should handle `SIGINT` and `SIGQUIT` -- upon catching one of these signals, it should inject the command `boom` into the preexisting telnet stream. You must use the recorded TCP information to guide the construction of the injected packets, so that the telnet server's network stack will accept the injected packets as legitimate.

Testing Your Hijacking Attack

First of all, `make hijack_telnet` should successfully compile the program.

Make sure the telnet server is running on the host (as described above).

From the VM, run:

```
sudo ./hijack_telnet 192.168.26.1 8263 eth0
```

From the VM (in a different terminal), connect to the server (as described above). Once the client has reached the `server>` prompt, go to the terminal window for `hijack_telnet` and press Control-C. If your attack works, then the telnet server on the host will print `BOOM!` and exit. The telnet client will probably hang or otherwise act strangely, since its TCP connection has now become desynchronized! So, you may have to kill the poor client from another terminal window using a command like `pkill telnet`.

Your attack should work regardless of what the user might have previously entered in the telnet client (other than `boom`, of course).

Finally, run `make test_hijack_telnet`, which is not a comprehensive test but should verify basic functionality.

> **Tip**
>
> As you debug your attack, remember that you can use your sniffer to explore why your attack might be failing!

Submitting

> **Important**
>
> Before submitting, make sure all your work is committed and pushed to the master branch of your fork, and make sure the Travis build is passing for master. You can verify by going to your fork's GitHub page, clicking on "commits", and looking for a green checkmark at the top of the list.

On the fork's GitHub page. click on "New pull request". The base fork should be the original repository (prefixed with `harvard-cs263`), and the head fork should be your fork (prefixed with your GitHub username). Then, click on "Create pull request" to submit your work! The title can be whatever, and the comment can be left blank (or non-blank if you have a note for the grader).

If you need to edit your submission before the deadline, just commit and push your new changes to the master branch of your fork. The original pull request will be automatically updated with those commits (of course, be sure to check the GitHub pull request page to verify).

> **Caution!**
>
> Do **not** click "Merge pull request" after submitting, as this will modify the original repository. We will merge your pull request when grading.

> **Caution!**
>
> The deadlines for all assignments are on Canvas. Deadlines are enforced to the minute (based on pull request/push times, not commit times), and the course late policy is a 10% deduction per 8 hours of lateness.
>
> Note that the Travis tests can take a while, and no testing-related extensions will be granted.

Deliverables and Rubric

"Mixed" grading means we will assign some points based on the result of the Travis test case(s). However, for this project, the tests do not fully cover the specification (for example, they only operate on `lo`), so we will also assign some points based on manual inspection and testing.

Criteria	Points	Grading method
`sniffer.c`	30	Mixed
`rst_http.c`	35	Mixed
`hijack_telnet.c`	35	Mixed