Very Suspicious Code: A Security Analysis of the VSCode Extension Ecosystem

Wiley Corning
University of Wisconsin–Madison

Amanda Xu
University of Wisconsin–Madison

Brianna Cochran
University of Wisconsin–Madison

Abstract

Visual Studio Code uses third-party extensions to provide a wide range of features and customization options to its users. However, its security model for extensions is weak, creating opportunities for attackers and risks for users. As with web browser extensions, VSCode extensions can be used as a vector for malicious code, and even honest extensions may expose vulnerabilities. In this paper we discuss the risks to which users are vulnerable and showcase multiple attacks that can be performed by malicious extensions. These malicious extensions have the ability to download and run arbitrary executables, control other extensions, impersonate trustworthy extensions, and act as spyware. We propose defenses to mitigate these attacks by sandboxing extensions, introducing transparent permissions, validating extensions when they are run, and improving the safety of the Extension Marketplace.

1 Introduction

Visual Studio Code (VSCode) is a popular source-code editor made by Microsoft and initially released in April 2015. As of February 2021, VSCode has 14 million users and is the most popular IDE of choice according to StackOverflow's 2021 Developer Survey [1].

Microsoft's goal is to have VSCode contain a broad set of capabilities and to be able to support developers using any programming language. This is a difficult feat for Microsoft to achieve alone given the ever-expanding space of languages. Introducing an extension ecosystem allows third-party developers to provide tools that serve every niche. To enhance this experience, extensions are built using the Extension API. Extensions cover a wide range of features, such as color themes, extending the workbench, enabling webpage viewing, supporting different programming languages, and debuggers. Anyone can create an extension and the process is well documented to encourage extension creation. However, extensions introduce security concerns since they can contain untrusted code and have access to a variety of user permissions. VSCode relies on a virus scan that is performed when publishing an extension and reputation-based security to protect users from malicious extensions. This loose model allows too much room for malicious extensions and carries immense consequences.

In this work, we address the following research questions to gain a better understanding of the VSCode extension ecosystem:

1. **Local risks:** What capabilities can a malicious VSCode extension access when installed on a victim's machine?
2. **Systemic risks:** What effects can be caused by malicious extension code in the longer term or beyond the original victim's device?
3. **Infiltration vectors:** How might a malicious extension trick an unwitting user into installing it?
4. **Weaknesses:** What vulnerabilities exist in honest extensions that could be exploited by an adversary?
5. **Defenses:** How might the security model of VSCode extensions be enhanced to mitigate these vulnerabilities?

The contributions for this paper include a formalized security model for the VSCode extension ecosystem, several attacks that showcase the weaknesses in VSCode's security model, and proposed defenses to strengthen the model.

2 Related Work

2.1 Security of VSCode Extensions

To date, there has been little published research on the topic of IDE extension security. The most directly relevant work [2] centers on an evaluation of the capabilities available to VSCode extensions. Its author implements a static analysis tool for Typescript to determine which sensitive APIs are included in an extension's AST, and applies this tool to measure the capabilities used across a dataset of popular extensions. Their

results show that popular extensions exercise a wide range of sensitive functions via both the Extension API and general Node libraries, particularly with respect to filesystem access, spawning child processes, and accessing web resources. Drawing inspiration from browser extensions and Android apps, the author proposes a permissions system and emulates it using their static analysis tooling. Due to the wide range of functionality available in npm packages, however, they conclude that properly implementing such a system would be difficult.

In comparison, our work seeks to develop a more comprehensive threat model. We present specific attacks that could be executed by a malicious extension, and also consider - from a human factors perspective - how such an extension could come to be installed. While our work considers the security of honest extensions, we do not primarily focus on measuring their capability usage. Our work also proposes a more specific set of defensive measures.

Other prior work [3] considers the vulnerabilities present in honest VSCode extensions. Certain popular extensions, such as *LaTeX Workshop*, were found to start a local web server that could be subverted by an remote attacker to gain access to the victim's filesystem. This vulnerability could be used by an attacker to exfiltrate confidential files and potentially execute arbitrary code. Our own work incorporates a static analysis of the top 100 installed extensions using SonarQube, but does not include a deep analysis of case studies. The authors of this piece recommend specific practices for hardening a local web server; our work instead focuses on defenses that could limit the damage inflicted by any extension that is compromised.

Another study [4] discusses the security risks of online coding platforms such as Visual Studio Online and AWS Cloud9. While this work is mainly focused on the threats introduced by online coding platforms, it includes a brief discussion of the security of extensions and plugins. The authors mention how VSCode extensions lack extensive security checks when publishing and permission checking during installation. As a demonstration, they create a malicious extension containing a reverse shell functionality, which would allow an attacker to perform full remote code execution with root permissions under a linked VM. Our work provides a more general and thorough analysis of the current VSCode security model. Our analysis does not include Visual Studio Online but presents a wide range of possible attacks.

2.2 Security in Web Browsers and Mobile OS

In contrast to IDE extensions, there is a significant body of literature addressing the security of browser extensions and mobile application platforms. Both of these domains have been exposed to significant adversarial pressure over multiple decades of heavy consumer adoption. As a result, robust threat models and effective security measures have been developed for both settings.

Google Chrome implements several measures for securing its extension ecosystem [5]. Chrome extensions are built within a privilege separation model (*isolated worlds*) that divides scripts that run in a system context from those that run within webpages. Extensions are appropriately sandboxed in memory to prevent them from being manipulated by malicious websites. Each extension is required to declare a manifest of its permissions, which the user must approve at the time the extension is installed. Together, these mechanisms are intended to support the principle of *least privilege*.

A study of 100 extensions [6] found that, while Chrome's model had a significant positive impact on security, 40% of those surveyed contained some form of vulnerability (e.g., using HTTP instead of HTTPS). Other work [7] has considered the forms of attack that could be performed by a malicious extension, assuming that Chrome's sandboxing works as designed; this research identified vulnerabilities that could be used to add the victim's browser to a botnet or sniff passwords. Its authors suggest a system of "micro-privileges" that would make Chrome's permission system more granular.

Today's mobile operating systems implement strict sandboxing and permissioning systems, limiting the exposure of the user's data and device functionality to malicious apps. On the Android platform, the various informal standards and goals that emerged early in development have since been codified into a robust security model [8]. This model is based on a set of core principles: *multi-party consent* is required for sensitive operations; an *open ecosystem* is provided for applications; *security is required for compatibility*; *factory reset* of a device places it in a safe state; and *applications are security principals*, given restricted permissions instead of being treated as full proxies for the user. Android implements multiple layers of protection to ensure that it is *safe by default* and provides *defense in depth*. For example, each app is assigned a separate SELinux user ID, restricting its filesystem access to its own storage area. All access to sensitive functionality is gated through the permissions system.

VSCode is built on the Electron framework, which uses the Chromium browser engine to perform rendering; in both VSCode and Chrome, extensions are primarily written in JavaScript. Many of the security principles underlying Android are also desirable for the IDE extension domain. However, we must also acknowledge that VSCode extensions present a distinct set of challenges. They are diverse in their needs and use cases, ranging from simple color themes to programming language support packs to third-party service integrations. Because they are hosted in a flexible environment with full user permissions, extensions may be developed in a way that is heterogeneous and difficult to sandbox. Such factors complicate the task of building a comprehensive security model, especially one that could be implemented without breaking existing honest extensions.

2.3 Supply Chain Attacks

As package-management tools have become increasingly prevalent, adversaries have sought to use these tools as a vector for injecting attacks into many dependent systems simultaneously. A recent review of supply chain attacks [9] includes many examples from the npm ecosystem, where widely-used packages have been hijacked to steal credentials, run crypto miners [10], or carry out other malicious actions. This risk is directly applicable to VSCode extensions, which are themselves Node packages and typically have npm dependencies. The mechanisms by which supply chain attacks are performed could also be used to hijack legitimate extensions: an attacker could steal the credentials of an application's publisher, or simply gain write access to an open-source extension through social engineering. In this scenario, the attacker could then publish a malicious version of the extension which would be installed to all of its users' devices through an automatic update.

2.4 Developers' Security Awareness

As we shall discuss in Section 3.2.2, VSCode relies on developers to perform their own due diligence when installing extensions. However, prior work has shown that security awareness among developers is not high [11], with typical developers taking security for granted, misestimating their own security knowledge, and neglecting to perform any security testing of their own code. Studies have also shown that developers often naively copy-paste insecure code from Stack Overflow, resulting in real-world vulnerabilities [12]. Such results have poor implications for how the average developer approaches the security of their work environment.

General computer users have been found to ignore even clear spyware warnings when installing software [13] and sophisticated interventions have been necessary to deter risky behavior [14]. This is notable given that VSCode allows extensions to be installed with a single click, a low level of friction that effectively encourages users to install new extensions quickly and indiscriminately. Attacks targeting developer tools are also increasingly common: in one recent case, a malicious XCode project impersonating legitimate open-source software was used to install a backdoor on victims' devices [15].

3 Background and Assumptions

3.1 Threat Model

We consider a generic adversary that may have any of a diverse range of objectives: theft of confidential data, installation of malware/spyware, distribution of malware to downstream software consumers, etc. This adversary is assumed to lack any privileged access to their target's device beyond what they are able to obtain through VSCode. We assume that the adversary may be able to act as a person-in-the-middle, and that they may be able to send HTTP requests to open sockets on the target's device.

Our adversary intends to use VSCode extensions as a way of conducting attacks against their victim. To understand extensions as an attack vector, we consider two distinct categories: *malicious* extensions, and *honest but vulnerable* extensions. A malicious extension is defined as one whose source code is at least partially controlled by the adversary. On the other hand, an honest but vulnerable extension is assumed to be outside of the adversary's control, but to contain some exploitable flaw that is known to the adversary. For malicious extensions, we will consider the means by which the victim could come to install them; we will assume honest but vulnerable extensions have already been installed by the victim for some legitimate purpose.

3.2 Current Security Model

VSCode does not have a formal security model published. We formalize the following model based on information found in VSCode's documentation and our own observations.

3.2.1 Single-party Consent

VSCode extensions rely on single-party consent rather than multi-party consent as in the Android platform security model [8]. Under the multi-party consent model, each individual party (e.g. user, platform, and developer) controls some data but cannot act on it without consent from other parties. The VSCode extension ecosystem exhibits a similar set of parties with the *user*, *VSCode platform*, and *extension developers*. However, currently any party can act on data without consent from other parties. For example, an extension can perform arbitrary filesystem manipulations with neither consent from the user, since an extension can be activated upon VSCode startup, nor consent from the VSCode platform. Although VSCode exposes a set of APIs that should be used to interact with the filesystem, this is not enforced and extension developers can opt to use nodejs equivalents instead [16]. This is different from browser extensions, which are forced to use the browser's APIs for filesystem access.

3.2.2 Extensions and Dependencies Both Trusted

Extensions have a wide range of capabilities including read and write access to the user's filesystem, access to the terminal, and the ability to make web requests. With only those three permissions, an extension can download and run executables, exfiltrate source code from the editor, log keypresses, inject malicious code, and much more. Section 4.1 contains a more in-depth discussion of possible attacks.

One security feature VSCode implements to protect users when opening untrusted files in the editor is called Workspace

Trust. Users are prompted to select whether they trust the authors of the files or not. If not, the workspace will open in Restricted Mode where automatic code execution, such as debugging, is disabled. Notably, this feature does not protect users from malicious extensions. Rather, the goal is to protect users when opening untrusted source files. Extensions have the option of specifying whether they are disabled in Restricted Mode, have limited functionality enabled, or fully enabled. This allows extension authors to decide if their extensions could potentially be dangerous if run on a workspace containing untrusted code. However, this relies on extension authors being honest and therefore does not protect against malicious extensions.

3.2.3 Reputation-based Security

The main tool users have at their disposal is to look at the install counts, ratings/reviews, and publisher for an extension to determine whether it can be trusted or not. Extensions have the option to link a GitHub repository but it is not required. Publishers of extensions have the opportunity to obtain a badge called "Top Publisher" that signifies the publisher has demonstrated commitment to its customers and the Marketplace through excellent policies, quality, reliability, and support. However the Top Publisher program is currently only applicable to publishers of Azure DevOps, which limits the coverage of extensions that are currently available. Users can see which extensions are running but must run a command from the command palette to do so. Even then, there is no transparency to users about what permissions each extension uses. There is no concept of restricting permissions or sandboxing extensions beyond Restricted Mode, which relies on an honest extension author, and the extension host, which only protects VSCode from extensions. Users have the option to report extensions. If an extension is determined to be malicious, it is removed from the Marketplace and uninstalled from all VSCode instances.

3.2.4 Extensions Isolated from VSCode

VSCode protects its stability and performance by isolating extensions and running them in their own process, the extension host. This ensures misbehaving extensions cannot compromise the stability of VSCode itself, particularly during startup. Extensions are also run lazily using activation events. This improves performance by not running extensions until the moment they are needed. However one might imagine a malicious extensions simply defining a wildcard "$\star$" activation event. Lastly, extensions cannot access the DOM of the VSCode UI. The main goal of the extension host is to protect the VSCode application from extensions but it does nothing to protect the user's system. The only assurance is that adversaries cannot perform a denial of service attack.

3.2.5 Marketplace Scans

The Marketplace claims to run a virus scan when extensions are published and updated to filter out extensions containing malware. But much like trusted/secure boot, there is nothing to guarantee the safety of the extension during runtime. For example, the extension source code itself might not contain any blatantly suspicious code but when activated, it could make a request to download a malicious executable and run it on the user's system. The Marketplace is also likely intentionally vague about what the virus scan entails. However all virus scans generally only contain a deny-list of known malware and does not protect users in a robust manner.

There are also scans to protect users browsing Marketplace pages. Firstly, there is a restriction on using SVGs for the icons, badges, and other images. SVGs are only allowed for badges if they are from trusted badge providers. Image URLs in the README.md and CHANGELOG.md must resolve to https URLs. Additionally, Marketplace pages allow embedded HTML but will neither load iframes nor execute JavaScript. Finally, unrelated to security, there is a content scan to prevent inappropriate or offensive content being published on Marketplace pages.

4 Evaluation

We detail specific attacks we implemented and tested that we could indeed publish an extension containing these attacks. See Appendix A for details on how we did so ethically. Next, we present possible infiltration vectors for malicious extensions. Finally, we briefly discuss our analysis of honest extensions.

4.1 Attacks[1]

4.1.1 Download and Run a Malicious Executable

Using node-fetch, it is straightforward for a malicious extension to download an executable or script file onto the victim's machine. While downloading an executable itself is not an immediate concern, running the executable would certainly be one. This can be accomplished most directly by invoking the `child_process` library from the attacker's extension to run the executable in a new process; such a process will then have the same full user permissions as VSCode itself.

By setting the extension's activation event to the wildcard "$\star$", no user interaction is needed for the executable to be run beyond starting the VSCode application. The location of the downloaded executable need not be associated with any VSCode directories and could therefore persist even if the extension or VSCode is completely uninstalled. This is generally true for any files an extension might create or modify.

[1] All source code for attacks available at https://github.com/WileyCorning/UW-CS782-vscode-demo

Furthermore, on Windows devices, a downloaded executable can be set to run when the operating system boots by placing a shortcut in the user's `Startup` folder - an action which requires no administrative privileges.

4.1.2 Exfiltrate Confidential Files

Extensions have full read and write access to the user's filesystem via the `fs` library. With unfettered filesystem access, a malicious extension can read and exfiltrate sensitive data from a victim's machine. For example, the extension can read the user's SSH key files (e.g. `id_rsa`) and send them to a server via a POST request through the `http` library. A malicious extension could even act as a disk-encrypting ransomware program.

4.1.3 Act as Spyware

An extension can act as spyware like a keylogger; in fact we have observed multiple extensions on the marketplace advertised as keyloggers. The only limitation is that modifier keys cannot be captured. Of course a keylogger on a victim's machine is not useful unless the attacker can access the data. This would be a simple task given the extension can make arbitrary requests to a server to exfiltrate the data as discussed above. Using a downloaded executable, the attacker could potentially spy on the victim in other ways such as recording audio from the microphone, video from the webcam, or video from the screen.

4.1.4 Tamper with Another Extension

VSCode stores the extracted source of installed extensions in the `.vscode/extensions/` directory normally located in a user's home directory. Since extensions have unfettered read and write access to the user's filesystem, a malicious extension can easily modify the source for other extensions. For example, imagine a situation with three extensions. There is an honest extension that exposes an API for other extensions to use, an honest extension that consumes that API, and a malicious extension that exposes its own version of the first extension's API and wishes to spoof the consumer into using the malicious version of the API. The malicious extension simply needs to change the consumer's package.json to declare itself as a dependency and modify the areas in the consumer's source code that call the API to use the malicious extension's publisher and name instead. An alternative version of this attack could be performed by directly modifying the first extension's source code; this technique is used in Section 4.1.5.

Neither of these forms of tampering is consistently detected by VSCode, which will run the altered code without checking its authenticity. It does sometimes provide a warning message saying "Extensions have been modified on disk. Please reload the window." However, in our testing, this message was not consistently displayed. On the other hand, VSCode does a background check to detect if the VSCode installation itself, not extensions, has been changed on disk and if so, the title bar displays "[Unsupported]". Any alterations will persist until an update delivers a fresh version of the extension or VSCode. The attacker may also prevent future updates by editing the version number of the first extension to a higher value, although the new version number will then be visible to the user as a possible indicator of tampering.

4.1.5 Inject Malware into Build Artifacts

To demonstrate the systemic risks from malicious extensions, we constructed an attack that permits the adversary to inject arbitrary content into Docker images. The attack targets users of the official VSCode Docker extension developed by Microsoft. It is executed in three phases. First, the user installs the attacker's extension. The attacker's extension immediately uses the VSCode API to find the local install directory of the Docker extension; it then uses the Node filesystem API to rewrite the Docker extension's source code. As published, this code consists of a single minified Javascript file that has been compiled from Typescript. We were able to clone the target extension's repository, insert our own payload into the Typescript source, and compile it into a tampered Javascript bundle; the attack then simply replaces the original file with our own.

In the second phase of the attack, the user is engaged in a typical Docker workflow: constructing a new container image from the contents of some local folder, including a `Dockerfile` that specifies the container's environment and entry point. When the contents of their container are in place, the user invokes the Docker extension's `buildImage` command, which will execute lower-level Docker commands to construct the image and make it visible within VSCode. However, our tampered version of the code injects additional logic into this command: before building the image, it writes a new script file into the folder and rewrites the `Dockerfile` to target this script as its entry point. The file itself performs some arbitrary attack logic and then invokes the original endpoint, keeping its operation stealthy. Once the image is built, these changes to the local filesystem are reverted, leaving the attack baked into the container image but invisible to source control. The final phase of the attack occurs when the image is deployed to a cloud server or to another user's machine, at which point the embedded attack logic will be executed.

This attack has several noteworthy features. Its phases are decoupled: removing the malicious extension will not undo its tampering with the Docker extension, and the attack will persist inside the tampered container images even if the Docker extension is removed or cleaned. While a Docker image can be inspected - for example, by serializing it as a `tar` file and extracting the contents - this requires an active and unusual intervention by the developer. There are many other types

of build artifact that could be hijacked similarly with even more opaque results, such as binary executables or libraries. Our implementation of the attack is executed entirely within the extension system; however, a malicious extension could just as easily tamper with the binary of a compiler to inject a Trojan attack of the kind discussed in [17].

4.2 Infiltration Vectors

4.2.1 Impersonating a Trustworthy Extension

To explore how users might download a malicious extension, we considered an impersonation attack. The VSCode Marketplace provides little protection against real extensions being impersonated, aside from the reputational cues provided by install count and star rating. An extension's appearance on the Marketplace is directly controlled by its `README.md` and `package.json` files; a basic form of impersonation can be performed simply by copying these files from a target extension to the attacker's extension. This attack requires only simple knowledge of the publication process, which is well documented. If the attacker then artificially increases their install count and adds a few positive reviews, it may be difficult for users to determine which listing is the original when viewing search results.

To simulate this attack, we created and published an extension which impersonates an honest third-party color theme named *Hop Light*. (See Appendix A for details on the content of our extension and the duration of this test.) Figures 1a and 1b are screenshots of the real and fake extension's Marketplace pages within VSCode, respectively. The most notable difference between the two is in their *Unique Identifier* field under the *More Info* section: the unique identifier for the fake extension contains a '0' instead of an 'o'. The extensions also have different publication times, and the impersonator contains an additional (likely removable) *Issues* link; otherwise, they are completely identical. A user who is not keen at verifying the extension or has no previous knowledge of the real extension can easily fall prey to this impersonation. After only 7 installs and 2 five-star reviews, our fake extension rose to the second search result in a marketplace search for "Hop Light" (the first being the legitimate extension).

Extensions may also provide a link to an associated git repository and website. However, there are no checks to ensure that the repository accurately represents the extension or that either link is genuinely associated with the publisher. An untrustworthy publisher could link to a repository containing a "clean" version of their extension while packaging added attack logic in the published version. When impersonating a legitimate extension, the attacker could simply link to the real extension's website and repository, as was done in our demonstration. The links are suggestive of proof that an extension is legitimate; however, we have shown that this implication can be misleading.

4.2.2 Hijacking a Trustworthy Extension

By default, VSCode automatically updates each installed extension whenever a new version is published. If an attacker can gain access to the publisher credentials for a widely-used extension, they can therefore deploy a malicious version to all of its users simultaneously. The attacker could obtain publisher credentials through some hacking technique (phishing, keyloggers, reusing a password exposed in a data breach, etc.), or through social engineering. Many extensions are open-source projects maintained by a community of volunteers; an attacker could infiltrate such a community by posing as a good-faith contributor until they are given write access, or they could fork an abandoned project and attract new users to the fork.

4.3 Vulnerabilities in Honest Extensions

Our threat model considers both malicious extensions and honest but vulnerable extensions. Determining the prevalence of the latter would require extensive static and dynamic analysis. Special care would need to be taken since extensions can also download arbitrary files during runtime. We ran SonarQube, an out-of-the-box static analyzer, on the top 100 extensions based on install counts as of December 1, 2021. The scan did not detect any vulnerabilities. SonarQube also detects security hotspots, which are pieces of code that need additional manual review to determine whether they are secure or not. The only security hotspot worth noting is a color theme extension, zhuangtongfa.Material-theme-3.13.6, fetching a resource from a CDN without verifying its integrity.

We also ran some preliminary scans to survey CVEs in honest extensions and their dependencies. We tried using both `npm audit` and Snyk but the data was noisy and we were not able to meaningfully extract any insights. Although our analysis did not find any vulnerabilities in honest extensions, prior work [3] has shown honest extensions that launch web servers can be vulnerable. A more thorough and targeted analysis of popular extensions is slotted as future work.

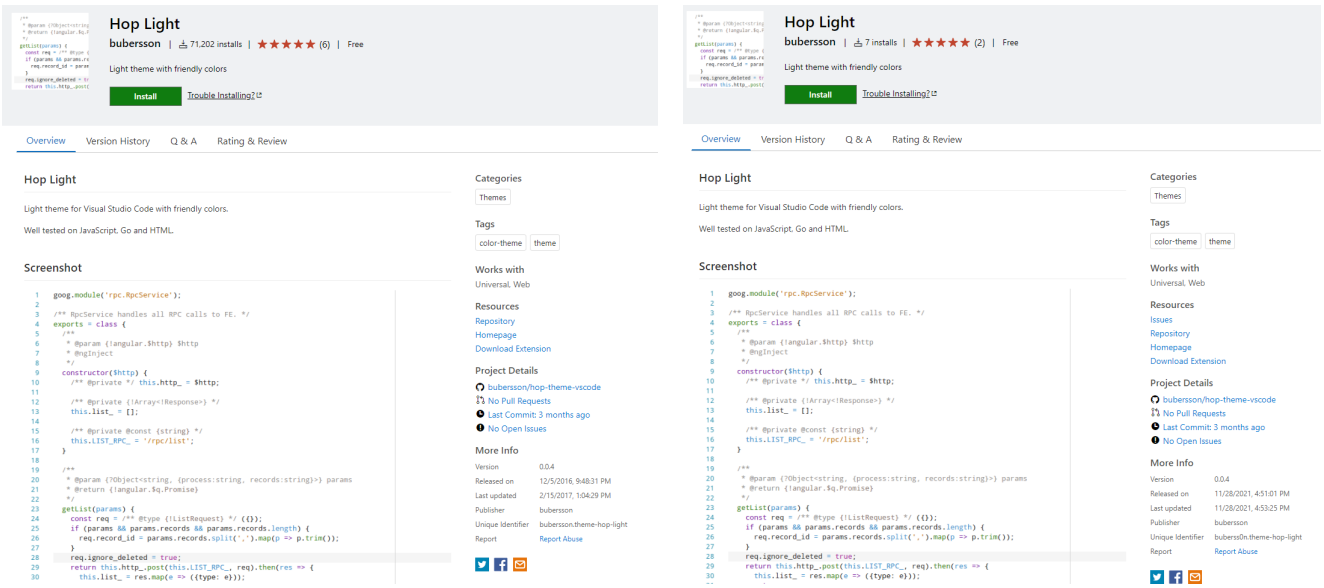
5 Discussion

5.1 Defenses

Given the weaknesses in the existing security model discussed in previous sections, we propose several defenses to mitigate the attacks.

5.1.1 Sandbox Extensions

The extension host process currently runs extension code directly, giving extensions full access to its permissions. We could introduce mechanisms for control by instead running extension code in a sandboxed environment, such as `vm2` [18],



(a) Marketplace view of the real extension named *Hop Light*

(b) Marketplace view of the fake extension impersonating *Hop Light*

Figure 1: Screenshots of the real and impersonating extension

that supports conditional passthrough of system APIs. An untrusted extension could then be prevented from accessing the filesystem, making web requests, spawning child processes, etc. without ahead-of-time authorization by the user. Sandboxing could be further enhanced by wrapping the system APIs in a layer that could apply more sophisticated permission logic or prompt the user for authorization at runtime. As of the writing of this paper, there is an open feature request to add extensions permission management and sandboxing from an individual unaffiliated with Microsoft [19].

Restrict filesystem access. Extensions' access to the filesystem should be restricted to relevant scopes. Taking inspiration from the Android security model, extensions could be granted access only to their own local storage folder and to the files and folders currently open in the VSCode editor. Adding this restriction would completely or partially mitigate many of the attacks presented. An extension could still download an executable to a location in the workspace but it would not be able to modify the user's `Startup` folder on a Windows device to run the executable when the operating system boots. Similarly, an extension would not be able to read sensitive data outside of the current workspace to extract secrets like SSH keys. Finally, an extension would not be able to modify files outside the current workspace and tamper with the source code of other installed extensions, which would completely mitigate that attack.

Restrict child process permissions. Extensions that spawn child processes are problematic from a sandboxing perspective, as the child process would exist outside of the sandbox and could access arbitrary OS functionality. Node's `child_process` API supports setting the `uid` of a child process when it is spawned [20]; this could be used to restrict the child processes spawned by an extension to have no greater capabilities than the extension itself. However, `uids` are not available on Windows devices, which could preclude the effectiveness of this defense. It may be more desirable to prevent extensions from spawning child processes at all; further work is needed to quantify this change would have on existing extensions.

5.1.2 Introduce permissions scheme

There should be a transparent permissions scheme available to users. Users have no way of knowing what permissions an extension has, uses, or needs without inspecting the source code. This makes it difficult to enforce extensions are following least privilege. We propose a permissions scheme that restricts different types of extensions to a set of appropriate permissions. Users would be able to see what kind of permissions an extension has when installing much like in the Android platform and browser extensions.

5.1.3 Extension code integrity

To protect extension source code from tampering on-disk, each extension should have a checksum or signature associated with the source code of the version published and installed. When activating an extension, this checksum should be verified against the source code being run. If this check fails, the extension should fail to activate with a message notifying the user the source code was potentially corrupted. This would protect against a malicious extension modifying

the installed source code for another extension. However, this defense would need to be paired with restricted filesystem access in order to fully protect the user since the integrity of the extension is only verified upon activation. With unfettered filesystem access, a malicious extension could modify the source after the target extension is activated. Another approach would be to encrypt extensions' content on-disk with a locally held key. VSCode might also consider disabling auto-update to mitigate the impact of compromised publisher credentials.

5.1.4 Stricter marketplace constraints

As discussed in Section 4.2.1, the Marketplace lacks sufficient protections against impersonation. Publisher display names should be required to be unique, or, alternatively, the publisher's unique identifier should be visibly displayed alongside their display name. In order to link to a webpage or git repository from their Marketplace page, a publisher should be required to prove their affiliation with it. This process could be as simple as requiring that a specific file in the repository or web domain contains the extension's unique identifier. Additional anti-impersonation measures could flag any new extension whose title and README content substantially resemble those of a more popular existing extension.

If an extension links to its source code in a git repository, additional measures can be taken to verify that the published version of the extension matches the repository contents. VSCode could provide a mechanism for building and publishing extensions on first-party servers, or for distributing extensions as source code (i.e., using `git clone`) and building them on the user's own machine. It could also connect to continuous integration and deployment services, such as those provided by GitHub, to confirm when extensions are published from these pipelines. Extensions that meet these "verified build" criteria could be given an additional security indicator, such as a shield or lock icon, on their Marketplace pages.

5.2 Future Work

The defenses proposed in Section 5.1 have not yet been prototyped, although we have identified specific tools and libraries that could be used to do so. In addition to implementing these measures, subsequent work should analyze the current landscape of extensions to determine how many would be disrupted by introducing more stringent permission controls. It will be necessary to develop more tailored static and dynamic analysis tools to detect which APIs each extension uses, as well as the semantics of these operations (e.g., what parts of the filesystem it touches and what URLs it requests). Such tools could also be applied to detect vulnerabilities in the wild, such as the web server issue discussed in [3], or suspicious behavior, such as a color theme that performs web requests.

As discussed in [4], online coding platforms provide a broader attack surface with different characteristics; more work is needed to understand the architecture of the extension host in VSCode's web-based version and the implications of this architecture for security. VSCode also supports connecting to remote hosts over SSH, which might result in different forms of vulnerability.

Because VSCode does not require any form of authentication to install an extension and thereby increase its installation count, we speculate that the installation count could be artificially inflated using a replay attack. More testing is necessary to confirm whether this attack is possible. In conjunction with this, it will be important to gain a better understanding of how users scrutinize extensions before installing them. Surveys and user studies could bring new insight on where developers find extensions, how deeply they examine them, and how susceptible they are to installing malicious extensions.

Our findings for VSCode have implications for the security of IDE extensions more broadly. Most popular IDEs, including Visual Studio, IntelliJ IDEA, and Eclipse, provide some kind of plugin or extension system. An investigation of these other tools might expose similar vulnerabilities or else provide examples of security measures that could be standardized.

6 Conclusion

We analyze the existing security model of the VSCode extension ecosystem and present several malicious extensions that are allowed under the current model. We show how an extension can impersonate a more popular extension to fool users into downloading the fake version as a potential infiltration vector for a malicious extension. Finally, we propose changes VSCode should consider to mitigate the attacks presented. We discuss the necessary steps to disclose our findings to Microsoft in Appendix B.

6.1 Individual Contributions

Overall, everyone in the group contributed equally to the project. Each member was responsible for implementing and writing about different attacks: Wiley, the Docker injection attack and demonstrations of download-and-run, clipboard snooping, and fingerprinting; Amanda, the API spoofing and discreet terminal attacks; and Bri, the impersonation attack. For the measurement pipeline discussed in Section 4.3, Wiley wrote the code to automatically download extensions and Amanda implemented and ran the scanning procedure. In writing this report, all members contributed throughout the document. Wiley was most involved in the Related Work, Defenses, and Future Work sections; Amanda, the Security Model, Vulnerabilities in Honest Extensions, and Defenses sections; and Bri, the Introduction and Abstract sections.

References

[1] Developer survey. https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-integrated-development-environment, 2021.

[2] Åström David. Implementation and evaluation of an emulated permission system for vs code extensions using abstract syntax trees, 2021.

[3] Raul Onitza-Klugman and Kirill Efimov. Deep dive into Visual Studio Code extension security vulnerabilities. *Snyk*, 2021.

[4] David Fiser. Security Risks in Online Coding Platforms. *Trend Micro*, 2020.

[5] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. 2010.

[6] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 97–111, 2012.

[7] Lei Liu, Xinwen Zhang, Guanhua Yan, Songqing Chen, et al. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[8] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. The android platform security model. *ACM Transactions on Privacy and Security (TOPS)*, 24(3):1–35, 2021.

[9] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2020.

[10] Ax Sharma. Popular npm Project Used by Millions Hijacked in Supply-Chain Attack. *Sonatype*, 2021.

[11] Hala Assal and Sonia Chiasson. Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 281–296, 2018.

[12] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305. IEEE, 2016.

[13] Nathaniel Good, Jens Grossklags, David Thaw, Aaron Perzanowski, Deirdre K Mulligan, and Joseph Konstan. User choices and regret: Understanding users' decision process about consensually acquired spyware. *I/S: A Journal of Law and Policy for the Information Society*, 2(2):283–344, 2006.

[14] Alessandro Acquisti, Idris Adjerid, Rebecca Balebako, Laura Brandimarte, Lorrie Faith Cranor, Saranga Komanduri, Pedro Giovanni Leon, Norman Sadeh, Florian Schaub, Manya Sleeper, et al. Nudges for privacy and security: Understanding and assisting users' choices online. *ACM Computing Surveys (CSUR)*, 50(3):1–41, 2017.

[15] Phil Stokes. New macOS Malware XcodeSpy Targets Xcode Developers with EggShell Backdoor. *SentinelLabs*, 2021.

[16] Vs code api. https://code.visualstudio.com/api/references/vscode-api#workspace, 2021.

[17] Ken Thompson. Reflections on trusting trust. In *ACM Turing award lectures*, page 1983. 2007.

[18] vm2. https://github.com/patriksimek/vm2, 2021.

[19] Dan Moorehead. Extension permissions, security sandboxing & update management proposal. https://github.com/microsoft/vscode/issues/52116, 2018.

[20] Child process. https://nodejs.org/api/child_process.html, 2021.

[21] Report an issue and submission guidelines. https://www.microsoft.com/en-us/msrc/faqs-report-an-issue?rtc=1, 2018.

Appendix

A Ethics

During the course of our investigations, we published two extensions to the public Marketplace. This was necessary to establish that the Marketplace lacked robust measures to catch potential malware and detect impersonation.

The first extension provided a set of commands that would demonstrate the potential capabilities of an adversary: downloading and running an executable file, reading the contents of the user's home directory, tampering with their clipboard contents, etc. Each of these functions would run only in response to the user executing a clearly labeled command, and would not deliberately cause a damaging effect (for example, the downloaded executable simply printed "hello world" in a console window). Furthermore, the extension itself was labeled in the marketplace as a test, and viewers of its store page

were instructed not to install it; we unpublished the extension immediately after uploading it.

Our second extension demonstrated the impersonation issue described in Section 4.2.1. It consisted of a `README.md` and `Package.json` file copied from a third-party extension, which, along with a similar publisher ID, was sufficient to produce a nearly-identical Marketplace page. The extension had no other content and would be inert if installed. Once we established that the impersonator would not be immediately removed from the marketplace, and that it could quickly rise in the search rankings, we removed it as well.

B Disclosure

We include all the necessary information to disclose our findings to Microsoft according to the Microsoft Security Response Center's FAQ [21]. The type of issue is a fundamentally weak security model. The affected product is VSCode and likely all versions that support extensions although our tests were all on version 1.62.3. To the best of our knowledge, we did not have any specific security updates installed. There are no special configuration required to replicate our attacks. We include step-by-step instructions to do so on a fresh install of VSCode along with our source code in our GitHub repository. We discuss the impact of the issue and how an attacker might exploit it in earlier sections. The next step would be to submit an official report containing these details.