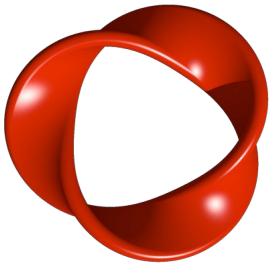

Secfault Security

1Password Mac Desktop App

Security Assessment

Report
FINAL

for

Agilebits Inc dba 1Password

4711 Yonge St., 10th Floor
Toronto,ON M2N 6K8 AgileBits

- hereafter called "Agilebits Inc dba 1Password" -

Document History

Version	Author	Date	Comment
0.1	Maik Münch	2022-04-13	Initial Draft Version
0.2	Gregor Kopf	2022-04-19	Additions
0.3	Dirk Breiden	2022-04-21	Internal Review
0.4	Gregor Kopf	2022-05-15	Included Customer Feedback
0.5	Dirk Breiden	2022-05-16	Internal Review
1.0	Gregor Kopf	2022-11-03	Final Version

Table of Contents

1 Executive Summary

Secfault Security was tasked by Agilebits Inc dba 1Password with a security review of selected components of the 1Password ecosystem, namely the new macOS application offering the "Universal Autofill" feature. The review has been performed in the time frame from 2022-04-04 to 2022-04-20. This document describes the results of the project.

During the review a number of issues, which are described in detail in section 4 of this document, have been identified. The more severe issues include lax checks in the updater implementation, which might be exploited by local attackers in order to elevate their privileges. Furthermore, issues in the code integrity checking logic have been identified, which could allow local attackers to perform modifications to the installed 1Password application. Other identified issues include weak authorization checks for XPC services.

Section 5 of this document provides a number of additional observations and recommendations for further strengthening the security aspects of the solution.

Overall, the reviewed codebase left a positive impression. The code is well-structured and readable and a large number of common possible security issues have been avoided. This indicates that the code has been implemented with security in mind.

After having received a draft version of this document, Agilebits Inc dba 1Password provided feedback on the identified issues, which can be found in section 6 of this document.

2 Overview

1Password is a password manager product developed and maintained by AgileBits Inc. The solution provides a secure place for customers to store various passwords, software licenses, and other sensitive information in virtual vaults.

Agilebits Inc dba 1Password tasked Secfault Security with a review of the new macOS version of the 1Password software, which includes a new feature called "Universal Autofill".

In section 2.1 of this document, a description of the project's scope is provided. Section 2.2 provides details on the test procedures.

2.1 Target Scope

The following source code repositories have been provided by Agilebits Inc dba 1Password for review:

- `core` at revision `5ab5ac`

Furthermore, Agilebits Inc dba 1Password provided the respective binaries for the above mentioned revision.

2.2 Test Procedures

The overall project followed a white-box approach, which means that Agilebits Inc dba 1Password provided the source code, the compiled binaries and technical documentation for the solution. Therefore, the solution has been analyzed by performing a source code review, as well as targeted dynamic testing.

The source code review has been performed in a manual fashion, i.e., without relying on automated vulnerability scanners or similar tools. Besides identifying possible classical implementation weaknesses, one main focus of the review was the identification of potential logic problems. This requires an in-depth understanding of the solution's inner workings, which is best achieved by a manual process.

The dynamic tests have been performed in a targeted fashion. On the one hand, this served the purpose of validating issues identified during the source code review. On the other hand, dynamic tests were also performed to obtain a better understanding of the overall solution and the interplay of its individual components.

2.3 Project Execution

The project has been executed in the time frame from 2022-04-04 to 2022-04-20 in 14 person days.

The consultants assigned to this projects were:

- Maik Münch
- Leonard König
- Gregor Kopf

3 Result Overview

An overview of the project results is provided in the following table.

Description	Chapter	Type	Severity
Integrity Verification Bypass (Unpacked App)	4.1	Code	Medium
Missing Quotes in Shell Command	4.2	Code	Not Exploitable
Weak XPC Client Validation	4.3	Design	Medium
Missing Focus Check in AutoType Implementation	4.4	Code	Low
Non-Atomic Verification Logic	4.5	Design	High
Symlink Attack in Updater Implementation	4.6	Code	High

Each identified issue is briefly described by its title, its type, its exploitability and by the impact of a successful exploitation. Technical details for the individual issues are provided in the respective sections of chapter 4 of this document. Details regarding the vulnerability rating scheme used in this document are provided in section 7.

4 Results

The issues identified during the project are described in detail in the following sections. For each finding, there is a technical description, recommended actions and - if necessary and possible - reproduction steps. For details regarding the used vulnerability rating scheme, please refer to section 7 of this document.

4.1 Integrity Verification Bypass (Unpacked App)

Summary

Type	Location	Severity
Code	ffi/core-node/src/ integrity_verification.rs	Medium

Technical Description

The 1Password application is built on top of the Electron framework. The implementation is partially done in native code, which is loaded as a shared object; other parts of the code are implemented in JavaScript. While the native code parts are signed using Apple's code signing technology, the script files are compiled into an archive (`app.asar`), which itself is not signed. Therefore, the code contains an integrity checking functionality, which aims to detect modifications to the `app.asar` file.

The evaluation of this integrity verification revealed that only packaged applications are subject to verification performed during load time of the core library.

The following excerpts from `ffi/core-node/src/integrity_verification.rs` illustrate that resource integrity verification is only performed on packaged apps and how packed apps are identified:

```
#[cfg(target_os = "macos")]
pub(crate) fn verify_asar_integrity(current_exe: &Path) -> Result<(), ()> {
    use op_crypto::blake3_unkeyed;

    // Unpackaged apps don't have ASAR archives.
    if !is_packaged(current_exe) {
        return Ok(());
    }
...
}
...
/// XXX: Should this be moved to `op-sys-info`?
#[cfg(target_os = "macos")]
```

```rust
#[inline]
pub(crate) fn is_packaged(current_exe: &Path) -> bool {
    // If the filename doesn't exist, somehow, then default to behaving like
the app is packaged.
    #[cfg(unix)]
    {
        current_exe
            .file_name()
            .map(|file_name| !file_name.eq_ignore_ascii_case("electron"))
            .unwrap_or(true)
    }

    #[cfg(windows)]
    {
        current_exe
            .file_name()
            .map(|file_name| !file_name.eq_ignore_ascii_case("electron.exe"))
            .unwrap_or(true)
    }
}
```

Please observe that if the `current_exe` is not considered to be packaged, integrity verification is not performed. An application is considered to be unpacked if its lower-case `file_name` is equal to `electron` respectively `electron.exe`.

By creating a hard link named `electron` to the `1Password` executable and executing the hard link instead of the original the integrity verification can be bypassed by a local attacker. This might enable an attacker to manipulate the `app.asar` contents to execute arbitrary JavaScript code in the context of the 1Password application circumventing potential mitigations.

Furthermore, this might allow malware to hide and persist malicious code.

Recommended Action

In order to mitigate this issue, it is advised to reconsider removing the "unpacked app" logic in release builds. Please also be aware of the issue described in section 4.5 of this document, which describes another issue in the integrity verification scheme and provides additional recommendations and remarks.

Reproduction Steps

To reproduce this issue please execute the following steps:

1. Start and exit the 1Password application
2. In a terminal, please install the `asar` Node package by e.g., executing `sudo npm install -g --engine-strict asar`
3. Then, please execute the script below in a terminal using e.g., `bash poc.sh`

4 Now, please observe that the console output indeed includes output generated by code inserted into the `main.js` file of the `app.asar` archive.

The following shell script (`poc.sh`) bypasses integrity verification and inserts JavaScript code executed on application startup and prints a message to the console.

```
cd /tmp
cp -r /Applications/1Password.app/Contents/Resources/app.asar* .
node /usr/local/lib/node_modules/asar/bin/asar.js extract app.asar
app.asar.unpacked
rm app.asar.unpacked/index.node
cp /Applications/1Password.app/Contents/Frameworks/index.node app.asar.unpacked
sed -i -e 's/setupMenu=()=>{if(!this/setupMenu=()=>{console.log("Hello from
main.js");if(!this/' app.asar.unpacked/main.js
node /usr/local/lib/node_modules/asar/bin/asar.js pack app.asar.unpacked
app.asar
cd -
cp /tmp/app.asar /Applications/1Password.app/Contents/Resources/
cd /Applications/1Password.app/Contents/MacOS
ln 1Password electron
./electron
```

4.2 Missing Quotes in Shell Command

Summary

Type	Location	Severity
Code	core/apple/macOS/FileManager+Authorization.swift	Not Exploitable

Technical Description

The review of the update logic revealed an unquoted variable used in a shell command. The following excerpt from the file `apple/macOS/FileManager+Authorization.swift` shows the respective part of the code:

```
private func trashItemAtPathWithForcedAuthorization(at url: URL) -> Bool {
...
  let trashFolder =
self.homeDirectoryForCurrentUser.appendingPathComponent(".Trash")

  if self.fileExists(atPath: trashFolder.path) {
    self.setenvWithString("TRASH_FOLDER", trashFolder.path, 1)
    return runShellWithAuthorization("1Password needs to update some of its
files, which requires the password you use to log in to your Mac.", "/bin/mv -f
\"$FILE_PATH\" \"$TMP_PATH\" && /bin/mv \"$TMP_PATH\" $TRASH_FOLDER")
  }
  else { // can't find trash, delete old version instead
  ...
  }
}
```

The `$TRASH_FOLDER` variable in the executed shell command is, contrary to the other variables, not quoted. Being user controlled to some degree, this might lead to issues such as command and argument injection or problematic globbing.

Being able to control the `$TRASH_FOLDER` variable, an attacker might be able to execute commands on behalf of the update process with elevated privileges ultimately leading to privilege escalation. These elevated privileges might then be used to execute subsequent attacks against the user and might help compromising 1Password's security posture.

It has to be noted that the affected function is currently not called within the code base and therefore cannot be exploited. However, it cannot be excluded that this function will be used in future iterations of the application and should therefore be addressed nonetheless.

Recommended Action

For the mitigation of this issue it is advised to quote the respective variables as a first step. Further, it should be ensured that variables used in shell scripts do not contain shell meta characters such as `*`, `?`, `;` and so on to avoid potential argument injection issues.

Ultimately, it should be avoided to rely on shell functionality whenever the same functionality could be implemented by library functionality of the programming language in question. This might help to eliminate multiple bug classes from emerging in the first place.

Reproduction Steps

This issue has been identified during a static source code review and has not been reproduced dynamically. Hence, no reproduction steps can be provided.

4.3 Weak XPC Client Validation

Summary

Type	Location	Severity
Design	XPC Server	Medium

Technical Description

During analysis of the XPC client validation it was identified that clients are presumably validated by their corresponding team identifier. If the team identifier of the client matches the one of the XPC's binary, clients are able to communicate with the XPC server and presumably invoke methods that might undermine the solutions security posture.

Please consider the following excerpt from `core/apple/CoreFoundation/CoreFoundation/ProcessValidation.swift`:

```
public static func verifySignatureOfSelfMatchesSignature(of client: SecureCode)
-> Bool {
  func signingInformation(_ client: SecureCode) -> [String: Any]? {
    switch client {
      case .secCode(let client):
        return CodeSignature.signingInformation(of: client)
      case .secStaticCode(let staticClient):
        return CodeSignature.signingInformation(of: staticClient)
    }
  }

  guard Self.verifySignedWithAppleCert(client),
    let clientSigningInfo = signingInformation(client),
    let clientTeamIdentifier = clientSigningInfo[kSecCodeInfoTeamIdentifier as
String] as? String else {
    os_log(.debug, "Failed to grab code signature information about the
client.")
    return false
  }

  // Grab information from ourselves.
  guard let selfClient = CodeSignature.selfCodeSignature,
    let selfSigningInfo = CodeSignature.signingInformation(of: selfClient),
    let selfTeamIdentifier = selfSigningInfo[kSecCodeInfoTeamIdentifier as
String] as? String else {
    os_log(.debug, "Failed to grab code signature information about
ourselves.")
    return false
  }
```

```
  let isEqual = clientTeamIdentifier == selfTeamIdentifier
  os_log(.debug, "Code signature team id of client == ourselves: %{public}@",
String(describing: isEqual))
  return isEqual
}
```

The `verifySignatureOfSelfMatchesSignature` function simply compares the team identifier in the code signatures of itself with the team identifier of the client. Please note that no version checks are performed and such were not identified to be performed anywhere else in the code base during static analysis.

While a local attacker cannot inject code into the newest versions of the 1Password application due to application of the Hardened Runtime capability, they might be able identify older versions of software with a matching team identifier that did not enable hardened runtime.

Due to the limited time budget and missing access to a software version which fulfills the mentioned requirements no dynamic tests were performed to validate this issue.

Recommended Action

To address this issue it is recommended to authenticate connecting processes by their team and bundle identifier and a minimum version should be enforced to ensure that mitigations such as the hardened runtime have been enabled.

To further improve the implementation and to mitigate potential issues around PID wraparounds it should be considered to validate the connecting processes' `audit_token` s.

Reproduction Steps

This issue has been identified during a static source code review and has not been reproduced dynamically. Hence, no reproduction steps can be provided.

4.4 Missing Focus Check in AutoType Implementation

Summary

Type	Location	Severity
Code	1Password Autofill/Brain/Brain.swift	Low

Technical Description

While reviewing the autofill implementation for macOS, it was found that the code performs a number of checks in order to ensure it targets the correct application window before sending keystrokes. The following excerpt from `Brain.swift` illustrates this:

```
                              // Step 3:
                              // Fill the operation value
                              do {
                                    try autoTypeStringNative(value:
operation.value, element: element)
                              }
                              catch {
                                    NSLog("@Autofill: Focused window changed
while performing auto-type string native")
                                    return
                              }
[...]
      // Auto-submit
      // TODO: Use post-fill action from Brain instead
      do {
            try await Task.sleep(nanoseconds: ONE_MS * 100)
      }
      catch {
            //
      }
      CoreLogging.log("typing enter")
      autoTypeEnter()
```

Please observe that the code in `autoTypeStringNative` contains the following check:

```
      for char in value {
            if !element.isFocused() {
                  throw AutoTypeStringError.elementLostFocus
            }
```

However, the implementation of `autoTypeEnter` does not:

```
func autoTypeEnter() {
```

```
      // Enter down
      let enterDown = CGEvent(keyboardEventSource: nil, virtualKey:
CGKeyCode(kVK_Return), keyDown: true)
      enterDown?.flags.remove(MODIFIER_KEYS)

      // Enter up
      let enterUp = CGEvent(keyboardEventSource: nil, virtualKey:
CGKeyCode(kVK_Return), keyDown: false)
      enterUp?.flags.remove(MODIFIER_KEYS)

      enterDown?.post(tap: .cghidEventTap)
      enterUp?.post(tap: .cghidEventTap)

      CGEvent(source: nil)?.post(tap: .cghidEventTap)
}
```

While the code repeatedly checks the currently focused window when auto-typing a string, it does not perform this check before pressing the enter key. Additionally, the code adds a 100ms delay before auto-typing the enter key. This might lead to sending keystrokes to windows that are not intended to receive them.

Exploiting this situation is however not completely trivial. One option for exploitation could be to focus a window such as a confirmation dialog, right before the enter key is sent by the code. However, without sufficient control over the target system, such an attack is not easily implemented. It should however be noted that besides an actively malicious attack, there could also be random circumstances (e.g., dialogs popping up right before the enter key is about to be pressed), which would likely have a negative impact on the user's experience.

Recommended Action

The general design of the autotype-based password filling solution does not allow to send keystrokes directly to a window - instead, key strokes are sent to the currently focused window. While the code aims to mitigate possible problems by checking the focused window before sending keystrokes, such checks are inherently prone to race conditions. That said, such race conditions might be hard to exploit for attackers without sufficient privileges on the target system, as such attackers would need to accurately time window focus switches.

In order to tighten this mitigation, it is hence advised to also check the currently focused window before generating the key press event for the enter key.

Reproduction Steps

This issue has been identified during a static source code review and has not been reproduced dynamically. Hence, no reproduction steps can be provided.

4.5 Non-Atomic Verification Logic

Summary

Type	Location	Severity
Design	ffi/core-node/src/ integrity_verification.rs	High

Technical Description

As described in section 4.1 of this document, the 1Password application implements an integrity checking logic, which aims to detect modifications to the `app.asar` file. While reviewing this implementation, it was found that is generally prone to race conditions.

The integrity checking logic for the `app.asar` file is invoked after the native code has been loaded; however, it cannot guarantee that the `app.asar` file it verifies is the same `app.asar` file that would be used by the Electron framework. In other words, the implementation does not act atomically when verifying/loading the `app.asar` file. This is a general shortcoming of the integrity checking implementation, which might be exploitable in order to perform modifications to the `app.asar` file.

Recommended Action

This issue cannot easily be addressed without changing the overall verification logic. Secfault Security is aware of the fact that this is a non-trivial change. A robust implementation would involve creating a signature on the `app.asar` file, which is checked at load time by the Electron framework. One possible work-around could be to prevent users from modifying the `app.asar` file by making use of file permissions. If, for example, the 1Password application was installed to a location that is not writable by regular users, this would at least prevent attackers with user privileges (e.g., in a malware scenario) from performing such modifications.

Reproduction Steps

This issue has been identified during a static source code review and has not been reproduced dynamically. Hence, no reproduction steps can be provided.

4.6 Symlink Attack in Updater Implementation

Summary

Type	Location	Severity
Code	core/apple/macOS/FileManager+Authorization.swift	High

Technical Description

While reviewing the 1Password updater implementation, it was found that the code does not properly take into account that an attacker might have modified the source and/or destination directories, so that they contain symbolic links. Please observe the following excerpt from `FileManager+Authorization.swift`:

```
            if FileManager.default.fileExists(atPath: dst) {
                  shellCommand = String(format: """
                                                                  /bin/rm -
rf \"$DST_PATH\" &&
                                                                  /bin/mv -
f \"$SRC_PATH\" \"$DST_PATH\" &&

/usr/sbin/chown -R %d:%d \"$DST_PATH\"
                                                 """,
                                                 sb.st_uid, sb.st_gid)
```

The code executes a shell command for moving an updated version of the 1Password application to its destination directory. Please note that this code is executed using the (deprecated) `AuthorizationExecuteWithPrivileges` function and hence runs with `root` privileges. It can be observed that the code contains a number of issues. On the one hand, the invocation of the `rm` command is not atomically tied to the `mv` command: after the `rm` command succeeded, the `$DST_PATH` directory could have been re-created by an attacker.

Furthermore, please note that the code does not check for the presence of symbolic links. If, for instance, `$DST_PATH` was a symbolic link pointing to another directory (e.g., to `/etc`), the code would copy files to the `/etc` directory. This might allow local attackers to elevate their privileges.

Recommended Action

Generally, it is recommended to reconsider the approach of executing shell commands with root privileges. If this is not possible, one mitigation could be to first change the ownership and access permissions of the source and target files, so that only the `root` user can access them, check that no links are present and subsequently replace the files in the destination directory.

Reproduction Steps

This issue has been identified during a static source code review and has not been reproduced dynamically. Hence, no reproduction steps can be provided.

5 Additional Observations

Secfault Security would like to point out a number of general observations and recommendations regarding the analyzed system in the following subsections.

5.1 Missing Validation of Shell Command Input

A cursory evaluation of used shell commands within the code based revealed that variables are not always validated before being used to construct a command being executed by an external shell. For example the following excerpt from the file `apple/macOS/1Password Autofill/Brain/Brain.swift` shows an instance of this:

```
func appleScriptToGetTerminalTty(bundleId: String) -> String {
  switch bundleId {
    case "com.apple.Terminal":
      return """
      tell app id "com.apple.Terminal"
          set myTty to tty of tab in window 1
      end tell

      return myTty
      """
  ...
  }
}

func collectTerminalElement(app: NSRunningApplication) -> AXUIElement? {
...
  guard let tty = scriptRunner.executeAndReturnError(&error).stringValue else {
    CoreLogging.log("Script Error: %@", String(describing: error))
    return .none
  }

  let hasEcho = shell("/bin/stty -f \(tty) | /usr/bin/grep 'icanon'")
  if !hasEcho.isEmpty {
    CoreLogging.log("Terminal in echo mode, filling disabled")
    return .none
  }
...
```

The result of the AppleScript invocation to retrieve the `tty` of a terminal is used to construct a shell command without prior validation. This might lead to potential command or argument injection issues and is considered to be best practice when working with shell scripts.

Secfault Security would like to recommend to validate all input to shell commands. If possible, it is advised to not directly execute commands through the shell as this is prone to missing quotation or

escaping of arguments. If executing external commands is required, it is advised to directly invoke the respective command using the `execv()` function family.

5.2 Exposure of Unsafe Functions to Frontend Code

While analyzing the implementation for possible exploitation paths for the issues described in sections 4.1 and 4.5, it was found that the 1Password application exposes functionality to the JavaScript frontend code, which might be abused in order to trigger memory corruption issues. One obvious instance is the function `unsafe_set_vibrancy`, which takes a window handle as an argument. A cursory inspection revealed that by providing an invalid window handle (e.g., the value `0x4141414141414141`), it appears to be possible to at least trigger an invalid memory access in the 1Password application.

While the application makes use of code signing and the hardened runtime feature, triggering and exploiting memory corruption issues might be a possible way for executing untrusted native code inside the application's memory space. This in turn would for instance allow for XPC communication with privileged helper services, such as the autofill service.

It is therefore recommended to review the functions exposed to the frontend JavaScript code for such patterns and to enforce a strict validation of their arguments. Ideally, the JavaScript code should not be able to provide "raw" arguments that are subsequently used to perform memory accesses.

5.3 Use of PIDs for Security Checks

While reviewing the autofill implementation, it was found that the code makes use of process IDs for performing security checks. For instance, please consider the following code excerpt from `1Password Autofill/Brain/Brain.swift`:

```
	let leafProcessPid = procInfo.first {
		!processPpids.contains($0.kp_proc.p_pid)
	}.map {
		$0.kp_proc.p_pid
	}

	guard let pid = leafProcessPid else {
		CoreLogging.log("Failed to get leaf process PID")
		return .none
	}

	let absolutePathToExecutable: String
	switch absoluteExecutablePathForProcess(pid: pid) {
		case .success(let path):
			absolutePathToExecutable = path
```


```
            case .failure(let e):
                  CoreLogging.log("Failed to get absolute path to executable for
process: %@", String(describing: e))
                  return .none
      }

      // Confirm that the leaf process is /usr/bin/sudo
      if absolutePathToExecutable != "/usr/bin/sudo" {
            CoreLogging.log("Absolute path to executable running in tty was
not /usr/bin/sudo, filling disabled")
            return .none
      }
```

It can be observed that the code aims to check whether `/usr/bin/sudo` is currently running in a terminal window, in order to ensure that auto-filling a users password is only performed when the legitimate `sudo` binary prompts for the password. However, on the one hand this scheme is inherently racy: it cannot guarantee that after checking the running binary, `sudo` still remains running. Furthermore, it can be observed that the checks are performed based on the progress ID of the program running in the target terminal. It should be noted that using process IDs for such purposes generally comes with the risk of possible PID wrap attacks. While this might not pose a direct threat in the above scenario, it should generally be highlighted that a more solid approach would be to use audit tokens.

6 Customer Feedback

After receiving a draft version of this document, Agilebits Inc dba 1Password reviewed the identified issues and provided feedback, describing their assessment. In order to provide full transparency, this feedback is included in the below sections.

6.1 Integrity Verification Bypass (Unpacked App) (Finding 4.1)

We have accepted the issue as a low severity issue. We are looking to include improved validations that prevent the identified issue in 1Password for Mac 8.8.

6.2 Missing Quotes in Shell Command (Finding 4.2)

We have accepted the issue as a best practice issue. 1Password for Mac 8.7.1 will contain correct shell quotes that prevent the identified issue.

6.3 Weak XPC Client Validation (Finding 4.3)

We have accepted the issue as a best practice issue. 1Password for Mac 8.7.1 will use audit tokens to validate XPC connections.

6.4 Missing Focus Check in AutoType Implementation (Finding 4.4)

We have accepted the issue as a best practice issue. 1Password for Mac 8.7.1 will contain improved validations at the time the filling action is performed.

6.5 Non-Atomic Verification Logic (Finding 4.5)

We have accepted the issue as a low severity issue. We were previously aware of this limitation in our local verification logic, and have worked with the Electron team to come up with an improved method of verifying Electron resources. A future version of 1Password 8 will contain these validations.

6.6 Symlink Attack in Updater Implementation (Finding 4.6)

We have accepted the issue as a best practice issue. We are discussing potential solutions to this fix, and anticipate that a future version of 1Password will contain mitigations.

7 Vulnerability Rating

This section provides a description of the vulnerability rating scheme used in this document. Each finding is rated by its type and its severity. The meaning of the individual ratings are provided in the following sub-sections.

7.1 Vulnerability Types

Vulnerabilities are rated by the types described in the following table.

Type	Description
Configuration	The finding is a configuration issue
Design	The finding is the result of a design decision
Code	The finding is caused by a coding mistake
Observation	The finding is an observation, which does not necessarily have a direct impact

7.2 Severity

The severity of a vulnerability describes a combination of the likelihood of attackers exploiting the vulnerability, and the impact of a successful exploitation.

Severity Rating	Description
Not Exploitable	This finding can most likely not be exploited.
Low	The vulnerability is either hard to exploit (e.g., because a successful exploitation requires significant prerequisites) or its consequences can be considered benign.
Medium	The vulnerability can be exploited (possibly under certain preconditions) and a successful exploit can be used to at least partially bypass the security guarantees of the solution.
High	The vulnerability can be exploited easily and a successful exploit bypasses one of the core security properties of the solution.
Critical	The vulnerability can be exploited easily and a successful exploit can be used to compromise systems beyond the scope of the analysis.

8 Glossary

Term	Definition
ID	Identification