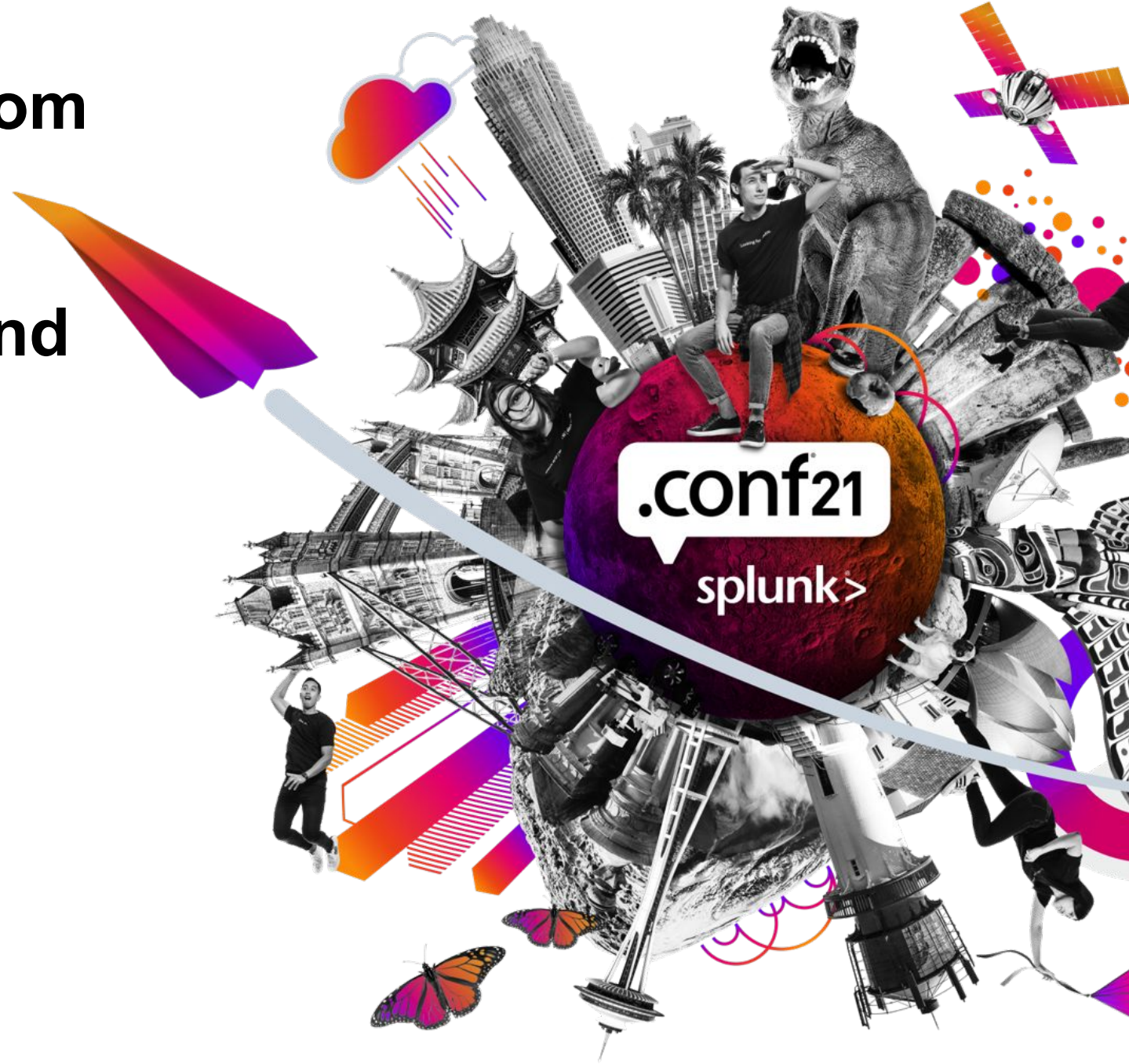


DEV1387B

# Noah Dietrich

Information Management Specialist |  
US Department of State





# Noah Dietrich

Information Management Specialist |  
U.S. Department of State

# About Me

“I’m a Virgo who likes long walks on the beach...”

Information Management Specialist with the U.S. Department of State

- Over the past 10 years I’ve lived in Mumbai, Rome, Eswatini, and Frankfurt

Open Source Contributor:

- Author of the Snort on Ubuntu documentation on [snort.org](https://snort.org).
- Contributor to [PulledPork3](#): The Snort 3 ruleset downloader.

Author of 2 Splunkbase Add-ons:

- [Snort 3 JSON Alerts](#)
- [CyberChef for Splunk](#)



# Agenda

## 1) Introduction

What's this all about then?

## 2) The Custom Search Protocol v2

An Overview

## 3) Getting Started: Logging

## 4) Rolling Your Own Custom Search Command

Avoid all the mistakes I made

## 5) Wrap-up



# Introduction



# About the CyberChef for Splunk Add-on

This presentation's origin story



- [CyberChef](#) is a web-based application for encryption, encoding, compression and data analysis.
- CyberChef also has a node.js API.
- I wanted to make all the functionality of CyberChef available as a single custom search command.
- I built the [CyberChef for Splunk](#) Add-on to make this happen, but had to implement Splunk's Custom Search Protocol v2 manually.

```
| cyberchef infield=data operation='ToBase64'
```

# About This Presentation

Everything I learned while building the CyberChef Add-on

All Examples will be in javascript / node.js

- A Custom Search Command can be written in any language.
- I am not a javascript expert by any means.

This will not be a complete breakdown of the Protocol

- I don't have a complete understanding of the protocol...
- But I know enough to make it work.

I will cover all of the quirks and gotchas that I ran into

- Which will save you a lot of frustration.

I will not talk about the Python API

# Expected Knowledge of the Viewer

“The files are *in* the computer?”

This presentation will make more sense if:

- You have experience programming in your language of choice.
- You have implemented your own custom search command using the Python SDK
- You have an understanding of the four different types of search commands (streaming, reporting, etc.).

Jacob Leverich's .conf 2018 presentation is a great resource:

- [Extending SPL with Custom Search Commands](#).
- Former Splunker who wrote the protocol.
- Describes the protocol, but with a focus on the Python SDK.
- It's not online anymore, but I have a copy of the slides available.



# Warning

“Here Be Dragons”

Implementing the Custom Search Protocol yourself is not easy

- There are a lot of edge cases to watch for.
- There is little documentation available.
- This presentation will make it easier though.

Make sure to choose a language that is powerful enough to do what you need

- Choose a language with good csv and json support (UTF-8).
- Make sure you can read directly from the stdin buffer.
- Splunk includes an older version of the node.js runtime.

I will not cover v1 of the protocol

- Ignore any online documentation that includes ‘intersplunk.py’.



# Custom Search Protocol - An Overview

Into the Weeds

splunk>

.conf21

# Overview: The Custom Search Protocol v2

Version 2 of the protocol is used when your custom search command:

- Includes **chunked=true** in **commands.conf**.

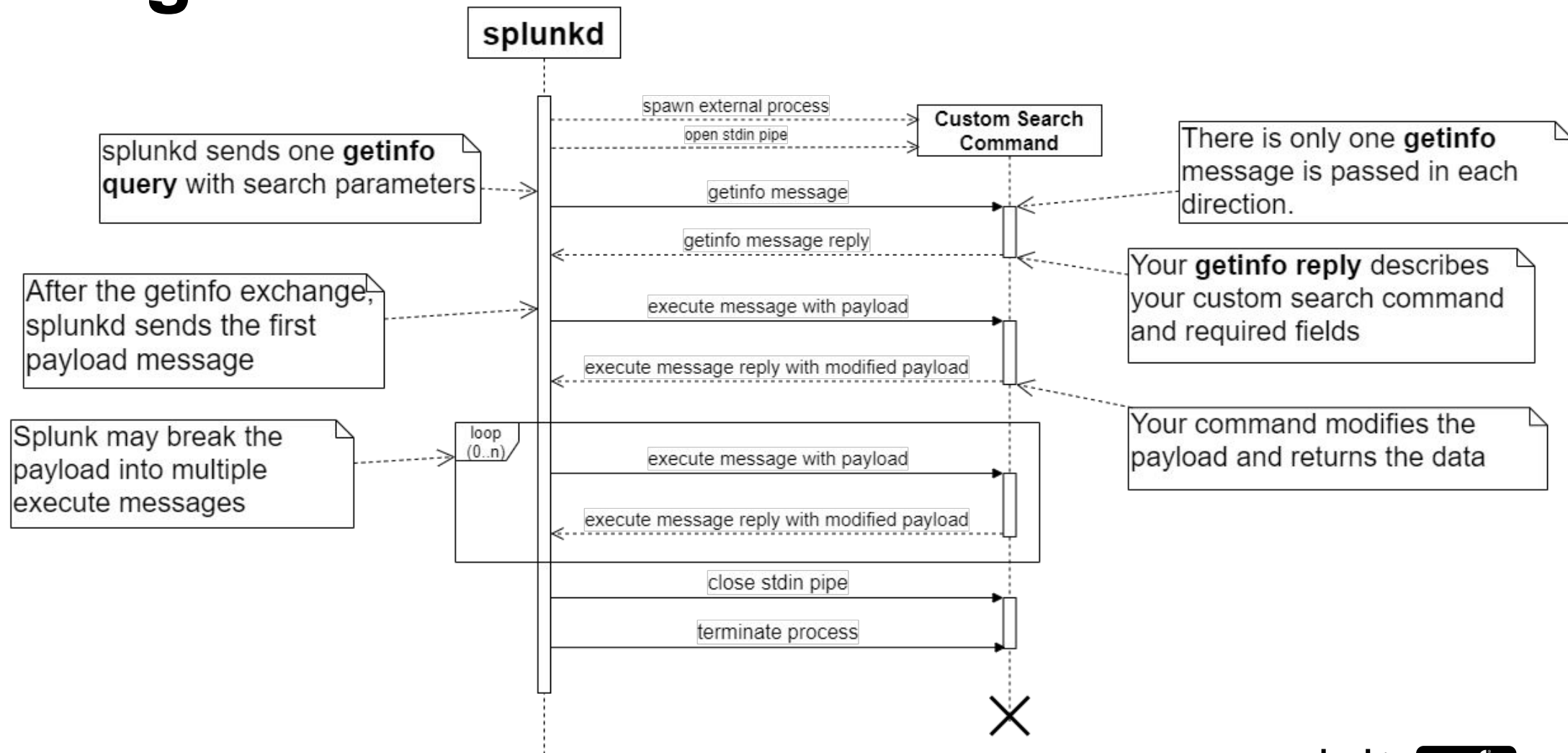
When Splunk sees a valid custom search command in your SPL:

- Splunk spawns an external process to execute the code indicated by your command.
- All communication between Splunk process and your code is via STDIN and STDOUT.
- Messages are in the Custom Search Protocol v2 format:
  - All text is UTF-8.
  - Parameters are Json Encoded.
  - Payload is csv encoded.

There are two message types passed:

- **GETINFO** messages to pass parameters between Splunk and your app.
- **EXECUTE** messages which contain the payload (data in csv format).

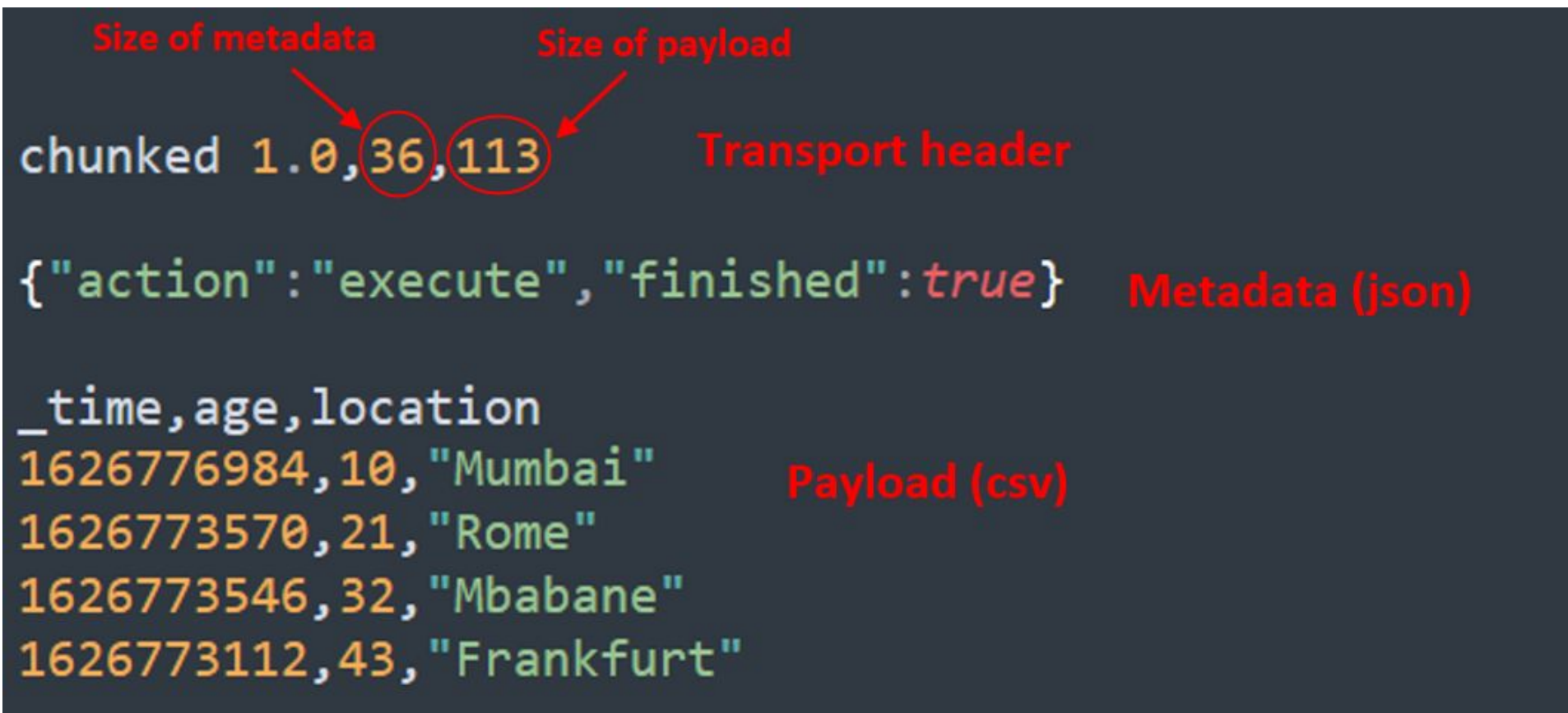
# Message Flow





# Message format

An example 'execute' message



The diagram illustrates the structure of a Splunk message. It shows a sequence of components: a transport header, a JSON metadata block, and a CSV payload block. Annotations with red arrows and circles identify specific parts: 'Size of metadata' points to the number 36 in the transport header; 'Size of payload' points to the number 113 in the transport header; 'Transport header' points to the entire 'chunked 1.0,36,113' line; 'Metadata (json)' points to the JSON object; and 'Payload (csv)' points to the CSV data lines.

```
chunked 1.0,36,113
{"action":"execute","finished":true}
_time,age,location
1626776984,10,"Mumbai"
1626773570,21,"Rome"
1626773546,32,"Mbabane"
1626773112,43,"Frankfurt"
```

# The `getinfo` Message

“Hello? Is anybody out there?”

Splunk first sends a `getinfo` message with all the search parameters.

- The parameters are all json-encoded.

Your program sends a `getinfo` message in response:

- Describes what type of command this is (streaming, reporting, etc.).
- If this command generates events (true or false).
- The field or fields you want Splunk to send you.

There is only one `getinfo` message passed each direction

After the `getinfo` exchange, execute messages are exchanged with the data payload (events) to process

# getinfo Message Examples

What you'll read from STDIN or write to STDOUT

The getinfo message from Splunk contains a lot of information:

```
chunked 1.0,585,0  
{ "action": "getinfo", "preview": false, "streaming_command_will_restart": true, "searchinfo": { "args": [ "count" ], "raw
```

The getinfo reply your program sends back to Splunk (writes to stdout) is simpler:

```
'chunked 1.0,84,0  
{ "type": "streaming", "generating": false, "required_fields": [ "count" ], "finished": false }
```

In our reply, we specify what type of command we are (ex: streaming, not generating)

# getinfo Json Metadata Fields from Splunk

```
1 {
2   "action": "getinfo",
3   "preview": true,
4   "streaming_command_will_restart": true,
5   "searchinfo": {
6     "args": ["count"],
7     "raw_args": ["count"],
8     "dispatch_dir": "C:\\Program Files\\Splunk\\var\\run\\splunk\\dispatch\\1627020566.14",
9     "sid": "1627020566.14",
10    "app": "custom_search_commands",
11    "owner": "admin",
12    "username": "admin",
13    "session_key": "EPkzOp1KhYt3cmYrE3K^ya79qKpmVpvD5foBKXskjbyb^Jj^8WzSoTWbVojP1Le0lqDeDId3VGU",
14    "splunkd_uri": "https://127.0.0.1:8089",
15    "splunk_version": "8.2.1",
16    "search": "%7C%20%20makesresults%20count%3D1%20%7C%20streamstats%20count%20%7C%20streammagic",
17    "command": "streammagic",
18    "maxresultrows": 50000,
19    "earliest_time": "1626933600",
20    "latest_time": "1627020566"
21  }
2 }
```



# getinfo - Metadata Response Options

Mostly these are optional

```
{
  "type": <string>,           // "streaming", "stateful", "events", "reporting"; streaming is the default.
  "generating": <bool>,       // Defaults to 'false'.
  "required_fields": [<string>,...], // Make sure to set this if you can.
  "maxwait": <number>,        // If set: Splunk will terminate this command after <number> seconds
                               // without a response.
  "streaming_preop": <string>, // SPL to run on the data before it is sent to this custom search
                               // command (streaming only).
  "finished": <bool>,         // Only set if Splunk sends you a 'finished' message or you're sending
                               // a terminating error.
  "error": <string>,          // Send a terminal error.
  "inspector": {               // Non-terminal messages to display in Splunk Web.
    "messages": [
      [ <string>, <string> ], ... ]
  }
}
```

# Types of Custom Search Commands

Your getinfo reply `type` field tells Splunk what type of command this is

- Determines what fields Splunk will include in the payload message sent to your command.
- Where your command will be run (distributed to indexers, search head only).
- If your command will modify the order of the events.

type	Global State	Reorder Events	Transforming	Needs all results	Example
streaming	No	No	No	No	eval
stateful	Yes	No	No	No	dedup
events	Yes	Yes	No	Yes	sort
reporting	Yes	Yes	Yes	Yes	stats

# execute Messages

“Execute Order 66”

```
chunked 1.0,36,113
{"action":"execute","finished":true}_time,age,location
1626776984,10,"Mumbai"
1626773570,21,"Rome"
1626773546,32,"Mbabane"
1626773112,43,"Frankfurt"
```

No newline between  
header and payload

```
chunked 1.0,36,84
{"action":"execute","finished":true}count,"_chunked_idx","__mv_count","__mv__chunked_idx"
1,0,,
2,1,,
3,2,,
4,3,,
5,4,,|
```

# One Last Thing on Spawning Processes

Yeah, I don't know why it does this either

Splunk will execute your custom search multiple times

- It'll only perform the getinfo portion of the message exchange before terminating the process.
- The first few times the dispatch\_dir will either be empty or reference a temp directory.
- You must reply to each one of these messages.
- Only one process will be the actual one that sends an execute message with a data payload.
- I think this is so Splunk can determine your command's requirements early on.

Logging recommendation: Make sure you log the PID for each log message





# LOGGING OPTIONS

stardate 99395.55

splunk>

.conf21

# Writing to `stderr`

Or: How to terminate the process poorly

If you write anything to STDERR:

- Splunk will terminate your custom search immediately.
- Splunk Web will only display: *“external search command exited unexpectedly.”*
- Any text written to STDERR will be added to the search.log.

Therefore, you should:

- Write error free code, or
- Make sure you’re wrapping all risky code with try/catch.

There are better options for logging:

- Error gracefully and display a message in Splunk Web.
- Log to the search’s dispatch directory.

# Example: Writing to stderr

Avoid this

```
process.stdin.setEncoding('utf8')
process.stdin.on('readable', () => {
  throw 'Guru Meditation.'
})
```

! Error in 'cyberchef' command: External search command exited unexpectedly.

! The search job has failed due to an error. You may be able view the job in the [Job Inspector](#).

```
[4544 searchOrchestrator] - Running process: "C:\Program Files\Splunk\bin\node.exe" "C:\Program Files\Splunk\etc\apps\conf_testing\bin\domagic_spl.js"
[4544 searchOrchestrator] - EOF while attempting to read transport header read_size=0
[12328 ChunkedExternProcessorStderrLogger] - stderr: C:\Program Files\Splunk\etc\apps\conf_testing\bin\domagic_spl.js:3
[12328 ChunkedExternProcessorStderrLogger] - stderr:         throw 'Guru Meditation.'
[12328 ChunkedExternProcessorStderrLogger] - stderr:         ^
[12328 ChunkedExternProcessorStderrLogger] - stderr: Guru Meditation.
[4544 searchOrchestrator] - Error in 'domagic' command: External search command exited unexpectedly with non-zero error code 1.
chOrchestrator] - PARSING: | domagic
```

# Gracefully Erroring Out

Send a getinfo message with the 'error' parameter set

You can do this at anytime (not just during initial getinfo exchange)

```
chunked 1.0,60,0  
{"finished":true,"error":"I'm sorry Dave, I can't do that."}
```

! Error in 'streammagic' command: I'm sorry Dave, I can't do that.

! The search job has failed due to an error. You may be able view the job in the [Job Inspector](#).



# Build Error Handling Into a Function

A javascript example

```
const halt_on_error = function (msg){  
  
  // Remove all newlines, backslashes, and double-quotes from 'msg'  
  msg = msg.replace(/(\r\n|\n|\r|\u00d0a)/gm, " ")  
  msg = msg.replace(/\\/g, "\\");  
  msg = msg.replace(/"/gm, " ")  
  
  const metadata = '{"finished":true,"error":"' + msg + '"}'  
  const transportHeader = 'chunked 1.0,' + metadata.length + ",0\n"  
  process.stdout.write(transportHeader + metadata)  
  
}
```

# Displaying Non-Terminating Warnings

You can display non-terminal warnings and errors to the user in Splunk Web

- Include in the initial getinfo message.
- Good for warning the user that they passed bad parameters or you can't connect to an optional external datasource.
- INFO level will log to 'info.csv' in your dispatch\_dir.

```
chunked 1.0,154,0
{"inspector":
  {
    "messages":
      [
        ["WARN", "OH DRAT"],
        ["ERROR", "Double-drat"]
      ]
  }
  "type": "streaming",
  "generating": false,
  "required_fields": ["count"],
  "finished": false
}
```

 The following error(s) and caution(s) occurred while the search ran. Therefore, search results might be incomplete. [Hide errors.](#)

- OH DRAT
- Double-drat

# Logging to the Dispatch Directory

My recommendation, especially during development of your command

This is highly recommended

- Note: you won't have the dispatch directory the first few times your program is called, so test for it.
- Should be similar to: `SPLUNK_HOME/var/run/splunk/dispatch/1627113595.30`.

During testing: Write your log to a set location

- You can hard-code your script to write to `/tmp/debug.log` or something similar for testing.

I've not had any luck appending to `search.log`

If you write to a local file (without a file path), it will write in your 'bin' directory

- Don't do this.

## Pitfalls, gotchas, and other weirdness

## The nitty-gritty details





# Parsing the Inbound Messages

What I figured out the hard way

There is no newline at the end of the json metadata

- This is true for all getinfo and action messages.
- You need to know how to read STDIN buffer directly (getinfo messages won't be newline-terminated).
- Most readline() type functions only know there is data on stdin when there is a newline terminating the text.
- My javascript solution is to use `process.stdin.on('readable', ...)`:

```
process.stdin.setEncoding('utf8')
process.stdin.on('readable', () => {
  data = process.stdin.read()
  ...
})
```



# More Parsing the Inbound Messages

What I continued to figure out the hard way

Expect to receive incomplete messages from stdin

- You will read stdin faster than Splunk writes stdout, leading to partial messages.
- That's why the payload length value from the message header is important to validate.
- If you read a partial message: Save that data and read stdin again.

Save the data from the getinfo message, as it isn't included in the execute messages

- start and end time
- args
- dispatch\_dir
- splunk\_version
- etc..

# Requesting Specific Fields

Only for streaming, stateful, and reporting commands

By default, Splunk will send you all fields. This is wasteful and slow

Parse the initial getinfo message to determine what fields are required

- (if needed).
- Send field name(s) back to Splunk in your getinfo reply as **required\_fields** (an array of strings).
- Only the requested fields will be included in the payload of the execute message from Splunk
- This makes the messages much smaller versus receiving all the fields.

Two examples:

```
{"type": "streaming", "required_fields": ["age"] }
```

```
{"type": "streaming", "required_fields": ["data", "sum"] }
```

# Parsing the Args

This can either be really hard, or you don't have to do it at all

The **args** and **raw\_args** are the parameters passed to your custom search command in Splunk Web

- Splunk parses these for you, and presents an array of strings.
- Splunk does no validation on the args, so it's up to you.
- Splunk breaks these down without any context, so you need to validate what it sends you.

If you're doing anything complex in your args, use your own BNF Grammar with a parser

- This is an advanced topic, but it's what I used for the CyberChef Add-on.
- I used the [Nearley](https://omrelli.ug/nearley-playground/) parser, along with the Nearly playground: <https://omrelli.ug/nearley-playground/>.

# Example data from the args

“Arrrrrghs”

For the SPL: ... | **domagic** a=one b = "two" "c"="three"

- The spaces surrounding the equal sign make Splunk parse these three parameters differently.

```
"args": ["a=one", "b", "=", "two", "c=three"],
"raw_args": ["a=one", "b", "=", "\"two\"", "\"c\"=\"three\""]
```

For the SPL: ... | **domagic** r="A \"b\" C" 's'="q\\r"

- Splunk escapes the parameters in the raw\_args.

```
"args": ["r=A \"b\" C", "'s'=q\\r"],
"raw_args": ["r=\"A \\\"b\\\" C\"", "'s'=\"q\\\\r\""]
```

- My solution: Join the raw\_args into a string, and parse it with a parser.

# searchbnf.conf

Not as useful as you would want it to be

Only provides information about your custom search command in Splunk Web

Does not parse the SPL parameters for correctness

- searchbnf.conf doesn't actually provide any limitations to what the user types.
- Incorrectly provides data to the user.

Search SPLUNK\_HOME for **searchbnf.conf** files, there are lots of great examples

The Search Assistant in Splunk Web only shows the first three examples from your searchbnf.conf file



# searchbnf.conf Limitations

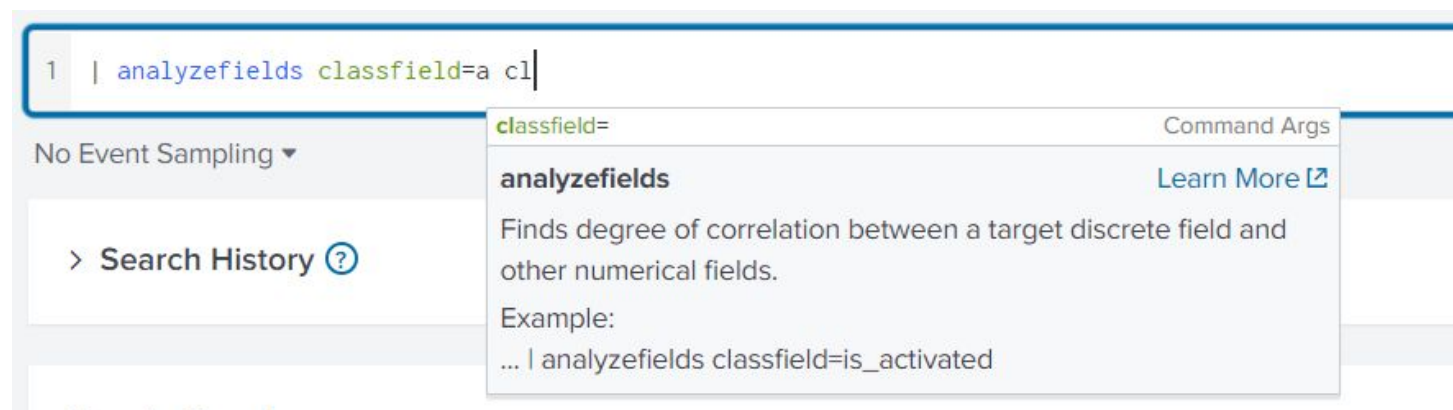
A simple example why searchbnf is not a validator

From `SPLUNK_HOME/etc/system/default/searchbnf.conf`: let's look at the **analyzefields** command

The syntax for this command specifies only one parameter: `classfield`:

- It is required, and it can only be used once.
- However: When you add the 'classfield' field multiple times in the SPL, the search assistant 'helpfully' highlights it for you.

```
#####  
# analyzefields  
#####  
[analyzefields-command]  
syntax = analyzefields classfield=<field>
```



# Quirks with Newlines

Newlines are platform-specific, but this shouldn't matter much

For the newlines between your header and the payload, and between the rows of the CSV payload (not the newlines that may be part of the fields in the payload):

Windows:

- On windows, all newlines in the header and Payload are `\n` (LF, UTF-8 character code 0x0a).

Linux:

- Newlines are `\r\n` (CRLF, UTF-8 character code 0x0d0a).
- Note that this is a single character (not two characters when the payload size is calculated).

# Choosing the Correct CSV Format

When parsing and writing the csv payload:

- The delimiter separating fields is a comma.
- The newline separating records is `\r\n` (CRLF) on linux, `\n` on windows.
- Any field that has double-quotes: Newlines (platform specific), or a comma should be double-quoted.
- Any field without these items should not be double-quoted.
- To include double-quotes inside your field, add a second double-quote to escape it.

CSV Record Field	Decoded Text	Notes
bob smith	bob smith	The field is not quoted because there are no special characters in the field (double-quote, newline, or comma).
"Bond, James Bond"	Bond, James Bond	The field is double-quoted because of the comma.
"I ""Like"" candy"	I "Like" candy	The field is quoted, and the quotes around "Like" are doubled.
"go...\naway"	go... away	The Field is quoted because of the newline (Windows System).

# Assorted Things I learned

That will make your life easier

Modifications to your program code do not require you to restart Splunk

Your custom search command **MUST** be all lowercase, and not contain an underscore

Your command is unknown if you get the error in Splunk Web:

- SEARCHFACTORY:UNKNOWN\_OP\_\_<<yourCommand>>".
- Fix: Did you export your command outside your app?
- Fix: In your commands.conf, is your command listed correctly?
- Fix: did you reboot Splunk after modifying commands.conf?

If your command uses Splunk's included node.js:

- Splunk comes with Node v8.17.0.

# Large Datasets

“Can'nae take any more, Captain!”

If your command might receive large amounts of data:

- Make sure you're only requesting the necessary fields.
- As you read the payload: write it to disk (the `dispatch_dir`) so you don't run out of space.
- Then process the data from disk one portion at a time.
- For my system, sending 10,000,000 simple events in a streaming command resulted in a *JavaScript heap out of memory* error.

In your `getinfo` reply, you specify the **type** as "events":

- This command requires the entire dataset before processing.
- Splunk will send the entire record (all fields) for each event in the dataset.
- At a minimum: The payload will be five times larger than with other commands for the same events.



# Wrapping-Up



# Key Takeaways

It's not easy... but it's not impossible

Use Python with the SDK if possible.

Use a programming language with:

- Strong UTF-8, Json, and CSV support, and which can read the stdin buffer directly (without a newline).

Get logging working first

- Start by logging all inbound and outbound messages to a log file.

Gracefully handle all possible errors

Use a BNF Grammar to parse the Args if you're doing anything complex

- Otherwise you'll have a mess of regular expressions.

# Additional Resources

GitHub Repository containing all the information I have:

- <https://github.com/NDietrich/Splunk-CustomSearchProtocol-v2>.
- Example code.
- Example messages passed for the different command types.
- Splunk Add-On to show each type of Custom Search Command.
- A copy of Jacob Leverich's .conf 2018 presentation slides.

## Splunk Resources:

- Custom Search Commands: <https://dev.splunk.com/enterprise/docs/devtools/customsearchcommands/>
- Splunk Python SDK: <https://github.com/splunk/splunk-sdk-python>.
- Commands.conf: <https://docs.splunk.com/Documentation/Splunk/latest/Admin/Commandsconf>.



# Thank You

Please provide feedback via the  
**SESSION SURVEY**

