Wolf at the Door: Preventing Install-Time Attacks in npm with Latch

Elizabeth Wyss
University of Kansas
Lawrence, KS, USA
ElizabethWyss@ku.edu

Alexander Wittman
University of Kansas
Lawrence, KS, USA
wittmanalex@gmail.com

Drew Davidson
University of Kansas
Lawrence, KS, USA
DrewDavidson@ku.edu

Lorenzo De Carli
Worcester Polytechnic Institute
Worcester, MA, USA
ldecarli@wpi.edu

ABSTRACT

The npm software ecosystem allows developers to easily import code written by others. However, manual vetting of every individual installed component is made difficult in many cases by the number of transitive dependencies brought in by installing popular packages. This has enabled attackers to propagate malicious code by hiding it deep into the dependency chains of popular packages. A particularly dangerous form of attack comes from malicious code embedded into package install scripts.

We tackle the problem of preventing undesirable install-time behavior by proposing Latch, a system for mediating install-time capabilities of npm packages. Latch generates permission manifests summarizing each package's install-time behavior and checks them against user-defined policies to ensure compliance. Policies in Latch are expressed in a rich formal policy language that covers a broad range of use cases. Our key insight is that expressive Latch policies empower users to define and enforce their own individualized security needs.

Evaluation of practical Latch policies on all publicly available npm packages and on a number of real-world attack packages demonstrates that our approach is effective in identifying and stopping unwanted behavior while minimizing disruption due to undesired alerts.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Empirical software validation**.

KEYWORDS

npm, supply chain security, install-time attack, policy language

ACM Reference Format:
Elizabeth Wyss, Alexander Wittman, Drew Davidson, and Lorenzo De Carli. 2022. Wolf at the Door: Preventing Install-Time Attacks in npm with Latch. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3488932.3523262

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9140-5/22/05.
https://doi.org/10.1145/3488932.3523262

1 INTRODUCTION

Many programming language ecosystems benefit from public repositories that allow any developer to upload a package that contains modular functionality for use in other software projects. Although these package repositories are undeniably useful, they can also be a vector for a *software supply chain* attack. In such an attack, the larger system is compromised through the functionality of an imported component, such as a package. Recently publicized software supply chain attacks have resulted in execution of crypto-mining code [51], exfiltration of credentials and other sensitive information [20], and other unwanted outcomes. While there are many ways in which software supply chain issues surface, one of the most dangerous involves exploiting the install-time package setup mechanism. In many repositories, packages are equipped with routines that allow for bootstrapping. Scripts embedded within these bootstrap mechanisms, under most conditions, execute when the package is installed. As a result, an attack can complete even if the actual package is *never run or imported* by the victim. In practice, setup scripts have been used as a vector for a variety of malicious behaviors [20, 29, 49–51], and there have been over one hundred documented attacks built upon this technique [13, 31].

Even when a package's installation scripts are not explicitly designed to do harm, they may still exhibit behavior that some developers would consider undesirable if they are poorly written or perform unnecessary operations. Potentially undesirable install-time operations have been discovered in many repository packages, including sending machine specifications, machine identifiers, and lists of installed packages to remote tracking APIs [24].

In this work, we tackle the risks of install-time software supply chain compromise by proposing Latch (**L**ightweight inst**A**ll-**T**ime **CH**ecker), a system capable of (i) capturing in a succinct but expressive manner the operations performed at install-time by any given package, and (ii) matching those operations to user-defined security policies, flagging cases where package behavior violates the intended policy. Thus, our approach defines a set of install-time capabilities, and precisely bounds the install-time behavior of each package within the set of allowed capabilities.

Our prototype of Latch is focused exclusively on the npm repository, since it is the largest package repository and the one from which the most attack reports have been derived [24]. However, we believe that our techniques could easily be extended to other language ecosystems. There are several aspects of npm that make it susceptible to install-time supply chain attacks. The bootstrapping mechanism of npm is that a package may invoke a shell script as part of its installation and configuration, and this script runs with the permissions of the installing user. In the case of globally-installed npm packages, this shell script must be run with root privileges (unless npm is configured to write to a non-root directory) [52]. Additionally, like in many package management ecosystems, npm packages may depend on each other. To maintain a complete system, the entire closure of transitive dependencies will be added in a single package installation request. This behavior means the compromising package may be added to the system unknowingly, as a dependency of the core application, or as a transitive dependency of another dependency. Latch makes npm more secure by gating an important avenue for attacks while ensuring that installation is restricted according to a policy that users can fine-tune to meet their definitions of undesirable install-time behavior.

Our design of Latch requires achieving a number of challenging goals: the system must be configurable, automated, and performant. We describe the importance of these goals, and the relevant challenges, below.

Configurability. There is no universal consensus about which behaviors should be disallowed by package installers. Different stakeholders in a package ecosystem may choose different definitions based on their objectives: registry maintainers seek to maintain the health of their package repositories by weeding out truly malicious behaviors, but they also encourage packages to be published [24]. Individual developers using packages may have stricter boundaries than registry maintainers for what they deem to be acceptable installation behavior, and these boundaries may differ from other package users. The lack of a single consensus of acceptable package behavior is a significant challenge which we overcome by introducing a novel policy language that empowers Latch users to express their own definition of acceptable package behavior. We also introduce two reasonable template policies aimed at the distinct security postures of developers and registry maintainers so that users of the system can use an existing policy, perhaps with additional tweaks to fit their needs. This flexibility allows Latch to be deployed both as a developer tool for managing personalized security guarantees and as a registry auditing tool for detecting malicious packages.

Automation. A key benefit of package management systems is that they automate the process of downloading, installing, and configuring complicated codebases (although this automation is a key factor in system compromises via the software supply chain). Thus, an additional design constraint of Latch is to be unobtrusive, minimizing effort on the part of registry maintainers, package authors, and package users. We achieve this goal by introducing the notion of a *behavior manifest*: a description of the security-relevant behaviors that a package exhibits at install-time. In Latch, manifests are batch-generated automatically by executing and tracing each package installation in a sandboxed environment without the need for manual intervention by developers. When a user attempts to install a package, Latch employs two lines of defense. First, the behavior manifest of the package is checked against the user's policy so that the installation can be aborted if the policy prohibits that package's behavior. Second, during package install-time, deviations from the user's policy are prevented via kernel-level security modules.

Performance. Package deployment for large codebases can require the installation of hundreds of packages. As such, the solution needs to be scalable and performant with respect to large, highly interdependent codebases. We address these performance challenges by enforcing Latch policies over cached manifests and by utilizing a live enforcement framework that incurs minimal overhead.

Overall, the contributions of our work are as follows:

- We designed a policy language that gives users the power to express what behaviors they are willing to allow (and, by omission, behaviors to disallow) for npm packages. We also crafted two template policies that capture the distinct security postures of developers and registry maintainers. Users can avail themselves of our system by directly using these policies with minimal customization necessary.
- We implemented an end-to-end prototype of Latch. Our implementation enforces policies in our language and includes an analysis system which can operate at ecosystem scale to infer manifests for npm packages without developer cooperation.
- We evaluated our system and showed that it can prevent malicious and undesirable behaviors while still allowing benign packages to be installed correctly and without prompting the user.

Additionally, we publicly release our code and supporting data, freely distributed via the Open Science Framework [9] and via a public GitHub repository:

`https://osf.io/pa8c2/?view_only=6083cfbdf7314b958172178ac05996b5`

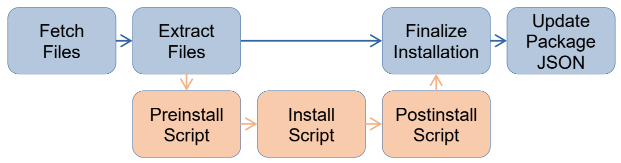
`https://github.com/elizabethwyss/Latch`

2 BACKGROUND

In this section, we give background information necessary to understand the need for the install-time permissions proposed in this work. We describe how the current landscape of package managers makes package installation a vector for undesirable and malicious behavior, and then we discuss how mitigations currently in place are insufficient.

2.1 npm

npm is a repository for software packages developed for the Node.js JavaScript runtime environment. Like other package repositories, it offers numerous benefits: it encourages code reuse and allows well-vetted, expertly-written codebases to be deployed by developers. npm contains millions of publicly available packages and has weekly download counts ranging from hundreds of millions to billions [38]. Additionally, dependencies comprise a significant portion of package code on npm. One recent study found that across



(a) Order of actions for a single npm package installation. Each of the script actions (red nodes) are only executed if the package defines these scripts.

```
{
  "name": "twilio-npm",
  "version": "1.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "postinstall": "bash -i >& /dev/tcp/4.tcp.ngrok.io/11425
  0>&1"
 },
 "author": "",
 "license": "ISC”
}
```

(b) twilio-npm install-time attack script (highlighted)

Figure 1: npm package installation hooks and attack

all npm packages, over 93% of all lines of code reside not in original package files, but rather in third-party dependencies [30].

Much of the complexity of package management is due to the interdependence of packages. Packages may require numerous dependencies, and these dependencies may transitively require dependencies of their own. When a user issues a command like `npm install webpack-dev-server`, the front-end constructs a tree of all required dependencies and then installs each package within the tree without the need for user involvement. While convenient, this dependency tree approach complicates the task for developers, who must decide whether to assign trust to dependencies.

2.2 Install Scripts

Most npm packages have limited configuration requirements beyond downloading JavaScript source code files and placing the files in a path that is known to the given project. In practice, however, there exists packages that require auxiliary bootstrapping actions during installation, such as writing configuration files or compiling code that might be used through a native interface. To automate these auxiliary tasks, npm packages are permitted to register shell scripts that are run in response to specific events in the installation procedure [40]. In particular, an npm package may register *preinstall* scripts that are invoked prior to the regular package installation, and *install* and *postinstall* scripts that are invoked when the installation is otherwise complete. The npm package installation process is summarized in Figure 1(a).

2.3 Installation Attacks

While shell scripts offer significant flexibility for package configuration, they also offer malicious packages significant capabilities to do harm to any system upon which they are installed. A simple but high-profile example of a package incorporating malicious install scripts is `twilio-npm` [50]. Despite its name, `twilio-npm` is unrelated to the popular Twilio cloud platform and exhibits no runtime functionality. However, it contains an install script that opens a reverse shell to an internet destination. The attack is displayed in Figure 1(b). Despite being removed from npm only a few days after its publication, `twilio-npm` had already affected 370 installation victims.

Malicious install scripts like that of `twilio-npm` have been used in practice for a wide variety of attacks, including damaging system integrity [29], exfiltrating credentials on the system [20], disrupting the operation of the host machine [62], and giving attackers access to the victim system [49]. These threats are enabled because package install scripts run with the privileges of the user invoking npm, which may be the administrator for system-wide installations, and is frequently an account that has privileges to access web infrastructure.

2.4 Mitigations in Place

Recent versions of npm do include some mitigations for install-time attacks and potentially undesirable behavior. However, they do not satisfy the needs of registry maintainers and developers since they are coarse-grained and either break package functionality (i.e., npm may be configured to ignore installation scripts entirely [42]), or place a large manual burden on the registry stakeholders (e.g., by using the npm audit option, which reports manually flagged security issues [37]). To the best of our knowledge there exists no solution providing fine-grained visibility and control over install-time scripts in npm. Such a solution, which we propose in our work, is necessary to ensure the containment of malicious and undesirable scripts while minimizing the disruption of packages that utilize installation scripts for benign purposes.

2.5 Software Supply Chain Security

Modern software infrastructure incorporates components from a variety of sources; the set of such components and their developers is referred to as the *software supply chain*. This approach to software development is flexible and cost-effective, but opens the door to supply-chain attacks, where software is compromised via injection of malicious code in its dependencies. These attacks represent a significant and costly problem [21, 43].

As supply-chain attacks are relatively new, so is their containment. A *software bill of materials* (SBOM)—a list of dependencies associated with a software artifact—can simplify software auditing, if deployed widely [17]. However, SBOM is still a relatively new concept that is undergoing standardization [4].

In practice, vetting software prior to deployment is oftentimes achieved by specialized tools such as Grafeas [5] and Kritis [6], which respectively store the output of vulnerability scanners and match such outputs against user-defined policies. This style of tools is designed to be integrated in software processes (most commonly CI/CD pipelines), so that vetting is performed transparently. Note that while Grafeas and Kritis provide infrastructure for security analysis of Kubernetes containers, they are orthogonal to the specific algorithms used for vulnerability analysis.

Finally, in the domain of package repository security, NodeSource provides commercial vetting of npm packages in the form of *certified modules* [7], which involves tagging npm packages with information about potentially security-relevant behavior. The goal is to enable users of the service to make informed decisions concerning the risk associated with each packages. Among other things, this service provides insight on whether a package executes install-time scripts; however, it does not provide details on what install-time scripts actually do. Package Analysis [27] is another tool that vets open source packages and exports information about the behaviors they exhibit. While this tool does provide some insights about the behaviors of package install scripts, it does not provide a mechanism for mediating undesirable package behaviors. Note that the technologies discussed above decouple detection and enforcement, which is typically necessary because not all security issues are practically relevant in all contexts (for example, the Kritis policy enforcement engine permits allow-listing CVE vulnerabilities in Kubernetes containers [3]).

Our work focuses on fine-grained identification and prevention of undesirable install-time behaviors in npm packages. It is based on the tried-and-true workflow outlined above which decouples the discovery of potential issue from the decision of whether such issues warrant blocking the installation/execution of the package. As such, we expect Latch to be easy to integrate with existing tooling and processes.

3 MOTIVATION

The npm registry ecosystem is comprised of distinct stakeholders with different use cases and security expectations. Because of these differences, it is necessary to consider the needs of all stakeholders involved when designing novel security protocols. We identify two distinct stakeholders in the npm registry ecosystem that are impacted by package install scripts: *registry maintainers*, who vet npm to remove malicious packages, and *developers*, who install packages to write software. The npm registry maintainers have stated that they wish to only remove packages that are explicitly malicious from the npm registry [39]. This vetting approach leaves significant room for packages to exhibit behavior that some developers would deem unacceptable.

A recent study [24] identified 22 npm packages that perform potentially undesirable install-time behaviors–including packages such as `npmtracker`, `igniteui-cli`, and `tysapi`, which send hardware configurations, software configurations, and unique machine identifiers to remote tracking APIs via installation scripts, and packages like `p4d-rpi-tools`, which uses an installation script to install additional npm packages within a nonstandard directory as the root user. Because different developers have different needs with respect to the privacy of their data and the integrity of their systems, the behaviors exhibited by these packages may be unacceptable for some developers. While these 22 packages were reported to npm, the npm registry maintainers stated that they would not remove the packages from the npm repository since they were not overtly malicious [24]. As such, registry maintainers and developers have distinct needs and expectations with regards to package install-time behaviors; registry maintainers need to detect explicitly malicious behaviors, and developers need to configure the behaviors they deem permissible. Thus, there exists no universally accepted threat model as different stakeholders have different security expectations.

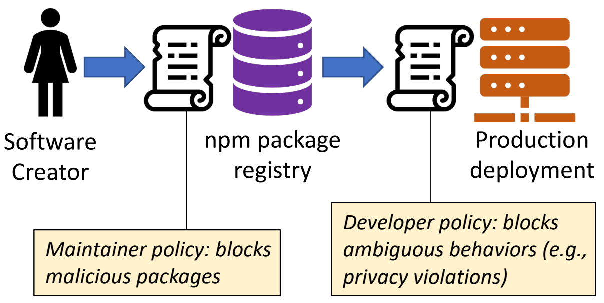

Software Creator
npm package registry
Production deployment
Maintainer policy: blocks malicious packages
Developer policy: blocks ambiguous behaviors (e.g., privacy violations)

Figure 2: Positioning of Latch within the software lifecycle

This motivates the need for a highly customizable and proactive defense against undesirable install-time behaviors.

We design Latch to satisfy the needs of both registry maintainers and developers. Registry maintainers can deploy a Latch policy to detect explicitly malicious behaviors as packages are uploaded to npm, and developers can deploy their own Latch policies to prevent packages from executing behaviors that they deem unacceptable. This process is summarized in Figure 2. In Section 5, we discuss reasonable default policies for both stakeholders.

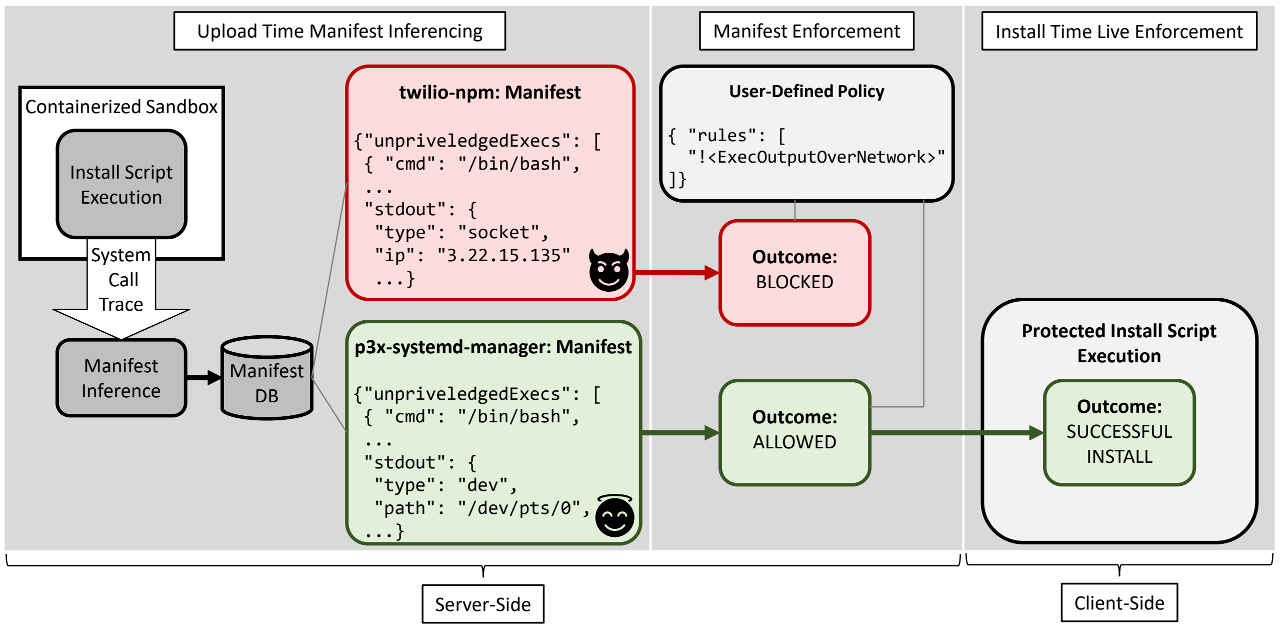
Review of Existing Package Behaviors. Using system-call tracing, we analyzed every install script on npm to determine how many exhibit behavior that developers may want to mediate.

At the time of our analysis, we found **1,493,231** distinct packages in npm, **36,438** of which contain an install script. However, this number understates the likelihood of an install script being encountered during npm package installation, since the scripts of all dependencies will be invoked. Furthermore, old versions of packages can be specified as dependencies. When considering all available versions, we found **385,798** packages that contain install scripts.

We found that many behaviors exhibited by install scripts may be undesirable to some developers. Many packages exhibited behavior that violates the best practices of npm: **178,870** packages create files within the `node_modules` directory, which can be done within the mediated environment of npm, with no need for a shell script[1]. **5,688** scripts are long-running (over 10 minutes of wall-clock time before we terminated them), potentially delaying or preventing an automated installation.

Many packages also contained behaviors that could break an OS deployment or cause data loss: **82,338** packages create files outside of the home directory, and **63** delete files outside of the home directory. Developers may be concerned about secrecy, or access/modification to sensitive files (such as **/etc/passwd**). We found **113,216** packages read one of more of these files, and **364** performed an update/modification. A number of packages also attempted to perform additional network connections: **104,400** connect to a local network host, and **102,900** connect to a remote host. These connections need not be declared by the package and may be used to exfiltrate information or alert a 3rd-party of the address from which the package is being run. We also found **3,235**

[1]We randomly sampled several packages that wrote to `node_modules`, and found none that required additional shell script capabilities for this behavior.



```
{"unpriveledgedExecs": [
 { "cmd": "/bin/bash",
 ...
 "stdout": {
  "type": "socket",
  "ip": "3.22.15.135"
  ...}
```

```
{"unpriveledgedExecs": [
 { "cmd": "/bin/bash",
 ...
 "stdout": {
  "type": "dev",
  "path": "/dev/pts/0",
 ...}
```

```
{ "rules": [
  "!<ExecOutputOverNetwork>"
]}
```

Figure 3: Three-phase Latch workflow. At upload time, packages manifests are created and cached offline. When a package installation is initiated, the user-supplied policy is checked against the manifest and interposes in case of a policy violation. At install-time, the install script's behavior is mediated by live enforcement.

scripts that execute a file that is created by that package, potentially hiding information from static analysis.

The distribution of install scripts in npm supports our observation that even strict mediation will not overwhelm npm users: most packages do not contain install scripts, so alerts will be rare (a detailed analysis is presented in Section 6). The behaviors exhibited by install scripts indicate that mediation is important; many of the behaviors could be system-breaking and at least indicate poor quality code that may become a weak point of the overall system. We emphasize that the behaviors we describe are not all overtly malicious, and thus motivate the need for expressive policies, rather than a one-size-fits-all approach, since some developers may be comfortable with these behaviors while others may not.

4 OVERVIEW

Latch is designed to limit the threat of install-time package attacks in two ways: (i) it allows registry maintainers to prevent overtly malicious packages from being published in the repository, and (ii) it empowers users to control the install-time behavior of packages. Operationally, a registry maintainer policy can be applied in the background whenever a package is uploaded. Similarly, whenever a user issues an `npm install` *p* command (for some package *p*), Latch will automatically ensure that package *p*, as well as the direct and indirect dependency packages of *p*, only perform install-time operations that are allowed by the user. Expressing permissions is done using a domain-specific policy language which we describe in Section 5.1. These policies operate over a *manifest* of permissions that a given package will use in its installation process. An important design goal of this system is that it works without the cooperation of individual package authors. In particular, we do not require that the package manifest be declared by the package author. Avoiding reliance on package authors is an important feature of our approach because of the number of existing packages currently available on the repository: there are 1,493,231 current packages, as well as 15,136,563 older versions of packages which are still accessible (and which other packages might depend upon). Requiring manual retrofitting on such a scale is infeasible; hence, we design Latch so that package manifests can be inferred independently—both for new and existing packages. Furthermore, we assume the goals of some package authors may run counter to the interests of package users: the package installer may contain behaviors that some developers find unacceptable and, in the extreme case, the installer may be explicitly malicious. Removing the requirement for package authors to cooperate with our system thus removes the reliance on a potentially untrusted entity. Therefore, the system needs to *infer* a given package's manifest and then *enforce* that the manifest is obeyed by the user's policy.

We envision that both registry maintainers and developers will want to deploy Latch. Registry maintainers benefit by using Latch to scan the npm repository for malicious packages, and developers benefit by using Latch to specify the exact install-time behaviors they deem permissible. Cooperation with registry maintainers also greatly simplifies the package manifest inference phase of Latch, since registry maintainers can generate and make available package manifests *en masse*. We present the high-level workflow of the Latch system in Figure 3, and we describe it in detail below.

4.1 Latch Workflow

Latch operates in three phases, a *manifest inference* phase, in which a package's manifest is generated, a *manifest enforcement* phase, in which the manifest is checked against the user's policy, and a

```
connect(3<TCP:[49835]>, sa_family=AF_INET, sin_port=htons(11425), \
  sin_addr=inet_addr("3.22.15.135"), 16) = 0
dup2(3<TCP:[10.0.2.15:44028->3.22.15.135:11425]>, 1</dev/pts/0>) = \
  1<TCP:[10.0.2.15:44028->3.22.15.135:11425]>
dup2(1<TCP:[10.0.2.15:44028->3.22.15.135:11425]>, 2</dev/pts/0>) = \
  2<TCP:[10.0.2.15:44028->3.22.15.135:11425]>
dup2(1<TCP:[10.0.2.15:44028->3.22.15.135:11425]>, 0</dev/pts/0>) = \
  0<TCP:[10.0.2.15:44028->3.22.15.135:11425]>
execve("/bin/bash", ["bash", "-i"], 0x55559cc91bd0) = 0
```

(a) Relevant subset of system call trace

```
{ "intent": "connect",
  "info": { "type": "AF_INET",
  "ip": "3.22.15.135", "port": "11425" }},
{ "intent": "exec",
  "info": {
    "cmd": "/bin/bash", "args":["bash","-i"],
    "stdin": {
      "type": "socket","ip": "3.22.15.135",},
...}}
```

(b) Relevant intents

Figure 4: Example system call trace and intents for `twilio-npm`

live enforcement phase, in which the package's install scripts are executed under the protection of a kernel level security module.

4.2 Manifest Inferencing

The manifest inference stage identifies all packages that define install scripts and computes a manifest of all actions performed on the operating system during the invocation of each script. The full process of inferring a package's manifest is depicted in Figure 3. To protect infrastructure, Latch sandboxes the installation process using Singularity containers [55].

A key observation that underlies our inference mechanism is that although package *run-time* behavior is highly diverse, package *install-time* behavior tends to be stable across multiple runs. As such, we use a dynamic-analysis approach whereby behaviors of the package under test are observed and recorded to a log file, which is then compiled into a package manifest. As inferencing is relatively time-consuming, we envision it to be performed asynchronously when a package is first uploaded to npm. The resulting manifest would then be cached in a manifest database.

Install-time operations are profiled by recording the resulting sequence of system calls using the *strace* tool[2]. To prevent *strace* from capturing non-script-related events (e.g., npm copying the package files in the appropriate directories), manifest inferencing utilizes a modified version of the package manager source code that only tracks the system calls invoked by the install script. As an example, Figure 4a lists the system calls collected during installation of the malicious `twilio-npm` package, whose installation script is represented in Figure 1b.

To aid in the extraction, we provide a minor operating system state abstraction to track data dependencies across system calls. For instance, the system call *mmap* loads a file into memory and *msync* syncs a memory mapped file to the file system. By noting the memory location, length, and flags of the *mmap* invocation, Latch can determine whether a process could manipulate the file before committing it by invoking *msync*. Latch also keeps track of the current working directory to disambiguate relative file paths.

After the install script has finished generating its system call trace, Latch interprets the trace into a collection of *intents* of all interactions between the script and the operating system. Figure 4b displays the set of intents generated from interpreting the system calls from Figure 4a. Each intent represents a relevant high-level operation—most commonly an access to the file system or a network resource—and related metadata. To generate manifest attributes, these intents are filtered and grouped according to each attribute's description. Using the example from Figure 4b, */bin/bash* and *3.22.15.135* would be added to the list of executables and remote IPs connected to, respectively.

After processing all system calls and filtering intents for an installation script, a manifest is generated and stored in the manifest database under the package name and version. We note that manifests are generated even if installation scripts crash during execution. It is important to capture the behaviors of these buggy install scripts since they may perform undesirable operations prior to crashing. Figure 3 shows a simplified version of the manifests for `twilio-npm` and `p3x-systemd-manager`. These manifests capture the fact that, while both packages execute `/bin/bash`, the former directs its output to a network socket, while the latter to a terminal.

4.3 Manifest Enforcement

When a user intends to install a package, Latch checks the user-defined policy against the manifests of all packages and package dependencies attempting to be installed (we discuss policy generation in Section 5.2). The manifest enforcement engine is invoked before spawning a package's install scripts to prevent unwanted behavior. All packages and package dependencies are analyzed according to the user's policy.

Each package's manifest is fetched from the manifest database and is then analyzed against each rule in the user's policy. Figure 3 illustrates this procedure and shows a simple policy that prohibits install-time scripts from executing external processes while redirecting output over the network. During manifest enforcement, policy rules are evaluated to a boolean value; in the example, the policy evaluates to *false* for `twilio-npm` and *true* for `p3x-systemd-manager`.

Inspired by other run-time enforcement systems such as SELinux [1], Latch supports two functioning modes, *warning* and *failure*. When in warning mode, rule violations generate alerts but do not halt package installation. When in failure mode, the same violations cause the installation to fail. These modes can be specified at the granularity of individual rules. When the policy of Figure 3 is executed in failure mode, the installation of `p3x-systemd-manager` completes but that of `twilio-npm` is correctly interrupted. When a

[2]While *strace* is Linux-specific, the Latch approach is general; *strace* can be swapped with other appropriate utilities under other operating systems with some implementation effort (we further analyze portability challenges in Section 7.2).

```
{"declarations": [
 "<allRemoteHosts> = [remoteHosts_preinstall] ~union
                     [remoteHosts_install] ~union
                     [remoteHosts_postinstall]",
 "<allFilesRead> = [filesRead_preinstall] ~union
                   [filesRead_install] ~union
                   [filesRead_postinstall]",
 "<<passwdFile>> = '/etc/passwd’”],

 "rules": ["<allRemoteHosts> == {}",
   "!(<allFilesRead> ~anymatches [<<passwdFile>>])"]}
```

Figure 5: Simplified example Latch policy. Rule 1 prevents connection to remote hosts. Rule 2 prevents reads from *etc/-passwd.*

policy violation occurs, the user is prompted with information pertaining to the packages violating the policy along with the policy rules being violated.

4.4 Live Enforcement

Even though the large majority of packages exhibit consistent install-time behavior, it is possible for install scripts to exhibit nondeterministic behavior not captured in a manifest that has been generated from a single run. These nondeterministic behaviors may be present due to differences in system architecture or due to adversarial evasion techniques such as logic bomb and time bomb attacks. To provide true policy enforcement in adversarial and nondeterministic settings, our solution needs to support live policy enforcement during installation time. The live enforcement process is depicted in Figure 3, which shows a protected install script execution for `p3x-systemd-manager`.

To provide live enforcement, Latch utilizes kernel level security module policies. We select AppArmor [2] as our security module of choice given its reputation, popularity, and relative ease of use as compared to similar kernel level security modules.

We note that existing kernel level security module policies do not offer the same level of expressibility as Latch policies that operate on manifests. As such, live enforcement provides a more restrictive, safe approximation of manifest enforcement. Ultimately, live enforcement enhances the security guarantees of Latch at the cost of weakened expressibility. Thus, we implement live enforcement as a configurable option in Latch that can be enabled or disabled by the user.

5 MANIFEST ENFORCEMENT POLICIES

A key component of our system is the use of user-defined policies to explicitly indicate what install-time behaviors are allowed by a program. To interface with Latch, our system provides a domain-specific policy language which allows users to express constraints on install-time manifests.

5.1 Policy Language

Our system's policy language is intended to be lightweight. It allows for fine-grained filtering with a high level of abstraction to construct a policy to be used for all packages. This language is centered around manifest attributes. It is implemented using a parsing expression grammar and allows for basic boolean and set operations. A simplified example policy is shown in Figure 5, and our default policy recommended for developers is shown in Appendix A.

We designed our policy language to allow a user to protect against any script capability model that can be extracted from a package's manifest. Each policy is a conjunction of predicates over permissioned behaviors consisting of a list of declarations and a list of rules. Declarations in a policy are computed values of a manifest that can be used in other declarations and rules. Rules in a policy are attributes over the manifest and declarations that the user allows (or denies, by negation) an install script to exhibit. This level of granularity permits the construction of both allow-list and block-list based policies.

The full grammar of our policy language and a collection of examples designed to aid in the construction of Latch policies are provided in our public GitHub repository:

https://github.com/elizabethwyss/Latch/tree/main/policy

5.2 Default Policies

Latch's design is based on the insight of separating *mechanism* from *policy*, which historically has been applied with great success in the security domain [46]. The enforcement engine provides the mechanism, while the user is tasked with defining policies appropriate to the use cases of interest, via Latch's policy language.

In practice, we expect that most users will use standardized policies defined by experts, with minimal customizations. In this section, we discuss the formulation of two empirically-derived template policies–one aimed at developers and one aimed at registry maintainers. We evaluate the template policies in Section 6.

Developer Policy. First, we consider a policy designed to meet the needs of a security conscious developer who wishes to disallow all security sensitive operations on their system. This policy guarantees that no install-time security sensitive operations occur on the system, but also prevents the installation of many packages with install scripts.

This policy's rules allow install scripts to print output to the terminal (/dev/pts or /dev/tty devices on Linux) and to read nonsensitive files, but prevents connecting to network hosts and writing to files. Appendix A shows our developer policy in full.

Maintainer Policy. Second, we consider a policy designed to meet the needs of registry maintainers who wish to scan the npm repository for malicious packages. Because the historical decisions of registry maintainers are available, we utilize a data-driven, learning algorithm-based approach formulated by discriminating known malicious and benign package install-time behavior as determined by historical takedowns of packages by npm maintainers. First, we build a labeled dataset by randomly sampling 375 well-known benign npm packages and supplement those with 375 malicious packages randomly oversampled (to avoid bias due to class imbalance) from a collection of 102 malicious packages [44] that have been removed from npm by the registry maintainers. Then, we build a CART decision tree classifier [14] to discriminate between benign and malicious packages, and we generate policy rules directly from the branches of the trained decision tree.

6 EVALUATION

To determine the effectiveness of our proposed system, we focus our evaluation on three research questions

- **RQ1:** Does manifest enforcement allow for effective discrimination between benign, potentially undesirable, and explicitly malicious packages? In Section 6.1, we show that our developer policy blocks 1.5% of all npm packages, 82% of tested potentially undesirable packages, and 100% of tested malicious packages. We contrast this with our maintainer policy, which blocks 0.013% of all npm packages, 14% of tested potentially undesirable packages, and 99% of tested malicious packages.
- **RQ2:** How frequently does the system result in interruption of developer workflow? In Section 6.2, we demonstrate that installation interruptions are caused by our developer policy in 1.6% of package installs and by our maintainer policy in 0.3% of package installs.
- **RQ3:** Is the system's performance reasonable at registry scale? In Section 6.3, we show that 90% of packages can have their manifest inferred in less than a minute, and 99% of packages can have their manifest enforced in less than a second.
- **RQ4:** Does live enforcement guarantee safety in potentially nondeterministic settings? In Section 6.4, we demonstrate that every installation script blocked by manifest enforcement is also blocked by live enforcement, and 93.8% of installation scripts allowed by manifest enforcement are also allowed by live enforcement.

6.1 Manifest Enforcement Policy Violations

First, we generate manifests for all npm packages and all known malicious packages in our datasets. Then, we evaluate our two template policies on every package manifest. We assess the effectiveness of our policies by analyzing the total number of policy violations caused by all packages, previously identified [24] potentially undesirable packages, and known malicious packages [44].

Effectiveness on npm Packages. Our template policies allow for the installation of packages that do not exhibit security-sensitive behavior. Ultimately, the ground truth of permissible package behavior lies within a user's policy, and as such, it is impossible to universally label all packages as either benign or malicious. Rather, we analyze the effectiveness of our template policies by measuring their violation rates across all available npm packages. We present the violation rates of each template policy in Table 1.1, separated by whether the violating package's version is the latest or old.

The less strict maintainer policy incurs far fewer policy violations than the more strict developer policy. Across all versions of all npm packages, the maintainer policy achieves a 0.013% violation rate, and the developer policy has a violation rate of 1.5%. These values align with our initial expectations when designing these policies. Both of these policies allow for a significant portion of security nonsensitive behaviors.

Table 1.1	**npm Package Violations by Package Version**	
Policy	**Latest (1,493,231)**	**Old (15,136,563)**
Developer	16,271	232,466
Maintainer	681	1,543

Table 1.2	**Malicious Package Violations**
Policy	**Dataset From [44] (102)**
Developer	102
Maintainer	101

Table 1.3	**Potentially Undesirable Package Violations**
Policy	**Dataset From [24] (22)**
Developer	18
Maintainer	3

Table 1: Policy violation counts for each template policy on our npm, potentially undesirable, and malicious datasets. Latest packages refer to the most up-to-date versions of packages, and old packages refer to all previous versions of packages.

Effectiveness on Malicious Packages. Our template policies are constructed to provide different approaches to prohibiting malicious behaviors. We present the results of analyzing our malicious package dataset on each of these policies in Table 1.2.

The more restrictive developer policy performs better at detecting malicious packages than the maintainer policy and reports violations in 100% of tested malicious packages. The maintainer policy is slightly less effective than the developer policy and reports policy violations in 99% of malicious packages, missing only a single instance of malicious behavior. This high recall of the maintainer policy is to be expected given that the malicious package dataset used to test it was also used to train it. The goal of the maintainer policy is to block the most widely applied forms of install script attacks, and we utilized a malicious package dataset that is comprised entirely of such attacks. As the malicious package landscape evolves, we expect the maintainer policy to be updated accordingly. Although these results do not show how our maintainer policy would generalize to unseen malware, they demonstrate that generating a policy using a realistic data-driven approach is effective in blocking behaviors deemed impermissible by registry maintainers.

The single instance of malicious behavior missed by the maintainer policy involves printing the user's *ssh* keys to the terminal. This behavior has similar permissions to benign behavior such as reading the */etc/passwd* file to determine a user's identity. Since reading files like */etc/passwd* could be done for benign purposes, our less strict maintainer policy does not catch this behavior. The maintainer policy could easily be tweaked to catch this behavior, although it would result in additional undesired alerts caused by installation scripts that read from similar files with benign intent. We believe that our maintainer policy in its current form reflects the security attitudes of the npm registry maintainers.

Effectiveness on Potentially Undesirable Packages. Our template policies are designed with the needs of different groups in mind. As such, the policies should perform vastly different on packages that exhibit potentially undesirable–but not overtly malicious–behavior. We present the results of analyzing our potentially undesirable package dataset on each of these policies in Table 1.3.

As expected, the developer policy disallows more potentially undesirable packages than the maintainer policy and reports violations in 82% of our potentially undesirable packages. The maintainer policy allows for more potentially undesirable behavior and disallows only 14% of potentially undesirable packages. Out of the four potentially undesirable packages allowed by the developer policy, one installs additional npm packages as part of a postinstall script, and the other three read information that is not security sensitive, including npm configurations and randomized identifiers. With regards to the maintainer policy, the three packages disallowed by the maintainer policy perform invasive user tracking, recording potentially identity compromising details about the package installer's system via a remote tracking API. The packages that were allowed by the maintainer policy but disallowed by the developer policy all send system architecture specifications and/or npm configuration information to remote tracking APIs. Given the goals of each template policy, we believe the violations of each policy align with the security needs of the users they are designed for.

Suspicious Install-Time Behaviors. We now review install-time behaviors found on npm that triggered the developer policy.

A number of packages either updated, modified metadata on, renamed, or deleted a file outside the user's home directory. These events were almost all due to mistakes in the package's install script. For instance, if a package builds files from its source code, it may have a folder *build* within its directory on the user's system. Such a package will also have a script to remove the contents of this folder before re-building these files. However, instead of removing the intended folder with a relative path, *./build*, some packages attempt to remove a folder with an absolute path, */build*. While this could be detrimental to a user with important files in this folder, packages exhibiting this mistake had very few downloads and are likely to never be installed by a real user. Also, we observed that when these issues occur, they tend to be resolved in successive versions of the same package. Incidentally, this observation suggests that LATCH may be applicable as a testing tool to identify unexpected install-time behaviors prior to uploading/releasing a package.

The exception to this behavior was the *opsie* package, which wrote to the */dev/initctl* and */run/initctl*. Writing to these files is a consequence of executing the *reboot* command on Linux. After further investigation, the intent of this package is to delete a user's unsaved work upon installation by rebooting the system without forewarning. We reported this package to the npm security team, and it has since been removed from npm.

Another potentially problematic behavior we found is updating user scripts executed upon startup or login. We singled out such updates because an attacker could put in a backdoor or execute malicious commands without the user's knowledge. There were 364 instances where the *~/.bashrc*, *~/.bash_profile*, and *~/.profile* files were updated 281, 80, and 14 times, respectively. From the 364 packages, we manually analyzed a sample of scripts that updated each file. Each of the analyzed packages used these files to either set environment variables or to load package tools.

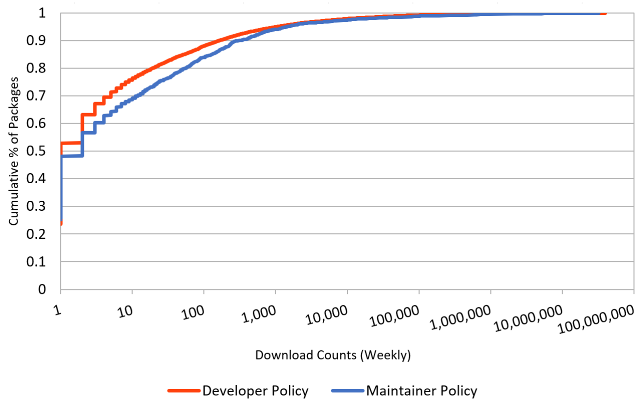


Figure 6: Distribution of download counts over the packages violating each template policy. A majority of policy violations for each of our template policies occurs in packages that are rarely downloaded.

We also found the creation and execution of a file to be a suspicious behavior. There were 3,235 packages exhibiting this behavior, and we performed a manual analysis on a portion of these packages. The most common cause of this behavior occurs when a package checks that compilation or download of a binary was successful by running that binary (usually with a *version* option).

All of these suspicious behaviors were detected using LATCH manifest attributes and thus can be blocked via LATCH policy rules.

6.2 Impact on User

Each template policy has instances of package installations halted across npm because of a policy rule violation. This subsection analyzes how frequently a user will encounter a package with such violation. A good metric to measure how often our system will intervene during the installation process is the download counts of packages that incur a policy rule violation. We gathered download counts for all packages violating each of our template policies to analyze the trade-offs between the different security levels and how often development workflow will be interrupted.

Download Counts and Popularity. Package download counts do not directly represent the number of users installing the package. npm mirrors and bots download packages regularly for analysis. Downloads from these sources are indistinguishable from user downloads and are recorded in the overall download count. The creators of npm estimate a package can be downloaded up to 50 times per day without ever being installed by an actual user [41].

Using this estimate, we classify a package with fewer than 350 downloads per week as likely never installed by an actual user. Recent research identifies packages with more than 100,000 downloads per week to make up less than 1% of npm and account for a majority of all downloads [56]. The analysis of this subsection shows that it is rare for any user to be hit by a policy violation.

Each violation shown in Table 1, represents a single version of one of the 36,438 distinct packages with installation scripts that we analyzed. Download counts represent all downloads of all versions of a package, thus we convert the policy violation numbers into

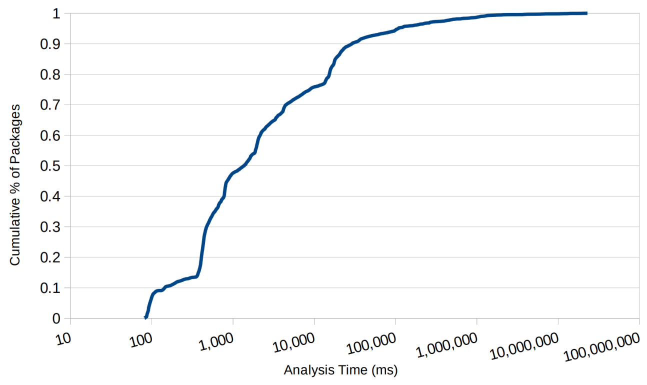


Figure 7: Distribution of total analysis time to produce a manifest for each package.

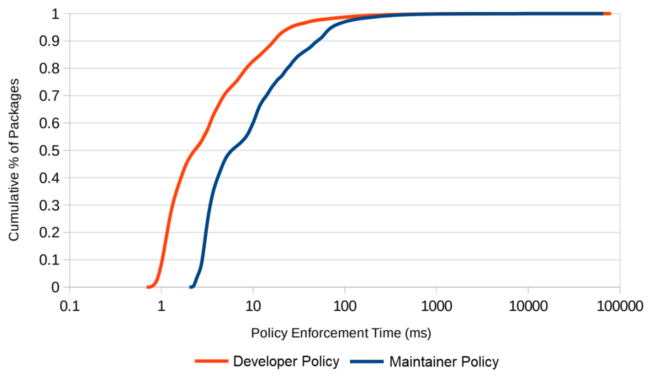


Figure 8: Distribution of time to enforce each of our template policies over manifests.

distinct package policy violations for this analysis. The 248,737 and 2,224 violations of the developer and maintainer policies represent only 24,345 and 1,775 distinct packages, respectively.

We gathered download counts for each of the distinct packages expressed by the policy violations and present the results grouped by popularity classes in Figure 6. Over 92.5% and 90.3% of the developer and maintainer policy violations respectively are committed by packages we estimate to likely never be installed by an actual user. The remaining packages which violate the developer and maintainer policies account for 435,471,572 and 78,075,730 weekly downloads, respectively. These numbers are absolute maximums considering the download count data provided by npm does not include a package version distribution; it is very probable these figures are reasonably smaller.

Workflow Interruptions. The ideal number of warnings given to a user notifying them of a policy violation during package installation depends on the level of security of the policy. It is difficult to quantify the degree of security of a given policy, however we have presented two template policies suited to the distinct goals of different stakeholders in the npm ecosystem.

The estimated portion of package downloads which will result in a warning from our system is approximately 1.6% for the developer policy and 0.3% for the maintainer policy. In other words, our system generates a single warning, on average, every 63 package installs for the developer policy and every 356 package installs for the maintainer policy. We consider these acceptable rates for users considering the goals of each policy.

6.3 Performance

Manifest Inference. The npm ecosystem is the largest and one of the fastest growing open-source software registries in the world, growing at rate of over 850 new packages per day[3]. Of these packages, we estimate about 21 will define install scripts. The registry, on average, also sees around 3,000 existing package updates per day, of which we estimate about 74 will also contain install scripts. The manifest inference phase of our system needs to be able to accurately and promptly compute manifests to keep pace with the growth of npm. We show that our manifest inference system can match the growth of npm.

The size of system call trace files directly impacts the time required to infer a manifest. In our analysis, the total size of system call trace files range from one kilobyte to over 4 gigabytes. The size of these files are proportional to the complexity of the script being executed (e.g. simply printing a message versus downloading and compiling a large codebase). We recorded the time it takes to infer manifests for different packages and present our data in Figure 7.

Around 90% of packages can have a manifest inferred in less than a minute. A majority of the remaining packages take less than 10 minutes to infer a manifest. When extracting manifests, there are a few outliers that take over an hour to analyze. This is due to the size of the system call trace files being analyzed. There are approximately 5,000 system calls per 1 megabyte of system call file and Latch can process around 1,000 system calls per second.

Manifest Enforcement. Large codebases can require the installation of hundreds of packages and their dependencies. Our solution needs to provide behavior restriction without incurring large overhead costs. The complexity of the policy and the size of the manifest directly influences how fast the analysis is performed. We performed timing analysis on each of our template policies over our datasets and show the results in Figure 8. Our maintainer policy is more complex, consisting of 73 declarations and 15 rules, and we consider it to represent a reasonable upper bound to the analysis runtime. The time taken to compute declarations and rules depends on the size of the manifest. In our analysis, manifest size ranges from 4KB to over 300MB.

Over 99% of all packages have both policies enforced over their manifests in less than 1 second. We believe this enforcement time to be reasonable since we experimentally measured the average runtime of npm install scripts to be 12 seconds. There are a few packages that take over a minute to enforce a policy on their manifests, but it is important to note that these packages have install scripts that take over 5 minutes to complete. Manual analysis showed that these long-running install scripts typically perform operations such as compiling native addon modules and executing large testing suites. Given the rarity and comparatively smaller enforcement

[3]http://www.modulecounts.com/

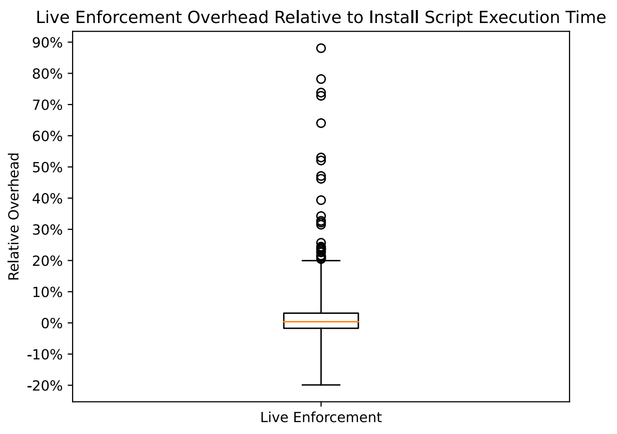


Figure 9: Boxplot distribution of install script overhead incurred by live enforcement. The median overhead of 0.37% is depicted in orange, and the quartiles form the edges of the box. The whiskers represent typical execution time variation across repeated install script runs, which we measured to be roughly 20% in either direction.

times of these lengthy installation scripts, we believe the performance of LATCH manifest enforcement is practical for deployment.

6.4 Live Enforcement

Due to the potential for install scripts to exhibit nondeterministic behaviors not declared in their manifests, our system must provide a safe and reasonably performant mechanism for live policy enforcement. We investigate the extent to which live LATCH policy enforcement is practical and safe using the AppArmor [2] kernel level security module.

Given that live enforcement would offer greater protections for developers installing packages, we opt to translate our default developer policy into an AppArmor policy and measure the effectiveness of this approach. Due to differences in the expressivity of LATCH and AppArmor, we seek to construct a more restrictive approximation of our default developer policy.

Performance Impact. We experimentally evaluate the performance of live enforcement and find that AppArmor handling live enforcement increases the overhead of package installation scripts by less than 1% on average, with a worst-case scenario of 88% overhead. This overhead is most pronounced for operations that frequently access files, which requires performing many additional live policy checks. The distribution of live enforcement overhead across all installation scripts is depicted in Figure 9.

Effectiveness of Live Enforcement. We experimentally evaluate the effectiveness of AppArmor handling live enforcement by executing the install scripts of the most recent versions of all 36,438 npm packages that declare installation scripts and tracking policy enforcement violations reported by AppArmor. We find that 93.8% of installation scripts allowed by manifest enforcement of our default developer policy are correctly allowed by our corresponding AppArmor policy. We further find that 100% of installation scripts denied by manifest enforcement of our default developer policy are correctly denied by our corresponding AppArmor policy.

To investigate why some installation scripts permitted by manifest enforcement are denied by live enforcement, we manually examine a random sample of 50 instances where this occurred. We find that 44% of these cases occurred due to buggy installation scripts that crash during execution. In 12 instances, the installation script failed to finish executing during manifest inference, and live enforcement terminated the script at a later point in the script. In the 10 other buggy installation scripts, live enforcement prevented the execution of operating system native binaries that serve the sole purpose of performing disallowed behaviors (e.g. the Linux `install` binary), while manifest enforcement found that those binaries crashed and as such did not exhibit any disallowed behaviors. We note that preventing the execution of operating system native binaries that explicitly perform disallowed behaviors was a more restrictive design choice made in constructing our AppArmor policy so that safety is preserved in cases where the successful execution of scripts is not deterministic. In the remaining 56% of cases where an installation script allowed by manifest enforcement was disallowed by live enforcement, AppArmor's coarse granularity with respect to network sockets caused the installation scripts to be incorrectly denied. We note that the incongruity between LATCH and AppArmor's handling of network sockets could be resolved through the combined use of AppArmor and additional security tools, but we view this to be more of an engineering challenge than a scientific one, and as such we leave this as future work.

Based on the results of our experimental evaluation of live enforcement, we believe live enforcement of LATCH policies via AppArmor to be practical and effective. Despite the slight discrepancies between the expressiveness of LATCH and AppArmor, we demonstrate that AppArmor live enforcement achieves an effective and safe approximation of LATCH policies, and as such, we believe that the development of independent live enforcement frameworks is unneeded. Despite this satisfactory result, however, we believe that further developments in kernel level security modules and related tools could improve the expressivity of live enforcement policies, and we leave this as a relevant avenue for future work.

7 DISCUSSION

7.1 Summary of Results

As shown in Table 1, our template policies mitigate a significant number of malicious and potentially undesirable behaviors while still allowing for the installation of a large portion of npm packages. As discussed earlier, the developer policy was the most successful at preventing malicious behavior although it incurred more policy violations. The maintainer policy was the most successful template policy at allowing for the most npm package installations while still reporting nearly all known malicious packages.

npm Package Policy Violations. Given the generally low popularity of most packages causing policy violations, it is very rare that any user would encounter a policy exception. We recommend these policies for developers and registry maintainers to make npm more secure for both parties. Since package installation is only performed once for most developer workflows, limited installation interruptions are acceptable for the security benefits provided.

Performance Impact. It is important for our system to be performant, both at package analysis time and package installation time. As shown in Figure 7, manifest inferencing occurs quickly for most all packages. Very few packages (61 packages, or < 0.0002% of packages with install scripts) invoke a degenerate behavior in which the npm binary is recursively called from within an install script, forcing a nested analysis of Latch, with a large increase in overhead. Recursive npm invocation is considered bad practice, but it could be handled in Latch with additional implementation effort. We stress that manifest inferencing can occur for many packages in parallel, which greatly improves Latch's scalability.

Figure 8 shows that manifest enforcement also occurs quickly for most all packages and policies. Latch imposes an average increase in installation time of about 3.2% and 6.2% for the developer and maintainer policies, respectively. We believe this result to be reasonable as the slowdown caused by Latch is essentially unnoticeable.

As shown in Figure 9, the overhead imposed by live enforcement is negligible in almost all cases. Live enforcement incurs an average install script overhead of less than 1%, and we believe this performance to be more than satisfactory for practical deployment.

7.2 Limitations

Portability. Although the Latch approach is not operating system-specific, porting Latch between different operating systems and their versions poses some challenges. Available system calls and their precise behaviors vary between operating systems and some of their versions; however, Latch manifest attributes and live enforcement capabilities are higher-level abstractions of these low-level behaviors. This level of abstraction means that many operating system updates will not break the functionality of Latch. Despite this, different operating systems and some operating system versions will require implementation effort to resolve differences. We believe that these portability issues are non-trivial but feasible to resolve if Latch were to be deployed in practice.

Adversarial Evasion. While adversarial evasion techniques that rely on nondeterministic install script behavior, such as logic bomb and time bomb attacks, are prevented by the live enforcement phase of Latch, there still exists techniques that an adaptive adversary could utilize to potentially circumvent the protections provided by Latch. One such approach would be to identify exploitable gaps between the precise low-level behaviors of system calls and how they translate into higher-level abstractions in Latch manifest attributes and live enforcement capabilities. Despite potential adversarial evasion, Latch improves the security of package install scripts and significantly raises the bar for adversaries trying to execute package install-time attacks.

8 RELATED WORK

Package Repository Security Analyses. Previous work has noted the existence of security issues within package managers and the impact of dependency chains on package security. A number of previous researchers have studied the overall impact of vulnerable and malicious package issues within npm, with some focus particularly on the use of dependencies [18, 23, 28, 47, 58, 62]. Abdalkareem *et al.* provide a study on the use of trivial packages, which increase the install-time codebase [10]. Koishybayev *et al.* describe a system to remove unused package dependencies, but their system does not mediate the behavior of a package if its use cannot be eliminated [30]. Davis *et al.* analyze the prevalence of regular expression denial of service (ReDoS) attacks across the npm and PyPI repositories [22], and Staicu *et al.* investigate how ReDoS attacks within npm affect real-world web servers [53]. While the focus of our work is in defending npm, we note that other package managers such as apt have also been an object of study for similar issues [11, 16, 33]. Finally, the problem of malicious and/or vulnerable dependencies falls within the more general problem area of supply-chain security [15, 32, 48, 57, 61], i.e., the study of how incorporating dependencies in a software artifact can diminish the security of that same artifact. Differing from these works, we tackle containment of a specific security issue, namely undesired installation operations within repository packages.

Manifest-Based Permission Systems. Our work is partially inspired by permission systems based on an install-time manifest, as widely deployed in the Android mobile operating system. Numerous works have built upon the manifest-based permission model to assess the safety of untrusted software [19, 35, 60] or to measure the extent of Android's manifest-based security [12].

Unlike work that builds upon existing manifests, we include manifest inferencing as part of our system. As such, we do not require the cooperation of developers to declare the capabilities of packages (in Android, developers are responsible for crafting a manifest and including it as part of an app).

Malicious Package Detection. Some previous work has focused on flagging packages based on signals of vulnerable or malicious behavior, through analysis of the code or metadata. Multiple works have proposed static analysis techniques to detect vulnerable package code [8, 26, 34, 36]. Synode, by Staicu *et al.*, combines static analysis and runtime enforcement to detect and secure packages with command injection vulnerabilities [54]. Garrett *et al.* created a system for detecting malicious updates to existing packages [25]. Duan *et al.* applied a combination of static analysis, dynamic analysis, and metadata analysis to detect malicious packages [24]. Taylor *et al.* proposed TypoGard, a tool for assessing if a package has a name which is suspiciously similar to a more popular alternative [56]. Most similar to our work is Mir, by Vasilakis *et al.*, which implements a read-write-execute permission model to mediate access to fields within Node.js libaries [59]. While these approaches share our goal of protecting users from undesirable behavior, none of them focus on the detection and prevention of malicious and undesirable install-time behavior.

Ohm *et al.* [45] proposes a tool to raise awareness to developers or users of new system changes between versions of packages. Much like the built-in audit capabilities of npm, this tool requires manual analysis of new system changes to proceed or halt installation, whereas our tool fully automates this process.

9 CONCLUSION

This paper presents a mechanism for limiting the install-time behaviors of npm packages through the use of a novel permission system.

We show that the install-time security behaviors of npm packages can be discovered automatically using dynamic analysis, where we summarize behavior in an install-time manifest. We propose a light-weight policy language to approve manifests and show that several template policies prevent most malicious and undesirable package behavior while still allowing almost all benign package installations. We further protect users in real-time by providing live enforcement of policies as install scripts are executed. Our evaluation shows that mediating package installation is a meaningful and effective protection. We present a solution that can be deployed without modification of package repositories and show that automatically creating and deploying manifests as part of a repository is feasible; we hope that repositories consider deploying this enhancement.

10 ACKNOWLEDGEMENTS

We thank our shepherd, Cris Staicu, and the anonymous reviewers for their insightful feedback that greatly aided us in improving this work. This work was partially supported by a generous gift from the Google Open Source Security Team.

REFERENCES

[1] 2020. SELinux Project. https://github.com/SELinuxProject
[2] 2021. AppArmor. https://gitlab.com/apparmor/apparmor
[3] 2021. Creating Attestations with Kritis Signer | Binary Authorization. https://cloud.google.com/binary-authorization/docs/creating-attestations-kritis.
[4] 2021. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.
[5] 2021. Grafeas/Grafeas. https://github.com/grafeas/grafeas.
[6] 2021. Grafeas/Kritis. https://github.com/grafeas/kritis.
[7] 2021. NodeSource. https://docs.nodesource.com/ncmv2/docs#overview
[8] 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/li-song
[9] 2022. Open Science Framework. https://osf.io
[10] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why Do Developers Use Trivial Packages? An Empirical Case Study on Npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 385–395.
[11] Anish Athalye, Rumen Hristov, Tran Nguyen, and Qui Nguyen. 2014. *Package Manager Security*. Technical Report. https://pdfs.semanticscholar.org/d398/d240e916079e418b77ebb4b3730d7e959b15.pdf
[12] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C. van Oorschot. 2012. Understanding and Improving App Installation Security Mechanisms through Empirical Analysis of Android *(SPSM '12)*. Association for Computing Machinery.
[13] K Bertus. 2018. Cryptocurrency clipboard hijacker discovered in pypi repository. https://medium.com/@bertusk/
[14] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. 1984. *Classification and regression trees*. CRC press.
[15] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *SANER*.
[16] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. 2008. A look in the mirror: Attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and communications security*. 565–574.
[17] Seth Carmody, Andrea Coravos, Ginny Fahs, Audra Hatch, Janine Medina, Beau Woods, and Joshua Corman. 2021. Building Resilient Medical Technology Supply Chains with a Software Bill of Materials. *npj Digital Medicine* 4, 1 (Feb. 2021), 1–6.
[18] Kyriakos Chatzidimitriou, Michail Papamichail, Themistoklis Diamantopoulos, Michail Tsapanos, and Andreas Symeonidis. 2018. Npm-miner: An infrastructure for measuring the quality of the npm registry. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 42–45.
[19] Pern Hui Chia, Yusuke Yamamoto, and N Asokan. 2012. Is this app safe? A large scale study on application permissions and risk signals. In *Proceedings of the 21st international conference on World Wide Web*. 311–320.
[20] Catalin Cimpanu. 2020. Microsoft spots malicious npm package stealing data from UNIX systems. https://www.zdnet.com/article/microsoft-spots-malicious-npm-package-stealing-data-from-unix-systems/
[21] Lucian Constantin. 2020. SolarWinds Attack Explained: And Why It Was so Hard to Detect | CSO Online. https://www.csoonline.com/article/3601508/solarwinds-supply-chain-attack-explained-why-organizations-were-not-prepared.html.
[22] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact of Regular Expression Denial of Service (ReDoS) in Practice: An Empirical Study at the Ecosystem Scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/3236024.3236027
[23] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 181–191.
[24] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium*. Internet Society.
[25] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. 2019. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 13–16.
[26] François Gauthier, Behnaz Hassanshahi, and Alexander Jordan. 2018. AFFOGATO: Runtime Detection of Injection Attacks for Node.Js. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 94–99. https://doi.org/10.1145/3236454.3236502
[27] OSSF Securing Critical Projects Working Group. 2022. Package Analysis. https://github.com/ossf/package-analysis
[28] Joseph Hejderup. 2015. *In Dependencies We Trust: How vulnerable are dependencies in software modules?* Master's thesis. Delft University of Technology.
[29] Vanessa Henderson. 2017. Open-Source Packages With Malicious Content. https://www.veracode.com/blog/research/open-source-packages-malicious-intent
[30] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 121–134.
[31] J Koljonen. 2019. Warning! is rest-client 1.6.13 hijacked? https://github.com/rest-client/rest-client/issues/713
[32] R. G. Kula, C. D. Roover, D. German, T. Ishio, and K. Inoue. 2014. Visualizing the Evolution of Systems and Their Library Dependencies. In *IEEE VISSOFT*.
[33] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. 2016. Diplomat: Using delegations to protect community repositories. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 567–581.
[34] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. *Detecting Node.Js Prototype Pollution Vulnerabilities via Object Lookup Analysis*. Association for Computing Machinery, New York, NY, USA, 268–279. https://doi.org/10.1145/3468264.3468542
[35] Jeffrey Mcdonald, Nathan Herron, William Glisson, and Ryan Benton. 2021. Machine Learning-Based Android Malware Detection Using Manifest Permissions. In *Proceedings of the 54th Hawaii International Conference on System Sciences*. 6976.
[36] Benjamin Barslev Nielsen, Behnaz Hassanshahi, and François Gauthier. 2019. Nodest: Feedback-Driven Static Analysis of Node.Js Applications. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 455–465. https://doi.org/10.1145/3338906.3338933
[37] npmjs.com. [n.d.]. audit (accessed 02/2021). https://docs.npmjs.com/cli/v7/commands/npm-audit.
[38] npmjs.com. [n.d.]. npm. https://www.npmjs.com/
[39] npmjs.com. [n.d.]. npm Open-Source Terms. https://www.npmjs.com/policies/open-source-terms
[40] npmjs.com. [n.d.]. scripts (accessed 02/2021). https://docs.npmjs.com/cli/v6/using-npm/scripts.
[41] npmjs.org. [n.d.]. numeric precision matters: how npm download counts work (accessed 02/2021). https://blog.npmjs.org/post/92574016600/numeric-precision-matters-how-npm-download-counts-work.
[42] npmjs.org. [n.d.]. Package install scripts vulnerability (accessed 02/2021). https://blog.npmjs.org/post/141702881055/package-install-scripts-vulnerability.
[43] Chris O'Donnell. 2018. The 'event-Stream' Vulnerability. https://medium.com/@codfish/the-event-stream-vulnerability-6acd4c515aae.
[44] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, Cham, 23–43.

[45] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards Detection of Software Supply Chain Attacks by Forensic Artifacts *(ARES '20)*. Association for Computing Machinery, New York, NY, USA, Article 65, 6 pages.

[46] Vern Paxson. 1999. Bro: A System for Detecting Network Intruders in Real-Time. *Comput. Netw.* 31, 23-24 (Dec. 1999), 2435–2463.

[47] Brian Pfretzschner and Lotfi ben Othmane. 2017. Identification of Dependency-based Attacks on Node.Js. In *ARES*.

[48] H. Plate, S. E. Ponta, and A. Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *ICSME*.

[49] Ax Sharma. 2020. NPM Nukes NodeJS Malware Opening Windows, Linux Reverse Shells. https://www.bleepingcomputer.com/news/security/npm-nukes-nodejs-malware-opening-windows-linux-reverse-shells/

[50] Ax Sharma. 2020. Trick or Treat: That 'twilio-Npm' Package Is Brandjacking Malware in Disguise! https://blog.sonatype.com/twilio-npm-is-brandjacking-malware-in-disguise

[51] Ax Sharma. 2021. Copycats imitate novel supply chain attack that hit tech giants. https://www.bleepingcomputer.com/news/security/copycats-imitate-novel-supply-chain-attack-that-hit-tech-giants/

[52] Sindre Sorhus. 2020. Install npm packages globally without sudo on macOS and Linux. https://github.com/sindresorhus/guides/blob/main/npm-global-without-sudo.md

[53] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[54] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2018. SYNODE: Understanding and Automatically Preventing Injection Attacks on NODE.JS. In *NDSS*.

[55] Sylabs. 2020. Home | Sylabs.Io. https://sylabs.io/

[56] Matthew Taylor, Ruturaj Vaidya, Drew Davidson, Lorenzo De Carli, and Vaibhav Rastogi. 2020. Defending Against Package Typosquatting. In *International Conference on Network and System Security*. Springer, 112–131.

[57] Jørgen Tellnes. 2013. *Dependencies: No Software is an Island.* Master's thesis. The University of Bergen.

[58] Ruturaj K. Vaidya, Lorenzo De Carli, Drew Davidson, and Vaibhav Rastogi. 2019. Security Issues in Language-based Sofware Ecosystems. *CoRR* abs/1903.02613 (2019). arXiv:1903.02613 http://arxiv.org/abs/1903.02613

[59] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing Dynamic Library Compromise on Node.Js via RWX-Based Privilege Reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1821–1838. https://doi.org/10.1145/3460120.3484535

[60] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. 2012. Droidmat: Android malware detection through manifest and api calls tracing. In *2012 Seventh Asia Joint Conference on Information Security*. IEEE, 62–69.

[61] A. A. Younis, Y. K. Malaiya, and I. Ray. 2014. Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. In *HASE*.

[62] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 995–1010.

A DEFAULT DEVELOPER POLICY

```
{
"declarations": [
    "<anyTimedOut> = [timedOut_preinstall] || [timedOut_install] || [timedOut_postinstall] ||
    [timedOut_preuninstall] || [timedOut_uninstall] || [timedOut_postuninstall]",
    "<allRemoteHosts> = [remoteHosts_preinstall] ~union [remoteHosts_install]
    ~union [remoteHosts_postinstall] ~union [remoteHosts_preuninstall] ~union [remoteHosts_uninstall]
    ~union [remoteHosts_postuninstall]",
    "<allLocalHosts> = [localHosts_preinstall] ~union [localHosts_install] ~union [localHosts_postinstall]
    ~union [localHosts_preuninstall] ~union [localHosts_uninstall] ~union [localHosts_postuninstall]",
    "<allLocalNetworkHosts> = [localNetworkHosts_preinstall] ~union [localNetworkHosts_install]
    ~union [localNetworkHosts_postinstall] ~union [localNetworkHosts_preuninstall]
    ~union [localNetworkHosts_uninstall] ~union [localNetworkHosts_postuninstall]",
    "<allMetadataMods> = [metadataMods_preinstall] ~union [metadataMods_install]
    ~union [metadataMods_postinstall] ~union [metadataMods_preuninstall]
    ~union [metadataMods_uninstall] ~union [metadataMods_postuninstall]",
    "<allFilesWritten> = [filesWritten_preinstall] ~union [filesWritten_install]
    ~union [filesWritten_postinstall] ~union [filesWritten_preuninstall]
    ~union [filesWritten_uninstall] ~union [filesWritten_postuninstall]",
    "<allFilesRenamed> = [filesRenamed_preinstall] ~union [filesRenamed_install]
    ~union [filesRenamed_postinstall] ~union [filesRenamed_preuninstall]
    ~union [filesRenamed_uninstall] ~union [filesRenamed_postuninstall]",
    "<allFilesDeleted> = [filesDeleted_preinstall] ~union [filesDeleted_install]
    ~union [filesDeleted_postinstall] ~union [filesDeleted_preuninstall]
    ~union [filesDeleted_uninstall] ~union [filesDeleted_postuninstall]",
    "<allFilesCreated> = [filesCreated_preinstall] ~union [filesCreated_install]
    ~union [filesCreated_postinstall] ~union [filesCreated_preuninstall]
    ~union [filesCreated_uninstall] ~union [filesCreated_postuninstall]",
    "<allFilesRead> = [filesRead_preinstall] ~union [filesRead_install] ~union [filesRead_postinstall]
    ~union [filesRead_preuninstall] ~union [filesRead_uninstall] ~union [filesRead_postuninstall]",
    "<<Terminal>> = '\/dev\/(pts\/[0-9]*|tts[0-9]*|null)’”,
    "<<sensitive>> = ['\/etc\/passwd', '\/etc\/shadow', '\/etc\/group', '\/etc\/gshadow',
    '\/etc\/profile', '\/etc\/pam[.]d.*', '\/proc\/cmdline', '\/etc\/system[.]d', '\/etc\/rc[.].*',
    '\/etc\/init[.].*', '\/dev\/shm\/instances\/[.]bash_profile',
    '\/volatile\/instances\/[.]bash_profile', '\/nfs\/volatile\/instances\/[.]bash_profile',
    '\/home\/user\/[.]bash_profile', '\/dev\/shm\/instances\/[.]bash_login',
    '\/volatile\/instances\/[.]bash_login', '\/nfs\/volatile\/instances\/[.]bash_login',
    '\/home\/user\/[.]bash_login', '\/dev\/shm\/instances\/[.]profile',
    '\/volatile\/instances\/[.]profile', '\/nfs\/volatile\/instances\/[.]profile',
    '\/home\/user\/[.]profile', '\/dev\/shm\/instances\/[.]bashrc', '\/volatile\/instances\/[.]bashrc',
    '\/nfs\/volatile\/instances\/[.]bashrc', '\/home\/user\/[.]bashrc', '\/dev\/shm\/instances\/[.]ssh.*',
    '\/volatile\/instances\/[.]ssh.* ', '\/nfs\/volatile\/instances\/[.]ssh.*', '\/home\/user\/[.]ssh.*',
    '\/etc\/bash[.]bashrc', '\/etc\/profile[.]d.*', '\/etc\/hosts', '\/etc\/resolv[.]conf']"
],
"rulesFail": [
    "!<anyTimedOut>",
    "<allRemoteHosts> == {}",
    "<allLocalNetworkHosts> == {}",
    "<allLocalHosts> == {}",
    "<allMetadataMods> == {}",
    "<allFilesWritten> ~matchesall [<<Terminal>>]",
    "<allFilesRenamed> == {}",
    "<allFilesDeleted> == {}",
    "<allFilesCreated> == {}",
    "!(<allFilesRead> ~anymatches [<<sensitive>>])"
],
"rulesWarn": []
}
```