**Module 2.3 – Socket Related Structures and Network Byte Ordering.**

**Required reading material:**

* [1] Brian “Beej Jorgensen” Hall, “Beej's Guide to Network Programming, v3.1.11”. April 2023. <https://beej.us/guide/bgnet/html/split/>
  + Chapter 3, 4, and 5.
* [3] Jon Erickson, “Hacking the Art of Exploitation 2nd ed”. No Starch Press. February 2008. ISBN: 978-1593271442. <https://learning.oreilly.com/library/view/hacking-the-art/9781593271442/>
  + Read chapter 0x04, section 0x420 -Sockets. Pages: 198-203.
* [4] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff, “The Sockets Networking API: UNIX® Network Programming Volume 1, Third Edition”. Addison Wesley. November 2003. ISBN: 0-13-141155-1. <https://learning.oreilly.com/library/view/the-sockets-networking/0131411551/>
  + This book provides a more in-depth/technical explanation for the topics covered in this module.
  + Read Chapter 3 (Sockets Introduction), sections 3.1 – 3.8. <https://learning.oreilly.com/library/view/the-sockets-networking/0131411551/ch03.html>
* [2] Lewis Van Winkle, “Hands-On Network Programming with C". Packt Publishing. May 2019. ISBN: 9781789349863. <https://learning.oreilly.com/library/view/hands-on-network-programming/9781789349863/>
  + Read chapter 5, Name/Address Translation functions, pages: 132-126

In this module we’ll cover the socket related structures necessary for any network application and we’ll also cover the topic of network byte ordering and its corresponding functions.

As we learned in the previous module, there are a lot of functions and a handful of structures that are required to write the simplest of network applications, we’ll dive into each of those structures. The required reading material does a phenomenal job covering all the relevant structures in detail, so we’ll summarize each of them here and provide whatever information might be missing from the books and try to clarify the more complex topics.

**Network Byte Ordering vs Host Byte Ordering and why it is important.**

In later modules we’ll cover the topic of data serialization in greater detail but for the purpose of this module, we’ll cover the basics of data transmission across different machines and how data is represented as it crosses the wire.

If we think back to an Assembly programming course, data organization, or Operating System course, we’ll remember that different processors and computer architectures represent bits and bytes in different ways. There are two common ways to write bytes on a computer system, Big-Endian and Little-Endian, both of which refer to whether the least or most significant byte is the trailing byte in a message.

We'll use the two-byte hexadecimal string: 0xb34f as our example string.

Big-Endian (also known as Network Byte Order): this is the commonly agreed upon byte ordering where bytes are represented in the sequence the way you'd expect them to. That is, the byte string 0xB34F is represented as 0xB3, 0x4F on a Big-Endian system.

Little-Endian (known as Host Byte Order) on the other hand, stores the bytes in reversed order, that is the string 0xB34F would be stored in memory as the sequential bytes 0x4F followed by 0xB3. There are computer systems and processors which use this data representation method.

Also keep in mind that numbers are represented in two different ways: short (two bytes) and long (four bytes) and they need to be converted or handled appropriately whenever you send data over a network.

As we can expect, having two different data representation methods poses a problem when transmitting data over a network. To overcome this issue, the socket API provides a set of functions to convert between the different data representation methods. The big point we're trying to make here is that you'll want to convert your data to Network Byte Order as you send out over the network and likely convert it to Host Byte Order when you receive it on the endpoint (depending on the processor: Intel vs other processors, etc.). This might not be a big issue in this course since we’ll almost always be running our server and client applications on the same system and so we’ll not write our code with this mind. This issue becomes a real concern once you start transmitting data across different networks, such as the internet, where data passes through multiple devices where they might use different processors with different architectures.

What’s the best strategy when writing code that needs to account for network vs host byte ordering?

* You just get to assume the Host Byte Order isn’t right, and you always run the value through a function to set it to Network Byte Order. The function will do the magic conversion if it has to, and this way your code is portable to machines of differing endianness.
* Basically, you’ll want to convert the numbers to Network Byte Order before they go out on the wire and convert them to Host Byte Order as they come in off the wire.

A black and white box with text

Description automatically generated

**Socket Related Structures**

Read the required material for a detailed explanation and purpose of each structure. I’ve added comments and brief explanations to each structure and function call to help make sense of the more complex pieces, but this section is no substitute for reading the material and won’t make too much sense without doing so.

For this section, we’ll use the showip.c program from “Beej's Guide to Network Programming” chapter 5. With only 60 lines of code, it covers a lot of the structures that we will use when writing networking applications. <https://github.com/beejjorgensen/bgnet/blob/main/source/examples/showip.c>

A screenshot of a computer code

Description automatically generated

And we’ll use chapter 3.3 Structs to explore each structure.

For all the structures covered here, visit the Linux man-pages for each structure / function.

Before we can understand what, these structures are and how they are used it helps to understand what this function is doing. Let’s start with getaddrinfo() on line 28.

The online man page: <https://man7.org/linux/man-pages/man3/getaddrinfo.3.html> goes into greater detail but I’ve taken the function definition and added comments explaining each parameter.

A screen shot of a computer code

Description automatically generated

Getaddrinfo() will return information on a particular host name (such as its IP address) and load up the host information into a struct of type “sockaddr” for you, taking care of the gritty details ( whether it is IPv4 or IPv6). It replaces the old functions gethostbyname() and getservbyname().

In other words, the purpose of this function is to populate an “addrinfo” structure with the requested information. It knows what information to request from a remote system by using a structure of type addrinfo as its hint structure (which we populate). All that means is that we are pre-populating a template structure with the type of data we want, getaddrinfo() uses our template structure and requests that data from the remote host. More on this later.

Note: In legacy code, getaddrinfo() was not available and you will see programmers manually initialize addrinfo.

I took the showip.c file and added comments explaining what the code is doing. I’ve added comments for each of the functions and structures used in the code. See showip.c in the module’s directory.

Let’s peel back the layer a bit and try to dissect each of the structures.

**struct addrinfo{…}**

This structure is a more recent invention and is used to prepare the socket address structures for subsequent use. It’s also used in host name lookups, and service name lookups: getaddrinfo() and getnameinfo().

We can inspect the structure's definition by browsing to /usr/include/netdb.h on a modern Linux system.

A screenshot of a computer

Description automatically generated

The following pages contain all the technical details for the structure and each field.

* <https://linux.die.net/man/3/getaddrinfo>
* <https://man7.org/linux/man-pages/man3/getaddrinfo.3.html>

Here is the same structure with different annotations.

A computer screen shot of a black screen

Description automatically generated

**struct sockaddr{…}**

This structure describes a generic socket address and holds socket address information for many types of sockets.

A screen shot of a computer

Description automatically generated

Manually packing this structure can be tedious and error prone, so programmers created a parallel data structure called struct sockaddr\_in("in" stands for internet) to be used with IPv4.

This is an IPv4 socket address structure, commonly called an "Internet socket address structure".

A computer screen shot of a black screen

Description automatically generated

A screenshot of a computer program

Description automatically generated

And lastly, we cover **struct sockaddr\_storage{…}**:

This structure is designed to be large enough to hold both IPv4 and IPv6 structures. For some calls, sometimes you don’t know in advance if it’s going to fill populate the struct sockaddr{} with an IPv4 or IPv6 address. For that reason, we pass in this parallel structure very similar to struct sockaddr except larger, and then cast it to the type you want/need.

A screen shot of a computer program

Description automatically generated

We have covered the relevant structures needed to write simple networking applications and are ready to start writing programs of our own and start digging into the socket API set of functions.