

RSA

# PARSERS

**A Treatise on Writing Packet Parsers for Security Analytics**

1<sup>st</sup> Edition, Revised 10/11/2017

William Motley

## CONTENTS

<b>Preface</b>	<b>5</b>
Intended Audience	5
Prerequisites	5
Scope	6
About the Examples	6
About the Author	7
<b>Introduction</b>	<b>8</b>
It's All About the Meta	8
Meta is Answers	8
Parsers Are Not Signatures	9
Parsers Do Not Sessionize	11
<b>ONE: Fundamental Concepts</b>	<b>12</b>
Parsers See Payload	12
Request and Response Streams	13
Session	14
Multiple Parsers Extract Meta from the Same Sessions	15
<b>TWO: What to extract, How to get to it</b>	<b>17</b>
Meta Should be Interesting	17
Meta Values Must be Available	18
Index Keys	19
Tokens	20
Write the Parser	22
<b>THREE: Essential LUA</b>	<b>23</b>
Lua Command Line Interpreters	23
Special Values	24
Blocks	25
Lines	25
Comments	26

Variables	26
Functions	32
Operators	34
Conditional – If Then Else Elseif	36
Tables	37
More Lua Resources	43
<b>FOUR: Parser Basics</b>	<b>45</b>
Declarations	45
Token Matches	50
Extracting Values	52
Make Sure Values Were Extracted	58
Registering Values as Meta	59
Basic Parser Example	60
Loading and Enabling a Parser	61
<b>FIVE: Debugging</b>	<b>63</b>
Decoder Tuning for Parser Development and Debugging	64
Meta Scrub	65
Syntax Checking with nw-api.lua	66
Syntax Checking with the Log	67
Import a Pcap	67
Watch the Log	68
Reload the Parser	69
Review the Meta	69
Reset the Decoder	70
pcall()	70
<b>SIX: Beyond the Basics</b>	<b>74</b>
Removed Lua Functions	74
“self”	74
Use Small Payload Objects	75
Globals	76
Use Tokens Instead of payload:find()	80
Return Values in Conditionals	80

	3
Loops	81
String Manipulation	85
Multiple Numeric Values	92
bitwise operations	93
Service Identification	96
nw.isRequest() nw.isResponse()	97
Building Tables	98
Calling Other Functions	100
Network Meta	101
<b>SEVEN: Finishing Touches</b>	<b>105</b>
alert.id	105
Parser Version	106
Comments	106
Remove Debugging	107
<b>EIGHT: Miscellaneous</b>	<b>108</b>
payload.equal()	108
nwlanguagekey.getPathDefaults()	108
createPathMeta()	109
setKeys()	110
Renaming a Payload Object	111
Conditional Assignment	112
Register Meta Directly from Payload	114
Take Advantage of Meta Scrub	115
Optimizations	115
<b>NINE: Advanced</b>	<b>120</b>
Stream Objects and Packet Objects	120
nwsession	120
nwstream	120
nwpacket	122
Alternate Character Sets	125
Modules	126
Shared Values	128

	4
Options	129
<b>Appendix</b>	<b>130</b>
nw-api.lua	130
Example Parsers	130
Example Modules	131
<b>Errata</b>	<b>132</b>

## PREFACE

I've previously written two guides to writing parsers, but each assumed the reader already had some idea of what they were doing. The second, which covered Lua parsers, even required knowledge of Flex. At the time, I intended to eventually expand it to comprehensively cover all aspects of writing a parser even for the complete newcomer. This book is the fulfillment of that vision.

## INTENDED AUDIENCE

This book is intended for anyone who wants to write packet parsers for use with RSA Security Analytics.

However, it is not official documentation. And while there are no secrets revealed, it is most directly targeted at those internal to RSA itself.

On the other hand, I believe that anyone who works with SA in any capacity, especially customers and potential customers, would benefit from reading “*Meta is Answers*” and “*Parsers are Not Signatures*” from the Introduction. As well, any analyst whether they intend to write a parser or not may further benefit from reading the entirety of Section One, *Fundamental Concepts*.

## PREREQUISITES

The only prerequisites are experience examining network data, even just pcaps in Wireshark, and basic knowledge of TCP/IP and the OSI model.

Familiarity with Lua isn't strictly necessary - though having written code or scripts in any language (but especially Lua) would be helpful. A knack for either pattern recognition or boolean algebra certainly won't hurt.

Ultimately, understanding the subject of your parser is more important than knowing all the ins and outs of writing the parser itself. By writing a parser you are automating something you would otherwise do manually, teaching a machine as it were. If you don't understand what you are looking for, you can't write a parser to do it for you.

Start simple and don't get overwhelmed. Good parsers don't have to be complex, and you don't have to throw every technique and capability into every parser. Understand what it is you want to

accomplish, and focus on the individual steps to get you there. Even the most complex parsers start as broad strokes, with details and convolutions built layer by layer.

## SCOPE

Parsers, whether log or packet, are the very foundation of Security Analytics. However, this book is focused exclusively on **packet** parsers.

It covers parser-writing from the ground up. I've tried to include everything I think about and use when writing a parser, even if I don't discuss each explicitly, mention only in passing, or refer to some other documentation. Whatever I later realize I've omitted (and I'm sure there'll be something) I'll add to a future revision. As well, if when looking through the example parsers in the Appendix, you puzzle over something I explained poorly here, or neglected entirely, let me know and I'll do my best to get it into a future revision.

Most of the commands use either NwConsole or the REST API. This is because that is what I use and am familiar with. Of course everything can be done via the SA interface, but I don't use it myself for parser development. The inner workings of the console commands and REST calls are beyond the scope of this book; I simply show you what to type to accomplish a particular task.

## ABOUT THE EXAMPLES

Most of the snippets in this book are not necessarily intended to demonstrate a “best” way of doing anything. Many of them have caveats; some are even labelled as “Don't do this”. They are intended only to demonstrate a particular concept, functionality, or capability. Even for the snippets which are intended to demonstrate a “best practice”, understand what the code is doing so that you can apply the best practice, don't just copy it.

Conversely, the parsers in the Appendix are the same as they exist in Live at the time of writing. So of course I consider them to be exemplary. If you find an issue with any of them, please let me know, or open a Support ticket, including if possible a pcap that demonstrates the problem.

## ABOUT THE AUTHOR

I'm content technical lead for RSA ASOC. I mostly just write parsers. If it is a parser in Live, chances are I wrote it.

My professional experience before coming to RSA (via NetWitness) was in wide area network engineering of both large enterprise and service provider networks. How that led me to NetWitness is a story for another time.

I am not a programmer. I can sometimes make sense of simple C code, but I certainly can't write any. I also am not a security guru by any stretch of the imagination. Thankfully, neither of these are required skills for writing parsers.



## INTRODUCTION

### IT'S ALL ABOUT THE META

Security Analytics is all about meta: presenting meta, querying meta, storing meta, correlating meta, aggregating meta, analyzing meta, reporting meta.

But first there has to be meta to present, query, store, correlate, aggregate, and analyze, and report. Parsers are ultimately responsible for **all** of that meta.

What about feeds? App rules? ESA? Data science? None of those originate meta. All of them create meta *from* meta:

- Feeds match against individual items of meta from individual sessions
- App rules match against multiple meta items from individual sessions
- ESA correlates and aggregates meta items from multiple sessions
- Data Science applies algorithmic analysis to meta items from multiple sessions

“Meta” is the fundamental, foundational concept of Security Analytics, and parsers are responsible for all meta. If a parser registers bad meta, everything else falls apart. If you’re going to write a parser, the stakes are high.

### META IS ANSWERS

Have you ever picked through a pcap in Wireshark looking for “what”, “who”, “where”?

I’m guessing you’ve done that many, many times. Wouldn’t it be nice if there was a way to pick through those sessions automatically? A way to answer those questions without having to pick through the same sessions the same way over and over?

That’s exactly what parsers do – they find the answers, and register them as meta.

What kind of session was it?	service
Who initiated the session?	ip.src
What credentials did they present?	username, password
What did they access?	directory, filename
What did they do?	action
... and so on ...	...

So “meta” is the answers to the questions. By registering meta, parsers answer the questions – the same questions an analyst has when picking through a pcap in Wireshark.

## PARSERS ARE NOT SIGNATURES

Meta is not an IDS alert. Meta is not a big red flag waving “bad stuff here!” Likewise parsers are not IDS signatures.

IDS signatures are static; they say, “raise an alert if you see <whatever>”.

A parser is dynamic; it says, “answer all these questions when you see <whatever>”. That isn’t to say that a parser can’t do the same thing as an IDS signature – rather that a parser is capable of so much more.

The recent “Heartbleed” vulnerability is a good example of this. The definition of a successful Heartbleed attack is a TLS session which contains both:

- a) a TLS heartbeat request in which the amount of data requested is greater than the amount of data provided
- b) a subsequent TLS heartbeat response in which the amount of data provided is the same as the amount of data requested

Here’s what the IDS signature for Heartbleed as provided by US-CERT looks like:

```
alert tcp any [!80,!445] -> any [!80,!445] (msg:"FOX-SRT - Suspicious - SSLv3 Large Heartbeat Response"; flow:established,to_client; content:"|18 03 00|"; depth: 3; byte_test:2, >, 200, 3, big; byte_test:2, <, 16385, 3, big; threshold:type limit, track by_src, count 1, seconds 600; reference:cve,2014-0160; classtype:bad-unknown; sid: 1000000; rev:4;)
```

(Reproduced from: <http://ics-cert.us-cert.gov/alerts/ICS-ALERT-14-099-01E>, 04/30/2014)

It looks for the hex digits 0x180300 at the beginning of a TCP packet which isn’t using ports 80 or 445, followed by a hex value greater than some arbitrary size. Is that a Heartbleed attack? Maybe. Was it successful? Who knows. How many false positives is the analyst going to track down and examine before giving up and ignoring the signature entirely?

More importantly, how many false **negatives** will slip by when:

- those three hex digits aren’t at the beginning of a packet (TLS is a streaming protocol, after all)
- it is a TLS 1.0 (0x18 0x03 0x01), TLS 1.1 (0x18 0x03 0x02), or TLS 1.2 (0x18 0x03 0x03) heartbeat instead of an SSL 3.0 heartbeat
- the amount of data requested is smaller than some arbitrary threshold?

How can a parser do better? Instead of what a signature can do:

*“when you see Heartbeat response (of some arbitrary size), raise an alert!”*

A parser could answer questions:

*“when you see a Heartbeat request, does it request more data than it provides, and does the server respond with the amount of data requested?”*

The TLS\_lua parser detects Heartbleed by examining TLS heartbeat requests and responses themselves, looking for the conditions which define a successful Heartbleed attack. The TLS\_lua parser is in the Appendix if you’d like to have a look for yourself.

However, this is just a step up from being a signature – it only detects one thing, a “Heartbleed” attack. The better you can extricate yourself from an IDS-bound mindset, the more powerful parsers can be for you.

Consider some meta from a hypothetical session:

service	80
action	get
directory	/
filename	arch59_win32.exe
extension	exe
alias.host	svoemesto.ru
tld	ru
client	Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)
server	NginX 1.0.13
content	application/octet-stream
filetype	windows_executable

Suspicious? Any .ru domain automatically raises eyebrows, and nginx has a poor reputation as well. Downloading an executable is risky, and may be against policy. Yet none of those qualities, alone or even together, is a smoking gun.

When analysts have to spend most of their time examining pcaps just because of a flagged a TLD, or a server vendor, an IP address on some blacklist, or whatever, they quickly become overwhelmed.

What if the headers were spoofed to look completely benign? Here’s another hypothetical session, without a suspicious TLD or server:

service	80
action	get

directory	/
filename	potato.jpg
extension	jpg
alias.host	idahofarmsupply.com
tld	com
client	Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0)
server	Apache/2.2.22
content	image/jpeg
filetype	windows_executable

Ruh-roh. The client requests a jpeg and the server claims to respond with a jpeg, but actually responds with an executable.

What would an IDS signature or blacklist have to say about this session? Unless the exploit code is known in advance and exactly matches a signature – nothing.

## PARSERS DO NOT SESSIONIZE

Parsers only see sessions that have already been reconstructed by Decoder. What does that mean?

As Decoder watches network traffic, it determines the beginning of each session, keeps track of all the packets in the session, determines when the session is over, then compiles all the packets that constituted that session. This is known as “sessionization”.

Parsers can’t do any of that.

If the session uses some odd or proprietary layer 2 through 4 protocol, chances are Decoder has no idea how to reconstruct it. In that case, either parsers will never see the session or what they see will be munged beyond recognition.

Likewise, you don’t need a “GRE” parser, or a “VLAN” parser, or an “MPLS” parser. Decoder does all that, not parsers.

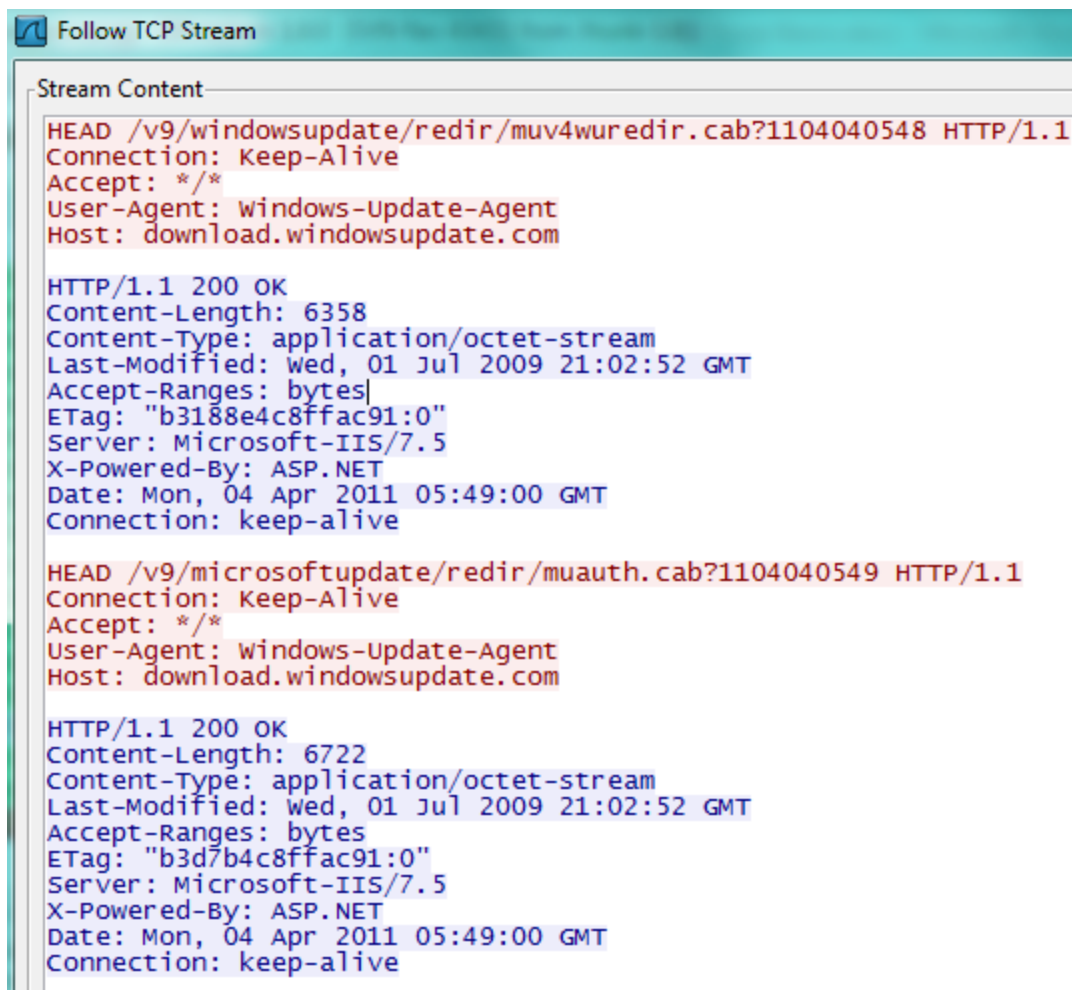
Generally, parsers are only concerned with the stuff in layers 5 – 7. That isn’t to say you can’t extract values from lower-layer fields – rather that if Decoder doesn’t understand the session in the first place, the parser will never have the opportunity.

## ONE: FUNDAMENTAL CONCEPTS

### PARSERS SEE PAYLOAD

Decoder performs sessionization, in which all of the packets which constitute a session are compiled. Then Decoder strips all the headers and footers from each packet – what’s left is **payload**. Payload is what Decoder presents to parsers.

If you’ve ever used the “Follow TCP Stream” capability of Wireshark, you’ve seen what this looks like.



```

Follow TCP Stream

Stream Content

HEAD /v9/windowsupdate/redirect/muv4wuredir.cab?1104040548 HTTP/1.1
Connection: Keep-Alive
Accept: */*
User-Agent: windows-update-agent
Host: download.windowsupdate.com

HTTP/1.1 200 OK
Content-Length: 6358
Content-Type: application/octet-stream
Last-Modified: Wed, 01 Jul 2009 21:02:52 GMT
Accept-Ranges: bytes
ETag: "b3188e4c8ffac91:0"
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
Date: Mon, 04 Apr 2011 05:49:00 GMT
Connection: keep-alive

HEAD /v9/microsoftupdate/redirect/maauth.cab?1104040549 HTTP/1.1
Connection: Keep-Alive
Accept: */*
User-Agent: windows-update-agent
Host: download.windowsupdate.com

HTTP/1.1 200 OK
Content-Length: 6722
Content-Type: application/octet-stream
Last-Modified: Wed, 01 Jul 2009 21:02:52 GMT
Accept-Ranges: bytes
ETag: "b3d7b4c8ffac91:0"
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
Date: Mon, 04 Apr 2011 05:49:00 GMT
Connection: keep-alive

```

Parsers do not “see” packet headers, frame headers, packet footers, or frame footers. Parsers only see payload.

(There is still a way for parsers to extract values from headers and footers, but that is an advanced topic covered in Section Nine.)

Don't confuse packet or frame headers with higher layer headers. IP and TCP headers are not payload. HTTP headers *are* payload, as seen in the example above.

## REQUEST AND RESPONSE STREAMS

A session is a conversation between two participants (we're talking unicast here, don't overthink this): the host which began the conversation, and the host which responded. Typically these correspond to a client and a server, respectively.

All of the packets from the initiator to the responder are the "request stream".

All of the packets from the responder to the initiator are the "response stream".

There are all sorts of mechanisms by which Decoder determines the initiator and the responder – all that is beyond the scope of this guide. Suffice it to say that it *almost* always gets it right. Keep in mind that the initiator may be the "server" in the protocol – that doesn't matter; it is still the request stream regardless.

Again, you've already seen this.

Request stream:

```
Stream Content
HEAD /v9/windowsupdate/redirect/muv4wuredir.cab?1104040548 HTTP/1.1
Connection: Keep-Alive
Accept: */*
User-Agent: windows-update-agent
Host: download.windowsupdate.com

HEAD /v9/microsoftupdate/redirect/mauth.cab?1104040549 HTTP/1.1
Connection: Keep-Alive
Accept: */*
User-Agent: windows-update-agent
Host: download.windowsupdate.com

HEAD /v9/windowsupdate/redirect/muv4wuredir.cab?1104040549 HTTP/1.1
Connection: Keep-Alive
Accept: */*
User-Agent: windows-update-agent
Host: download.windowsupdate.com

HEAD /v9/microsoftupdate/redirect/muv4muredir.cab?1104040549 HTTP/1.1
Connection: Keep-Alive
Accept: */*
User-Agent: windows-update-agent
Host: download.windowsupdate.com
```

Response stream:

```
Stream Content
HTTP/1.1 200 OK
Content-Length: 6358
Content-Type: application/octet-stream
Last-Modified: Wed, 01 Jul 2009 21:02:52 GMT
Accept-Ranges: bytes
ETag: "b3188e4c8ffac91:0"
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
Date: Mon, 04 Apr 2011 05:49:00 GMT
Connection: keep-alive

HTTP/1.1 200 OK
Content-Length: 6722
Content-Type: application/octet-stream
Last-Modified: Wed, 01 Jul 2009 21:02:52 GMT
Accept-Ranges: bytes
ETag: "b3d7b4c8ffac91:0"
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
Date: Mon, 04 Apr 2011 05:49:00 GMT
Connection: keep-alive

HTTP/1.1 200 OK
Content-Length: 6358
Content-Type: application/octet-stream
Last-Modified: Wed, 01 Jul 2009 21:02:52 GMT
Accept-Ranges: bytes
ETag: "b3188e4c8ffac91:0"
Server: Microsoft-IIS/7.5
X-Powered-By: ASP.NET
Date: Mon, 04 Apr 2011 05:49:00 GMT
Connection: keep-alive

HTTP/1.1 200 OK
Content-Length: 6253
Content-Type: application/octet-stream
Last-Modified: Wed, 01 Jul 2009 21:02:52 GMT
Accept-Ranges: bytes
```

## SESSION

A session is the entire request stream followed by the entire response stream.

The actual order of request and response packets in a session is typically something like:

request packet	
	response packet
request packet	
	response packet
request packet	
	response packet

However, that is **not** the order in which a parser sees the session.

Instead, a parser sees the entire request stream followed by the entire response stream:

```
request packet
request packet
request packet
response packet
response packet
response packet
```

The entire request stream is seen before any of the response stream.

So while an HTTP session may consist of:

```
GET /pageone.html HTTP/1.0
                                HTTP/1.0 200 OK
                                <html> ...
GET /pagetwo.html HTTP/1.0
                                HTTP/1.0 200 OK
                                <html> ...
```

What is presented to a parser is:

```
GET /pageone.html HTTP/1.0
GET /pagetwo.html HTTP/1.0
HTTP/1.0 200 OK
<html> ...
HTTP/1.0 200 OK
<html> ...
```

This may seem counter-intuitive, but it actually makes parser writing easier. For one thing, you don't have to be concerned with packet boundaries or bouncing between streams trying to skip ACKs.

## MULTIPLE PARSERS EXTRACT META FROM THE SAME SESSIONS

If you're familiar with log decoding, you understand that a log message is only parsed by a single log parser. This is **not** true of packet parsers.

In packet parsing, every parser has the opportunity to parse every session. This is essential to flexibility. At its simplest, each parser only has to do one thing, instead of every parser having to do everything.



Consider a webmail session which contains an executable. If a session could only be parsed by one parser, then a single parser would have to contain the logic for:

- extracting HTTP meta
- extraction mail meta
- detecting executables

What if it were an SMTP session? Different parser (or worse, logic for SMTP would have to be added to the same parser). What if you wanted to detect PDF files? You'd have to add that logic to all parsers.

Instead,

- the HTTP parser registers HTTP meta
- the MAIL parser registers mail meta
- the windows\_executable parser registers the presence of the executable

All three parsers extract meta from the same session.

For an SMTP session,

- the SMTP parser registers SMTP meta
- the MAIL parser registers mail meta
- the windows\_executable parser registers the presence of the executable

Again, all three parsers extract meta from the same session.

For pdf detection, the fingerprint\_pdf parser would detect the presence of a pdf in either the webmail or SMTP (or any other) session.

## TWO: WHAT TO EXTRACT, HOW TO GET TO IT

Before you write each and every parser, there are two questions you must answer:

- a) What meta do you want to extract?
- b) What tokens do you use?

If you don't have answers to those questions, the parser will be a mess and probably won't ever come together.

If you have good solid answers to those questions, the parser will almost write itself.

### META SHOULD BE INTERESTING

Storage in Security Analytics is finite, and costly. For most customers, retention time is a concern for compliance, historical visibility, and budgeting. Every meta value occupies storage; every meta value contends for retention with every other meta value.

Query response is a frustration for analysts. Real-time analytics is pointless if the analyst can't see the results for hours. Every meta value slows down query response just a little bit – the cumulative effect of lots of meta values is that query response slows down a lot.

Clutter reduces visibility. The more meta values on the screen, the harder it is to pick out the important stuff. This causes analysts to immediately look at “risk” meta, and nothing else – completely missing attacks like the “jpg -> executable” example from the Introduction.

All three of these issues are exacerbated by meta that isn't forensically valuable. Keep that in mind when you are deciding what meta values the parser should extract.

If the parser is intended for Live, always assume that every customer will enable it. Interesting meta solves a use case that is relevant to many if not all customers.

There are usually alternate ways to satisfy a use case. Consider a request such as, “I want all the values of all HTTP transfer-encoding headers.”

That header is sometimes used for command and control, and detecting nefarious use of it is a valid use case. Yet overwhelmingly those values are forensically useless. Registering them as meta would bloat the metadb (slowing query response, reducing visibility, and decreasing retention).

As an alternative, since a “transfer-encoding” header should only contain certain values, the parser could check and alert if the header contains an invalid value. That should result in a

greater than 99% reduction of potential meta while satisfying the use case and keeping the meta “interesting”.

(It’s also worth noting that while such a capability is mildly signature-like, it can’t be accomplished with a signature...)

If you are writing a parser in a Professional Services capacity sometimes you don’t have the luxury of saying “no.” When a customer can’t be convinced there’s a better way and insist on shooting themselves in the foot, you’re getting paid to hand them the gun. But at least you’re only handing the gun to one customer, not to every customer (via Live).

## META VALUES MUST BE AVAILABLE

It should seem obvious that in order to extract and register values as meta, the values must be present and visible in the session.

What may not be obvious are the conditions in which those requirements cannot be met.

A general rule of thumb is:

If you can see it in “Follow Stream” (Wireshark), you can extract it.

If you can’t see it, you can’t extract it.

## Parsers cannot perform decryption

- a) SSL/TLS – something like Netronome can help here, by feeding Decoder decrypted sessions
- b) Encrypted files – nothing can be done about these, a parser may detect the filetype and even that the file is encrypted, but it can’t see into the encrypted contents

## Parsers cannot perform decompression

- a) Zip, rar, et al – a parser may detect the filetype and even extract filenames from the archive headers, but it can’t unpack the archive contents
- b) Microsoft Office documents are either “Compound Document Format” (95-2003) or zipped XML (2007+) – a parser can’t unpack the contents of the documents

### Parsers cannot perform large-scale de-encoding

- a) HTTP server content-encoding such as “chunked”, “deflate”, and “gzip” - This is the biggest issue with parsing HTTP contents. Server replies, which contain the actual web pages, images, files, et al, are often content-encoded or transfer-encoded. If an organization uses HTTP proxies, it would be wise to strip “Accept-Encoding” headers from all HTTP requests. Remember the *If you can see it in ‘Follow Stream’...* rule.
- b) MIME base64 encoding, such as email attachments – A parser can de-encode base64, but best practice is to only de-encode small chunks, just enough to get the information the parser needs, not entire multi-meg files.

### Sessions may not be understood by Decoder

- a) If it some odd or proprietary layer 2-4 protocol, chances are Decoder has no idea how to properly sessionize it
- b) ICAP and HTTP proxies can contain requests from several clients, and responses from several servers, causing complications

### Important information may not be in the stream at all

- a) For screen protocols such as RDP and VNC, and even TN3270/5250, the meaning of areas of the screen is known to the server but isn’t in the stream (e.g., a click or text input at coordinates x,y means one thing, but at coordinates i,j means something else)
- b) ICAP will contain the HTTP request *or* response, never both.

### INDEX KEYS

All meta values are registered and organized with index keys. Think of index keys as categories:

bob	username
delete	action
/var/log/	directory
wtmp	filename

Ask yourself to what category does this value belong, and register it appropriately. Like bad meta, even if good meta is registered with inappropriate keys then everything else falls apart.

Furthermore, as much as possible you should restrict yourself to the standard index keys. Nothing is stopping you from making up your own keys and registering meta with them. But consider:

- If it can't be categorized into a standard index key, it probably isn't interesting
- Non-standard index keys won't be consumed by standard content such as,
  - Feeds
  - ESA
  - Data Science algorithms

## TOKENS

When you pick through a pcap manually, you're scanning for certain things:

- a) Things that should always be there, to let you know you're looking at what you think you are, something you're interested in rather than something you're not
- b) Things that are near the answers (values) you're look for

Parsers do the exact same. Those are called "tokens".

### Tokens should always be there

Tokens let the parser know that the session is one that it is interested in. If the tokens aren't there, the parser isn't interested.

Tokens essentially say, "when you see this, do something." If "this" isn't seen, then "something" isn't done.

(FIXME: Move to "Globals") An intermediate technique is to use a combination of tokens, e.g.,

A or B or (C and D)

## Tokens should be unique

The parser will do “something” each and every time it sees “this”. It doesn’t know yet whether the session is one that it is interested in or not.

The fewer times the parser is run on sessions it isn’t interested in, the better Decoder will perform and the less chance there is of registering bad meta.

A good token is longer than a couple of bytes, even though 2 bytes is the technical minimum. Decoder won’t even load a parser that declares a 1-byte token.

The best tokens are at least 6 bytes. Longer is gooder.

“Carriage return, line feed” (0x0D 0x0A) is a terrible token. Don’t ever use it. Not only is it too short, but it is extremely common and will hit in all kinds of sessions the parser won’t care about at all.

A practical minimum is 4 bytes. A 3-byte token will still “hit” far too often even in random data. Even a 4-byte token will hit fairly often – be careful, don’t rely solely on 4 bytes if you can help it at all.

## Tokens should be near values

You’ll need a way of determining where the values you want to register begin and end. So tokens which are value boundaries accomplishes two things at once - “here’s where we want to be” and “here’s the value we want to register”.

Many times you can’t feasibly tokenize value boundaries such as when the resulting token would be too short, or it just doesn’t make sense to do so. In those cases, the closer your tokens are to the values you want to extract the better. The parser will have to do less moving around and “find”-ing, and Decoder will perform better as a result.

## Tokens are static

Even though the values you want to extract are dynamic, the tokens you use to get to them aren’t.

Tokens are strings of characters or sequences of hex digits (or combinations of both).

Well, really tokens are always sequences of hex digits – under the hood, characters are represented by their UTF-8 byte value, e.g.

FOO = 0x46 0x4F 0x4F

For that reason, tokens are also case-sensitive. “FOO” won’t match “Foo” or “foo”.

Tokens are not regular expressions. To be blunt, if you think you need to use a regular expression, you need to come up with a better token.

## WRITE THE PARSER

Once you understand how sessions are presented to parsers, and have concrete answers to:

- a) What do you want to extract?
- b) How will you know you are where you want to be?
- c) How will you know where the values you want begin and end?

Writing the parser itself is just a technical exercise in knowing how to use the parser language itself.

## THREE: ESSENTIAL LUA

The goal of this section is provide just enough of an introduction to Lua to facilitate writing basic parsers. This is not intended to stand as a comprehensive reference of Lua itself. For more information, see the resources listed in “More Lua Resources”. The book “*Programming in Lua*” is especially recommended.

### LUA COMMAND LINE INTERPRETERS

Having a Lua CLI will be indispensable, both for learning Lua and later for developing your parsers.

For learning Lua, follow the “Programming in Lua” e-book in one window with the CLI in another.

For parser development, you will be able to:

- a) Type (or paste) your functions and algorithms into the CLI – and even run them with dummy data to ensure they do what they are supposed to
- b) Load entire parser files into the CLI. They won’t actually parse anything there, but the CLI will alert you if there are any syntax errors that would prevent the parser from loading on Decoder. If it loads in the CLI, it will load in Decoder.

### Lua For Windows

Download the binary installer from Google Code:

<https://code.google.com/p/luaforwindows/>

### Lua for OSX, BSD, and Linux

There is no binary distribution for Unix variants. Instead you’ll need to compile from source yourself. There are various web sites that have detailed instructions.



## SPECIAL VALUES

### NIL

The concept of NIL is very important in Lua.

NIL is a non-value. NIL means “no value”, “empty”, “undefined”. It specifically denotes the absence of any useful or meaningful value whatsoever.

NIL is not 0.

- 0 / “zero” is a value
- NIL is not a value

Undeclared variables are NIL.

Declared variables that haven’t been assigned a value are NIL. Unless a variable is assigned a non-NIL value, it effectively does not exist. If you assign NIL as the value of a variable, you have deleted the value of the variable entirely.

Attempting to use a NIL variable in an arithmetic or string operation **will** result in an error:

```
a = nil
b = a + 1 ERROR
```

Don’t interpret this to mean that you can’t assign a value to variable that is currently NIL:

```
a = nil
b = 1
a = b + 1 OK
```

NIL is still valid in Boolean operations, in which it evaluates to false:

```
a = nil
b = true
if a or b then OK
```

### TRUE AND FALSE

A value may be a literal, boolean TRUE or FALSE. This is not a string “true” or string “false”.

Nor are FALSE and NIL equivalent. FALSE is still a value, while NIL is the absence of value.

## -1 (“minus one”)

This means “the end” or “the last”. It could mean “the end of the string”, or “the last byte of the payload object”, depending upon what it is being used with.

You can also use -2, -3, and so on. These mean “second to last”, “third from last”, etc., respectively.

## BLOCKS

A block is a logical set of statements, such as a function, if, or while loop.

A block may (and usually does) consist of smaller blocks, such as an if-block within a function.

A block is almost always terminated with “end”.

For example:

```
function myFunction(a, b)
  if a == b then
    while a == b do
      a = a + 1
    end
  end
end
```

In the above:

- everything between “function” and the last “end” is a block
- everything between “if” and the 2<sup>nd</sup>-to-last “end” is a block
- everything between “while” and the first “end” is a block

Indentation and nesting is purely stylistic, for readability. Blocks do not need to be nested. There is no special significance to indentation.

## LINES

There are no special terminating requirements for lines, such as semicolons.

You may even combine multiple statements and even multiple blocks into a single line:

```
if a == b then c = c + 1 end if c == 2 then c = c + 1 end a = b b = c
```

That said, please don't.

## COMMENTS

Comments in Lua are denoted by a pair of hyphens:

```
-- This is a comment
if varA > varB then
    -- another comment
end
```

Everything beyond the comment marker through the end of the line is ignored.

Statements may appear on the same line as a comment:

```
varA = 1 -- initialize varA
varB = 2 -- initialize varB
```

Everything up until the comment marker is evaluated.

Comments may also span multiple lines:

```
if varA > varB then
    --[[ This is a comment that
        spans multiple lines.
    --]]
```

To mark the beginning of a multi-line comment, append two opening square brackets to a comment marker.

To mark the end of a multi-line comment, append two closing square brackets to a comment marker.

## VARIABLES

### Naming Variables

Variable names may be any string of letters, digits, and underscores that does not begin with a digit.

The following are all valid variable names:

```
i
j
x100
_xy
myLongVariableName
my_10th_REALLY_LONG_VARIABLE_NAME
```

Variable names are case-sensitive:

```
myVar

is not the same variable as

myvar

nor is

MYVAR
```

### *Reserved Words*

There are some reserved words that cannot be used as variable names because they already have meaning in Lua:

```
and  break  do    else  elseif  end  false
for  function if    in    local  nil   not
or   repeat  return then  true   until while
```

Also, there are names that you shouldn't use for your variables in a Lua parser because they overlap some parser-specific functions:

byte	getPorts	getTcpSeqAndFlags	int32
equal	getRequestStream	getTimestamp	int8
find	getResponseStream	hasSessionNext	len
getAddresses	getSessionNext	hasSessionPrevious	pos
getDestination	getSessionPrevious	hasStreamNext	short
getFirstPacket	getSize	hasStreamPrevious	sub
getLastPacket	getSource	isRequest	tostring
getNextPacketPayload	getStats	isResponse	uint16
getPacketPayload	getStreamNext	int	uint32
getPayload	getStreamPrevious	int16	uint8

Variables don't need to be declared or initialized. You can just start using a variable name.

However, it will still often make sense to initialize a variable by giving it a default value before you use it, or to scope it locally instead of globally (see “Variable Scope”, below).

## Types

The most common value “types” in lua are,

- string
- number
- table
- function

A variable in lua is not explicitly tied to a type as it is in some languages. The “type” is only a description of the value held by the variable, which may be changed. If a variable holds a string, it is a type “string”. If that same variable is subsequently assigned a numeric value, it is then a type “number”.

```
foo = "1"    → type is string
foo = 1      → type is number
```

## type()

The lua function `type()` returns the type of value passed to it.

```

type("1")  →  string
type(1)    →  number

```

Most commonly, a variable is passed to `type()` in order to determine the type of value currently held by that variable.

```

foo = "1"
type(foo)  →  string
foo = 1
type(foo)  →  number

```

## Variable Scope: Globals and Locals

“Scope” refers to whether a variable has a value (or even exists) outside of the block in which it was used.

**global** variables retain their values and may be referenced outside of the block in which they are assigned a value

**local** variables are NIL outside of the block in which they are declared

Any variable not declared as “local” is automatically “global”. But you don’t have to declare that the variable is “local” each time you use it or assign a value to it, only the first time within its block.

Example of “global”:

```

function funcOne()
  varA = 1
end

function funcTwo()
  varB = varA + 1
end

```

Assuming that `funcOne()` was called before `funcTwo()`, the value of `varA` would be available to reference in `funcTwo()`.

Example of “local”, identical to the above except that `varA` is declared as local:

```

function funcOne()
  local varA = 1
end

```

```
function funcTwo()
  varB = varA + 1 ERROR
end
```

varA is local to funcOne(). In funcTwo() it has no value – it is NIL.

A locally scoped variable retains its value within sub-blocks:

```
function funcOne()
  local varA = 1
  if varA == 1 then
    varB = varA + 1 OK
  end
end
```

But when you leave the block in which a local was declared it goes back to NIL:

```
function funcOne()
  if someGlobal == 1 then
    local varA = 1
  end
  varB = varA + 1 ERROR
end
```

Why bother with locals?

- Efficiency: Because locals only exist within their blocks, it is much faster to access a local variable, and you don't waste memory holding onto values you don't need - garbage collection takes place as soon as you leave a block.
- Errors: you are much less likely to make a logic error by accidentally referencing a variable that has a value set from another block

Use local variables whenever possible.

## Assignment

Assigning a value to a variable is very simple:

```
local myNumber = 100
myOtherNumber = myNumber
myString = "one hundred"
```

Variables may hold other types of values besides numbers and strings as well:

- functions
- tables
- truth (TRUE / FALSE)

Furthermore, variables are not tied to a specific type of value. Any type of value may be held by any variable:

```
a = 100
a = "one hundred" OK
a = "100" OK
a = true OK
```

### *Multiple Assignment*

More than one variable may be assigned a value at the same time.

```
a, b, c, d = 1, 2, "foo", true
```

The variables to be assigned are on the left, separated by commas. The values to be assigned are on the right, also separated by commas. This is equivalent to,

```
a = 1
b = 2
c = "foo"
d = true
```

If there are more variables than values, the extra variables are NIL. This is not an error.

```
a, b, c = 1, 2
```

is equivalent to,

```
a = 1
b = 2
c = nil
```

Likewise, if there are more values than variables, the extra values are silently ignored.

```
a, b, c = 1, 2, 3, 4
```

is equivalent to,

```
a = 1
b = 2
c = 3
```



The “4” is dropped, not being assigned to any variable.

## FUNCTIONS

Lua does not have a “main” function. However it does have functions, and functions may call other functions.

Furthermore, functions in Lua are “first-class values”. What does that mean?

*It means that, in Lua, a function is a value with the same rights as conventional values like numbers and strings. Functions can be stored in variables (both global and local) and in tables, can be passed as arguments, and can be returned by other functions.*

(<http://www.lua.org/pil/6.html>)

Essentially, functions are another type of value that is stored by variables.

Given that, function names are really just variable names. Thus function names must abide by the same rules as variable names (no leading digit, etc.)

Functions accept values that are passed to them. These values are assigned to variables declared in the function definition itself. Such variables are automatically local to the function – they do not have to be otherwise declared, initialized, or reset (but nor are they available outside of the function).

Example function variable passing:

```
function funcOne(a, b, c)
    local d = a + b + c OK
end
```

Just as with ‘Multiple Assignment’, fewer variables may be passed to a function than it expects. This is not automatically an error. Values would be assigned in order (e.g., “a”, then “b” ...) Any variable that does not get a value will be NIL.

In the example above, if only two values were passed to funcOne rather than three, then c would be NIL,

```
local d = a + b + c ERROR
```

The important thing to note is that passing fewer values did not cause the error directly. Rather that since fewer variables were passed, a variable ended up being NIL – when that variable was referenced, the error was thrown.

More variables may be passed to a function than it expects. This is also not an error. Again, values would be assigned in order, and any extra values would be simply ignored. In the case of extra values, no variable would get a value of NIL.

## Calling a Function, Assigning Return Values

Functions are called by simply referencing them by name:

```
funcOne()
```

Including any values you want to pass, separated by commas:

```
funcOne(1, 2)
```

Values passed may themselves be variables, or even a combination:

```
myVarA = "foo"
funcOne(1, myVarA)
```

You can assign return values (if any) from a function to a variable:

```
local myVarC = funcOne(myVarA, myVarB)
```

Just as with ‘Multiple Assignment’, more or fewer variables may be assigned returned values:

```
myVarC, myVarD = funcOne("foo")
```

If `funcOne()` returns two values, they will be assigned to `myVarC` and `myVarD` in order.

If only one value is returned, it will be assigned to `myVarC`.

If more than two values are returned, the extras will be silently ignored.

## Libraries of Functions

Many functions are stored in “libraries”, and calling them is a little different. For example, all standard functions pertaining to string manipulation are stored in the “string” library.

To call a function in a library, you append the name of the function to the name of the library:

```
varB = string.find(varA, "foo")
```

“string” is the name of the library, “find” is the name of the function

Behind the scenes, libraries are really Lua tables (covered later in this section) which store functions. What you are actually doing is referencing a table named “string” and an entry in that table named “find”.

However, there is another syntax that you will also need to be aware of (note the colon below):

```
position = payload:find("foo")
```

Section Eight will describe what’s going on behind the scenes here. For now, just be aware of that subtle difference.

## OPERATORS

### Arithmetic

Lua supports the common arithmetic operators:

- + addition
- subtraction
- \* multiplication
- / division

Lua also adheres to the standard order of operations:

- 1) parentheses and brackets
- 2) exponents and roots
- 3) multiplication and division
- 4) addition and subtraction
- 5) left – to – right

### Relational

Lua uses the common relational operators:

- > greater than
- < less than
- >= greater than or equal too
- <= less than or equal too

As well as,

== equal to  
 ~= not equal to

Be careful with “==”, don’t confuse it with assignment.

a == 1 **ERROR**

if a = b then **ERROR**

## Boolean

Logical operators (and, or, not) have standard boolean meanings.

Truth may also be evaluated directly. These two statements are functionally identical:

```
if sawEndOfLine == true then
  ...
if sawEndOfLine then
  ...
```

For efficiency, Lua will use shortcuts when evaluating “and” / “or”:

- if the left term of an “and” is FALSE, the right term is not evaluated
- if the left term of an “or” is TRUE, the right term is not evaluated

## *NIL in boolean logic*

When used in a conditional, NIL evaluates to FALSE. But there’s a more sophisticated way to use NIL as well.

```
if a == nil then
  print(“a has no value”)
else if not a then
  print(“a is false”)
else
  print(“a has a value”)
end
```

This technique can be used to differentiate whether a condition is truly false, or is simply unknown yet.

## Precedence

When mixing operators, precedence is evaluated in the following order:

```

^
not
* / %
+ -
..
< > <= >= ~= ==
and
or

```

It is probably best to always use explicit parentheses to avoid ambiguity.

## CONDITIONAL – IF THEN ELSE ELSEIF

An if-block evaluates a condition, and only if the condition is true does it executes the statements within the block.

```

if a == b then
    c = a + b
end

```

Only if “a” is equal to “b” will the statement “c = a + b” be executed.

The keyword “then” is required.

Lua also supports “else” and “elseif”:

```

if a < b then
    c = b - a
else
    c = a - b
end

```

If the evaluation is false, then the statements within the else-block are executed instead.

```

if a < b then
    c = b - a
elseif a == b then
    c = 0

```

```

else
    c = a - b
end

```

If the initial condition is false, then the “elseif” condition is evaluated. If it is true then the statements within the elseif-block are executed instead.

But if the “elseif” condition is also false, then the statements in the else-block are executed.

Any value other than NIL and FALSE is TRUE. An empty string is still a value, therefore it is true.

```

emptyString = ""
if emptyString then → TRUE

```

## TABLES

Tables are like “arrays”, “lists”, and “matrices” of other languages. They can perform the functionality of all of these, and more. Just about any data structure you may need can be represented with a table efficiently.

Tables may hold any values: strings, numbers, boolean, functions, and even other tables. Furthermore, tables are not limited to holding a particular type of value: a single table may hold any combination of types of values simultaneously.

This section won’t do justice to working with tables. For better understanding, refer to Chapter 11 of *Programming in Lua*.

### Declaring a Table

Unlike other variables, tables must be declared. You can’t just start adding values to a table without having declared it as a table.

```
myTable = {}
```

The “{}” is required in order to declare “myTable” as a table.

Like variable names, tables names may be any string of letters, numbers, and underscores that doesn’t begin with a number and isn’t a reserved word.

Also like variables, tables have a scope (“global” or “local”). If you don’t declare a table as “local” then it is by default “global”.

```
local myTable = {}
```

Tables may grow – the size of a table isn't specified or restricted. Values may be added to a table at any time.

## Simple Tables

The simplest tables consist of index:value pairs.

The type of value may all the be same,

<u>index</u>	<u>value</u>
1	"A"
2	"B"
3	"C"
4	"D"
5	"E"

Or mixed,

<u>index</u>	<u>value</u>
1	"a"
2	6549
3	true
4	function(a,b) if a and b then local c = a + b return c end end
5	"Lorem ipsum dolor sit amet"

## Table Indexes

Indexes may be numeric (e.g., "1", "2", "3") or named (e.g., "name", "address", "phone").

### Numeric Indexes

As in the examples above, the index may be numeric. Some things to keep in mind with numeric indexes:

- Don't use index "0".

Lua is a 1-based language. Using an index of 0 won't result in an error, but referring to index 0 can sometime have unpredictable results. So best to avoid it altogether.

- It's okay to leave gaps.

You won't cause errors by having index 1, 4, 9, 16... But it may be easier to retrieve values later if there aren't gaps.

- It's okay to skip around

You won't cause an error by assigning a value to index 2 before assigning a value to index 1. But again it may be easier to retrieve the values later.

### *Named Indexes*

Index may also be named with a string,

<u>index</u>	<u>value</u>
"firstName"	"Elwood"
"lastName"	"Blues"
"streetAddress"	"1060 W. Addison"
"city"	"Chicago"
"state"	"IL"

You don't even have to stick to one or the other within the same table – you could have some table entries indexed numerically and some named.

### *Table Length*

It's often handy to know how many entries there are in a table. Fortunately, lua provides an easy way to get this value:

```
#myTable
```

returns the number of entries in "myTable"

Note that this works only for numeric indexes with no gaps.

- length returned for a table indexed by name will be "0" (not NIL)



- length returned begins with “1” up to the first gap
  - indexes 1, 2, 3, 5 returns “3”
  - indexes 2,3,4,5 returns “0”
- an error results from trying to get the length of a table that doesn’t exist
 

```
myTable = nil
tableLength = #myTable ERROR
```

## Referencing Table Entries

Table entries may be used in all operations just like variables. To use the value of a table entry, reference the name of the table and the index of the entry.

There are two valid syntaxes for referencing table entries.

### *Square Brackets*

This is the most flexible syntax. It may be used with numeric or named indexes.

To refer to a numeric index, use the numeric value, unquoted.

Table named “planets”, with indexes 1, 2, and 3:

```
planets[1] = “mercury”
planets[2] = “venus”
planets[3] = “earth”
```

To refer to a named index, use the index name within quotes:

Table named “mercury”, with indexes “type”, “diameter”, “distanceFromSun” and “inhabited”:

```
mercury[“type”] = “terrestrial”
mercury[“diameter”] = 4880
mercury[“distanceFromSun”] = 0.38
mercury[“inhabited”] = false
```

You can also use a variable to refer to an index, by using a variable name without quotes:

```
planet = 3
if planets[planet] == “earth” then
```

```
    print("hello world")
end
```

## Dots

This syntax may only be used with named indexes. Variables cannot be used:

```
if mercury.diameter < earth.diameter
    print("It's a small world after all.")
end
```

“Dot” syntax is used for calling library functions:

```
local endOfLine = string.find(line, "\013\010")
```

“string” is a table

“find” is a named index entry in the table

## Adding and Removing Table Entries

Add an entry to a table is just like assigning a value to a variable - simply give an index a value.

```
myTable[3] = "gamma"
myTable[4] = "delta"
```

Likewise, to remove an entry simply give it a value of NIL.

```
myTable[5] = nil
```

## Complex Tables

Any table entry may itself hold another table.

```
planets = {}
planets["mercury"] = {}
planets["mercury"]["type"] = "terrestrial"
planets["mercury"]["distanceFromSun"] = 0.38
```

```

planets["mercury"]["diameter"] = 4880
planets["mercury"]["moons"] = false
planets["jupiter"] = {}
planets["jupiter"]["type"] = "gas"
planets["jupiter"]["distanceFromSun"] = 5.2
planets["jupiter"]["diameter"] = 142984
planets["jupiter"]["moons"] = 67

```

“planets” is a table. It contains entries “mercury” and “jupiter”, each of which is also a table.

Subtables do not need to be structured the same as each other – they may each have their own “layout”, one may be indexed numerically and another with named indexes, another may be mixed etc.

Referring to subtables is similar to referring to table entries.

```

if planets["earth"]["inhabited"] then
  print("hello world")
end

```

## Copying Tables

Unlike normal variables, a copy of a table is the *same object* as the original. The copy is simply a different name by which to refer to the same table. The ramification of this is that if you modify the “copy” you are also modifying the original. This is true even if the copy is more locally scoped than the original.

```

> originalTable = {}
> print(originalTable["a"])
nil
> do
>> local newTable = originalTable
>> newTable["a"] = "not nil"
>> end
> print(originalTable["a"])
not nil

```

Don’t fall into a trap of treating a copy of a table as a different object – it isn’t.

## MORE LUA RESOURCES

### The Lua Reference Manual (free)

Just the basics.

<http://www.lua.org/manual/5.2/>

### *Programming in Lua, Third Edition*

If you are serious about writing parsers, buy this book. I learned lua by working through the e-book open in one window with the CLI open in another.

Paper: ISBN 978-8590379850

E-Book: <http://store.feistyduck.com/products/programming-in-lua>

### Lua-Users Wiki (free)

A wealth of information.

<http://lua-users.org/wiki/>

The tutorials are especially recommended.

<http://lua-users.org/wiki/TutorialDirectory>

### Lua BitOp API Documentation (free)

Bitwise operations in parsers are provided by the LuaJIT BitOp Module.

<http://bitop.luajit.org/api.html>

### *Lua Programming Gems*

Worthy of consideration once you've finished *Programming in Lua*.

Paper: ISBN 978-85-903798-4-3

E-Book: <http://store.feistyduck.com/products/lua-programming-gems-ebook>

Chapter 2 “Lua Performance Tips” is available for free.

<http://www.lua.org/gems/sample.pdf>

## FOUR: PARSER BASICS

The basic functionality of a parser could be simplified to the following steps:

1. match a token
2. find the beginning of a value
3. find the end of the value
4. extract the value
5. register the value as meta

Use a text editor to write your parsers. Even Notepad, vi, or Emacs is fine.

I use EditPadPro. If you're already familiar with a text editor you like such as Notepad++, use that.

Just don't use Word or any other program that leaves formatting characters all over the file – Decoder won't know what to do with them and the parser won't load.

## DECLARATIONS

First, you'll need to define a few things:

- a) the name and description of the parser
- b) what index keys the parser will register values with
- c) what tokens the parser is looking for

### `nw.createParser()`

The parser name and description must be the first thing declared in the parser. It will usually be the very first line of the parser file itself.

It is simply a line that looks like:

```
local demoParser = nw.createParser("Demo_Parser", "Demonstration of Lua")
```

Breaking down each of the elements:

<code>local</code>	<code>required</code>
<code>demoParser</code>	an internal name for the parser, not visible in the UI

```

= nw.createParser( ... ) required
"Demo_Parser"           display name for the parser for the UI and REST API
"Demonstration of Lua"  display description of the parser for the UI

```

The internal name is arbitrary – you can use anything you like, but it must follow the same naming rules as a variable. Essentially, the entire parser is being stored in a variable referenced with this name.

The display name is the name shown in the user interface used directly in the REST API, so it should make sense. It must not contain spaces.

The display description is shown in the user interface. It is free-form text and may contain spaces, but try to keep it short.

## setKeys()

Next you must declare which index keys the parser intends to register values with.

It should immediately follow the parser declaration itself.

```

demoParser:setKeys({
  nwlanguagekey.create("action"),
  nwlanguagekey.create("username"),
  nwlanguagekey.create("alias.ip", nwtypes.IPv4),
})

```

Each of the elements:

demoParser	the internal name of the parser
:setKeys({	required
nwlanguagekey.create( ... ),	required
"action"	index key that will be used to register meta
"username"	another index key that will be used
"alias.ip"	another index key
nwtypes.IPv4	index key format, if not "text" (see below)

## nwtypes

Most index keys are "text". For these index keys, you do not need to specify a format. However, for any key that isn't just text, you must specify its format.

You can determine the format of a key from the “format” parameter of the key’s entry in the index xml:

```
<key description="IP Aliases" level="IndexValues" name="alias.ip" format="IPv4"/>
```

The format for “alias.ip” is “IPv4”. But you can’t just use “IPv4” in `setKeys()`.

Instead you must use the nwtype which corresponds to IPv4, which is “nwtypes.IPv4”. The full list of nwtypes is enumerated in the `nw-api.lua` file, and most of them are self-explanatory.

You can also determine the format of a key from a REST “language” call:

```
http://decoder:50104/sdk?msg=language&size=255
```

The REST API returns numerical format values. For example, 65 = “text” and 128 = “IPv4”. For more information, reference the API documentation.

## setCallbacks()

At the end of your parser, you must declare the tokens the parser is looking for.

```
demoParser:setCallbacks({
  ["atoken"] = demoParser.someFunction,
  ["another token"] = demoParser.anotherFunction,
})
```

Breaking down each of the elements:

<code>demoParser</code>	the internal name of the parser
<code>:setCallbacks({</code>	required
<code>["atoken"] =</code>	a token value to be matched, quoted and bracketed
<code>demoParser</code>	the internal name of the parser (again)
<code>.</code>	literal dot character
<code>someFunction</code>	the name of the function in the parser to run if the token matches
<code>["another token"]</code>	another token value to look for as well
<code>anotherFunction</code>	a different function in the parser to be run if this token matches

Different tokens may call the same function. This is often useful for tokenizing variations of a token.

```
demoParser:setCallbacks({
  ["HOST: "] = demoParser.hostHeader,
  ["Host: "] = demoParser.hostHeader,
```



```
[“host: ”] = demoParser.hostHeader,
})
```

### *String Tokens*

Typically, tokens are literal string values:

```
“hello”
“supercalafragilisticexpialadocious”
```

Including punctuation such as spaces and colons:

```
“hello world”
“HOST: ”
```

Strings are case-sensitive:

```
“HOST: ”
“Host: ”
“host: ”
```

### *Special Characters in Tokens*

Some characters have special meanings:

^	matches the beginning of a line
\$	matches the end of a line
%	escape character
“	double quote

For example, to match only at the beginning of a line

```
[“^HOST: ”] = demoParser.hostHeader,
```

Or to match only at the end of a line:

```
[“ HTTP/1.0$”] = demoParser.httpVersion,
```

If you need to use any special characters in your token, escape them with “%”, which strips the next character of any special meaning:

```
[“Say, %”Hello, world!%””] = demoParser.helloWorld,
```

For a literal “%”, you must escape it as well:

```
[“50%% of all statistics are fabricated.”] = demoParser.statistics,
```

### *Hex Values in Tokens*

Hex byte values may also be used in token strings. However, you must use the decimal value of the hex, not the hex digit itself.

<u>hex</u>	<u>decimal</u>
0x00	\000
0x01	\001
0x02	\002
...	...
0xff	\255

For example:

```
[“\102\111\111\013\010”] = demoParser.foo,
```

### *Mix-n-Match*

You may mix characters, punctuation, special characters, and hex:

```
[“^%”P\067\032L0\065\068 \076\069\TT\069\082.%”$”] = demoParser.funcE,
```

### *Port Tokens*

Port values may be used as tokens as well, although the use of ports is discouraged – remember that just because it uses port 80 doesn’t mean it is HTTP.

To use a port value, specify the port without quotes:

```
[80] = demoParser.portEighty,
```

A port will match if it is either the source port or destination port of the session.

## Event Tokens

You may often want to match when certain events occur in the session, such as when a session or stream begins.

```
[nwevents.OnSessionBegin] = demoParser.beginningOfSession,
```

The full list of possible events is listed in `nw-api.lua`

There are some restrictions on events, however:

- You won't get a session begin or end if no declared token in the parser matches in the session
- You won't get a stream begin or end if no declared token in the parser matches in the stream

Also, be careful with `OnRequestBegin`, `OnRequestEnd`, `OnResponseBegin`, and `OnResponseEnd`. While very useful, they can be misleading in some circumstances:

- If the session only has one stream, which happens to be the server -> client stream, it will likely be considered to be the request stream, not the response stream – so `OnResponseBegin` and `OnResponseEnd` won't match.
- Sessionization sometimes even gets a two-stream session backwards, in which case `OnRequestBegin` and `OnRequestEnd` will match on the server -> client stream, and vice versa.

## Meta-Callbacks

An advanced technique is to match when meta is registered with a specified index key. As an aside, this is essentially how feeds and app rules work as well.

You must specify which index key the parser is interested in.

```
[nwlanguagekey.create("alert")] = demoParser.onAlert,
```

The parser will run the function "onAlert" each time meta for "alert" is registered. The meta could be registered by a parser, feed, app rule, or anything else.

## TOKEN MATCHES

When a token matches (whether a string, port, event, or meta-callback) the function specified for that token in `setCallbacks()` is executed.

## Functions

You’ve seen a few simple functions already in this book:

```
function funcE(a, b, c, d)
    local e = a + b + c + d
end
```

Function definitions:

- begin with the literal word “function”
- followed by a name for the function (same rules as variable names)
- specify local variables to hold values passed to the function
- contain blocks of statements
- end with the literal word “end”

Matches for string/hex tokens will be passed three variables, commonly referred to as:

- token – an index into the `self.tokens` table (covered later), not the token value itself
- first – position in the stream payload where the first byte of the token matched
- last – position in the stream payload where the first byte of the token matched

However, other types of tokens are passed different values:

string / hex	token, first, last
port	port value
events	(none)
meta-callbacks	index, meta value

It is within these functions that all the real work of the parser is done.

Each function in a parser will have this structure:

```
function parserName:functionName(token, first, last)
    ... stuff ...
end

function
parserName      the literal word “function”
:              the internal name of the parser
               a literal colon (not dot)
```

<code>functionName</code>	a name for the function
<code>token</code>	a variable to hold the value passed for “token”
<code>first</code>	a variable to hold the value passed for “first”
<code>last</code>	a variable to hold the value passed for “last”

The name of the function may be anything, but must follow the same rules as naming a variable.

The variables “token”, “first”, and “last” may also be named something else, but it is easiest just to name them as such.

If the function doesn’t use “last”, it may omit it. If for instance the function only uses the value of “first”.

Be very careful however – to get a value for “first”, a value for “token” must be accepted even if it won’t be used. Likewise, to get a value for “last”, values for “token” and “first” must be accepted even if they won’t be used.

Note that a colon must be used to delineate the parser name from the function name.

## EXTRACTING VALUES

### Payload Objects

Until now, “payload” has been presented conceptually. In a parser, however, payload is an “object” with which the parser interacts to find and extract values. A payload object is a value stored in a variable, just as any other value.

Always name this variable “payload”, and always declare it as local. You can break these rules later after you are more comfortable – but even then you should have a good reason to, otherwise it’s always best to keep things simple.

### `nw.getPayload()`

To give payload a value, use the function `nw.getPayload()`

```
local payload = nw.getPayload()
```

The variable `payload` will be given a payload object representing the entire stream (request, or response) in which the token matched. It is not given the session, just the stream.

For all of the functions which search and extract values from payload objects, you must first have a payload object. If you have not done a `nw.getPayload()`, but try to use one of those functions, an error will be thrown.

It usually isn't necessary or even desirable to get the *entire* stream. A payload object occupies a portion of Decoder's memory - the smaller the payload object, the better Decoder will perform. "*Use Small Payload Objects*" in Section Six covers how to get less than the entire stream.

## Position

In order to extract values, of course you must know where they are. Each byte location in the payload is a "position", just as every character in a text file or every byte in a hex editor.

Lua is a 1-based language. So the first byte in the payload is "position 1".

As noted above, a function for a token match is passed three variables, "token", "first", and "last". The values of "first" and "last" are positions within the session payload:

first: the byte position in the payload where the first byte of the token occurred

last: the byte position in the payload where the last byte of the token occurred

Positions are "on" bytes, not "between" bytes.

A token match for "GET " (including the trailing space) in the payload below would be passed the following values for "first" and "last":

```
first = 1  
last = 4
```

(remember positions are 1-based)

Stream Content																				
00000000	47	45	54	20	2f	41	42	43	44	45	46	47	48	49	4a	4b	GET /ABC	DEFGHIJK		
00000010	4c	4d	4e	4f	50	51	52	53	54	55	56	57	58	59	5a	20	LMNOPQRS	TUVWXYZ		
00000020	48	54	54	50	2f	31	2e	31	0d	0a	55	73	65	72	2d	41	HTTP/1.1	..User-A		
00000030	67	65	6e	74	3a	20	4f	70	65	72	61	2f	39	2e	38	30	gent: Op	era/9.80		
00000040	20	28	57	69	6e	64	6f	77	73	20	4e	54	20	36	2e	31	(window s	NT 6.1		
00000050	3b	20	55	3b	20	65	6e	29	20	50	72	65	73	74	6f	2f	; U; en)	Presto/		
00000060	32	2e	31	30	2e	32	32	39	20	56	65	72	73	69	6f	6e	2.10.229	Version		
00000070	2f	31	31	2e	36	31	0d	0a	48	6f	73	74	3a	20	65	6c	/11.61..	Host: el		
00000080	62	2e	6e	65	74	0d	0a	41	63	63	65	70	74	3a	20	74	b.net..A	ccept: t		
00000090	65	78	74	2f	68	74	6d	6c	2c	20	61	70	70	6c	69	63	ext/html	, applic		
000000A0	61	74	69	6f	6e	2f	78	6d	6c	3b	71	3d	30	2e	39	2c	ation/xm	l;q=0.9,		
000000B0	20	61	70	70	6c	69	63	61	74	69	6f	6e	2f	78	68	74	applica	tion/xht		
000000C0	6d	6c	2b	78	6d	6c	2c	20	69	6d	61	67	65	2f	70	6e	ml+xml,	image/pn		
000000D0	67	2c	20	69	6d	61	67	65	2f	77	65	62	70	2c	20	69	g, image	/webp, i		
000000E0	6d	61	67	65	2f	6a	70	65	67	2c	20	69	6d	61	67	65	mage/jpe	g, image		
000000F0	2f	67	69	66	2c	20	69	6d	61	67	65	2f	78	2d	78	62	/gif, im	age/x-xb		
00000100	69	74	6d	61	70	2c	20	2a	2f	2a	3b	71	3d	30	2e	31	itmap, *	/;*;q=0.1		
00000110	0d	0a	41	63	63	65	70	74	2d	4c	61	6e	67	75	61	67	..Accept	-Languag		
00000120	65	3a	20	65	6e	2d	55	53	2c	65	6e	3b	71	3d	30	2e	e: en-US	,en;q=0.		
00000130	39	0d	0a	41	63	63	65	70	74	2d	45	6e	63	6f	64	69	9..Accep	t-Encodi		
00000140	6e	67	3a	20	67	7a	69	70	2c	20	64	65	66	6c	61	74	ng: gzip	, deflat		
00000150	65	0d	0a	43	6f	6e	6e	65	63	74	69	6f	6e	3a	20	4b	e..Conne	ction: K		
00000160	65	65	70	2d	41	6c	69	76	65	0d	0a	0d	0a				eeP-Aliv	e....		

Keep this screenshot in mind for the examples below.

## payload:find()

The function `payload:find()` searches the payload object for a string and/or hex value. Note that it is called using a colon (":") instead of a dot (".").

Don't confuse using `payload:find()` with using tokens. Use `payload:find()` when you have matched a token and need to find something else relative to that token. Especially don't use `payload:find()` to search an entire stream (unless you want your Decoder to drop packets).

Three values are accepted as parameters:

1. The string for which to search (required)
2. Position in the payload object to begin the search (defaults to "1" if omitted)
3. Position in the payload object to end the search (defaults to "-1" if omitted)

You should always specify where the search should begin and end. If you do not, then the entire payload object is searched (remember that "-1" means "the end"), which is very hard on Decoder and will likely cause it to drop packets.

Search begin and search end positions are inclusive, meaning that both the search begin position and search end position will be included in the payload bytes to be searched.

The values passed may be literal,

```
local found = payload:find("foo", 16, 32)
```

or variables (or a mix of both),

```
local searchBegin = 16
local searchEnd = 32
local found = payload:find("foo", searchBegin, searchEnd)
```

How do you know where to begin and end the search?

Remember that the token match function is passed values for “first” and “last” representing the positions in the stream where the matched token began and ended. You should have a good idea where the string you are searching for is, relative to where the token would match.

For example, a token “GET “ (including the trailing space) matching in the “Stream Content” screenshot above would be passed first = 1 and last = 4. If you wanted to extract the path from the GET request, you already know that it starts at position 5. Likewise, you would expect that the end of the line must be within some number of bytes of the token.

Continuing with the example, if you wanted to extract the path, you’ll need to know search for the end of the line. You would begin the search from the position 1 byte after “last” (remember that searched positions are inclusive), so you could pass “last + 1” as the position to begin the search.

Likewise, you would expect that the end of the line must be within some number of bytes from the token – 100 bytes should be reasonable. So, you could pass “last + 100” as the position to end the search.

Conveniently, you don’t even have to create new variables:

```
endOfPath = payload:find(" HTTP/1.1\013\010", last + 1, last + 100)
```

If a search is successful, the position where the search string begins is returned. The return value is a position; it is not a distance. In this example, “endOfPath” would be assigned the value “31”.

If a search is not successful – the search string is not found within the payload positions specified – then the returned value is NIL, and NIL is assigned to the variable even if it already had a value.

You can easily test whether the search was successful, by checking if a value was returned,

```
if endOfPath then
```

If endOfLine has a value, the conditional will evaluate to true. If endOfLine is NIL, the conditional will evaluate to false.



## payload:tostring()

Now that you know where the value to be extracted begins and ends, you can read the value itself. For string values, the function to use is `payload:tostring()`

Two values should be passed to `payload:tostring()`

1. The position of the first character to be extracted (optional)
2. The position of the last character to be extracted (optional)

These positions are inclusive.

Even though they are optional, you should always specify a begin and end. If you do not, then begin will default to the first byte of the payload object and last will default to the last byte of the object – in other words the entire payload object will be read as a string, which shouldn't be what you want.

```
local path = payload:tostring(last + 1, endOfPath - 1)
```

Why “`endOfPath - 1`”? Remember that `payload:find()` returns the position of the first byte of the search term, and that `payload:tostring()` is inclusive. If just “`endOfPath`” had been specified, then `path` would include the space at the beginning of “HTTP/1.1”

If the `payload:tostring()` is successful, then a string will be returned.

If the `payload:tostring()` is unsuccessful, then `NIL` will be returned.

Continuing with the example, `path` would get the value,

```
/ABCDEFGHIIJKLMNOPQRSTUVWXYZ
```

If `payload:tostring` is used to read something that isn't a string, that isn't an error. Strings are just sequences of hex values, so the return value assigned to the variable is really just a sequence of hex values as well – but you can't perform arithmetic on them or otherwise use them as numbers.

To read numeric values, a set of different functions is required.

## payload:uint8()

## payload:uint16()

## payload:uint32()

These three functions read bytes of payload and return them as unsigned integers. Which you use depends upon how many bytes and values you require:

`payload:uint8()` returns 1 byte  
`payload:uint16()` returns 2 bytes  
`payload:uint32()` returns 4 bytes

*Reading 3-byte numeric value is covered in “Bitwise Operations”, Section Six.*

Each accepts two values as parameters:

1. Position of the first byte to be read (defaults to “1” if omitted)
2. Position of the last byte to be read (defaults to first byte + 1, 2, or 4 if omitted)
3. Whether the value is to be read as little-endian (boolean, defaults to FALSE if omitted)

You should always specify the first byte – if you do not then the first byte of the payload object will be used, which is usually not what you want.

You don’t have to be as strict about specifying the last byte. It will default to the appropriate number of bytes for the function (1, 2, or 4).

You only need to specify TRUE for little-endianness in those cases where you know the value to be extracted is represented as little-endian in the payload object. Otherwise, you can just leave it out.

If you do need to specify little-endian, you still don’t need to specify “last byte” – the parsing engine will figure out what you mean.

Some examples, using the screenshot on page 53,

<u>function</u>	<u>hex bytes</u>	<u>return value</u>
<code>payload:uint8(1)</code>	0x47	71
<code>payload:uint8(2, 2)</code>	0x45	69
<code>payload:uint16(3)</code>	0x5420	21536
<code>payload:uint16(5, 6)</code>	0x2f41	12097
<code>payload:uint16(7, 8, true)</code>	0x4243	17218
<code>payload:uint16(9, true)</code>	0x4445	17732
<code>payload:uint32(11)</code>	0x46474849	1179076681
<code>payload:uint32(15, 18)</code>	0x4a4b4c4d	1246448717
<code>payload:uint32(19, true)</code>	0x4e4f5051	1364217678
<code>payload:uint32(23, 26, true)</code>	0x52535455	1431589714

`payload:int8(), payload:byte()`  
`payload:int16(), payload:short()`  
`payload:int32(), payload:int()`

These functions work exactly the same as their “uint” counterparts – except that they return signed integers rather than unsigned.

Be very careful if you use them, which should be almost never. Lua fully supports signed values, but they can lead to all sorts of unexpected behavior if you aren’t expecting them.

`payload:len()`

This function returns the number of bytes in the payload object. It has no parameters, and doesn’t extract any values from the payload.

## MAKE SURE VALUES WERE EXTRACTED

Every time, after extracting any value, always check to make sure a value was extracted.

I really can’t emphasize this enough. If a value was not extracted but the parser doesn’t check, then when the parser tries to reference the variable it will be NIL and an error will result. Always checking that a value was actually extracted will eliminate a great portion of logic errors in the parser.

Doing so is simple:

```

local myString = payload:tostring(pos, pos + length - 1)
if myString then
    ...

```

- If myString has a value (is not NIL) the condition is TRUE and the if-block will be run.
- If the `payload:tostring()` failed, myString will be NIL and the condition will be FALSE.

The only alternative is to check in advance that there are enough bytes to be extracted, such as by using `payload:len()`

```

local payloadLength = payload:len()
if payloadLength >= pos + length - 1 then

```

```
local myString = payload:tostring(pos, pos + length - 1)
...
```

## REGISTERING VALUES AS META

Once you have matched a token, found the beginning of a value to extract, found the end of a value to extract, and read the value itself – you’ll want to register that value.

### `nw.createMeta()`

This is the function which registers values as meta.

It requires two values as parameters:

1. The index key to be used for the meta value
2. The meta value itself

The index key must have been declared with `setKeys()`. In `nw.createMeta()` it must be in the form,

```
self.keys.keyname
```

or

```
self.keys["keyname"]
```

Such as:

```
self.keys.username
self.keys["password"]
```

That syntax should be familiar – it is the same as when referencing a named table entry (which is exactly what you are doing).

If the key name contains a “.” such as “alias.host”, then the second form must be used,

```
self.keys["alias.host"]
```

The meta value may be literal or a variable.

Some examples,

```
nw.createMeta(self.keys.username, "Bob")
```

```

nw.createMeta(self.keys.password, somePassword)

nw.createMeta(self.keys["alias.host"], clientHost)

```

## BASIC PARSER EXAMPLE

The following is a complete basic parser, using only the concepts covered so far.

The index key “*path*” is not a standard index key. It is used for this example only.

```

local getPath = nw.createParser("get_path", "extract path from HTTP GET requests")
getPath:setKeys({
    nwlanguagekey.create("path"),
})
function getPath:httpGet(token, first, last)
    local payload = nw.getPayload()
    local endOfLine = payload:find(" HTTP/1.1\013\010", last + 1, last + 100)
    if endOfLine then
        local path = payload:tostring(last + 1, endOfLine - 1)
        if path then
            nw.createMeta(self.keys.path, path)
        end
    end
end
getPath:setCallbacks({
    ["^GET "] = getPath.httpGet,
})

```

This parser contains all of the basic elements of “parsing”:

- Declarations are done with `nw.createParser()`, `setKeys()`, and `setCallbacks()`
- A token is declared (“^GET “) which will execute a function upon matching (“httpGet”)
- The beginning of a value to be extracted is determined (“last + 1”)
- The ending of a value to be extracted is determined (“payload:find”)
- The value is extracted (“payload:tostring”)
- The value is registered (“nw.createMeta”)

Of course, there is a lot more that *could* be done, e.g.

- How do you know you’ve matched an HTTP GET request?
- Do you really need the entire stream payload just to register the path?
- What if you want to register directory, filename, and extension instead of a full path?
- What if you want to extract and register more values from the same session?

## LOADING AND ENABLING A PARSER

Parsers should be named with the extension “.lua”, and placed in the “parsers” directory of Decoder,

```
/etc/netwitness/ng/parsers/
```

When you add a new parser or make changes to an existing parser, you must reload the parsers,

*NwConsole:*

```
/parsers reload
```

When a new parser is loaded, it is automatically enabled.

To explicitly enable a parser, make sure it is not listed in /decoder/parsers/config/parsers.disabled

*NwConsole:*

```
cd /decoder/parsers/config
```

```
get parsers.disabled
```

or

```
get /decoder/parsers/config/parsers.disabled
```

Any parser not listed is enabled. However, that doesn’t mean your parser loaded (see *Section Five: Debugging*).

Conveniently, there is a special character “\*” (asterisk) which means “all parsers.

```
decoder:50004 /> get /decoder/parsers/config/parsers.disabled
```

```
[*]
```

(All parsers are disabled)

Any parsers listed after the asterisk are enabled.

```
decoder:50004 /> get /decoder/parsers/config/parsers.disabled
[*,NETWORK,HTTP_lua]
```

(All parsers are disabled except for NETWORK and HTTP\_lua, which are enabled.)

To modify `parsers.disabled` use “set”. It is often useful to disable all parsers except for the one you are developing.

*NwConsole:*

```
cd /decoder/parsers/config
set parsers.disabled *,get_path
```

or

```
get /decoder/parsers/config/parsers.disabled *, get_path
```

(Disable all parsers except `get_path`.)

The name of the parser is the “display name” from the parser’s `nw.createParser` declaration.

```
local getPath = nw.createParser("get_path", "extract path from HTTP GET requests")
                        ^^^^^^^^^
```

## FIVE: DEBUGGING

Debugging a parser is a very manual process. There are no automated tools to help you.

Debugging itself is more art than science. What you need when debugging is information (*What is the parser doing, and why?*) These are techniques for getting information from your parser.

Debugging typically occurs in three stages, each building upon the previous:

### Stage One:

- 1) Check for syntax errors
- 2) Make changes
- 3) Repeat

### Stage Two:

- 1) Check for syntax errors
- 2) Import a pcap
- 3) Watch the log
- 4) Make changes and reload the parser
- 5) Repeat

### Stage Three:

- 1) Check for syntax errors
- 2) Import a pcap
- 3) Watch the log
- 4) Review the meta
- 5) Make changes and reset Decoder
- 6) Repeat

No errors, and correct meta? Congratulations! Otherwise, repeat...



## DECODER TUNING FOR PARSER DEVELOPMENT AND DEBUGGING

**Important:** Do **not** apply any of these settings to a production Decoder.

Apply them only to a Decoder dedicated to parser development.

There are several tuning options on Decoder which affect how sessions are parsed. These settings prevent Decoder from exhausting its resources on very large or long-running sessions.

For parser development and debugging, I turn them all off. True, that doesn't provide the same environment as a production Decoder. But for parser development you need to know that a "session" is a complete TCP/IP session no matter how large, small, long, or short, and that the tokens will match no matter where in the session they occur.

```
/decoder/config set assembler.size.max 0
```

For a large session, `assembler.size.max` specifies that when the session size exceeds the threshold, Decoder considers the session to be complete. At that point it will be sessionized and parsed. Any further packets will be sessionized as a different session. Setting this to "0" disables it – no session will be truncated due to size.

```
/decoder/config set assembler.size.min 0
```

Similar to `assembler.size.max`. "0" may be the default, but I always set it anyway, just to make sure.

```
/decoder/config set assembler.timeout.packet 0
```

For a long-running session, `assembler.timeout.packet` specifies that when the time since the last packet exceeds the threshold, Decoder considers the session to be complete. At that point it will be sessionized and parsed. Any further packets will be sessionized as a different session. Setting this to "0" disables it – no session will be truncated due to time.

```
/decoder/config set assembler.timeout.session 0
```

Similar to `assembler.timeout.packet`.

```
/decoder/config set assembler.session.pool 50000
```

I'm not sure of the specific ways in which `assembler.session.pool` affects sessionization, but it certainly has an effect. It can't be turned off, but it can be set to a large number.

```
/decoder/parsers/config set parse.bytes.max 0
```

For a large session, `parse.bytes.max` specifies that no tokens will match beyond this threshold. Decoder simply stops looking for any tokens in the remainder of the session, meaning that the remainder of the session is not parsed at all. Setting this to “0” disables it – Decoder will continue looking for tokens in every session regardless of how far into the session they may be.

```
/decoder/parsers/config set parse.bytes.min 0
```

If no token from any enabled parser has matched in a session by this threshold, then Decoder stops looking for tokens in the remainder of the session and the remainder of the session is not parsed at all. Setting this to “0” disables it – Decoder will continue looking for tokens in every session regardless of whether any have already matched.

The most important to disable is “`parse.bytes.min`”. When developing and debugging a parser, usually all other parsers are disabled. So if the first token of your parser occurs beyond this threshold in your test session, your parser never has the opportunity to parse the session, and you are left wondering why your parser didn’t register any meta.

## META SCRUB

This isn’t “debugging” per se, but you should be familiar with what occurs to meta values when you register them.

The “meta scrub” process isn’t something that is done in a parser. Rather it is done in the backend, when meta is registered by a parser.

Among other things, meta scrub

- truncates at the first non-printable character

If a parser registers,

`username: Alice0x0d0aBob`

it is truncated to,

`username: Alice`

All characters from the first non-printable character onward are discarded, whether or not they are themselves printable.

- truncates at 255 bytes

If a parser registers a meta value longer than 255 bytes, all extra bytes are discarded.

## SYNTAX CHECKING WITH NW-API.LUA

You should have a Lua CLI of some sort, whether Windows, Mac, or Unix. In the *Appendix* you'll find the file "nw-api.lua". Save that file to somewhere *other than* the "parsers" directory of Decoder.

Open the CLI, and type:

```
dofile('/path/to/nw-api.lua')
```

Any spaces in path names must be escaped with a backslash:

```
dofile('/somewhere\ over\ the\ rainbow/nw-api.lua')
```

Then type:

```
dofile('/path/to/parser.lua')
```

Again, replacing path and parser filename as appropriate.

No news is good news. If loading your parser in the CLI doesn't produce an error, it will load in Decoder. It may not do what you want, but it will load.

If there is a syntax error, it will produce a descriptive error message which includes a line number.

```
demoParser.lua:8: ')' expected near 'local'
```

```
An error on line 8 of 'demoParser.lua'
```

It will stop at the first syntax error, so there may be others.

Once you have fixed the syntax error and saved the parser file, simply repeat the dofile command.

```
dofile('/path/to/parser.lua')
```

You don't need to load nw-api.lua again unless this is a different instance of the CLI, such as if you closed it and reopened.

Repeat this process until loading the parser in the CLI no longer produces errors.

## SYNTAX CHECKING WITH THE LOG

Checking syntax in a Lua CLI with `nw-api.lua` is far easier – it loads in the CLI it will load in Decoder.

But if for whatever reason you cannot use a CLI, the Decoder log can still tell you what you need to know. The caveat is that you can't cause Decoder to attempt load your parser and issue the log messages at will. Instead, you must either

- a) restart Decoder, or
- b) issue a `"/parsers reload"` then import a pcap

If there are syntax errors in a parser, the log will display the same errors that would have been displayed in the lua CLI.

```
LUA_ERRRUN: [string "exeOffset.lua"]:8: ')' expected near 'local'
```

An error on line 8 of 'demoParser.lua'

Once you have fixed the syntax error and saved the parser file, you must either restart Decoder, or issue a `"/parsers reload"` then import a pcap again, to check the log for further syntax errors.

## IMPORT A PCAP

It's best to test a parser with a known pcap rather than with live traffic – at least until you are sure the parser works correctly.

Choose a pcap demonstrative of the subject of the parser. Ideally the same you examined to determine what to register and how to get to it, so that you are already familiar with what the parser should register if it works correctly.

*NwConsole:*

```
import /path/to/pcap
```

or in Windows,

```
import C:\path\to\demo.pcap
```

You can do this over and over – the pcap will be imported, parsed, and meta registered each time.

## WATCH THE LOG

Each time you import a pcap, have a “tail -f” (or similar) running on Decoder’s log file. There may be logic errors that won’t show up from a syntax check (such as referencing a variable with a NIL value) but will show up when the parser is run on a session.

When a parser encounters an error, Decoder will stop the parser. No more token matches will occur or functions be run from that parser until the current capture or import stops.

`nw.logDebug()`  
`nw.logInfo()`  
`nw.logWarning()`  
`nw.logFailure()`

You may also register your own log messages, from within the parser itself.

Each of these sends a message to the log. Which you use depends upon what syslog level you want the message to be logged at. That level must be enabled by being listed in

```
/logs/config/log.levels
```

By default, “debug” is not enabled. I typically use `nw.logInfo()`

The syntax for each is identical,

```
nw.logInfo("This message will be logged.")
```

You may also use the contents of variables,

```
local endOfLine = payload:find("\013\010", position, position + 32)
if endOfLine then
    -- log the position at which endOfLine was found
    nw.logInfo(endOfLine)
    ...
end
```

Logging is very handy to determine exactly where a function encounters a problem.

```
nw.logInfo("1")
...
nw.logInfo("2")
...
```

```
nw.logInfo("3")
```

```
...
```

If you see “1” and “2” in the log, but not “3”, then you know the problem is between those two points of the function.

Be very sure to remove all logging statements from the parser after you have finished debugging it.

## RELOAD THE PARSER

If everything didn’t go smoothly, and chances are they won’t at first, you’ll be making some changes to the parser. Once you have made those changes, you need to let Decoder know that the parser has been modified.

```
/parsers reload
```

The next time you import the pcap, the parser will run with your latest changes.

Unfortunately, if you’ve introduced a new syntax error Decoder won’t log it right away unless you do a syntax check with the CLI first. You won’t know until you import a pcap again, and only if you are watching the log.

## REVIEW THE META

The next step is to actually look at the meta registered by the parser, to ensure that it matches what should have been registered.

The easiest way to get at the meta is with a “reports” REST call.

```
http://decoder:50104/sdk/app/reports?filter=time=1-u
```

This provides a very Investigator-like view of keys and values. You can even drill, pivot, and get to the detailed meta and session views by clicking on the number in parentheses.

```
User Account
bob (1)
```

The number in parentheses itself is the number of sessions in the current drill with that meta value. Clicking it provides the detailed meta view for all the sessions with that meta value.

The meta listed in the detailed meta view of a session is the same order in which it was registered.

A basic tabular view of meta is provided by a “query” REST call.

```
http://decoder:50104/sdk?msg=query&id1=0&id2=0&size=1677721
```

Or for plaintext instead of tabular, add ‘&force-content-type=text/plain’ to the end. Plaintext output is good for saving to a file and comparing with ‘diff’.

Meta listed from a query call is also in the order in which it was registered.

## RESET THE DECODER

Unlike the old Investigator thick client, Decoder has no concept of local collections. Each time you import a pcap, the meta is added to the same “collection”. This is fine when you are fixing syntax and simple logic errors and importing the same pcap over and over.

But when you are fixing issues having to do registered meta values, you want to know that the meta you see came from this import of the pcap, not any previous import.

So you have to reset the Decoder, removing all previous meta. Unfortunately there is no easier way to do this.

```
/decoder reset data=1 force=1
```

Decoder will delete all sessions, pcaps, and meta, then shut itself down.

Restart Decoder, and import your pcap. The meta will from that import only, no previous import.

No errors and correct meta? Congratulations! Otherwise, make changes, reload parsers, import pcap, and watch the log.

## PCALL()

**IMPORTANT: This is **not** a way to hide a poorly written parser.**

A lua “pcall” is a protected call – a way of catching exceptions before they escalate.

Instead of calling a function normally:

```
local foo = someFunction(bar)
```

You can use a pcall to catch a returned error:

```
local status, foo = pcall(someFunction(bar))
```

If all goes well in `someFunction()`, then “status” will get a value of `TRUE`, and “foo” will get the value returned by `someFunction()`.

If however there is an exception in `someFunction()` which throws an error message, then “status” will get the value of `FALSE`, and “foo” will get the error message. This is not a magic bullet, however – an exception still occurred.

You can also encapsulate an entire function within a `pcall`:

```
if pcall(
    function()
        -- do stuff
    end) then
    -- do more stuff
else
    -- an exception occurred, do something else
end
```

How is this useful for a parser? Two ways.

**(A) When you don’t know if something is going to fail or not**, but don’t want to bail and throw an error to the log if it does, then encapsulating it into a `pcall` is very handy.

If you were to do this:

```
local varA = someTable[1][5][1][someIndex]
```

then if `someTable[1][5]` didn’t exist (or `someTable`, or `someTable[1][5][1]`, or...) then your parser would bail entirely and an ugly error message would be written to the log.

To prevent that, you could check each element (which would work, but be extremely tedious):

```
local varA
if someTable
and someTable[1]
and someTable[1][5]
and someTable[1][5][1]
and someTable[1][5][1][someIndex] then
    varA = someTable[1][5][1][someIndex]
end
```

Or instead, you could use a `pcall`:



```

local varA
pcall(function() varA = someTable[1][5][1][someIndex] end)
if varA then ...

```

If `someTable[1][5][1]` (for example) didn't exist, `varA` simply wouldn't get a value – but your parser would continue and no error message would be written to the log (that's why you still have to check “if `varA` then”).

**(B) Parsers unfortunately encounter unexpected circumstances.** No matter how hard you try to anticipate and account for every possibility, some session will befuddle even the most carefully crafted parser.

When that happens, the parser could leave ugly errors all over the logs causing angst, consternation, and doubt. Or, it could use a `pcall`.

You can actually wrap an entire callback function in a `pcall`:

```

function demoParser:foo(token, first, last)
  local status, error = pcall(function(token, first, last)
    -- do stuff
  end, token, first, last)
  if not status then
    nw.logFailure(error)
  end
end

```

Here's what's going on in this example:

- Everything from ‘`function(token, first, last)`’ through ‘`end,`’ is a parameter of the `pcall` function itself - the entire “anonymous” (un-named) function is being passed as a parameter.
- The `pcall` is also being passed the parameters required by the anonymous function
- The anonymous function is being passed those same parameters by `pcall`
- If the function doesn't throw an error, everything goes on as normal
- If the function does throw an error, then `status` will be `FALSE` and the value of `message` will be logged.

**A caveat:** *If you call another function from within a `pcall`'d function and the subsequent function returns an error, the `pcall` will not get the error message – instead it will get a literal function object (remember, functions are themselves just variable values). The resulting error in the log will read something like “expected string, got function”. In that case, you'll need to comment out the `pcall` to see the actual error.*

What if you don't want an error logged at all? Remove the “if not `status` then” block.

What if you want to log error messages during debugging, but not otherwise? Make the following changes:

```
local debugParser = true -- somewhere near the top of the parser
...
function demoParser:foo(token, first, last)
  local status, error = pcall(function(token, first, last)
    -- do stuff
  end, token, first, last)
  if not status and debugParser then
    nw.logFailure(error)
  end
end
```

- if you set debugParser to FALSE or NIL, or comment it out, no error messages will be logged
- if you set debugParser to TRUE or any other value, error messages will be logged

## SIX: BEYOND THE BASICS

Most parsers are more sophisticated than the example in Section Four. This section details some of the intermediate-to-advanced functionalities and capabilities available, concentrating on that which is frequently relevant to parsers.

Much more is possible than is presented here - owing mostly to the capabilities of Lua itself.

### REMOVED LUA FUNCTIONS

First however I should mention Lua capabilities that are *not* available to a parser. A lua parser runs in something of a sandbox - some capabilities were removed to ensure that no parser could interfere with the behavior of another, or of the system.

The following are removed from the Lua implementation available to parsers (may not be exhaustive):

- the entire IO library
- the entire OS library
- the entire debug library
- coroutines
- the ability to modify `_G` (the “global” table) in any way
- `table.setmetatable()`

### “SELF”

This term was used in some syntax in Section Four, and will be used much more henceforth.

Think of “self” as referring to “this parser”.

```
self.keys    = this parser's keys
self.tokens = this parser's callbacks
```

Behind the scenes, “self” is the name of a table and “keys”, “token”, etc. are indexes in that table. A parser may add entries into the “self” table, so long as the critical indices such as “keys” and “tokens” are not overwritten.

## USE SMALL PAYLOAD OBJECTS

One of the most important things that can be done in a parser to prevent performance degradation is to use small payload objects – meaning less than the entire stream, preferably as small as possible, perhaps containing only the values you need to extract.

### `nw.getPayload()`

The `nw.getPayload()` function was demonstrated above without parameters, but actually accepts two parameters:

1. position of the first byte of payload to be returned (defaults to “1”)
2. position of the last byte of payload to be returned (defaults to “-1”)

These positions are inclusive.

Since “from” defaults to “1” and “to” defaults to “-1”, without parameters `nw.getPayload()` will retrieve the entire stream.

How do you know how much payload to get? This is a very similar exercise to considering how far within a payload object to search using `payload.find()`. Your token position will be known from the values passed as `first` and `last`. If you know that your value is within a certain number of bytes from the token, then only get that much payload.

```
function getPath:httpGet(token, first, last)
    local payload = nw.getPayload(last + 1, last + 100)
```

### `payload.sub()`

The `payload.sub()` function is logically similar to `string.find()`. It returns a subset of the current payload object.

It accepts two parameters,

1. The position within the current payload object of the first byte to be returned (inclusive)
2. The position within the current payload object of the last byte to be returned (inclusive)

This is useful to “whittle down” a payload object, when a parser can determine, from a larger payload object, that the values it is interested in are within a smaller subset of the payload. Remember that it is more efficient for the parser to hold and extract values from a smaller

payload object than a larger one. It is also useful for registering meta directly from payload (covered in Section Eight).

### All Positions are Relative to a Payload Object

All of the functions which search and extract values from payload objects accept positions as parameters. If “payload” is not the entire stream (which it shouldn’t be if you’ve heeded the advice about using small payload objects), then the positions passed to those functions are relative to that payload object, not the entire stream.

For example, given the following stream:

```
payload = nw.getPayload()

payload: 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 6f 6c 6f 72 20 73 69
position: 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

word = payload.toString(7, 12) → “ipsum”
```

Positions 7 through 12 of the entire stream contains the word “ipsum”.

If the payload object is not the entire stream, then the same position numbers refer to a different set of bytes.

```
payload = nw.getPayload(7, 18)

payload: 69 70 73 75 6d 20 64 6f 6c 6f 72 20
position: 1  2  3  4  5  6  7  8  9 10 11 12

word = payload.toString(7, 12) → “dolor”
```

## GLOBALS

You should always use local-scoped variable whenever possible. But if you need to keep some sort of state or other value between token match functions, you’ll need to use a global (a local will be NIL outside of its function).

### Using Globals Safely

*This sub-section concerns best practices. These aren't rules, per se. Rather, consider this to be very strong advice.*

A “global” variable never loses its value – not even between sessions.

**Not even between sessions.** The next time the parser is run, even if a token matched in a different session entirely, that variable will still have a value. If for example the parser uses a globally-scoped variable to flag that something has been seen in the session, then in the next session the parser will think that same something has been seen in that session too.

You must reset global variables yourself. Nothing does it for you.

### *How Not to use Globals*

Don't think that you can use a global to perform multi-session correlation or aggregation in a parser by allowing a global to keep its value between sessions.

**You can't.**

Decoder is multi-threaded. Each parser “instance” is run in its own thread. Any two sessions may or may not be parsed in the same thread. A global set in one thread will not have the same value in a different thread.

*Shared Values* in Section Nine provides a method of keeping variables between threads in a thread-safe manner.

### *When to Reset Globals*

A global may be reset at any time, but generally they should be reset either between streams or between sessions:

- Reset between streams if you don't need the state outside of that stream
- Otherwise, reset between sessions.

### *Keeping Track of Globals*

Since you'll be resetting some globals between streams, and the rest between sessions, it's easiest to hold them in two tables:

```
self.streamVars    globals that should be reset between streams
```

```
self.sessionVars   globals that should be reset between sessions
```

Remember, "self" is a table – so you can add anything you want to that table whenever you like so long as you don't overwrite something important (such as "keys", or "callbacks", or a function name).

Each and every global variable should be kept in one of those two tables. Every other variable should be declared as local.

```
local payload = nw.getPayload(last + 1, last + 32)
local endOfline = payload:find("\013\010", 1, -1)
...
self.sessionVars.sawFoo = true
self.streamVars.methodPosition = last
```

This has the benefit of providing a single "place" to keep all globals, instead of having to remember what you named each of them when you want to reset all of them.

### *Resetting Globals*

The best time to reset `self.sessionVars` is at the beginning of the session. Likewise, the best time to reset `self.streamVars` is at the beginning of the stream.

This is due to two things:

- a) Callbacks for `OnSessionBegin` or `OnStreamBegin` won't occur unless a declared token matches in the session or stream, in which case you'll want "clean" globals without residual values
- b) Callbacks for `OnSessionBegin` and `OnStreamBegin` are guaranteed to occur so long as some token has matched in the session or stream, whereas callbacks for `OnSessionEnd` and `OnStreamEnd` are not guaranteed

First you'll need to declare functions for those callbacks.

```
function demoParser.resetSessionVars()
    self.sessionVars = {}
end
```

```
function demoParser.resetStreamVars()
  self.streamVars = {}
end
```

Note that `self.sessionVars` and `self.streamVars` are not assigned `NIL`. Rather they are declared as empty tables. This simultaneously resets them and saves you from having to declare them as tables at some other point in the parser.

And you'll need to declare callbacks for `OnSessionBegin` and/or `OnStreamBegin`:

```
demoParser:setKeys({
  [nwevents.OnSessionBegin] = demoParser.resetSessionVars,
  [nwevents.OnStreamBegin] = demoParser.resetStreamVars,
  ...
})
```

Of course if you only have `streamVars`, you don't need a function and callback for `sessionVars` (and vice versa).

## Keeping State with Globals

Now that you know how to use globals safely, what can you do with them? The primary use is to keep track of what the parser has seen or done already in the session.

For example, how can you be reasonably certain that a single token such as "GET" belongs to a set of HTTP request headers?

```
[("^GET ") = demoParser.httpGET,
```

You'd want to see more.

```
[(" HTTP/1.1$") = demoParser.httpVersion,
```

But when the `httpVersion` token matches, how will the parser know whether the `httpGet` token had matched or not? With a global:

```
function demoParser.httpGet(token, first, last)
  self.streamVars.sawGet = true
end
```



```
function demoParser.httpVersion(token, first, last)
  if self.streamVars.sawGet then
    ...
```

## USE TOKENS INSTEAD OF PAYLOAD:FIND()

This technique combines the concepts of small payload objects with globals.

In the example above, both “^GET “ and “ HTTP/1.1\$” were tokens. Everything between them is “path”, just as in the *Basic Parser* example.

So instead of using payload:find() in the function httpGet(), just note where it occurred with a global. Then in the function httpVersion() just extract everything between them.

```
function demoParser.httpGet(token, first, last)
  self.streamVars.pathBegin = last + 1
end

function demoParser.httpVersion(token, first, last)
  if self.streamVars.getPosition then
    local payload = nw.getPayload(self.streamVars.pathBegin, first - 1)
    local pathString = payload:toString(1, -1)
    nw.createMeta(self.keys.path, pathString)
  end
end
```

## RETURN VALUES IN CONDITIONALS

The return value of a function may be compared directly in a conditional. The return value itself is lost after the comparison since it isn’t assigned to a variable.

In its most simple form, a check can be made for success or failure. Remember that, in a conditional (“if”, “elseif”, “until” etc.) any value other than FALSE or NIL is TRUE. If the parser doesn’t need to know the return value of a function, only that there *was* a return value (implying success) or was *not* a return value (implying failure).

For example, instead of,

```
local foundFoo = payload:find("foo", position, position + 12)
if foundFoo then
```

the `payload:find()` could be directly evaluated:

```
if payload:find("foo", position, position + 12) then
```

- If “foo” is found within position through position + 12, then the condition is TRUE
- Otherwise it is FALSE
- The position that “foo” was found is lost

Another use is to compare the return value in some way.

```
if bit.band(someValue, 1) == 1 then
```

- If the “1” bit is set in the value held by the variable `someValue`, then the condition is TRUE
- Otherwise it is FALSE

A very common use of that technique is checking payload length.

```
if payload:len() >= N then
```

- If the payload object is larger than or equal to *N* bytes, then the condition is TRUE
- Otherwise it is FALSE

## LOOPS

Using loops will become an essential tool for writing more advanced parsers. However, care must be taken to not allow loops to run out of control.

An out of control loop will continue to run happily until `/parsers/config/lua.instruction.limit` is exceeded – that’s the only built-in safety net. Best practice is to default each loop iteration to not run again.

There isn’t anything parser-specific about loop syntax itself. This is just Lua – the book *Programming In Lua* does a much better job of documenting them.

### repeat until

A “repeat” loop will always run at least once. Everything between “repeat” and “until” is a block. At the end of the block, the conditional in the “until” statement is evaluated – if TRUE then the entire block is executed again, if FALSE then the block is not executed again.

```
local x = 1
repeat
  x = x + 1
until x == 10
```

In the first pass, x is given a value of 2 and the loop is run again. On the 9<sup>th</sup> pass, x is given a value of 10 – the loop exits.

Note that x was declared as local outside of the block. If x was declared as local inside of the block, then it would lose its value on every iteration of the loop:

```
repeat
  local x = 1
  x = x + 1
until x == 10
```

This loop will run infinitely.

In the first pass, x is given a value of 1, then a value of 2, and the loop is run again. On the 9<sup>th</sup> pass, x is given a value of 1, then a value of 2, and the loop is run again. And so on.

Consider the following modification:

```
repeat
  local x
  x = x + 1 ERROR
until x == 10
```

This results in an error because x is NIL.

That isn't to say that locals can't be declared inside of a loop. Just be aware that they will begin as NIL for every iteration of the loop; they will not retain any value from a previous iteration.

To default a “repeat until” loop to not run again,

```
repeat
  local loopControl = 0
  ...
  ...
  if ... then
    loopControl = 1
  end
```

```

        end
    end
until loopControl == 0

```

The variable `loopControl` determines whether the loop will run again, rather than some other condition. It is declared as local inside of the loop because it doesn't need to exist outside of the loop, nor should it retain its value on each iteration. It is immediately given a value of "0", which based upon the "until", means that the loop will not be run again. Only if all the logic in the block completes successfully will it be given a value of "1", allowing the loop to run again.

## while

A "while" loop is very similar to a "repeat until" loop, except that the condition is evaluated first. This means that a "while" loop is not guaranteed to run at least once – if the condition initially evaluates to FALSE then the loop is not run at all.

```

local x = 0
while x < 10 do
    x = x + 1
end

```

To default a "while" loop to not run again,

```

local loopControl = 1
while loopControl == 1 do
    loopControl = 0
    ...
    ...
    ...
    loopControl = 1
end
end
end
end

```

Again, the variable `loopControl` determines whether the loop will run again, rather than some other condition. However, it is declared outside of the loop because otherwise it would be NIL when evaluated.

```

while loopControl == 1 do ERROR
    local loopControl = 0
    ...

```

## for

A “for” loop is used when you know in advance how many times a loop needs to run.

```
for n = 1, 10 do
    print(n)
end
```

The loop will be run 10 times.

- In the first iteration, n will have a value of “1”
- In the second iteration, n will have a value of “2”
- And so on...

## ipairs, pairs

These are a special form of “for” loop, used exclusively with tables. They will iterate through every entry in the table, returning the index and value of each entry.

- `ipairs` will only iterate over numeric table entries
  - beginning with 1
  - counting upwards
  - stopping at the first gap or when there are no more entries
- `pairs` will iterate over named or numeric table entries
  - gaps don’t matter
  - but no order is guaranteed

For example, (try this in the CLI)

```
local myTable = ({ [1] = “A”, [2] = “B”, [3] = “C” })
for i,j in ipairs(myTable) do
    print(i, j)
end
```

will output,

```
1    A
2    B
3    C
```

The variables `i` and `j` will be given the index and value respectively on each iteration. You can use whatever variable names you like. Those two variables will automatically be local to the `for` loop.

If you do the exact same, but replace “`ipairs`” with “`pairs`”, the output will be the same.

If you try this:

```
local myTable = ({ ["A"] = 1, ["B"] = 2, ["C"] = 3 })
for i,j in ipairs(myTable) do
    print(i, j)
end
```

you will get no output at all – “`ipairs`” only iterates over numeric table entries.

If you do the exact same, but replace “`ipairs`” with “`pairs`”, the output may be,

```
A 1
C 3
B 2
```

Using “`pairs`” does not guarantee any sort of order to the returned results.

The benefits of “`ipairs`”

- order is guaranteed
- more efficient (faster) than “`pairs`”

The benefits of “`pairs`”

- works with named table entries
- won’t stop at a gap

Generally, `ipairs` is preferable but obviously can’t be used with named table entries or if gaps may be present in numeric entries.

## STRING MANIPULATION

Sometimes a parser doesn’t just extract a string from the payload and register it. Instead it needs to do something with or to that string first. The “`path`” from Section Four is a typical example. Instead of registering the entire string, perhaps it should be broken down into directory, filename, extension, and `querystring` meta.

The parser could loop `payload:find()` to get to the last directory marker, and another to get to the extension, then `querystring`, extracting and registering each in turn. That is certainly a valid technique.

But another valid technique is to extract the entire path as a string, then break down the components of the string itself. That is but one of many uses possible with the Lua string library.

## Regular Expressions

Lua does not support regular expressions. *Programming in Lua* explains:

*[A] typical implementation of POSIX regular expressions takes more than 4000 lines of code. This is about the size of all Lua standard libraries together. In comparison, the implementation of pattern matching in Lua has less than 500 lines. Of course, the pattern matching in Lua cannot do all that a full POSIX implementation does. Nevertheless, pattern matching in Lua is a powerful tool, and includes some features that are difficult to match with standard POSIX implementations.*

## Patterns in Lua

The concept of “patterns” in lua consists of:

- character classes
- character sets
- special characters

**Important:** Patterns are not valid in `tokens` or `payload:find()`  
They are valid only in string functions.

## Character Classes

Lua character classes are somewhat similar to character classes in regular expressions, in that they match any member of the set.

.	all characters
%a	all letters
%c	control characters (typically 0x00 – 0x1f and 0x7f – 0xff)

%d	digits
%l	lowercase letters
%p	punctuation
%s	whitespace
%u	uppercase letters
%w	alphanumeric (%a and %d, combined)
%x	hexadecimal digits
%z	“the characters whose representation is 0

Further, all characters not in a particular class may be referenced by using the uppercase version of the class, e.g.

%A	all non-letters
%C	all non-control characters
%D	all non-digits
...	and so on ...

### *Special Characters*

Some characters have special meaning or are reserved (unusable) in a pattern.

( )	reserved (capture)
%	escape – removes special meaning from the next character
+	1 or more repetitions
-	0 or more repetitions, shortest sequence possible
*	0 or more repetitions, longest sequence possible
?	the preceding character is optional (0 or 1 occurrence)
[ ]	reserved (character set)
^	beginning of string
\$	end of string
“ “	(see below)

To use a character which is reserved or has a special meaning as a “normal” character, it must be escaped.

%(	→ matches an open-parenthesis
%%	→ matches a literal % character

The semantic difference between “-“ and “\*” is best illustrated with an example. Given the string “ABBBBBBC”,



B- → B  
 B\* → BBBBB

Quotation characters are not actually, strictly speaking, reserved. They must be used with care. If the function's parameters are encapsulated with double-quotes, then a double-quote within a pattern must be escaped. Alternately, if the parameter is encapsulated with double-quotes then single-quotes may be used freely within the pattern. Likewise if the parameter is encapsulated with single-quotes, single-quotes within the pattern must be escaped but double-quotes may be used freely.

### *Character Sets*

A “character set” is a character class defined within the pattern itself. It may be a combination of classes, or single characters, or both, encapsulated within square brackets.

[%p%s] → any punctuation or whitespace  
 [AEIOU] → any uppercase vowel  
 [%d\_] → any digit or underscore

Ranges are also acceptable.

[A-E] → A, B, C, D, or E

Character sets may be negated with ^, in which case any character not belonging to the set will be matched.

^[%p%s] → any non-punctuation non-whitespace character  
 ^[AEIOU] → any non- uppercase vowel  
 ^[A-E] → any character that isn't A, B, C, D, or E

### *string.find()*

The `string.find()` function searches for a string or pattern inside of a string, returning the “from” and “to” (inclusive) positions within the larger string at which the search matched.

Three parameters are accepted:

1. the string to be searched
2. a string or pattern to search for
3. the character position in the string to begin the search (optional, defaults to 1)

For example,

```
local largeString = "ABCDEFGHJKLMNOPQRSTUVWXYZ"
local beginMNO, endMNO = string.find(largeString, "MNO") → 13, 15
```

Just because a function such as `string.find()` returns multiple values doesn't mean they have to be used. This is also valid:

```
local beginMNO = string.find(largeString, "MNO") → 13
```

If the search term isn't found then the return value is `NIL` so `NIL` is assigned to the variable. Just as with `payload:find()`, check that the search term was actually found.

```
local beginMNO = string.find(largeString, "MNO")
if beginMNO then
  ...
end
```

Only the first match is returned. To search for further matches, set the begin position to one character beyond the match – a loop is good for this.

```
local beginSearch = 1
local found
repeat
  local loopControl = 0
  found = string.find(someString, somePattern, beginSearch)
  if found then
    beginSearch = found + 1
    loopControl = 1
  end
until loopControl == 0
```

When the loop exits, `found` will have the position of the last match.

## `string.gmatch()`

Like `string.find()`, `string.gmatch` searches for a string or pattern within a string. However, instead of returning the position of the first match, it acts as an iterator: performing some action for every match found within the string.

Again, three parameters are accepted:

1. the string to be searched
2. a string or pattern to search for
3. the character position in the string to begin the search (optional, defaults to 1)

Patterns are standard lua patterns (see “Patterns in Lua”, page 85).

For example,

```
local testString = "A1B2C3"
for digit in string.gmatch(testString, "%d") do
    print(digit)
end
```

will output,

```
1
2
3
```

A more practical example, to iterate over substrings delineated by a dot:

```
local hostname = "www.example.com"
for tag in string.gmatch(hostname, "[^%.]+") do
    print(tag)
end
```

## string.sub()

The `string.sub()` function extracts a portion (substring) of a string as another string.

Three parameters are accepted:

1. the string from which the new string is to be extracted
2. the position within the string to begin (inclusive)
3. the position within the string to end (inclusive, optional, defaults to -1)

For example,

```
local largeString = "ABCDEFGHJKLMNOPQRSTUVWXYZ"
local beginMNO = string.find(largeString, "MNO")
if beginMNO then
    local subMNO = string.sub(largeString, beginMNO + 6) → "MNOPQRS"
```

Note that the substring is 7 characters, not 6. Positions are inclusive.

## string.gsub()

The `string.gsub()` function performs substitution. It replaces some or all of one string with another.

Use of `string.gsub()` is rare in a parser. Typically, values to be registered should be as pristine as possible with no modification, interpretation or translation. Reserve substitution for circumstances in which some character(s) in the string would prevent it from being registered, such as unprintable characters (see *Meta Scrub*, Section Five).

It requires three parameters:

1. the string in which substitutions are to be made
2. a string or pattern to be replaced
3. a string with which to replace

Every occurrence of (2) is replaced with (3).

This is a common way of removing unprintable characters from values.

```
someString = string.gsub(someString, "%c", "")
```

Every control character is replaced with an empty string – effectively removing it.

For advanced needs, the “replace with” doesn’t even have to be another string – a function may be used instead. The matched string to be replaced may be given to the function as a parameter, and the return value will be used as the replacement value.

## append

Append simply concatenates two or more strings. It is not a function - it’s used similar to an arithmetic operation.

Each string to be concatenated are delineated with a pair of consecutive dots, with a space on either side of the dots.

```
local greeting = "Hello" .. " " .. "world!" → Hello world!
```

Variables may be used as well, so long as hold string values. Combinations of literal strings and variables may also be used.

```
local helloString = "Hello"
local greeting = helloString .. " world!" → Hello world!
```

## tonumber()

This is a lua function that will convert a string representation of a number into a numeric representation.

```
local stringOne = "1"
local numericOne = tonumber(stringOne) → 1
```

If the value passed isn't representative of a number, the result is nil.

```
local stringOne = "one"
local numericOne = tonumber(stringOne) → nil
```

Since conversion to numeric is automatic for arithmetic operations, this function is only really useful when a value needs to be explicitly converted to a number for some other reason.

```
local stringOne = "1"
local numericThree = stringOne + 2 → 3
```

## tostring()

This complementary lua function will convert a value to a string representation.

```
local numericOne = 1
local stringOne = tostring(numericOne) → "1"
```

## nw.base64Decode()

This is a parser function which accepts a base64-encoded string, returning an unencoded string.

```
base64String = "SGVsbG8gd29ybGQh"
unencodedString = nw.base64Decode(base64String) → "Hello world!"
```

## MULTIPLE NUMERIC VALUES

The sections detailing `payload:uint8()` et al showed how to get a single value from a length of bytes. It is also possible to get multiple values from a set of bytes.

Without multiple assignment, to get three 1-byte values you'd have to do:

```
local byte1 = payload:uint8(position)
local byte2 = payload:uint8(position + 1)
local byte3 = payload:uint16(position + 2)
```

Or, multiple assignment could be used instead to get more than one value at the same time.

If a “to” position is specified that is longer than the number of bytes returned by the relevant function (1 for `uint8`, 2 for `uint16`, etc.), then multiple values are returned:

`payload:uint8(1, 2)` → two 1-byte values

`payload:uint16(3, 8)` → three 2-byte values

`payload:uint32(9, 24)` → four 4-byte values

Using multiple assignment, to get three 1-byte values you could:

```
local byte1, byte2, byte3 = payload:uint8(position, position + 2)
```

- `byte1` gets the value at `position`
- `byte2` gets the value at `position + 1`
- `byte3` gets the value at `position + 2`

## BITWISE OPERATIONS

Lua parsing includes the functionality of the `BitOp` module from LuaJIT. Definitive documentation is found on its web site (see *More Lua Resources*, Section Three). It is only briefly covered here.

**Important:** All values run through a bitwise operation are normalized to 32-bit signed integers. This behavior can cause unexpected results if you do not account for it.

### `bit.bor()`

Performs a bitwise “or” between two values.

```
bit.bor(1,2) → 3
```

```
bit.bor(2,4) → 6
```

```
local x = 5
local y = 9
local z = bit.bor(x, y) → 13
```

### bit.band()

Performs a bitwise “and” between two values.

```
bit.band(3, 1) → 1
bit.band(3, 2) → 2

local x = 5
local y = 9
local z = bit.band(x, y) → 1
```

### bit.lshift()

### bit.rshift()

Performs a bitwise left or right shift on the given value of the given number of bit positions.

```
bit.lshift(1, 2) → 4
bit.rshift(4, 2) → 1

local x = 16
local y = bit.lshift(x, 4) → 256
local z = bit.rshift(x, 4) → 1
```

### bit.rol()

### bit.ror()

Similar to left and right shift, except that bits shifted out of one side are shifted back on the other side.

```
bit.rol(1, 8) → 256
bit.ror(512, 4) → 32
```

Remember that values are normalized to 32-bit signed integers.

```
bit.rol(1, 32) → 1
bit.rol(1, 31) → -2147483648
```

### bit.bswap()

Swaps the bytes of the value, e.g. 0x00010203 → 0x03020100.

```
bit.bswap(66051) → 50462976
```

### 3-byte numeric values

There is no such function as `payload:uint24()`. So to get a 3-byte value, use a combination of multiple return values and bitwise functions.

For big-endian:

```
local threeBytes
local byte1, byte2, byte3 = payload:uint8(position, position + 2)
if byte1 and byte2 and byte3 then
    byte1 = bit.lshift(byte1, 16)
    byte2 = bit.lshift(byte2, 8)
    threeBytes = bit.bor(byte1, byte2)
    threeBytes = bit.bor(threeBytes, byte3)
end
```

For little-endian, just assign the variables in reverse - then the shifts are the same as above:

```
local threeBytes
local byte3, byte2, byte1 = payload:uint8(position, position + 2)
if byte1 and byte2 and byte3 then
    byte1 = bit.lshift(byte1, 16)
    byte2 = bit.lshift(byte2, 8)
    threeBytes = bit.bor(byte1, byte2)
    threeBytes = bit.bor(threeBytes, byte3)
end
```

Note that before any operations are performed on `byte1`, `byte2`, or `byte3`, they are checked to make sure that they are not `NIL`.



## SERVICE IDENTIFICATION

### `nw.setAppType()`

Some parsers intend to identify sessions as a particular application protocol (e.g., “HTTP”, “SMTP”, etc.)

Application meta is index key “service”. The value of service is numeric, generally corresponding to the well-known port used by the protocol (e.g., 80 = HTTP). However, it is very important to understand that port is not protocol – just because it may use port 80 doesn’t mean it is HTTP.

For a mapping of service value to protocol, refer to the `index-decoder.xml`, under ‘key description=“Service Type”’.

For a parser which intends to register meta for service, two things are required:

A) a third field in the parser declaration:

```
local imapParser = nw.createParser(“IMAP”, “IMAP protocol”, 143)
```

The value “143” corresponds to the alias for IMAP (again, not the port number.)

B) a statement, somewhere in the parser:

```
nw.setApptype(143)
```

The parameter “143” should be the same as declared in `nw.createParser()`

When the `nw.setApptype()` function is executed, meta for the “service” key is registered with the value of the parameter. In the example above, this would identify this session as being “IMAP”.

Note that this occurs **even if** some other parser has already set service meta. Last in wins.

### `nw.getAppType()`

A related function is `nw.getAppType()`. Instead of setting service meta, it retrieves it.

```
local service = nw.getAppType()
```

The currently set value for “service” (if any) is returned. If there is no currently set value, “0” is returned, not NIL.

This is particularly useful to ensure that service has not already been set for the session by some other parser.

```
if nw.getAppType() == 0 then
  nw.setAppType(...
```

- The condition will evaluate to true if there is no currently set value for “service”

## **nw.isRequest()**

## **nw.isResponse()**

Often it is important in a parser to know whether a token was matched in the request stream or the response stream. The token may have different significance depending upon in which stream it was matched, or a token match may not be relevant or interesting unless it is in a particular stream.

These two functions provide an easy method to determine in which stream the token matched.

- `nw.isRequest()` will return `TRUE` if the token matched in the request stream. Otherwise it will return `FALSE`.
- `nw.isResponse()` will return `TRUE` if the token matched in the response stream. Otherwise it will return `FALSE`.

The most common use is to use directly evaluate them in a conditional.

```
if nw.isRequest() then
  ...
end

or

if nw.isResponse() then
  ...
end
```

It is important to note that these are not foolproof. Decoder sometimes (though rarely) gets the request and response streams backwards. More commonly, if there is only one stream then it is entirely possible for Decoder to interpret the server to client stream to be the “request stream”.

## BUILDING TABLES

The easiest way to build a table is to simply add a value to an index, as show before:

```
someTable[1] = "A"
someTable[2] = "B"
someTable[3] = "C"
```

But there are other techniques for building tables that are easier or more efficient.

### automatic indexing

This technique only works for numerically indexed tables.

The length of a table may be retrieved with #tableName, e.g.,

```
tableLength = #myTable
```

That value can also be used to add an entry to the end of the table:

```
myTable[#myTable + 1] = someValue
```

This is logically identical to,

```
local tableLength = #myTable
myTable[tableLength + 1] = someValue
```

which is also logically identical to,

```
local tableLength = #myTable
tableLength = tableLength + 1
myTable[tableLength] = someValue
```

### table.insert()

The `table.insert()` function is logically identical to “automatic indexing”, but much more efficient. Like automatic indexing, it only works for numerically indexed tables.

It accepts two parameters,

- 1) the name of the table in which to insert a value
- 2) the value to be inserted

It will add, to the end of the table, an index with the specified value.

```
table.insert(myTable, "D")
table.insert(myTable, someValue)
```

Whenever possible, use `table.insert()` in preference to “automatic indexing”.

### `table.remove()`

The `table.remove()` function doesn’t build a table, instead it removes entries from a numerically indexed table.

It accepts two parameters,

- 1) the name of the table from which to remove an entry
- 2) the index of the entry to be removed

For example,

```
table.remove(myTable, 5)
```

The *value* at index 5 will be removed from the table.

Be aware that `table.remove()` will **not** leave a gap in the index numbers. All the values above the removed value will be moved “down”. In the example above, there will still be an index 5, but it will now contain the value previously held in entry 6:

```
myTable
...
4 = "D"      4 = "D"
5 = "E"  →  5 = "F"
6 = "F"...   6 = "..."
```

### `table constructors`

When the values to be added to a table are known in advance, such as with a lookup table, instead of building a table entry by entry, build it with a “constructor”.

Instead of,

```

userTable["A"] = "alice"
userTable["B"] = "bob"
userTable["C"] = "charlie"

```

or

```

errorTable[1] = "not found"
errorTable[2] = "access denied"
errorTable[3] = "pc load letter"

```

The tables could be constructed as:

```

userTable = ({
  ["A"] = "alice",
  ["B"] = "bob",
  ["C"] = "charlie",
})

```

and

```

errorTable = ({
  [1] = "not found",
  [2] = "access denied",
  [3] = "pc load letter",
})

```

For a numerically indexed table, the index numbers aren't even strictly necessary:

```

errorTable = ({
  "not found",
  "access denied",
  "pc load letter",
})

```

The table will be indexed numerically, with three entries (1, 2, and 3).

## CALLING OTHER FUNCTIONS

Functions may be written into a parser that are not called directly by a callback, so operations common to multiple functions may be written once and called when needed instead of included in each callback function.

The function is declared just as a callback function, accounting for whatever parameters the parser will pass to it.

```
function exampleFunction(varA, varB, varC)
    ...
end
```

To call the function from elsewhere in the parser,

```
self:exampleFunction(A, B, C)
```

- The function is called as `self:exampleFunction()`, not `exampleFunction()`.
- Any values the function expects are passed to it

Return values (if any) may be assigned to variables as desired,

```
local D = self:exampleFunction(A, B, C)
```

```
local E, F, G = self:exampleFunction(A, B, C)
```

Just as with any other function, if there isn't a return value then the variable will be assigned NIL.

The function is only available to the parser in which it was declared; it cannot be called by from another parser. *Modules*, Section Nine, will provide a technique to share functions among multiple parsers.

## NETWORK META

Generally, parsers are not concerned with the network characteristics of a session. An HTTP session is HTTP, regardless of whether it uses TCP port 80 or not, and just because a session uses TCP port 80 doesn't mean that it is HTTP. To the MAIL parser, it doesn't matter whether an email message is seen on TCP 25, TCP 110, TCP 143, or something else entirely.

In limited circumstances however it may be important for a parser to know some network characteristic of a session. DNS over TCP differs slightly from DNS over UDP, so a DNS parser needs to know which transport protocol was used for the session.

### `nw.getNetworkProtocol()`

Accepts no parameters.

Returns the numeric ethertype of the session:

<u>Protocol</u>	<u>Hex</u>	<u>Numeric</u>
IPv4	0x0800	2048
IPv6	0x86DD	34525
ARP	0x0806	2054
etc...		

### `nw.getTransport()`

Accepts no parameters.

Returns multiple values:

- 1) the numeric layer 4 protocol number of the session:

<u>Protocol</u>	<u>Number</u>
ICMP	1
TCP	6
UDP	17
etc...	

- 2) the source port of the session (only for TCP and UDP)
- 3) the destination port of the session (only for TCP and UDP)

### `nw.getSessionSource()`

Accepts no parameters.

Returns, as a string, the source IP address of the session.

### `nw.getSessionDestination()`

Accepts no parameters.

Returns, as a string, the destination IP address of the session.

### `nw.getSessionAddresses()`

Accepts no parameters.

Returns multiple (as strings),

1. source IP address
2. destination IP address

```
local srcIP, dstIP = nwsession.getAddresses()
```

### **nw.getSessionPorts()**

Accepts no parameters.

For TCP or UDP, returns multiple,

1. source port
2. destination port

```
local srcPort, dstPort = nwsession.getPorts()
```

### **nw.getSessionStats()**

Accepts no parameters.

Returns multiple:

1. number of streams in the session
2. number of packets in the session
3. number of bytes in the session (including headers and footers)
4. number of *payload* bytes in the session (excluding headers and footers)

For example,

```
local streams, packets, bytes, pBytes = nw.getSessionStats()
```

### **nw.getStreamStats()**



The complementary function `nw.getStreamStats()` returns information about the current stream.

Accepts one optional parameter, a stream object (see *Streams*, Section Nine). If omitted, then the default is the stream in which the token currently matched.

Returns multiple:

1. number of packets in the stream
2. number of bytes in the stream (including headers and footers)
3. number of *payload* bytes in the stream (excluding headers and footers)
4. number of packets in the stream that were retransmissions
5. number of payload bytes that were in retransmitted packets

## SEVEN: FINISHING TOUCHES

### ALERT.ID

*Skip or ignore this subsection on “alert.id” if you aren’t writing a parser to be included in Live.*

Any parser to be included in Live which registers an alert must follow these rules:

- No parser should register meta using index key “alert”, under any circumstances, ever.
- Nor should any parser register meta *directly* using index keys “risk.informational”, “risk.suspicious”, “risk.warning”, “threat.category”, or “threat.source”. Instead, it must register meta “alert.id” which will be matched by the alertid feeds to register the risk meta.

It used to be that parsers, feeds, and app rules registered all alert-type meta to the “alert” key. There was no rhyme or reason to the alerts – some were merely bits of miscellaneous information, some were big red flags. Nor was there any tracking or repository of alert meta – each simply registered whatever it wanted.

To rectify this, the “alert.id system” was put in place. It has two parts:

- a) alerts are categorized, in order of increasing severity:
  - a. risk.informational – not necessarily indicative that further analysis is warranted, but could be in combination with other alerts or meta
  - b. risk.suspicious – may warrant further analysis, especially in combination with other alerts or meta
  - c. risk.warning – very suspicious, even by itself worthy of further analysis

*Note that the description of each includes the phrase “further analysis” – no alert, even a risk.warning, constitutes a definitive determination.*

- b) risk meta values are kept in the alertid feeds:
  - a. alertids\_informational – matches alert.id meta to register risk.informational meta
  - b. alertids\_suspicious – matches alert.id meta to register risk.suspicious meta

- c. alertids\_warning – matches alert.id meta to register risk.warning meta

The parser must register a unique alert.id value which is mapped to risk meta by the alertid feeds.

If you have access to the Perforce repository, you may check out the alertid feeds, assign your own value(s) as appropriate, submit the feeds back to Perforce, and use the value(s) in your parser.

If you don't have access to Perforce, you should still write your parser to use alert.id values not register risk meta directly. Just use fake/temporary values, and note that permanent values must be assigned by someone before the parser goes through QE and is published in Live.

## PARSER VERSION

All parsers in Live are encrypted. So there's no way to open the parser file to determine, for example, if it is different than the latest version in Live.

To solve this, the parser should log its name and version at the 'debug' level. If Decoder is configured for 'debug' (/logs/config/log.levels), that information will be logged when the parser is loaded.

Just after `nw.createParser(...)` add a line:

```
nw.logDebug("name version")
```

Every time the parser is revised, update the version.

## COMMENTS

Of course you are leaving comments throughout your parser explaining your logic (aren't you?). But that's not what I'm referring to here.

Every parser should have a block of comments at the top which provide basic information about the parser itself.

**Description:** a long description, more than the few words from `nw.createParser()`, but no more than a sentence or two.

**Version:** a history of the major revisions of the parser, including date, author, SA version, and the changes made.

**Dependencies:** any other parsers or feeds which should be enabled along with the parser. Typically this will be the alertid feeds if the parser registers alert.id meta to be turned into risk meta, nwill.lua if the parser requires a function within nwill, or some other parser if the parser requires a meta-callback for some index key.

**Conflicts:** parsers can't really "conflict", but if the parser duplicates to some degree the functionality of another parser it should be listed here.

**Standard Index Keys:** a list of the standard (OOTB) index keys that the parser will register meta with

**Custom Index Keys:** a list of the non-standard index keys that the parser will register meta with

**Alert.Id:** if the parser registers alert.id meta to be turned into risk meta, list the alert.id values and associated risk meta here.

**Usage Notes:** intended for end users, broadly explaining what the parser does and how it may be best used.

**Options:** Details of each parser option (covered in Section Nine), including its default value and how it affects the functionality of the parser.

**Implementation Notes:** intended for other content authors, explaining the logic of the parser, and listing any external documentation such as RFCs, to aid in the maintenance and bugfixes.

**Todo:** functionality which could or should be added to the parser in a future revision

Each of the Example Parsers in the Appendix contains such a block of comments.

## REMOVE DEBUGGING

If you've added any statements which log information or register meta for debugging purposes – **don't forget to remove them.**

## EIGHT: MISCELLANEOUS

For completeness, this section contains alternative techniques and functionalities to some methods described previously. There isn't anything "new" here, just different ways of accomplishing the same thing.

### `payload:equal()`

The function `payload:equal()` simply compares a payload object to a string - returning true if they are the same, otherwise returning false.

It requires one parameter:

1. the string to compare to the payload object

For example, given a payload object:

```
0x46 0x4f 0x4f
```

```
if payload:equal("FOO") → TRUE
```

```
if payload:equal("Foo") → FALSE
```

```
if payload:equal("foo") → FALSE
```

This is logically identical to:

```
local someString = payload:tostring(1, 3)
if someString == "FOO" then
  ...
end
```

### `nwlanguagekey.getPathDefaults()`

This is an alternative to specifying each of "directory", "filename", and "extension" in `setKeys()`.

Instead of,

```
someParser:setKeys({
  ...
  nwlanguagekey.create("directory"),
```

```

        nwlanguagekey.create("filename"),
        nwlanguagekey.create("extension"),
        ...
    })

```

A parser could,

```

someParser:setKeys({
    ...
    nwlanguagekey.getPathDefaults(),
    ...
})

```

Except as a convenience, this is not particularly useful at this time. It is intended as a future-proofing for possible capabilities.

## createPathMeta()

The createPathMeta() function is an alternative to individually registering meta for directory, filename, and extension.

Given an example variable named "path" containing, instead of extracting each element and registering them,

```

... find last / and extract directory as "pathDir" ...
... extract remainder as "pathFile" ...
... find last . and extact extension as "pathExt" ...
nw.createMeta(self.keys.directory, pathDir)
nw.createMeta(self.keys.filename, pathFile)
nw.createMeta(self.keys.extension, pathExt)

```

A parser could simply,

```

parserName:createPathMeta(path)

```

The each path element will automatically be extracted and registered as appropriate.

An alternative is presented in the nwll.lua module (Appendix). The differences are:

- Missing Path Elements
  - createPathMeta() will, for any missing element (such as extension) register "<none>" as its value

- `nwll.extractPathElements()` function will simply return NIL for any missing path element
- Normalization of directory slash direction
  - `createPathMeta()` will normalize backslashes in directory to forward slashes
  - `nwll.extractPathElements()` function will preserve backslashes

In both cases I prefer the approach taken by `nwll.extractPathElements()`, that's why I wrote it. But the choice is yours.

## setKeys()

There is an alternative method of specifying the index keys with which parser will register meta, and using those keys in `nw.createMeta()`. You won't see this in any of the example parsers in the Appendix; it is presented here so that you will be aware of it, and use it if you choose.

Instead of (for example),

```
parserName:setKeys({
  nwlanguagekey.create("action"),
  nwlanguagekey.create("username"),
})
```

A parser could,

```
local metaAction = nwlanguagekey.create("action")
local metaUsername = nwlanguagekey.create("username")
parserName:setKeys({
  metaAction,
  metaFile
})
```

"metaAction" and "metaUsername" are variables, which can be named as any valid variable name, and which hold the index key

Essentially this adds a layer of abstraction. Rather than referring to index keys directly ('self.keys.keyname'), they may be referenced with a variable instead.

In order to register meta if keys have been specified as variables, an alternate use of `nw.createMeta()` is required.

Instead of:

```
nw.createMeta(self.keys.action, "delete")
```

use:

```
nw.createMeta(metaAction.handle, "delete")
```

- “metaAction” is the variable name holding `nwlanguagekey.create(“action”)`
- “handle” is the literal word “handle”

## RENAMING A PAYLOAD OBJECT

Until now, you’ve been advised to always name the payload object variable “payload”.

However, there may be circumstances in which you need to interact with more than one payload object at a time. You can’t just name them both “payload”, because they will overwrite each other.

In reality, you don’t have to name the payload object “payload”; it just makes it easier on yourself if you do.

Remember that the various payload functions `payload:find`, `payload:uint8`, `payload:tostring`, et al, are called using a colon (":") instead of a dot ("."). What’s going on behind the scenes is that the string “payload” before a colon is actually a variable being passed as the first parameter of the function (“find”, “uint8”, “tostring” ...)

```
payload:tostring(1, -1)
  ^^^      ^^^
variable function
```

By naming your payload object “payload”, you are always passing the variable which holds the payload object as the first parameter.

So you could, in theory, do something like:

```
local foo = nw.getPayload(x, y)
local bar = foo:tostring(1, -1)
```

However, a better way is to instead use the more explicit name of the function, and manually pass the payload object as the first parameter:

```
local foo = nw.getPayload(x, y)
local bar = nwpayload.tostring(foo, 1, -1)
```



Why is this better? I’ve seen the first method (e.g., `foo:tostring()`) perform oddly, with unexpected results, but admit that I don’t know why. Whereas the second method (e.g., `nwpayload.tostring(foo, ...)`) works correctly every time.

Note that when using the “proper name” of the function (e.g., `nwpayload.tostring()`) it is called with a dot, not a colon.

All of the “nwpayload” functions listed in `nw-api.lua` (*Appendix*) may be called using their “proper name”,

e.g., either

```
payload:uint8(...)
```

or

```
nwpayload:uint8(payload, ...)
```

Just remember that the second syntax requires:

- dot (“.”), not colon (“:”)
- the payload object variable be passed as the very first parameter

## CONDITIONAL ASSIGNMENT

This is something of a Lua shortcut to assign a value to a variable based upon a condition without explicitly using an “if”. It is used in some of the example parsers in the *Appendix*.

The most common use is to initialize a variable (such as a table) only if it doesn’t already have a value.

```
myTable = myTable or {}
```

Is logically identical to,

```
if not myTable then
    myTable = {}
end
```

- If `myTable` doesn’t already exist, then it is declared as an empty table
- Otherwise it is left alone

This is very handy when localizing a global.

```
local count = self.sessionVars.count or 1
```

- If `self.sessionVars.count` exists then `count` is given its value
- Otherwise `count` is given a value of “1”

Another common use it to make sure a variable has a value (nil-check) before performing an operation on it:

```
count = count and count + 1
```

is logically identical to,

```
if count then
  count = count + 1
end
```

- Only if `count` has a non-NIL value will it be incremented.

As well,

```
varC = varA and varB and varA .. varB
```

is logically identical to

```
if varA and varB then
  varC = varA .. varB
end
```

- Only if both `varA` and `varB` have non-NIL values will they be concatenated.

A less common use it to check that a variable has a particular value before performing an operation on it.

```
count = (count and count < 4 and count + 1) or count
```

is logically identical to

```
if count and count < 4 then
  count = count + 1
end
```

- Only if `count` exists and is less than 4 will it be incremented.

Why add “or count” to the end? Without it, `count` would get a value of `FALSE` if `count` is greater than or equal to 4. With it, `count` will retain its value in that case.

## REGISTER META DIRECTLY FROM PAYLOAD

It is not strictly necessary to extract a value as a string (or number) before registering it. Instead, part (or all) of a payload object itself may be registered. The bytes in the payload object will automatically be converted to a string.

This does **not** remove any requirements for knowing where the value is, the beginning and ending positions within in the payload object of the value to be registered.

### Register a Payload Object

The `nw.createMeta()` function will accept an entire payload object as the value to be registered. The entire payload object, from the first byte to the last, will be converted to a string as registered as meta.

How is that useful? With `payload:sub()`,

```
local endOfLine = payload:find("\013\010", pos, pos + 32)
if endOfLine then
    local payload = payload:sub(pos, endOfLine - 1)
    nw.createMeta(self.keys.username, payload)
end
```

Or even,

```
local endOfLine = payload:find("\013\010", pos, pos + 32)
if endOfLine then
    nw.createMeta(self.keys.username, payload:sub(pos, endOfLine - 1))
end
```

### Register Positions within the Stream

An alternate form of the `nw.createMeta()` function will use positions within the stream of the value to be registered.

Three parameters are accepted,

1. The index key to be used for the meta value, using 'self.keys.key' syntax
2. The position in the **stream** of the beginning of the value
3. The position in the **stream** of the end of the value (optional, defaults to -1)

IMPORTANT: Note the term “*stream*”, not “*payload object*”. When used in this way, the positions are relative to the entire stream, regardless of the current payload object. Be very careful.

```
nw.createMeta(self.keys.username, position, position + length - 1)
```

## TAKE ADVANTAGE OF META SCRUB

This is convenient for registering meta values which you know will be terminated by a non-printable character such as a carriage return. Instead of using `payload.find()` to search for the carriage return, reading the characters up to it, then registering that string – either read some arbitrary number of characters you know will include the carriage return, or even better just register the payload beginning at the first character of the value, and let meta scrub terminate the value for you.

For example, given the token “Username: “, and knowing that the username itself will be no more than 16 characters terminated by a carriage return:

```
function demoParser.onUsername(token, first, last)
    local payload = nw.getPayload(last + 1, last + 17)
    nw.createMeta(self.keys.username, payload)
end
```

## OPTIMIZATIONS

There are a few techniques which will help a parser perform better and be less of a burden to Decoder.

The greatest improvements to be had are to,

- a) use small payload objects (Section Six – Use Small Payload Objects)
- b) limit the length of `payload.find` (Section Four – Extracting Values)
- c) avoid string manipulation whenever possible

Make a habit of doing those things, right from the start.

For the rest presented here, don’t be overly concerned with them until you are generally comfortable writing parsers.

## Use `table.concat()` instead of `append`

For concatenating several strings, this technique provides several advantages.

Behind the scenes, strings are immutable. When strings are concatenated, new strings are created, and memory has to be copied around. As more strings are concatenated, the memory requirements grow exponentially. This is not unique to Lua – many languages behave similarly.

The `table.concat()` function ameliorates this issue. It returns the concatenation of its entries.

- the table must be numerically indexed
- entries must be strings or numbers (numbers are converted to strings)
- there must be no gaps

```
local someTable = ({
    [1] = "AB",
    [2] = "CD",
    [3] = 567,
    [4] = "GHI"
})
local concatenatedTable = table.concat(someTable) → ABCD567GHI
```

For example, instead of:

```
local varA = payload:tostring(pos, pos + 5)
local varB = payload:tostring(pos + 6, pos + 8)
...
local varF = payload:tostring(pos + 26, pos + 29)
local concatenatedString = varA .. varB .. varC .. varD .. varE .. varF
```

The following is logically identical, but significantly more efficient:

```
local tempTable = {}
table.insert(tempTable, payload:tostring(pos, pos + 5))
table.insert(tempTable, payload:tostring(pos + 6, pos + 8))
...
table.insert(tempTable, payload:tostring(pos + 26, pos + 29))
local concatenatedString = table.concat(tempTable)
```

**Do not do this, ever:**

```
local concatenatedString = ""
for i,j in ipairs(tempTable) do
    concatenatedString = concatenatedString .. j
end
```

Want proof? Try this exercise: open a lua CLI and type,

```
tempTable = {}
for i = 1, 1000000 do
    table.insert(tempTable, i)
end
catString = ""
for i,j in ipairs(tempTable) do
    catString = catString .. j
end
```

Now go catch up on email, run some errands, come back in a while – it probably still won't be finished. Once you've given up and closed the window, open it again and type this instead,

```
tempTable = {}
for i = 1, 1000000 do
    table.insert(tempTable, i)
end
catString = table.concat(tempTable)
```

Don't blink.

## Localize Globals

Accessing the value stored in a local variable is much more efficient than from a global. If a global is referenced more than twice in a block – localize it to take advantage of that efficiency.

What does “localize it” mean? Consider the following simple example:

```
local count = self.sessionVars.count
if count and count >= 2 and count <= 5 then
    count == count + 1
    self.sessionVars.count = count
end
```

- 1) the value of self.sessionVars.count (global) is assigned to count (local)
- 2) count is compared and incremented
- 3) the value of count (local) is assigned back to self.sessionVars.count (global)

Functions may be localized as well, since functions are really just variables that hold the function itself.

```

local stringLength = string.len
local lengthA = stringLength(A)
local lengthB = stringLength(B)
local lengthC = stringLength(C)

```

- The function `string.len` is assigned to a local variable `stringLen`
- Note that parentheses are not used when localizing the function. The function is not called – only the variable name holding the function is referenced.

You may be tempted to localize an entire library (such as “string”) rather than the individual function within the library – **don’t**. Only localize the individual functions in a library that will be used.

Care must be taken if localizing the “payload:” functions.

- The “`nwpayload.`” form of the function must be called,
- not the “`payload:`” method

See “*Renaming a Payload Object*”, Section Eight.

## Localize Locals

The same techniques used to localize globals may be used to further localize a variable which itself is local to a parent block. The efficiency gains are smaller but in aggregate may be significant, especially within a loop.

```

local stringFind = string.find
local lastDirectory
repeat
    local loopControl = 0
    local stringFind = stringFind
    local found = stringFind(path, “/”, lastDirectory + 1)
    if found then
        lastDirectory = found
        loopControl = 1
    end
until loopControl == 0

```

- 1) The `string.find` function is localized in the parent block as `stringFind`
- 2) It is localized further in the repeat block

## Use Lookup Tables

Instead of a long chain of “if... elseif... elseif... elseif...”, a lookup table can accomplish the same task using only one evaluation rather than several.

For example, instead of,

```
if errorCode == 1 then
    nw.createMeta(self.keys.error, "not found")
elseif errorCode == 2 then
    nw.createMeta(self.keys.error, "access denied")
elseif myVar == 3 then
    nw.createMeta(self.keys.error, "pc load letter")
end
```

Construct a table consisting of the values that are to be matched against:

```
local errorTable = ({
    [1] = "not found",
    [2] = "access denied",
    [3] = "pc load letter",
})
```

then determine if the entry exists in the table:

```
if errorTable[errorCode] then
    nw.createMeta(self.keys.error, errorTable[errorCode])
end
```



## NINE: ADVANCED

Don't attempt to implement the techniques in this section until you are very comfortable writing parsers. These are "cutting-edge" recently-added parsing capabilities. Some of them are not necessarily well-tested or otherwise documented.

Try not to think of sessions in terms of "packets". It is very rare that a parser needs to examine sessions at a packet level.

### STREAM OBJECTS AND PACKET OBJECTS

Much like payload objects, streams and packets may be values stored in a variable as an object. This section lists the available functions to get and interact with streams and packets in a parser.

A packet object contains all of the bytes of a packet, including its headers, footers, and payload. Values may be extracted from a packet object using a set of functions similar to payload functions.

#### NWSESSION

`nwsession.getRequestStream()`

`nwsession.getResponseStream()`

Returns a stream object of the request or response stream, respectively.

`nwsession.getFirstPacket()`

`nwsession.getLastPacket()`

Returns a packet object of the first or last packet of the session, respectively.

#### NWSTREAM

The `nwstream` functions are used to interact with stream objects.

### `nwstream.getPayload()`

Returns a payload object from the specified stream.

Accepts three parameters:

1. the name of a variable holding a stream object (defaults to the current stream)
2. position of the first byte of payload to be returned (defaults to “1”)
3. position of the last byte of payload to be returned (defaults to “-1”)

For example,

```
local reqStream = nwsession.getRequestStream()
local payload = nwstream.getPayload(reqStream(1, 32))
```

The `nwstream.getPayload()` function is actually identical to `nw.getPayload()`, which also accepts a stream object as an optional parameter.

```
local reqStream = nwsession.getRequestStream()
local payload = nw.getPayload(reqStream, 1, 32))
```

### `nwstream.getFirstPacket()`

### `nwstream.getLastPacket()`

Returns a packet object of the first or last packet of the session, respectively.

Accepts one parameter:

1. the name of a variable holding a stream object (defaults to the current stream)

For example,

```
local packet = nwstream.getFirstPacket()

local reqStream = nwsession.getRequestStream()
local packet = nwstream.getFirstPacket(reqStream)
```

## NWPACKET

The nwpacket functions are used to get information about and extract values from packet objects.

### `nwpacket.getTimestamp()`

Requires one parameter - the name of a variable holding a packet object.

Returns two values:

1. the capture time in seconds (epoch)
2. milliseconds

### `nwpacket.getSize()`

Requires one parameter - the name of a variable holding a packet object.

Returns the number of bytes in the packet, including header, payload, and footer bytes.

### `nwpacket.getTcpSeqAndFlags()`

Requires one parameter - the name of a variable holding a packet object.

Returns multiple,

1. the TCP sequence number in the packet
2. the TCP flags present in the packet

For a non-TCP packet, “0” will be returned for each (not NIL).

`nwpacket.byte()`

`nwpacket.int8()`

`nwpacket.uint8()`

`nwpacket.int16()`

`nwpacket.uint16()`

`nwpacket.int32()`

`nwpacket.uint32()`

`nwpacket.tostring()`

These functions are each equivalent to their `nwpayload` counterparts.

Each requires, as the first parameter, the name of a variable holding a packet object.

`nwpacket.hasSessionNext()`

`nwpacket.hasSessionPrevious()`

Requires one parameter – the name of a variable holding a packet object.

Returns `TRUE` if another packet follows (or precedes, respectively) the specified packet in the session. Otherwise returns `FALSE`.

Note that packets will “appear” in the order that they were captured, regardless of the stream to which they belong and regardless of whether they arrived out of sequence order. Furthermore, all retransmissions are included – if the next packet was a retransmission of the current packet, it is still the “next” packet.

`nwpacket.getSessionNext()`

`nwpacket.getSessionPrevious()`

Requires one parameter – the name of a variable holding a packet object.

Returns the following (or preceding, respectively) packet in the session, if there is a packet to be retrieved. Returns `NIL` otherwise.

Note that just as with `nwpacket.hasSessionNext()` and `nwpacket.hasSessionPrevious()`, packets will “appear” in the order that they were captured, regardless of the stream to which they belong and regardless of whether they arrived out of sequence order. Furthermore, all retransmissions are included – if the next packet was a retransmission of the current packet, it is still the “next” packet.

```
local packet = nwsession.getFirstPacket()
if packet then
    ...
    if nwpacket.hasSessionNext(packet) then
        packet = nwpacket.getSessionNext(packet)
        if packet then
            ...
```

### `nwpacket.hasStreamNext()`

### `nwpacket.hasStreamPrevious()`

Requires one parameter – the name of a variable holding a packet object.

Returns TRUE if another packet follows (or precedes, respectively) the specified packet in the stream. Otherwise returns FALSE.

Packets will “appear” in the order that they were captured, regardless of whether they arrived out of sequence order.

However, retransmissions are **omitted** – if the next packet was a retransmission of the current packet, it is skipped. This is a notable difference from `nwpacket.hasSessionNext()` and `nwpacket.hasSessionPrevious()`.

### `nwpacket.getStreamNext()`

### `nwpacket.getStreamPrevious()`

Requires one parameter – the name of a variable holding a packet object.

Returns the following (or preceding, respectively) packet in the session, if there is a packet to be retrieved. Returns NIL otherwise.

Packets will “appear” in the order that they were captured, regardless of the stream to which they belong and regardless of whether they arrived out of sequence order.

However, retransmissions are omitted – if the next packet was a retransmission of the current packet, it is skipped. This is a notable difference from `nwpacket.getSessionNext()` and `nwpacket.getSessionPrevious()`.

```

local reqStream = nwsession.getRequestStream()
if reqStream then
    local packet = nwstream.getFirstPacket(reqStream)
    if packet then
        ...
        if nwpacket.hasStreamNext(packet) then
            packet = nwpacket.getStreamNext(packet)
            if packet then
                ...
            end
        end
    end
end

```

### `nwpacket.getPayload()`

Accepts one parameter – the name of a variable holding a packet object.

Returns a payload object of the payload from the packet. This payload object may be used just as any other payload object such as from `nw.getPayload()`.

## ALTERNATE CHARACTER SETS

When string meta values are registered, they are assumed to be UTF-8, unless otherwise specified.

The function `nw.createMeta()` actually accepts another optional character encoding parameter not mentioned previously to specify that the string is encoded in something other than UTF-8, such as “UTF-16LE”, “windows-1256”, etc.

The encoding parameter should be passed after the value itself, or after the “from” and “to” parameters if those are used instead.

The parameter itself is the name of the character set.

```

nw.createMeta(self.keys.filename, varFilename, “UTF-16LE”)

```

## MODULES

The use of modules allows a parser to call a function from another file.

The caveats are:

- The function must itself be written in lua
  - it can't be a third party binary "lua module"
- A module is not itself a parser
  - meta cannot be registered within the module
- The function called must be passed any variables it needs
  - it won't have access any variables in the parser, not even globals
- Any library that a function in the module needs must be explicitly required,

```
local string = require('string')
```

The module file must have a .lua (or .luax, for encrypted) extension and reside in the parsers directory of Decoder.

Inside the module file, the line

```
module("moduleName")
```

Where "*name*" is the name of the module.

must appear before any function definitions.

Unlike in a parser file, function names are not tied to a specific parser using `parserName:functionName()` Instead,

```
function functionName(...)
```

Any parser that will call a function defined in a module must explicitly require the module,

```
local moduleName = require('moduleName')
```

Calling a module function from a parser is similar to calling a library function,

```
moduleName.functionName()
```

Two modules are included in the Appendix.

## nwll.lua

The nwll module is used by many lua parsers in Live. It currently contains the following functions (see the notes in the module file in the Appendix for detailed information on usage):

```
nwll.determineHostType()
nwll.extractPathElements()
nwll.extractUrlElements()
nwll.readASNlength()
nwll.decodeASN()
nwll.extractKerberos()
nwll.urldecode()
nwll.decodeQuotedPrintable()
nwll.convertEBCDIC()
nwll.winErr()
nwll.friendlyOID()
```

## debugParser.lua

The debugParser module is used to aid in the writing and debugging of a parser. It currently contains only two functions (see the notes in the module file in the Appendix for detailed information on usage):

```
debugParser.logPayload()
debugParser.logTable()
```

Here's a neat trick to see the entire environment of a lua parser:

**Don't do this on a production decoder!**

1. Write a parser that
  - a. has only a callback for OnSessionBegin
  - b. requires the debugParser module from the Appendix



- c. in the callback put only  
`debugParser.logTable(_G)`
2. Enable the parser, reload parsers
3. Import a pcap which contains only one session
4. Watch the log – there will a lot of information

## SHARED VALUES

Shared Values allows a value to be held and referenced by the same parser but in a different session. The possible uses for this include,

- parse a session differently depending upon what has occurred in a previous session
- count some event or value across sessions
- basic correlations of events or values across sessions

As mentioned when “globals” were covered, the biggest hurdle to referencing a variable value set by a parser in a prior session is threadedness. There’s no guarantee that the same thread will get the later session. Shared Values solves this problem in a thread safe manner.

This opens up entire realms of possibilities never before possible in a parser.

However, there is of course a cost – the performance impact of Shared Values has not been well tested. Given the thread locking that must take place, it is possible that excessive use of Shared Values could cripple a decoder.

No current “production” parser uses this capability.

Tread carefully.

### parser:setSharedValue(name, value)

Give value “value” to shared variable “name”.

```
local varC = varA + varB
myParser:setSharedValue("alpha", varC)
```

### `parser:getSharedValue(name)`

Retrieve the value of shared variable “name”.

```
local varA = myParser:getSharedValue("alpha")
```

### `parser:removeSharedValue(name)`

Delete the value of shared variable “name”.

```
myParser:removeSharedValue("alpha")
```

## OPTIONS

Options don’t exist yet. They are entirely a figment of my imagination. You’ll see them in the example parsers, but there is no way to expose options outside of the parser itself.

As such, this section is reserved for when options are a reality.

## APPENDIX

Descriptions for the files included in the zip archive along with this book. I'm no longer embedding them directly in the word document.

### nw-api.lua

The nw-api.lua file does nothing on its own. It is not a parser, it contains no actual functionality. It is not required for parsing in any way.

It serves as the definitive documentation of all parser-specific lua API functions available to a parser.

It can also be loaded as a lua “dofile” in the lua CLI, for checking the syntax of a parser (see *Five: Debugging*).

Do **not** put it in the “parsers” directory of Decoder – Decoder will try to load it and throw an error.

## EXAMPLE PARSERS

These parsers are the same as they exist in Live at the time of writing. They range in order from very simple (fingerprint\_jpg) to very complex (TLS).

### fingerprint\_jpg.lua

The “fingerprint\_jpg\_lua” parser is really just a signature. It looks for a jpeg header, and registers “filetype: jpg” upon a match. Parsers don't get any more simple than this.

### gnutella.lua

The “gnutella\_lua” parser is a little more complicated, though it is still very signature-based. It requires seeing a combination of tokens in order to identify the Gnutella protocol. However, it does not extract any meta such as filenames.

### **fingerprint\_rar.lua**

The “fingerprint\_rar\_lua” parser is a significant step up in complexity. But it really consists of three individual functionalities, (a) rar v4, (b) rar v5, and (c) base64-encoded. Work through each in turn rather than trying to grok the entirety at once.

### **mail.lua**

“MAIL\_lua” parses email messages themselves – not SMTP, or IMAP, etc.

I recently rewrote this parser from scratch. I’ve included both the new and the old.

The old saves the position of each header found until the end of the header block, at which point it retrieves a payload object for each header individually from which to extract values.

The new gets a payload object of an entire block of headers, converts it to a lua table, then iterates that table.

### **tls.lua**

“TLS\_lua” is one of the more complex parsers in Live. The “Implementation” notes do a decent job of providing the overall logic to get you started.

It extracts certificates in a chain as payload objects stored in a global to be referenced later when other tokens match.

I intend to rewrite it, using a less convoluted and hopefully more efficient technique such as used in (old) MAIL\_lua, or perhaps leverage decodeASN() in nwl to extract the certificate chain as a lua table.

## **EXAMPLE MODULES**

### **nwl.lua**

The “nwl” module is distributed in Live. It contains functions written in Lua that any parsers may use so that each does not need to include those functions individually, which provides consistency and ease of maintenance.

### **debugParser.lua**

The “debugParser” module is not distributed in Live. It was written solely to aid in the debugging of parsers during development, and has no useful purpose otherwise.

## ERRATA

Need to add discussion of callbacks being independent, discrete events.

Modify “tokens should always be there” section – remove boolean confusion.

Need to add method of using meta-callbacks when resetting globals OnSessionBegin:

- If no token matches occur at all, then the meta callback occurs but SessionBegin ***does not***.
- If any token matches before the callback occurs, then the callback occurs after SessionBegin
  1. Session Begin
  2. Token Match
  3. Meta-Callback
  4. Token Match
- If all tokens match after the callback occurs, then the callback occurs before SessionBegin
  1. Meta-Callback
  2. SessionBegin
  3. Token Match
  4. Token Match

**Summary:** Don’t mix tokens and meta-callbacks in the same parser unless you really *really* need to. Doing so safely is difficult to impossible depending on what you are attempting.

Need to add section on writing log parsers in lua.

- Does not preclude log from being parsed by normal log parsers
- All meta-callbacks occur at end of session, in no particular order
- Lua transforms in C2