Implement the indiscriminate shell (**Individual effort**)

February 28, 2018

1 Assumptions

- You have certified as a basic developer
- You are an experienced C programmer
- You are familiar with the raw socket API on UNIX
- You are familiar with the C headers on UNIX which have to do with network-protocols
- You are familiar with the `dlopen`-family of library calls
- You are familiar with the `fork`, `execve`, `pipe`, and `dup` system calls on UNIX

2 Artifacts

You must submit your solution source code using Git no later than two weeks after you begin this exam. Your source code will consist of the following files:

- `Makefile`,
- README,
- `ish.c`,
- `ishd.c`,
- `plugin.c`,
- `plugin.h`, and
- `plugin-icmp.c`.

The code you submit must build when the command `make` is executed in your source directory. The `Makefile` must compile all C code using GCC with the `-Wall -Werror` flags, and the C compiler must emit neither warnings nor errors during the build process. Furthermore, `valgrind` must not find any notable memory errors in your programs.

Here is a reasonable `Makefile`:

```
all: ish ishd plugin-icmp.so

CFLAGS=-g -Wall -Werror

ish: ish.c plugin.c
	gcc $(CFLAGS) $^ -ldl -o $@

ishd: ishd.c plugin.c
	gcc $(CFLAGS) $^ -ldl -o $@

plugin-icmp.so: plugin-icmp.c
	gcc $(CFLAGS) -fpic -shared $^ -o $@

clean:
	rm -f ish
	rm -f ishd
	rm -f plugin-icmp.so
```

3 Instructions

Your task is to implement a client and server which together provide a remote “indiscriminate” shell. You will write the `ish` client and `ishd` server. `Ish` should behave as follows, assuming `ishd` is running on the remote host 192.168.1.100,

```
$ sudo ./ish 192.168.1.100 cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
$ ▮
```

(The use of `sudo` is necessary due to the use of raw sockets.)

`ish` takes two or more arguments. This first (here `192.168.1.100`) indicates a remote host running `ishd`. The rest of the arguments indicate a command to run on the remote hosts (here `cat /etc/passwd`). `Ish` encodes the command and passes it to `ishd` over the network. `Ishd` then executes the command, captures its outputs, and sends this output back to `ish`. `Ish` prints the command output it receives from `ishd`. `Ish` need only support non-interactive commands which do not read from standard in. The following paragraphs specify the details of how `ish` and `ishd` must work.

Permitted libraries Your implementation of `ishd` may use only glibc (or another standard C library). This exam forbids the use of any other library.

Command execution Your implementation of `ishd` must directly use the `fork`, `execve`, `pipe`, and `dup` system calls to facilitate running a command and capturing its output. The use of higher-level functions such as `system` or `popen` is not permitted.

Error handling Robustly program. Your programs must appropriately handle the runtime errors which might result from system calls. Take advantage of UNIX’s `errno` and `perror` as well as the `%m` format string found in the `printf` series of functions. Print the errors indicated by the remote host using the mechanisms described below.

Inputs and outputs The first argument to `ish` must be a valid IPv4 address in dotted-decimal form. As they are intended to be directly typed, the subsequent arguments must be printable ASCII text (e.g., no embedded ‘\0’ characters). In contrast, the output of the commands which `ishd` runs on a remote host need not include only ASCII text. For example, the following should copy the command `ls` from the remote host to the local host in such a way that it will run if granted the correct permissions:

```
$ sudo ./ish 192.168.1.100 cat /bin/ls > /tmp/ls
$ sudo chmod +x /tmp/ls
$ /tmp/ls
...
$ ▮
```

(Note that this example makes assumptions about the similarity of the software on the local and remote hosts.)

Pluggable modules This exam prescribes a wire format (documented below); however, the need could conceivably arise to extend your program to support other wire formats. For this reason, you must implement your networking code as a shared object which `ish` and `ishd` load at runtime. You will use the `dlopen` family of functions to do this.

The interface for your pluggable module should be defined in plugin.h (you are free to define other things in this file too):

```c
#include <netinet/ip_icmp.h>
#include <dlfcn.h>

enum {
      MSG_REQUEST,
      MSG_REPLY,
      MSG_REPLY_ERR,
      MSG_REPLY_FRAG,
      MSG_REPLY_DONE,
};

struct fntable {
   void (*perror) (const char *s);
   int (*socket) (void);
```

```
    ssize_t (*sendto) (int sockfd, const void *buf, size_t len, int flags, const struct sockaddr
          *dest_addr, socklen_t addrlen);
    ssize_t (*recvfrom)(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr,
          socklen_t *addrlen);
};

struct fntable *plugin_load(const char *filename, const char *fntable_symbol_name);
void plugin_unload(void);
```

The source file `plugin-icmp.c` will contain the ICMP-specific pluggable module. It will define three functions according to the declarations above:

`plugin_icmp_perror` Print the error string associated with the most recent failed operation. Such an error might have resulted from a local system call, or it might have been transmitted from the remote host.

`plugin_icmp_socket` Obtain a raw network-protocol socket.

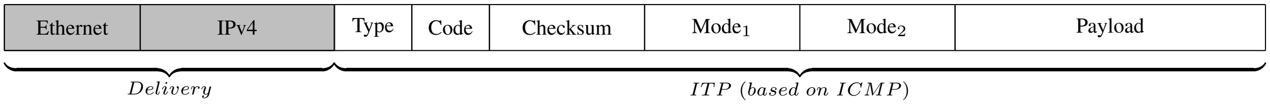
`plugin_icmp_sendto` Given a payload buffer and length, build a new buffer which encapsulates the payload in an ICMP message. Address this message to the given host and send it on the given socket. The flags parameter should define the message type (`MSG_REQUEST`, `MSG_REPLY_ERR`, and so on but not the catchall `MSG_REPLY`) with the effect of setting the type, code, checksum, and mode fields as described in the wire-format section below. This function should return the number of bytes written (to include any header bytes added) or a -1 to indicate an error. On error, this function should store an error string such that it can be printed using `plugin_icmp_perror`.

`plugin_icmp_recvfrom` Given a payload buffer and length, read a message from the given socket. Place the ICMP payload portion of this message (i.e., strip any protocol headers to include ICMP) in the given buffer. The flags parameter should define the message type expected (`MSG_REQUEST` or `MSG_REPLY` but not the particular `MSG_REPLY_X` types) with the effect of ignoring the messages that do not match the specified type category. Update the sender address and address length as with `recvfrom`. This function should return the number of bytes received (excluding any header bytes stripped) or a -1 to indicate an error. On error, this function should store an error string such that it can be printed using `plugin_icmp_perror`. Returning a zero value indicates that the remote end sent an end-of-transmission message.

`Plugin-icmp.c` will also define a global variable of type `struct fntable` which contains pointers to these three functions. Name this variable `plugin_icmp_fntable`.

You should define the functions `plugin_load` and `plugin_unload` in `plugin.c`. The `plugin_load` function should load a pluggable module from the given filename (i.e., `plugin-icmp.so`) and then obtain a pointer to its function table (i.e., `plugin_icmp_fntable`). The `plugin_unload` function should free the pluggable module. Both `ish.c` and `ishd.c` should use these functions to load and unload the module.

Wire format Your implementation of `ish` and `ishd` *must* communicate using the wire format and protocol described here. We call this the ICMP Transport Protocol (ITP). ITP makes use of ICMP's payload field to transmit application data.



Field	Size (bytes)	Description
Delivery fields		
Ethernet header	14	Standard Ethernet header
IP header	20–60	Standard IPv4 header
ITP fields		
Type	8	8 = request or 0 = reply
Code	8	Always set to 0
Checksum	16	Checksum of ICMP message
Mode$_1$	16	Request: always `0xd000` Reply: `0xdead` if data transmission `0xbaad` if error `0xfeed` if end of transmission (Network byte order)
Mode$_2$	16	Request: always `0x000d` Reply: `0xbeef` if data transmission `0xf00d` if error `0xface` if end of transmission (Network byte order)
Payload	$\leq$ 452	Arbitrary payload bytes

The checksum field contains a standard ICMP checksum. The following C code will calculate this checksum assuming it is initially set to zero. Pass a pointer to the ICMP message to `_update_icmp_checksum`. The `nbytes` argument should account for the entire ICMP message (i.e., bytes `Type`–`Payload`).

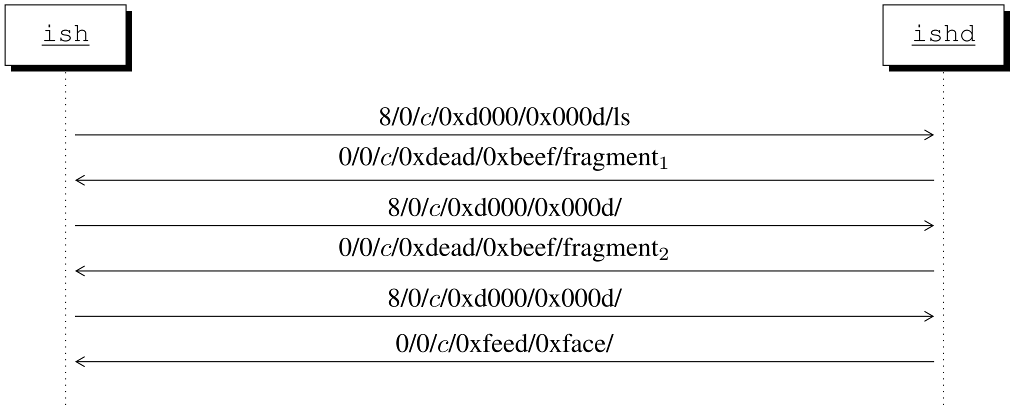
```c
static void
_update_icmp_checksum(unsigned short *ptr, int nbytes)
{
    long sum;
    unsigned short oddbyte;
    unsigned short answer;
    struct icmphdr *icmph = (struct icmphdr *) ptr;

    sum = 0;
    while (nbytes > 1) {
          sum += *ptr++;
          nbytes -= 2;
    }

    if (nbytes == 1) {
          oddbyte = 0;
          *((unsigned char *) & oddbyte) = *(unsigned char *) ptr;
          sum += oddbyte;
    }

    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;

    icmph->checksum = answer;
}
```

Protocol A protocol exchange starts when `ish` submits an ITP message for remote execution. This message bears a `Type` field set to 8 and `Mode`$_1$ and `Mode`$_2$ fields set to zero. The payload contains an encoded command line. This encoding is similar to a standard shell's input: each token is separated by an ASCII space, and the command bears a '\0' terminator. A request must fit in a single ITP message, and thus the protocol restricts command lines to 452 bytes. (Here we assume the appropriate link- and network-layer headers are also present.)

An `ishd` server receives such an encoded command, executes the command, and transmits the output of the command back to `ish`. These messages bear a `Type` field of 0. Messages containing data bear a `Mode`$_1$ field set to `0xdead` and a `Mode`$_2$ field set to `0xbeef`. The `ish` client must acknowledge each output fragment with a request bearing a zero-length payload. `Ishd` waits for an acknowledgment before sending the next fragment.

The final message contains an empty payload and bears the mode fields of `0xfeed` and `0xface`, respectively. Upon receiving this terminating message, `ish` should stop reading from the network and exit. `Ish` should not acknowledge the terminating message. For example:

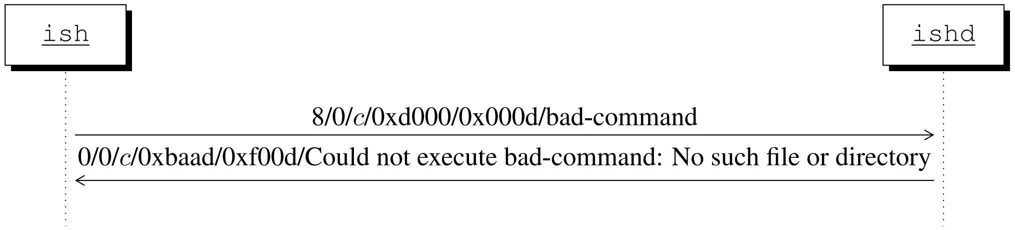
```
$ sudo ./ish 192.168.1.100 ls
```



```
[output from ls]
```

If an error prevents `ishd` from executing a command, then `ishd` should send a message with its `Mode`$_1$ field set to `0xbaad`, its `Mode`$_2$ field set to `0xf00d`, and a payload containing an error message. Upon receiving such an error message, `ish` should print the message to standard error, stop reading from the network, and exit. For example:

```
$ sudo ./ish 192.168.1.100 bad-command
```


ish
ishd
8/0/c/0xd000/0x000d/bad-command
0/0/c/0xbaad/0xf00d/Could not execute bad-command: No such file or directory

```
Could not execute bad-command: No such file or directory
```