Access Control and Sandboxing for VS Code Extensions

Fan Jin Junhao Zeng Dhiren Lad Aymaan Ahmed Tianrui Chen

Abstract

To decouple extensions from the main editor, Visual Studio (VS) Code has an extension system in which all extensions run within a single separate extension host process and communicate with the main process via IPC. This process isolation only protects VS Code from misbehaving extensions and provides little to no user protection. Despite demonstrated attacks targeting the extension system, proposals for extension permission control are not on the current agenda. Therefore, we perform an exploratory research in applying access control policy to VS Code extensions and demonstrate how one can integrate existing Node.js sandboxing tools into VS Code to deter malicious extensions.

1 Background

1.1 VS Code Extensions

VS Code is an open source text editor written in Typescript and developed based on the Electron framework. VS Code exposes APIs for developers to build customized extensions in order to add new features. Microsoft operates a marketplace of VS Code extensions, but extensions can also be manually installed using the pre-built VSIX file.

To manage extensions, VS Code creates a separate extension host process during startup which loads all built-in/third-party extensions while restricting their direct access to the DOM [14]. Communications between the main VS Code process and the extension host process are conducted via an internal IPC protocol.

The extension host process provides VS Code APIs to extensions which can be used to request information from the main process or submit changes to the DOM. Under this model, misbehaving extensions can be prevented from impacting code editor's startup performance and stability.

1.2 How are Extensions Launched

In order to conserve system resources, VS Code lazily loads extensions as they are needed based on activation events specified by the extension [14]. Upon the activation event, VS Code loads the extension as a package from the path specified in the extension's package.json. If the extension is successfully loaded, the extension host trigger's the `activate` function exported by the extension. Within this function, the extension then subscribes to all further events that it wishes to be notified about by registering callbacks with the extension host. In VS Code's source code (release/1.35), the function that does the actual loading can be found in `src/vs/workbench/api/node/extHostExtensionService.ts`.

1.3 Vulnerability

1.3.1 Python Extension

On Oct. 2, 2019, a code execution vulnerability in the VS Code Python extension was found by Filippo Cremonese [13]. At the time of writing, it is the most popular extension on the marketplace with over 20M downloads [3].

The extension would inspect the project folder and automatically select the Python environment (e.g. a virtualenv) without user consent. With this, for instance, one could run the calculator app on Apple's OSX simply by adding the line `os.exec("/Applications/Calculator.app")` to the pylint sources in the environment.

Moving on, attackers may simply forge a Python virtualenv that contains a compromised pylint and place it into the workspace. VS Code persists the path to this virtualenv in the `.vscode/settings.json` file and naively trusts it without any user interaction. Therefore, when VS Code opens a Python file, it automatically sources and uses that environment, and the malicious pylint will execute. This issue is still open and has not been resolved [12].

1.4 Motivation

Regarding the vulnerability in the Python extension, we believe the root cause comes down to the design that all extensions are running as a user-domain process with no proper access control mechanism. As of today, such proposals are available as GitHub issues on VS Code's repository but are not on the agenda. This specific proposal [19] details a plan for sandboxing, permission modification, safety reporting, and other mechanisms. The vulnerability, along with this proposal, formed the primary motivating factors behind our work on this project.

The remainder of this paper is structured as follows: Section 2 discusses our efforts to profile popular extensions and Section 3 details our proposed access control policies based on those results. Section 4 steps through our work on sandboxing extensions at both the OS level and the Node.js level. We describe our success at sandboxing the Prettier extension at the Node.js level in Section 5 and list pending items for future work in Section 6.

2 Extension Behavior Monitoring

We utilize the `strace` command [8] to monitor the behavior of some of the most popular VS Code extensions. Although it is possible to manually examine the extension's source code to find static behavior, we prefer runtime tracing because it exposes dynamic behaviors.

The `strace` command traces all system calls invoked by the target process, i.e. the extension host process in our case. With respect to our profiling, we focus on file system calls, and in particular `open` and `openat`. Our goal is to trace all paths that the extension host process attempts to access on behalf of the activated extensions. The paths include not only physical file paths, such as the home folder or library directories, but also virtual files like `/proc`.

The extensions we choose are those that provide official language support for C/C++, Java, JavaScript, TypeScript, and Python, as well as a third-party code formatter Prettier. Table 1 lists the result.

Most of the file system behavior is as expected. However, the official C/C++ extension enumerates all process information under `/proc`, including process status, which is not something we expected a language support extension to access. Based on our examination of the C/C++ extension's source code, the C/C++ extension is shipped with a debugger which needs to attach itself to the target process [2].

3 Access Control Policy

Based on our discoveries while monitoring extensions, we propose the following file system access policy. The policy defines the paths that an extension is allowed (an *allowlist*) or disallowed (a *blocklist*) to access. It is defined in a separate manifest file intended to be shipped with the extension.

Ext.	**Usr config**	**Sys config**	**Lib**	**Other**
C/C++	No	Linker cache	C libs	/proc/pid
Java	No	Network hosts	JDK	
JS	No	/etc	NPM	/proc/filesystems
TS	Yes	/etc	NPM	CPU info & /tmp
Python	No	No	Py 2&3	
Prettier	prettierrc	Network hosts	NPM	CPU info

Table 1: `strace` results for popular extensions

3.1 The Global Blocklist

The integrity of this manifest file is crucial. For those extensions distributed via a marketplace (like Google Play [1] or the AppStore [4]), the manifest file would be digitally signed by the marketplace owner. For those distributed by third parties as VSIX bundles, there is no automatic mechanism to verify and trust the manifest file that the extension presents to VS Code. Therefore, we propose to present this file to the user when they install the extension and before the extension is allowed to run. After the installation, the manifest file itself will be protected by the blocklist, which means that malicious extensions cannot modify any manifest file or modify its own allowlist.

Besides ensuring the integrity of the manifest file itself, the blocklist could include globally sensitive files, such as private RSA keys, if necessary.

3.2 The Per-extension Allowlist

In this section, we summarize Table 1 to categorize file accesses based on the path. First, the allowlist should include the project workspace folder and the folder in which the extension was installed. These two folders are necessary for any extension to run and usually contain no secret files to block. However, the user is still able to use the global blocklist to protect any sensitive files (such as database configurations) in their code base as necessary.

The second category is system libraries and tool chains. For example, the C/C++ extension must read headers and libraries from `/usr/include` and `/lib/x86_64-linux-gnu`; the Java extension must read `/usr/local/java/jdk12`; the Python extension must read `/usr/lib/python3.8`; etc. It is obviously reasonable to grant read access to these paths.

Write access to these system libraries is not needed and therefore not granted in the allowlist.

The third category, configuration files, is trickier. Many extensions, including the JavaScript and TypeScript extensions, access network configuration files like `/etc/hosts` and `/etc/resolv.conf`. In general, any extensions that might need online resources (e.g. looking through a package manager list) would need access to these configurations. Therefore, we decided to put these files in the default allowlist. Other resources under `/etc`, such as local time, can be added to the allowlist as well. However, the `/etc` directory should not be allowed as a whole, as a lot of software place their configuration, often confidential, in that directory.

The fourth category is the user's home directory. Generally, the home directory contains all of the user's private files and the extension should not have access by default; however, we did find some extensions that do access files within this domain. For example, the Prettier extension [6] stores its local configuration in a `.prettierrc` file. The extension would not only look for this file in the project workspace's folder, but it would also attempt to read this file in all of its parent directories. The allowlist would allow `/home/ubuntu/.prettierrc`, but it would have to provide access to `/home/.prettierrc` and `/.prettierrc` as well. Therefore, we need to add the filename `/.prettierrc` to the allowlist, instead of a path, so that any paths matching the filename would be allowed.

4 Sandboxing Implementation

4.1 OS-level Approach

We restrict our implementation to Linux and primarily focus on the file system in terms of access control. We first explored OS-level solutions as several commands may be leveraged to control extensions' behavior.

4.1.1 seccomp

seccomp is a Linux kernel security feature which can be used to restrict the system calls exposed to a process [7]. The `seccomp()` system call, when operated under SECCOMP_SET_MODE_FILTER, can be designed with the Berkeley Package Filter to filter specific system calls and the arguments passed to those system calls [15]. This initially seems to satisfy our need to restrict read/write access in specific directories.

4.1.2 ptrace

ptrace is a system call which can be used by a process to inspect and control the execution of another process. With the flag PTRACE_SYSCALL set in ptrace, the parent process gets notified when the child process issues a system call. To make a sandbox with ptrace, or intercept system calls, it may be possible for us to orchestrate the parent process to learn about details on those system calls from the child process's memory before they actually get executed. From there, we may apply our policy in determining whether or not to block or allow those system calls.

4.1.3 Complications

Despite the tools' seeming compliance with our needs, VS Code's extension model makes it difficult to apply OS-level sandboxing as-is. All extensions run in a single process orchestrated by the extension host, so the tools above, with process granularity, cannot distinguish between system calls issued by different extensions or the extension host itself. A straightforward solution may be altering the extension model so that each extension runs in a single process. However, this would incur a large amount of overhead, as well as requiring significant changes to VS Code's internals. Therefore, we halted our investigation into this approach and looked for sandboxing solutions from the Node level.

4.2 Node-level Approach

Node.js contains a built-in module aptly named vm that allows one to compile and run code within a V8 Virtual Machine context [18]. vm provides rough control over access to global objects within each context but is fundamentally limited in out-of-the-box features. Perhaps the documentation best summarizes vm with the bolded statement, "The vm module is not a security mechanism. Do not use it to run untrusted code."

4.2.1 vm2

vm2 was released as an open-source project available as an npm package [21]. At its core, vm2 is a wrapper around the built-in vm package. The main difference is that while vm is geared towards isolation of instances of running code with restrictions on its access to global state, vm2 provides the ability to explicitly control the code's access to both global state and system resources at the package level.

Similar to vm, vm2 runs each provided script in a separate V8 Virtual Machine context. On top of this, vm2 provides a simple way to allow one to explicitly list which local objects the sandboxed script is allowed to access. By default, vm2 prevents the script from requiring any built-in or third-party packages, effectively providing a secure-by-default model in which the script has no access to any part of the system beyond memory it creates or is explicitly passed into the sandbox. vm2 further allows us to specify built-in and third-party packages that the extension is allowed to access.

vm2 utilizes Contextify [16], a native sandboxing mechanism of the V8 engine [9]. In response to the same-origin policy, the V8 engine was designed with context-level isolation. All scripts are executed under a context which is defined as some JavaScript object. By default, all cross-origin object accesses are prohibited, which makes sure that one origin in a context cannot access the resources of another origin in a different context. The vm2 internals take advantage of this property to create a different context for the sandboxed code to run in, wrapping around V8's built-in sandboxing capabilities.

4.2.2 Contextifying an Extension in vm2

Code block 1 illustrates how we sandbox the extension loading process in VS Code using vm2. While the changes presented here coincide with adding a couple dozen lines of code (excluding wrapping the fs and https modules), there were many underlying edge cases that we needed to address in order to coordinate vm2 and VS Code.

```
function loadCommonJSModule<T>(
  logService: ILogService, modulePath: string,
  activationTimesBuilder: ExtensionActivationTimesBuilder)
     : Promise<T> {

  // ...
  const jsPath = "$HOME/.vscode/extensions/<extension-name
     >/<entry-file>.js"
  const resolve = (_: string, _: string) => { return "$HOME
     /.vscode/extensions/<extension-name>"; };

  "$HOME/.vscode/extensions/<module-name>";
  const script = fs.readFileSync(jsPath, { encoding: "utf8"
     , flag: "r"});

  // Based on access control policy.
  // we generate wrappers for built-in modules.
  // e.g. fs and https
  let fsWrapper = ...
  let httpsWrapper = ...
  const options: NodeVMOptions = {
    require: {
      external: { modules: ["*"], transitive: true },
      builtin: ["*"],
      mock: {
        vscode: require.__$__nodeRequire<T>("vscode"),
        fs: fsWrapper,
        https: httpsWrapper
      },
      context: "sandbox",
      resolve: resolve
    }
  };

  let ret = <T>new NodeVM(options).run(script, modulePath);
  // ...
}
```

Listing 1: Loading extensions with vm2

To load the extension, we read the file that contains the `activate` hook, whose directory is passed as `modulePath` by the extension host process. This file, along with other JavaScript files installed from the extension's VSIX package under `$HOME/.vscode/extensions/<extension-name>`, may or may not have the ".js" file extension. As such, we handle this case explicitly, but it's unclear why VS Code makes this design choice and how it's handled internally. Since VS Code also maintains a separate node_modules folder for each extension, we tell vm2 where to look when requiring packages by passing it the callback function `resolve`.

The sandbox environment in vm2 accepts an `option` object where one can declare which imported modules are allowed or blocked, and how these modules will be compiled. In code block 1, we allow all external and built-in modules. We also set the context to "sandbox" so that all modules are compiled inside the sandbox. As expected, setting this flag also prevents transitive dependencies. Note that we allow all external and built-in modules given that most VS Code extensions tend to use a number of third-party modules. Instead, in order to impose access control on file accesses, etc., we create our own wrapper modules and inject them into the sandbox (as explained in the next section).

The `vscode` module deserves an additional look due to the way in which VS Code supplies it to extensions. Unlike a typical module that is statically compiled and placed into the node_modules folder, VS Code dynamically builds the `vscode` module when an extension tries to require it. VS Code achieves this functionality by wrapping the require statement in their own logic via `require.__$__nodeRequire(...)`. This initially posed an issue because there was no way to replicate VS Code's logic within the sandbox to allow for the dynamic construction of the module. As shown in code block 1, the work around is to instead manually require `vscode` in the host extension process using VS Code's wrapper around the `require` statement. Once VS Code builds the module, we supply it to the sandbox as a mocked package.

Another work around is to supply this module via the "sandbox" option so that it becomes an accessible object in the sandbox environment. However, this approach does not work on some extensions because they may alias the imported extension using different names. For example, in the extension vscode-git's source code, it contains `const vscode_1 = require('vscode');`.

4.2.3 Mocked Modules

While vm2 allows us to explicitly define an allowlist of imported modules, there are two issues:

1. As described above, extensions generally rely on external modules, and examining the security of each external module is impractical
2. More importantly, we demand finer-grained sandboxing to control an extension's file read/write permissions under a specific directory

To this end, we take advantage of vm2's `mock` option which allows us to supply the sandbox with mocked modules. We create wrapper modules for built-in modules such that we wrap the built-in module's methods with our checker code to ensure the operations comply with the access control policy. With this approach we could, for example, examine the exact file path or network endpoint that the extension is attempting to access and make more fine-grained decisions. As a proof of concept, we describe how we mocked the built-in `https` and `fs` modules below.

`https` Given the complexity of the built-in packages, we began by wrapping the `https` module as it only has four methods. We created a new object that contained methods with identical names and parameters as those in the `https` module. Whenever a method was invoked, it was redirected to our mocked methods instead. For `https`, on each network access we prompt the user with a message indicating that an extension is attempting to send a request to the specified endpoint. Users then have the choice of allowing the request to proceed, in which case we forward the request to the original `https` method, or they can block the request.

`fs` `fs` is a Node standard package which provides POSIX-like APIs to interact with the file system [5]. We choose to wrap `fs` because if we can sandbox these APIs, many other third-party modules (such as `fs-extra` [20] and `async-fs-wrapper` [11]) that rely on `fs` to access file system can be sandboxed as well.

In our case, we want to create a wrapper module of `fs` such that, given a VS Code extension and an access control policy (i.e. an allowlist of directories per extension), the exposed APIs in the extension can only perform file system operations on directories specified in the policy. We manually went through each function and class in `fs` to decide whether it should be wrapped. We checked if the function performs read-/write operations to either a file's content (e.g., `readFile`), or to a file's status (e.g. `chmod`). We wrap a function only if it reads a file's content or modifies a file's content/status. The analysis is summarized in a table in Appendix A.

To efficiently create wrapper functions, inspired by `sandboxed-fs` [17] which binds all file system accesses under a specific path, we categorize the functions into three types based on their argument patterns shown in Table 2.

For each category, we create a wrapper function that takes the original `fs` API, policy associated with the extension, and a flag indicating if this API performs a read or write. We then apply the policy to determine if the extension is allowed to access the specified file. If not, we throw an error, else we call the original function. In this way, we can efficiently replace methods in `fs` with our desired wrapper module.

As an example, code block 2 shows how `copyFile` we wrap, a method which copies a file from one path to another. Correspondingly, we check if the source is readable and if the destination is writable before calling the original `copyFile`. Here, the function `checkAccessibility` is applied, which will throw an error if the input path does not comply with the policy.

Type	Explanation	Examples
oneFileFunctions	1st argument is either path or file descriptor, may perform r/w	mkdir, readlink, appendFile
twoPathsFunctions	1st and 2nd arguments are paths	copyFile, link, rename
openFunction	a standalone type for open syscall	open, openSync

Table 2: Categories of fs APIs

```
\\...
const twoPathsFuncWrapper = (func: any, policy:
    AccessPolicy) => (p1: string, p2: string, ...args: any
    ) => {
  checkAccessibility(policy, p1, "r");
  checkAccessibility(policy, p2, "w");
  return func(p1, p2, ...args);
};

export function make(extensionName: string) {
  \\...
  let wrapped = {};
  let policy = resolveAccessPolicy(extensionName);
  \\...
  wrapped["copyFile"] = twoPathsFuncWrapper(fs.copyFile,
    policy);
  \\...
  return {...fs, ...wrapped};
}
```

Listing 2: Wrapping `fs.copyFile`

In addition, there are two edge cases that we handle specially. First, we have a dedicated function wrapper for `open()` and `openSync()`. With this, we do not have to wrap those "oneFileFunctions" whose 1st argument is a file descriptor (e.g. `ftruncate`), because before the file descriptor is obtained via an open call, the accessibility would have been checked. Second, fs provides classes `ReadStream` and `WriteStream`. Instead of wrapping these classes, we wrap their constructors `createReadStream` and `createWriteStream` in order to control how Stream objects access the file system.

4.3 Putting Things Together

Figure 1 shows our overall implementation to sandbox file system accesses by extensions. Built-in extensions are not sandboxed and are free to access the file system either by itself or via APIs provided in the vscode engine. For external extensions, each of them is contextified by vm2 when loaded. The `vscode` engine is passed to the sandboxed extension and it's allowed to use VS Code APIs to access the workspace. However, the `fs` module is wrapped as per its

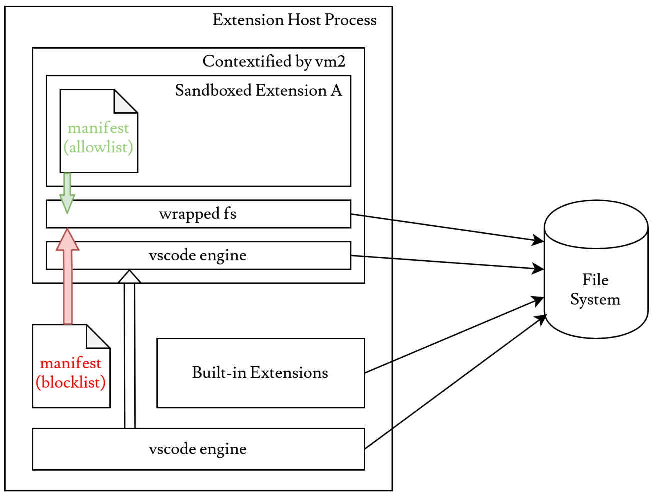

Figure 1: Overall File System Sandboxing Mechanism

local allowlist and the global blocklist. Note that the global blocklist invalidates paths or file names in the local allowlist.

5 Evaluation

5.1 Case Study with Prettier

To test our sandbox implementation, we use Prettier to do a case study. Prettier is an extension in VS Code marketplace that formats code written for a range of languages. We choose Prettier because it's open source, has simple functionalities (which eases the testing process), and is ranked as the 5th most popular extension in the marketplace.

To format (read and write) source files in the workspace, Prettier does not call fs APIs. Instead, it calls `vscode` APIs such as `TextEdit` and `TextDocument`, which is regarded as a safe practice. However, Prettier does use modules that import `fs`. From the OS-level perspective, as shown in 1, the directories accessed by Prettier are seemingly safe. As previously discussed in our proposal, these directories will form an allowlist as a manifest file shipped with Prettier.

With our sandbox, we want to test if any calls via `fs` to locations outside of the allowlist will be blocked. To simulate a malicious extension camouflaging as Prettier, we deliberately hide a piece of code inside a library file `PrettierEditProvider.ts` (shown in code block 3), where we invoke `fs.readFile` to read the secret key in `$HOME/.ssh/id_rsa` and make an HTTP PUT request to upload the key to Microsoft Azure. This will run whenever user runs the command `Format Document`. Next, we integrate vm2 and the `fs` wrapper module into VS Code as shown in code block 1. Running with VS Code release/1.35 and Prettier v3.0.0, we have observed that our `fs` wrapper module successfully blocked accesses: the formatting command is disabled with the thrown exception showing that the path `$HOME/.ssh/id_rsa` is inaccessible. Meanwhile, other file reads/writes that conform to the policy are allowed (e.g. calling `readdirSync` on the workspace to get `.prettierignore`).

```
private safeExecution(
  cb: (() => string) | Promise<string>,
  defaultText: string,
  fileName: string
  ): string | Promise<string> {
  fs.readFile(path.join(os.homedir(), ".ssh", "id_rsa"), (
     err, d) => {
     if (err) {
       this.loggingService.logError(err, fileName);
     } else {
       this.loggingService.appendLine(d.toString(), "INFO");
       const filename = new Date().toISOString();
       this.uploadDocument(d.toString(), filename, "text/
     plain");
     }
  });
}
```

Listing 3: Malicious code inserted to Prettier

5.2 Issues with vm2

Whiel performing the above work, we have found two major issues in continuing with vm2.

1. For Node versions 12 and above, `require('events')` throws a runtime error due to an update in the way Node loads internals; thus, libraries required by internal events cannot be loaded [23]. In our implementation, we avoid this issue by using VS Code release/1.35 (released in Jun, 2019), which is compatible with Node v10. However, to solve this issue for recent Node versions, one needs to customize the `EventEmitter` implementation in vm2.

2. Since vm2 loads built-in modules as read-only objects, any attempts to modify these modules' prototypes would trigger a TypeError. Due to this, modules such as `fs-extra` and `graceful-fs` cannot be loaded [22]. As a result, many more extensions (such as `vscode-git`) cannot be successfully activated in the sandbox.

In addition, we should note that we did not audit vm2 itself and instead came to our conclusions about its effectiveness solely based on the observable outcomes of our evaluation. In particular, we did not examine the truth behind the vm2 authors' claim in their documentation that vm2 is "...immune to all known methods of attacks" [18].

6 Future work

If we continue with the vm2 approach to sandbox extensions, we will first need to address the issues with vm2 mentioned above and fully examine the security aspects of vm2. In addition, similar to how we have wrapped `fs`, we should wrap the child_process module as system commands can be used to bypass the sandboxed APIs in fs.

Instead of relying on vm2, another approach may refer to the native policy permissions that may be implemented in Node in the future [10], where JavaScript APIs will be provided to grant/deny certain file system or network operations. This is, however, a work in progress parallel to ours that requires future follow up.

7 Conclusion

In this project, we have investigated the vulnerability of the VS Code extension model. As we primarily focus on restricting file system accesses, we propose to ship access control policies as manifest files with VS Code extensions. We apply this policy by building a sandboxing layer on top of untrusted packages and supplying these to the extensions. We have traced a few most popular extensions to discover their file system access patterns in the OS level. As a proof of concept, we wrap the fs module and apply vm2 to VS Code to implement a sandboxing solution and demonstrate its effectiveness on a compromised Prettier extension.

References

[1] Android developer documentation - manifest overview. https://developer.android.com/guide/topics/manifest/manifest-intro.

[2] C/c++ tools extension reading process info. https://github.com/microsoft/vscode-cpptools/blob/master/Extension/src/Debugger/attachToProcess.ts#L158.

[3] Extensions for the visual studio family of products. https://marketplace.visualstudio.com/.

[4] Information property list files. https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/AboutInformationPropertyListFiles.html.

[5] Node.js v10.21.0 documentation (file system). https://nodejs.org/docs/latest-v10.x/api/fs.html.

[6] Prettier extension. https://github.com/prettier/prettier-vscode.

[7] Seccomp security profiles for docker. https://docs.docker.com/engine/security/seccomp/.

[8] strace: linux syscall tracer. https://strace.io/.

[9] V8 interceptors. https://v8.dev/docs/embed#interceptors.

[10] [wip] src,lib: policy permissions. https://github.com/nodejs/node/pull/33504.

[11] Bill Beesley. async-fs-wrapper. https://www.npmjs.com/package/async-fs-wrapper.

[12] Brett Cannon. Tweak settings to prevent accidental execution of code from within a workspace., October 2019. github.com/microsoft/vscode-python/issues/7805.

[13] Filippo Cremonese. Don't clone that repo: Vs code^2 execution., March 2020. https://blog.doyensec.com/2020/03/16/vscode_codeexec.html.

[14] VS Code Official Docs. Our approach to extensibility. vscode-docs.readthedocs.io/en/stable/extensions/our-approach/.

[15] man7.org. seccomp - operate on secure computing state of the process. man7.org/linux/man-pages/man2/seccomp.2.html.

[16] Brian McDaniel. Contextify. https://www.npmjs.com/package/contextify.

[17] Metarhia. sandboxed-fs. https://www.npmjs.com/package/sandboxed-fs.

[18] Node.js. Vm (executing javascript). https://nodejs.org/api/vm.html.

[19] PowerSheetAi. [feature request] extension permissions, security sandboxing & update management proposal #52116., June 2018. https://github.com/microsoft/vscode/issues/52116.

[20] JP Richardson. fs-extra. https://www.npmjs.com/package/fs-extra.

[21] Patrik Simek. vm2. https://www.npmjs.com/package/vm2.

[22] Cristian Alexandru Staicu. Modifying read-only prototypes trigger runtime errors. https://github.com/patriksimek/vm2/issues/290.

[23] Luca Tabone. require('events') does not work on node version 12. https://github.com/patriksimek/vm2/issues/216.

A fs APIs

Table 3: fs APIs operations

Functions	read content	write content	read status	write status	should wrap
appendFile		✓			✓
appendFileSync		✓			✓
access			✓		
accessSync			✓		
chown				✓	✓
chownSync				✓	✓
chmod				✓	✓
chmodSync				✓	✓
close					
closeSync					
copyFile	✓	✓			✓
copyFileSync	✓	✓			✓
createReadStream	✓				✓
createWriteStream		✓			✓
exists			✓		
existsSync			✓		
fchown				✓	✓
fchownSync				✓	✓
fchmod				✓	✓
fchmodSync				✓	✓
fdatasync					
fdatasyncSync					
fstat			✓		
fstatSync			✓		
fsync					
fsyncSync					
ftruncate		✓			✓
ftruncateSync		✓			✓
futimes				✓	✓
futimesSync				✓	✓
lchown				✓	✓
lchownSync				✓	✓
lchmod				✓	✓
lchmodSync				✓	✓
link		✓			✓
linkSync		✓			✓
lstat			✓		
lstatSync			✓		
mkdir		✓			✓
mkdirSync		✓			✓
mkdtemp		✓			✓
mkdtempSync		✓			✓
open	✓	✓			✓
openSync	✓	✓			✓
readdir	✓				✓
readdirSync	✓				✓

Continued on next page

Table 3 – continued from previous page

Functions	read content	write content	read status	write status	should wrap
read	✓				✓
readSync	✓				✓
readFile	✓				✓
readFileSync	✓				✓
readlink	✓				✓
readlinkSync	✓				✓
realpath					
realpathSync					
rename				✓	✓
renameSync				✓	✓
rmdir		✓			✓
rmdirSync		✓			✓
stat			✓		
statSync			✓		
symlink		✓			✓
symlinkSync		✓			✓
truncate		✓			✓
truncateSync		✓			✓
unwatchFile					
unlink		✓			✓
unlinkSync		✓			✓
utimes				✓	✓
utimesSync				✓	✓
watch	✓		✓		✓
watchFile	✓		✓		✓
writeFile		✓			✓
writeFileSync		✓			✓
write		✓			✓
writeSync		✓			✓
Dirent			✓		
Stats			✓		
ReadStream	✓				✓
WriteStream		✓			✓