A Node.js Security Model

Intent

This document proposes a way to introduce a set of features to Node.js in order to achieve a reasonable security model based upon the Object-Capability Model and verification of resources loaded by Node.js. Node.js was not designed to be a sandbox, and the usage of potentially dangerous APIs by code running in a Node.js process remains valid. The main purpose is not to prevent improper usage of APIs but to enable trust to be enforced and auditing to be performed in a reasonable manner for applications deployed to Node.js environments. The security model presented here is not exhaustive; even if all of these proposals are adopted, OS-level techniques for securing Node.js processes (such as handling memory constraints and user privileges) will continue to be necessary. The implementation planning in this document is not meant to be complete, and has expectations of further development after initial implementation described here.

Loading of Untrusted Resources

Problem

When Node.js loads code to be evaluated by `require()`, `eval()`, or other means, Node.js has no policy mechanism to declare the loaded resource as untrusted or trusted. As a result, all code running into a node process has the ability to load and run additional arbitrary code using `eval()` (or its equivalents). All code with file system write access may achieve the same thing by writing to new or existing files which are loaded via `require()`. These capabilities provide a powerful attack vector for circumventing security precautions in Node.js applications.

Dynamic loading of resources is intrinsic to the programming model used by Node.js applications. As such, it is not possible to determine all potential resources used ahead of time to verify the program signature at startup. Instead, resources must be verified as they are loaded.

Dangerous API Access

Problem

Code loaded into Node.js applications may perform operations which are considered dangerous for security, deprecation, or other reasons. The effort required to audit a JavaScript resource partially depends on said resource's access to these dangerous operations. By limiting and tracking a loaded resource's access to these operations, we can remove potential attack vectors from said resource. In this way, we can improve the efficiency and reliability of security audits.

API concerns exist at various levels and need to be approached in different ways. Approaches to different concerns are listed in the sections below and intended to allow investigations of discrete topics when performing security audits.

Code Signing

Enables

- Module Auditing in Isolation
- Verification of Signatory
- Revocation of Trust

Description

Using signatures and OS level integrations with key chains, Node.js can create policies enforcing that well known resources are signed by well known signatories. When checked, code signing can assert that content is unchanged from the time of signing. For any resource loaded into the process, a digital signature could be verified for the contents of the resource. This allows contents to be audited by a signatory and passed to other parties. For any given signature, trust may be revoked due to expiration, security, or other concerns.

Implementation Concerns

When implementing a code signing infrastructure, different responses may be applied when a check fails. It should be configurable whether a failed check results in a warning, thrown error, or immediate process exit.

Due to the differences in OS APIs for runtime based code signing, a variety of metadata needs to be configured by the users who ultimately run an application. This metadata must be stored for verification purposes and includes but is not limited to:

- Public Key/Chain of Key Pair used to sign a Resource
- Location of a Resource
- Signature of a Resource

Implementation Planning

1. Implement a manifest format for packages which records the data necessary to verify the signatures of resources contained within said package.
    a. Likely can copy most work from HTTP Signed Exchange "Signature:" header
2. Allow manifests to be loaded at start of Node.js bootstrapping via an environment variable or command line flag. Something like `node --code-signing ./manifest.json main.js`
    a. Only accept file paths to manifests.
    b. No manifests should be loaded after bootstrapping.

    c. No runtime JS API for this should be exposed for this feature.
    d. Allow Integrity Checks on manifests via an environment variable or command line flag.
3. Perform a verification check inside of CJS Module wrapping, before mutating code and compiling to JS.

Subresource Integrity Checks

Enables

- Cross Module Auditing
- Module Auditing in Isolation

Description

Subresource integrity allows for exact matching of source text content without needing to use a signatory, but cannot be trusted without a secure way to verify the root integrity of an application. This may be sufficient if the means of passing in the root integrity is considered safe, but is not sufficient for all scenarios in which Node.js may be used.

A signatory may be unable to sign their own resource's dependencies, due to these dependencies originating from an external source. Despite this limitation, they may wish to provide a signature which establishes their trust in both their own resource and said resource's dependencies.

A concrete example of this would be the following scenario:

- Alice signs her module A
- Bob signs his module B
- Alice wishes her signature on module A to be valid only for a specific version of B

With naive code signing (without subresource integrity checks), Alice's signature could only depend on the contents of A, without regard for A's dependencies, and Bob would be able to update B in a way that Alice considers untrusted (most likely due to B's update not yet being audited). If A uses privileged APIs and makes calls to B in the data path to these APIs or otherwise exposes these APIs to B, then the trustworthiness of A depends on the trustworthiness of B. Alice will thus wish to audit B, and make her signature of A depend on an audited version of B.

For any resource, Node.js should be able to consume a digest produced by one resource to verify the expectation of loading another resource.

Implementation Concerns

There is currently no clear way to pass subresource integrity checks via JS source text in ESM.

Advances in cryptographic technology may result in particular digest algorithms being deprecated and replaced. As a result, the particular algorithms preferred for subresource integrity checks will likely change over time. A hard cutoff on the acceptance of one algorithm and migration to an alternative algorithm would likely be a strain on the package ecosystem; for this reason, simultaneous support of multiple digests would be advantageous. When signing infrastructure supports the use of both an older and a newer digest, signatories can establish a “sliding window” permitting the continued use of a deprecated digest during a limited upgrade period. This removes the need for a simultaneous, ecosystem-wide effort to deprecate a given digest algorithm. Such a capability is not required in the initial subresource integrity check implementation, but any specified design should be forward compatible with such a feature.

Also, there is the possibility that an integrity check may intentionally wish to match multiple dependency content bodies. This is not required for an initial implementation but should be possible under any design specified.

Implementation Planning

1. We can reuse the same format as "integrity" metadata that browsers use.
2. CJS integrity checks can be implemented via options to `require()`
    a. These checks should be out of band per TC39 and WHATWG designs
        i. Implementing another composable manifest format
    b. These checks must be validated even if the module comes from `require.cache`
    c. This likely should also be an option that can be passed to `require.resolve()`
3. Seek assistance from TC39 on expanding this to work for ESM.
4. Allow integrity checks to be loaded at start of Node.js bootstrapping via an environment variable or command line flag. Something like `node --integrity ./manifest.json main.js`

Removing Dangerous Features

Enables

- Removing need to audit various features at the process level

Description

An application may have no need for various features such as direct access to filesystem bindings. It should be possible for an application to completely remove or disable such features from its Node.js runtime, thus removing the need to audit for attack vectors which depend upon the removed features.

Implementation Concerns

If features are disabled at runtime, there is the possibility of leaving references to privileged APIs or resources in the application. Access to these references should be well defined even if the feature is disabled. Audits will need to be vigilant that any such "dangling" references are properly investigated even if features are disabled if using the runtime API. If revocable proxies are used for permissioned userland APIs, there may be identity discontinuity when non-permissioned internal usage of such APIs are used.

Implementation Planning

Please get involved in @addaleax's PR

Constraining APIs

Enables

- Allows usage of dangerous features within a subset of loaded modules, without requiring an audit of all modules for potential misuse of this feature.

Description

In many cases, more fine-grained control over access to dangerous APIs is necessary than enabling or disabling them process-wide. As mentioned in the introduction, the usage of potentially dangerous APIs is perfectly valid for Node.js applications. An application should be able to make use of these APIs in a controlled way by only making them available to specific resources with well-defined scopes.

Implementation Concerns

All of the concerns from Removing Dangerous Features apply here as well. In addition, since this does not completely disable/remove privileged APIs, audits will still need check API access across modules. Since this is done on a more granular basis than the process level, there should be a mechanism to prevent privilege escalation (such as a package without filesystem access directly loading another module with filesystem access.)

The implementation should allow a package to drop privileges and disable all bindings to privileged resources which it received via a runtime API. This is a package-level equivalent to the process-level privilege dropping discussed in Removing Dangerous Features. The benefits are similar to those mentioned in that section; this could potentially allow audits to analyze smaller portions of packages which make use of potentially dangerous APIs. Such a feature is not required for an initial implementation, but any accepted design should accommodate this possibility. If revocable proxies are used for permissioned userland APIs, there may be identity discontinuity when comparing bindings across module boundaries.

Implementation Planning

1. Implement a manifest format for packages to declare the permissions they require.
2. Allow manifests to be loaded at start of Node.js bootstrapping via environment variable or command line flag. (this could come from the same manifest as Code Signing)
    a. Only accept file paths to manifests.
    b. No manifests should be loaded after bootstrapping.
    c. No runtime JS API for this should be exposed for this feature.

    d. Allow Integrity Checks on manifests via environment variable or command line flag.
3. Introduce compartments around all CJS modules using membranes to limit access to privileged APIs (globals, results from `require()`, etc.)
4. Throw `EPERM` errors if `require()` ever loads a module in a package with escalated privileges compared to the current package.

Freezing Primordials

Enables

- Auditors may investigate packages without needing to review potential taint from monkey patching.

Description

The JS standard library and that of Node.js' core are able to be mutated by code loaded into the process normally. This means that malicious code can potentially gain access to privileged data or bindings by mutating these "primordials". Node.js should be able to prevent any mutation of primordials.

Here is a concrete example of an attack which gains access to potentially privileged data without direct access to the compromised API:

```
const https = require('https');
const createServer = https.createServer;
https.createServer = (...args) => {
  // send TLS keys to insecure location
  return createServer(...args);
};
```

Here is an example which patches the JS standard library rather than Node.js' core:

```
const parse = JSON.parse
JSON.parse = (string) => {
   if (string.includes(JSON.stringify('token'))) {
     // send token JSON to insecure location
   }
   return parse(string);
};
```

Such patching lets libraries provide complex instrumentation and backporting of features without host level support. These capabilities are of appreciable value but need to be constrained in order to provide the protections needed to make auditing tractable.

Implementation Concerns

Lots of companies are patching core in order to achieve various profiling and instrumentation capabilities which would not work with a frozen set of primordials. In addition, polyfills would also be unable to be implemented on frozen primordials. Libraries providing these features tend to be loaded as the first step of running applications. Handling these cases is a topic for future research. There should be a way for trusted resources to be loaded ahead of the main app and given the capability of mutating primordials, after which point primordials will be frozen and the main application will be loaded.

APIs that suffer from the "override mistake" will be a source of slow down. Means of addressing this should be investigated both via fixing the spec, and via vm implementation.

Implementation Planning

1. This is required and must be enabled for Constraining APIs to be feasible.
2. Allow primordials to be frozen at start of Node.js bootstrapping via an environment variable or command line flag. Something like `node --untaintable-builtins main.js`
    a. Future variations could include something like `node --primordial-mutation polyfill.js main.js`

Future Considerations

Dangerous Data Retention

Problem

With the advent of increasing side channels to gather data from shared address spaces (such as the Spectre and Meltdown attacks) there are increasing concerns about storing sensitive data in processes running code. In addition, the ability to convert values into opaque references that can be passed to resources without granting privileges is not new.

Related topics, such as research into Module Keys invested into to mitigate threats that are in the same heap by techniques that turn sensitive data into opaque values. Other techniques to hide data from the heap such as Secure Strings and Cross Origin Read Blocking should be investigated for techniques to keep sensitive data outside of the shared address spaces of the runtime.

User Interface To Security Policy Management

Problem

Sometimes users run programs without any privileges as a means to better ensure that a program only uses the required permissions. By allowing users to run code without permissions and then escalate permissions only when they are verified, users can gather context on why permissions are needed. To provide one example, when running `npm test` it might be unclear why filesystem access is required, but it could be clear that `npm install` does need filesystem access in order to place packages into their installed locations. A means by which users may alter the permissions of a running application should be possible via some sort of responsive prompting mechanism.