
jsluice



`jsluice` is a Go package and command-line tool for extracting URLs, paths, secrets, and other interesting data from JavaScript source code.

If you want to do those things right away: look at the command-line tool.

If you want to integrate `jsluice`'s capabilities with your own project: look at the examples, and read the package documentation.

Install

To install the command-line tool, run:

```
1 ►  
2 go install github.com/BishopFox/jsluice/cmd/jsluice@latest
```

To add the package to your project, run:

```
1 ►  
2 go get github.com/BishopFox/jsluice
```

Extracting URLs

Rather than using regular expressions alone, `jsluice` uses `go-tree-sitter` to look for places that URLs are known to be used, such as being assigned to `document.location`, passed to `window.open()`, or passed to `fetch()` etc.

A simple example program is provided here:

```
1 analyzer := jsluice.NewAnalyzer([]byte(`  
2     const login = (redirect) => {  
3         document.location = "/login?redirect=" + redirect + "&method=  
4             oauth"  
5     })  
6  
7 for _, url := range analyzer.GetURLs() {  
8     j, err := json.MarshalIndent(url, "", "  ")  
9     if err != nil {  
10         continue  
11     }  
12
```

```
13     fmt.Printf("%s\n", j)
14 }
```

Running the example:

```
1 ▶
2 go run examples/basic/main.go
3 {
4     "url": "/login?redirect=EXPR\u0026method=oauth",
5     "queryParams": [
6         "method",
7         "redirect"
8     ],
9     "bodyParams": [],
10    "method": "GET",
11    "type": "locationAssignment",
12    "source": "document.location = \"/login?redirect=\" + redirect + \"\
        \u0026method=oauth\""
13 }
```

Note that the value of the `redirect` query string parameter is `EXPR`. Code like this is common in JavaScript:

```
1 document.location = "/login?redirect=" + redirect + "&method=oauth"
```

`jsluice` understands string concatenation, and replaces any expressions it cannot know the value of with `EXPR`. Although not a foolproof solution, this approach results in a valid URL or path more often than not, and means that it's possible to discover things that aren't easily found using other approaches. In this case, a naive regular expression may well miss the `method` query string parameter:

```
1 ▶
2 JS='document.location = "/login?redirect=" + redirect + "&method=oauth"
   "'▶
3 echo $JS | grep -oE 'document\.location = "[^"]+"'
4 document.location = "/login?redirect="
```

Custom URL Matchers

`jsluice` comes with some built-in URL matchers for common scenarios, but you can add more with the `AddURLMatcher` function:

```
1 analyzer := jsluice.NewAnalyzer([]byte(`
2     var fn = () => {
3         var meta = {
4             contact: "mailto:contact@example.com",
5             home: "https://example.com"
```

```

6         }
7         return meta
8     }
9 `))
10
11 analyzer.AddURLMatcher(
12     // The first value in the jsluice.URLMatcher struct is the type of
13     // node to look for.
14     // It can be one of "string", "assignment_expression", or "
15     // call_expression"
16     jsluice.URLMatcher{"string", func(n *jsluice.Node) *jsluice.URL {
17         val := n.DecodedString()
18         if !strings.HasPrefix(val, "mailto:") {
19             return nil
20         }
21         return &jsluice.URL{
22             URL: val,
23             Type: "mailto",
24         },
25     }),
26 )
27 for _, match := range analyzer.GetURLs() {
28     fmt.Println(match.URL)
29 }

```

There's a copy of this example here. You can run it like this:

```

1 ▶
2 go run examples/urlmatcher/main.go
3 mailto:contact@example.com
4 https://example.com

```

`jsluice` doesn't match `mailto:` URIs by default, it was found by the custom `URLMatcher`.

Extracting Secrets

As well as URLs, `jsluice` can extract secrets. As with URL extraction, custom matchers can be supplied to supplement the default matchers. There's a short example program here that does just that:

```

1 analyzer := jsluice.NewAnalyzer([]byte(`
2     var config = {
3         apiKey: "AUTH_1a2b3c4d5e6f",
4         apiURL: "https://api.example.com/v2/"
5     }
6 `))

```

```

7
8 analyzer.AddSecretMatcher(
9     // The first value in the jsluice.SecretMatcher struct is a
10    // tree-sitter query to run on the JavaScript source.
11    jsluice.SecretMatcher{"(pair) @match", func(n *jsluice.Node) *
        jsluice.Secret {
12        key := n.ChildByFieldName("key").DecodedString()
13        value := n.ChildByFieldName("value").DecodedString()
14
15        if !strings.Contains(key, "api") {
16            return nil
17        }
18
19        if !strings.HasPrefix(value, "AUTH_") {
20            return nil
21        }
22
23        return &jsluice.Secret{
24            Kind: "fakeApi",
25            Data: map[string]string{
26                "key": key,
27                "value": value,
28            },
29            Severity: jsluice.SeverityLow,
30            Context: n.Parent().AsMap(),
31        }
32    }},
33 )
34
35 for _, match := range analyzer.GetSecrets() {
36     j, err := json.MarshalIndent(match, "", " ")
37     if err != nil {
38         continue
39     }
40
41     fmt.Printf("%s\n", j)
42 }

```

Running the example:

```

1 ►
2 go run examples/secrets/main.go
3 [2023-06-14T13:04:16+0100]
4 {
5     "kind": "fakeApi",
6     "data": {
7         "key": "apiKey",
8         "value": "AUTH_1a2b3c4d5e6f"
9     },
10    "severity": "low",
11    "context": {

```

```
12     "apiKey": "AUTH_1a2b3c4d5e6f",  
13     "apiURL": "https://api.example.com/v2/"  
14   }  
15 }
```

Because we have a syntax tree available for the entire JavaScript source, it was possible to inspect both the `key` and `value`, and also to easily provide the parent object as context for the match.