CMSC 491 Active Cyber Defense

Networking HW 2

Objective

Students will learn about how network protocols work and will become more comfortable idea with the concept that TCP simply transports bytes. It's up to the application what it does with those bytes.

Due date

This assignment is due Wednesday, Dec 5, 2018 at 6:59:59 PM.

Summary

For this assignment, you will write a client for a simple protocol, the **Cyber Resource Acquisition Protocol**. You can find the specification for this protocol here:
https://drive.google.com/file/d/1uRdup0T7jQSR3sX-1y-MbYqoXKWCmRKE/view

It should be possible to run your client as follows:

```
./client.py -u zack -p letmein --limit 3 --filter "type=service"
```

The filter parameter should be able to be left off, in which case the server will give all results, which should be printed.

Sample output:

```
$ ./client.py -u zack -p letmein --limit 2
Results:
--
type: protocol
name: http
rfc: 2616
--
type: protocol
name: htcpcp
rfc: 2324

$ ./client.py -u zack -p letmein --filter "name=http" --limit 2
Results:
--
type: protocol
name: http
rfc: 2616
--
type: service
```

```
port: 80
name: http
protocol: tcp
```

Your output may be slightly different (the results the server sends are not deterministic), but must contain a similar readout.

The server hostname and port should be either constants at the top of your program, or command line arguments, that's up to you.

Your program must exit cleanly once the searchResultDone message is received.

There is a **test server** running on notanexploit.club port 9090. There are no other servers in existence for this protocol, and we didn't give you server code, so I recommend testing against this one :) Note that this test server may log every message it receives, so please don't send it any of your real passwords! Currently, all usernames are valid, but the password must be "letmein". Secure, right?

This should make it much more difficult to Google other implementations of this assignment and copy paste them, without really understanding how they work.

How to approach this

I've taken the liberty to give you a step-by-step guide for how to start this assignment. Once you finish the step-by-step, I think you'll know how to complete the full assignment.

If you already know what you're doing, feel free to do something else. You'll notice we don't even require it to be written in Python, as long as we can easily get your program running (you should check with us before using something else, to make sure we know how to run it).

Part 1: Construct a bindRequest message

Start by reading the spec, and generally figure out the structure of the protocol. Ask questions on Slack if you need to. You shouldn't need to guess. Please ask protocol clarifications in #cmsc-491-791 so that everyone can see the answer, if you're not comfortable with that, you can DM a TA, and they will repost your question for you.

I recommend using the Python 3 bytes type to represent messages. It's somewhat unergonomic, but it works well enough. You can create a single byte bytes object with `myobj = bytes([0xFF])`, where `0xFF` is the value of the byte you want. You can then concatenate that object with `newobj = myobj + myobj2`.

Part 1.1: Construct a CHAR

Write a Python function that takes an int and converts it to a bytes object. Operation is undefined if the int is > 255, you're welcome to throw an exception or something.

Here's how mine works (my program here is named parser.py)

```
$ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import parser
>>> print(parser.char(14))
b'\x0e'
```

Part 1.2: Construct PSTRINGs

Write a function (I called it pstring()) that takes a Python string object as a parameter, and returns a bytes object that is a valid, encoded, “PSTRING”.

You’ll do this by UTF-8 encoding the string (this is a single function call in Python, don’t overthink it), then prepending a single byte, which is the length of the ENCODED value.

To test your code, you can encode the string “teststring” and verify the result is '0a74657374737472696e67'.

I obtained this result like this:

```
$ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import parser
>>> import binascii
>>> print(binascii.hexlify(parser.pstring("teststring")))
b'0a74657374737472696e67'
```

If you haven’t used it before, binascii.hexlify is a delightful function that gives you a pretty hexdump of data. The inverse is binascii.unhexlify.

Part 1.3 Construct the bindRequest message

Read the specification and figure out what a bindRequest consists of. Notice that it uses the 2 data types that you just wrote code to create. So combine it and add the tag byte.

Here’s some sample output so you can test yours.

```
$ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import parser
>>> import binascii
>>> print(binascii.hexlify(parser.bindRequest(1, "user", "pass")))
b'2301047573657204706173 73'
```

Part 2: Construct a parcel

“Parcel” is a term I made up for this protocol. Read the spec and see what it means. Write a Python function that creates one, given a bytes object.

Here’s some sample output:

```
$ python3
Python 3.5.3 (default, Sep 27 2018, 17:25:39)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import parser
>>> import binascii
>>> print(binascii.hexlify(parser.make_parcel(parser.bindRequest(1, "user",
"pass"))))
b'0c2301047573657204706173373'
```

Part 3: Connect to the server and send a bindRequest

Part 3.1: Connect to the server

For this, you will need to use sockets. Since this is the main point of the assignment, I will not give you all the details. You will need to research how sockets work in Python 3 and use them. You want the “socket” library. You’ll probably use the socket and connect functions.

Part 3.2: Send your bindRequest within a parcel

Again, consult the documentation. Note there is a HUGE difference between socket.send and socket.sendall. Make sure whatever you do, you ensure all data is sent (that’s a grading item :)

Part 4: Receive parcels from the server

This might be the easiest single part to mess up, but I’ve designed the assignment so that you’ll end up getting it right. With the comments here, I don’t think it’ll be hard.

You will note that the recv() method of the socket is defined to return **up to** the number of bytes you ask for. So what do you do if you didn’t get however many bytes you want? You try again, keeping a buffer of already received data.

I recommend writing a function that does this for you. Since parcels begin with their length, you can read the length, then read again until you have acquired the complete message.

This really isn’t hard, but I’ve assigned it a bunch of points because I want to emphasize this is important to write robust network code. If a syscall or library function says it may randomly fail and you need to retry it, that means *you need to retry it*, if you’re doing anything more important than a CTF challenge.

Part 5: Parse the bindResponse message

So now you have a function to call that will give you the inner contents of a single parcel from the network. (That means that function ensured you have the full length, then it removed the length byte and returned a buffer.)
You now need to figure out what that buffer means.

Read the specification and figure out what a Message is. You'll notice there's a CHAR that lets you figure out what type of message it is, so I'd check that byte, then call some function that parses message type X.

I use Python classes, where each message type is a class, that knows how to parse message of its own type. So I have central parsing code that checks the message type and returns an object of that type. For some message types, that is nested a few layers deep. For instance, my SearchResultEntry parsing code contains an array of Attribute objects.

At the base level, I have code that can parse CHARs and PSTRINGs

Your interface will differ, but here's some sample output that should help you test your parser.

```
>>> print(parser.SCM1Parser(binascii.unhexlify(
b'24771c5468697320697320612061206d6f7469766174696e67206d657373616765')))

SCM1([BindResponse(CRAPResult(resultCode=RES_SUCCESS, motivation="This is a
motivating message"))])
```

Make sure you write code to check the resultCode and print an error if it fails.

Part 6: Generate the searchRequest message

By this point, I think you'll know how to do this. I'd write a function to encode FILTERs, a function to encode OPTIONAL values, and then I'd write a function to encode searchRequests that uses those.

I'm not sure the API I made is optimal, but here's some sample output to compare yours to:

```
>>> binascii.hexlify(parser.searchRequest(4, parser.make_filter("type",
"protocol")))
b'30042b0474797065087072f746f636f6c'
>>> binascii.hexlify(parser.searchRequest(4, None))
b'300423'
```

Again, note that these sample messages are not yet encapsulated in parcels.

Part 7: Parse the searchResultEntry messages

Ok, I have to make you read the spec yourself at some point, and I think I explained enough about how to parse this above that you can do it!

Here's some sample data so you can test your parser:

```
>>> print(parser.SCM1Parser(binascii.unhexlify(
b'41030269640131046e616d6508436f6d7075746572056f776e657208536f6d65626f6479')))

SCM1([SearchResultEntry(count=3, attributes=[Attribute(name="id", value="1"),
Attribute(name="name", value="Computer"), Attribute(name="owner",
value="Somebody")])])
```

Part 8: Parse the searchResultDone messages

Yeah, I think you can do it at this point. Read the spec, ask in Slack if you have questions.

By the way, it goes without saying you have to actually SEND the searchRequest, and then RECEIVE the searchResultEntry and searchResultDone messages… but I'm saying it here in case someone missed that.

Part 9: Prettify your client

Now you need to add command line arguments. I highly recommend the Python argparse library (it's in the standard library, just import argparse). It's daunting the first time you read the docs, but it's actually super easy to use.

Next, you should add some documentation of the following:

- What language (and version) your program is written in
- How to run it / how to pass arguments
- How to change host/port of the server it's connecting to

Also, make the print out nicely readable like the sample at the top of the assignment.

Congrats, you're done!