Errata Security: How to teach endian

笔记本:	编程理论
创建时间:	2020/9/2 11:51
URL:	https://blog.erratasec.com/2016/11/how-to-teach-endian.html#.X08Ty8gzaUk

Errata Security

Advanced persistent cybersecurity

Monday, November 14, 2016

How to teach endian

On */r/programming* is this post about byte-order/endianness. It gives the same information as most documents on the topic. It is wrong. It's been wrong for over 30 years. Here's how it should be taught.

One of the major disciplines in computer science is *parsing/formatting*. This is the process of converting the *external format* of data (file formats, network protocols, hardware registers) into the *internal format* (the data structures that software operates on).

It should be a formal computer-science discipline, because it's actually a lot more difficult than you'd expect. That's because the majority of vulnerabilities in software that hackers exploit are due to parsing bugs. Since programmers don't learn about parsing formally, they figure it out for themselves, creating ad hoc solutions that are prone to bugs. For example, programmers assume external buffers cannot be larger than internal ones, leading to buffer overflows.

An external format must be well-defined. What the first byte means must be written down somewhere, then what the second byte means, and so on. For Internet protocols, these formats are written in RFCs, such as RFC 791 for the "Internet Protocol". For file formats, these are written in documents, such as those describing GIF files, JPEG files, MPEG files, and so forth.

Among the issues is how integers should be represented. The definition must include the size, whether signed/unsigned, what the bits means (almost always 2s-compliment), and the byte-order. Integers that have values above 255 must be represented with more than one byte. Whether those bytes go left-to-right or right-to-left is known as *byte-order*.

We also called this *endianness*, where one form is *big-endian* and the other form is *little-endian*. This is a joke, referring back to Jonathan Swift's tale *Gulliver's Travels*, where two nations were at war arguing whether an egg should be cracked on the big end or the little end. The joke refers to the Holy Wars in computing where two sides argued strongly for one byte-order or the other. The commentary using the term "endianess" is that neither format matters.

However, *big-endian* is how humans naturally process numbers. If we have the hex value 0x2211, then we expect that representing this number in a file/protocol will consist of one byte with the value 0x22 followed by another byte with the value 0x11. In

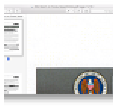
Errata Security On

Awesome Inc. theme. Powere

Popular Posts

You are committing a crime right now

Are you reading this blog? If so, you are committing a crime under 18 USC 1030(a) (better known as the " Computer Fraud & ...

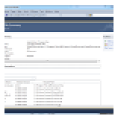


How The Intercept Outed Reality Winner

Today, The Intercept released documents on election tampering from an NSA leaker. Later, the arrest warrant request for an NSA contractor ...

SideJacking with Hamster

NOTE: you can download the program at http://www.erratasec.com/sidejacking.zip ; make sure to read the instructions. Others have done a be...



Extracting the SuperFish certificate

I extracted the certificate from the SuperFish adware and cracked the password (" komodia ") that encrypted it. I discuss how dow...

a *little-endian* format specification, however, the order of bytes will be reversed, with a value of 0x2211 represented with 0x11 followed by 0x22.

This is further confused by the fact that the *nibbles* in the byte will still be written in conventional, big-endian order. In other words, the big-endian format for the number 0x1234 is 0x12 0x34. however, the little-endian format is 0x34 0x12 -- not 0x43 0x21 as you might naively expect trying to swap everything around in your mind.

If little-endian is so confusing to the human mind, why would anybody ever use it? The answer is that it can be more efficient for logic circuits. Or at least, back in the 1970s, when CPUs had only a few thousand logic gates, it could be more efficient. Therefore, a lot of internal processing was little-endian, and this bled over into external formats as well.

On the other hand, most network protocols and file formats remain big-endian. Format specifications are written for humans to understand, and big-endian is easier for us humans.

So once you understand the byte-order issue in external formats, the next problem is figuring out how to parse it, to convert it into an internal data structure. Well, we first have to understand how to parse things in general.

There are two ways of parsing thing: buffered or streaming. In the buffered model, you read in the entire input first (like the entire file, or the entire network packet), then parse it. In the streaming mode, you read a byte at a time, parse that byte, then read in the next byte. Stream mode is best for very large files or for streaming data across TCP network connections.

However, buffered parsing is the general way most people do it, so I'll assume that in this guide.

Let's assume you've read in the file (or network data) into a buffer we'll call *buf*. Your parse that buffer at the current *offset* until you reach the *end*.

Given that, then the way you'd parse a *big-endian* integer *x* is the following line of code:

```
 x = buf[offset] * 256 + buf[offset+1];
```

Or, if you prefer logical operators, you might do it as:

```
 x = buf[offset]<<8 | buf[offset+1];
```

Compilers always translate multiplication by powers-of-2 into shift instructions, so either statement will perform the same. Some compilers are smart enough to recognize this pattern as parsing an integer, and might replace this with loading two bytes from memory and byte-swapping instead.

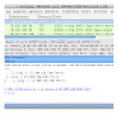
For a little-endian integer in the external data, you'd reverse how you parse this, like one of the following two statements.

```
x = buf[offset+1] * 256 + buf[offset];
x = buf[offset] + buf[offset+1] * 256;
```

If we were talking about JavaScript, C#, or a bunch of other languages, at this point the conversation about endianess would end. But if talking about C/C++, we've got some additional wrinkles to deal with.

That NBC story 100% fraudulent

Yesterday (Feb 5 2014) On February 4th, NBC News ran a story claiming that if you bring your mobile phone or laptop to the Sochi Olympics...



Bash 'shellshock' scan of the Internet

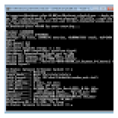
NOTE: malware is now using this as their User-agent. I haven't run a scan now for over two days. I'm running a scan right now ...

Bash 'shellshock' bug is wormable

Early results from my scan: there's about 3000 systems vulnerable just on port 80, just on the root "/" URL, without Host fiel...

Some notes about HTTP/3

HTTP/3 is going to be standardized. As an old protocol guy, I thought I'd write up some comments. Google (pbuh) has both the most popu...



Confirmed: LinkedIn 6mil password dump is real

Today's news is that 6 million LinkedIn password hashes were dumped to the Internet. I can confirm this hack is real: the password I use...

Notes on the Ashley-Madison dump

Ashley-Madison is a massive dating site that claims 40 million users. The site is specifically for those who want to cheat on their spouse. ...

Blog Archive

-

The problem with C is that it's a *low-level* language. That means it exposes the *internal* format of integers to the programmer. In other words, the above code focuses on the *external* representation of integers, and doesn't care about the *internal* representation. It doesn't care if you are using an x86 little-endian CPU or some RISC big-endian CPU.

But in C, you can parse an integer by relying upon the internal CPU representation. It would look something like the following:

```
x = *(short*)(buf + offset);
```

This code produces different results on a little-endian machine and a big-endian machine. If the two bytes are 0x22 and 0x11, then on a big-endian machine this produces a short integer with a value of 0x2211, but a little-endian machine produces the value of 0x1122.

If the external format is big-endian, then on a little-endian machine, you'll have to *byte-swap* the result. In other words, the code would look something like:

```
x = *(short*)(buf + offset);
#ifdef LITTLE_ENDIAN
x = (x >> 8) | ((x & 0xFF) << 8);
#endif
```

Of course, you'd never write code that looks like this. Instead, you'd use a macro, as follows:

```
x = ntohs(*(short*)(buf + offset));
```

The macro means *network-to-host-short*, where *network* byte-order is big-endian, and *host* byte-order is undefined. On a little-endian host CPU, the bytes are swapped as shown above. On a big-endian CPU, the macro is defined as nothing. This macro is defined in standard sockets libraries, like <arpa/inet.h>. There are a broad range of similar macros in other libraries for byte swapping integers.

In truth, this is not how it's really done, parsing an individual integer at a time. Instead, what programmers do is define a *packed* C structure that corresponds to the external format they are trying to parse, then *cast* the buffer into that structure.

For example, in Linux is the include file *<netinet/ip.h>* which defines the Internet protocol header:

```
struct ip {
#if BYTE_ORDER == LITTLE_ENDIAN
u_char ip_hl:4, /* header length */
ip_v:4; /* version */
#endif
#if BYTE_ORDER == BIG_ENDIAN
u_char ip_v:4, /* version */
ip_hl:4; /* header length */
#endif
u_char ip_tos; /* type of service */
short ip_len; /* total length */
u_short ip_id; /* identification */
short ip_off; /* fragment offset field */
u_char ip_ttl; /* time to live */
u_char ip_p; /* protocol */
u_short ip_sum; /* checksum */
```

struct in_addr ip_src,ip_dst; / source and dest address */*
};

To "parse" the header, you'd do something like:

*strict ip *hdr = (struct ip *)buf;*
printf("checksum = 0x%04x\n", ntohs(ip->ip_sum));

This is considered the "elegant" way of doing things, because there is no "parsing" at all. On big-endian CPUs, it's also a no-op -- it costs precisely zero instructions in order to "parse" the header, since both the internal and external structures map exactly.

In C, though, the exact format of structures in undefined. There is often padding between structure members to keep integers aligned on natural boundaries. Therefore, compilers have directives to declare a structure as "packed" to get rid of such padding, this strictly defining the internal structure to match the external structure.

But this is the wrong wrong wrong way of doing it. Just because it's possible in C doesn't mean it's a good idea.

Some people think it's faster. It's not really faster. Even low-end ARM CPUs are super fast these days, multiple issue with deep pipelines. What determines their speed is more often things like branch mispredictions and long chain dependencies. The number of instructions is almost an afterthought. Therefore, the difference in performance between the "zero overhead" mapping of a structure on top of external data, versus parsing a byte at a time, is almost immeasurable.

On the other hand, there is the cost in "correctness". The C language does not define the result of casting an integer as shown in the above examples. As wags have pointed out, instead of returning the expected two-byte number, acceptable behavior is to erase the entire hard disk.

In the real world, undefined code has lead to compiler problems as they try to optimize around issues. Sometimes important lines of code are removed from a program because the compiler strictly interprets the rules of the C language standard. Using undefined behavior in C truly produces undefined results -- quite at odds from what the programmer expected.

The result of parsing a byte at a time is defined. The result of casting integers and structures is not. Therefore, that practice should be avoided. It confuses compilers. It confuses static and dynamic analyzers that try to verify the correctness of code.

Moreover, there is the practical matter that casting such things confuses programmers. Programmers understand parsing *external* formats fairly well, but mixing *internal/external* endianess causes endless confusion. It causes no end to buggy code. It causes no end to ugly code. I read a lot of open-source code. Code that parses integers the right way is consistently much easier to read than code that uses macros like *ntohs()*. I've seen code where the poor confused programmer keeps swapping integers back and forth, not understanding what's going on, and simply adding another byte-swap whenever the input to the function was in the wrong order.

Conclusion

There is the right way to teach endianess: it's a parser issue, dealing with external data formats/protocols. You deal with in in C/C++ the same way as in JavaScript or C# or any other language.

Then there is wrong way to teach endianess, that it's a CPU issue in C/C++, that you intermingle internal and external structures together, that you swap bytes. This has caused no end of trouble over the years.

Those teaching endianess need to stop the old way and adopt the new way.

Bonus: alignment

The thing is that casting integers has never been a good solution. Back in the 1980s and the first RISC processors, like SPARC, integers had to be aligned on even byte boundaries or the program would crash. Formats and protocols would be defined to keep these things aligned most of the time. But every so often, a odd file would misalign things, and the program would mysteriously crash with a "bus" error.

Thankfully, this nonsense has disappeared, but even today a lot of processors have performance problems with unaligned data. In other words, casting a structure on top of data appears to cost zero CPU instructions, but this ignore the often considerable effort it took to align all the integers before this step was reached.

Bonus: sockets

The API for network programming is "sockets". In some cases, you have to use the *ntohs()* family of macros. For example, when binding to a port, you execute code like the following:

sin.sin_port = htons(port);

You do this not because the API defines it this way, not because you are parsing data.

Some programmers make the mistake of keeping the byte-swapped versions of IP addresses and port numbers throughout their code. This is wrong. Their code should keep these in the correct format, and only passed through these byte-swapping macros on the Interface to the sockets layer.

Tweet

Share 314

By Robert Graham

2 comments:

Krister Walfridsson said...

Vector instructions still tend to crash on misaligned data – and compilers are now smart enough to use vector instructions when the C pointer rules say that the data must be aligned – so “this nonsense” have not disappeared. See e.g. http://pzemtsov.github.io/2016/11/06/bug-story-alignment-on-x86.html for an example where calculating IP header checksum sometimes crashes due to this.

1:45 AM

palako said...

"the little-endian format is 0x34 0x12 -- not 0x43 0x21 as you might naively expect trying to swap everything around in your mind."

That's if your mind works in hex and instead of swapping groups of 8 bits you swap groups of 4 bits. If you think big vs small endian naively as in reversing the order, 0x12 0x34 is 0001 0010 0011 010, which reversed is 0010 1100 0100 1000, in hex 0x2C 0x48.

Non constructive comment, just for a bit of added fun to the argument. Great post.

9:18 AM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)