Secfault Security

2023 1Password CLI Pentest

Security Assessment

Report

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	Finn Westendorf	2023-05-05	First Draft
0.2	Maik Münch	2023-05-16	Additions
0.3	Jennifer Gehrke	2023-05-17	Additions
0.4	Gregor Kopf	2023-07-24	Added Customer Feedback
1.0	Gregor Kopf	2023-08-14	Final Version

Table of Contents

1 Executive Summary

Secfault Security was commissioned by Agilebits Inc dba 1Password with a retest of the security issues identified in their Command-line tool (1Password CLI) in March 2022. In addition, new features and commands, which are related to the CLI usage, were specified as subjects for further inspections. The audit has been performed in the time frame from 2023-05-02 to 2023-05-17. This document describes the results of the project.

Details on the project's scope and chosen testing approaches can be found in the sections 2.1.1 and 2.1.2. The results of the performed retests are listed in section 4.1. Four of the seven issues are considered to be addressed properly, while two mitigations were found to be incomplete. The inspection of one issue led to the discovery of a new related weakness that affects the IPC peer verification on Windows and is documented in section 4.2.1. A bypass for the approach chosen to prevent Time-of-check-to-time-of-use (ToCToU) attacks during the Authenticode signature validation was detected when using special file system types. Apart from this, no new security issues could be determined. One general recommendation related to the use of deprecated API endpoints, as observed in the course of testing the new item sharing features, was added in section 5.

The reviewed codebase left a positive impression. The code is well-structured and readable and has been implemented with security in mind.

2 Overview

1Password is a password manager product developed and maintained by Agilebits Inc dba 1Password Inc. The solution provides a secure place for customers to store and share various passwords, software licenses, and other sensitive information in virtual vaults. Agilebits Inc dba 1Password tasked Secfault Security with a retest and inspection of new features related to the 1Password CLI offered for software developers. In section 2.1.1 of this document, a description of the project's scope can be found. Section 2.1.2 provides details on the test procedures.

2.1.1 Target Scope

As part of the current security audit, seven issues identified during the project executed in March 2022[1] should be retested. Those correspond to the following sections of the original document:

- 4.1.1 Lax Parsing for Dotenv Files
- 4.1.2 Output of Escape Sequences
- 4.1.3 Race Condition in File Creation
- 4.1.4 Access to Parent Environment in op run
- 4.1.5 Missing Privilege Dropping in op run
- 4.1.6 Secret Data in Command Arguments
- 4.1.7 ToCToU Weakness in Windows Peer Verification

Secfault Security was further provided by Agilebits Inc dba 1Password with an open scope covering various new features related to the CLI usage. Internal documentation was made available for more complex aspects to enable Secfault Security to perform a selection of target topics themselves. This was based on the expected risk and probability of implementation flaws. Precedence was given to features that were not yet included in a security assessment.

The below aspects were communicated by Agilebits Inc dba 1Password as a basis for the project scope:

- Rate limiting based on throttling token introduced for Service accounts
- Utilization of database read replicas
- CLI support for item sharing
- CLI command for generating SSH keys
- Change from optional to mandatory use of the internal cache
- New CLI account type: Service account (previously assessed)

1 https://bucket.agilebits.com/security/SecfaultSecurity_Report_OP_Security_Assessment_v1.0.pdf

- CLI plugin support (previously assessed)

The next section provides details on the analyzed topics and the extent of the performed tests.

2.1.2 Test Procedures

As in previous iterations, the overall project was executed using a white-box approach. This means that source code, compiled binaries and technical documentation for the solution were provided upfront by Agilebits Inc dba 1Password. Therefore, the solution has been analyzed by performing a source code review in combination with targeted dynamic testing.

Secfault Security, in general, performs source code reviews 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. Given the overall good software quality, such an approach is assumingly more effective.

Prior exposure to the source code in previous engagement was used to efficiently identify relevant parts of the solution respective to the defined target scope and allowed for more in-depth review.

In addition to static analysis, 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. For example, the 1Password session analyzer plugin for Burp Suite[2] was used to analyze traffic generated by the CLI and web application. Further, an internal CLI build allowing to send raw requests that was provided by Agilebits Inc dba 1Password was utilized to validate implemented mitigations.

Being part of this assessment's scope, seven issues, identified in a previous audit[3] performed by Secfault Security in March 2022, were retested. During the retest it was identified that the issue regarding Windows peer verification might not be fully addressed. More details on this issue can be found in section 4.1.7 and 4.2.1 of this document.

Apart from the retest, the throttling token implementation was subject to an in-depth review. The generation of throttling-related secrets and UUIDs was inspected for cryptographic issues, such as using cryptographically insecure pseudo random number generators. Further, the throttling process was evaluated with focus on token verification, e.g., ensuring token expiry, as well as potential bypasses of the configured rate limits. Dynamic tests were performed to assess the general effectiveness of this active defense mechanism. No issues were identified during the audit with regard to this feature.

2 https://github.com/1Password/burp-1password-session-analyzer
3 https://bucket.agilebits.com/security/SecfaultSecurity_Report_OP_Security_Assessment_v1.0.pdf

Further, the item sharing functionality of the CLI tool was reviewed with focus on general implementation flaws. For this, a static analysis as well as dynamic tests were performed, e.g., to identify potential bypasses of potentially configured recipient allow lists. Additionally, the used API endpoints were inspected with the aim to correlate them with the API endpoints used for example by desktop applications. This revealed, that the CLI assumingly uses deprecated endpoints. Although no direct security implications could be identified when reviewing the endpoints in question, this fact has been documented as an additional observation in section 5 of this report for the sake of completeness.

Service accounts were identified to be another interesting area and therefore were assessed during this audit. Given the complex implementation and the fact that service accounts have been audited in a previous assessment[4] already, this feature could not be fully covered during project execution. Therefore, a subset of interesting areas was selected for inspection. First and foremost, the addition of rate limiting and the accompanying addition of a throttling key to the service account credentials has been evaluated. Service account creation and secret generation were analysed statically with focus on cryptographic flaws. Further, dynamic testing was used to evaluate the general implementation. These tests included for example privilege escalations and token revocation. Please note, that the service account feature could only be covered briefly, mainly focussing on the addition of rate limiting. Thus, no general conclusion can be provided regarding this feature's security posture.

In addition, the implementation of the new `ssh generate` CLI command was reviewed. This was found to offer limited clear functionality, selecting strong cryptographic primitives with regards to random generation and the supported SSH key types. No concerns raised during the inspection.

2.2 Project Execution

The project has been executed in the time frame from 2023-05-02 to 2023-05-17.

The consultants assigned to this project were:

- Finn Westendorf
- Maik Münch
- Jennifer Gehrke
- Gregor Kopf

4 https://bucket.agilebits.com/security/378.2202.Recurity_Labs-Report-Service_Accounts-v1.2.pdf

3 Result Overview

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

Description	Chapter	Type	Severity	Status
Lax Parsing for Dotenv Files	4.1.1	Code	Low	Partially Fixed
Output of Escape Sequences	4.1.2	Code	Low	Closed
Race Condition in File Creation	4.1.3	Code	Low	Partially Fixed
Access to Parent Environment in op run	4.1.4	Code	Low	Won't Fix
Missing Privilege Dropping in op run	4.1.5	Code	Medium	Closed
Secret Data in Command Arguments	4.1.6	Code	Low	Closed
ToCToU Weakness in Windows Peer Verification	4.1.7	Code	Medium	Closed
Windows Peer Verification depends on File System Type	4.2.1	Code	Medium	

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 Retest

This section lists all issues that were identified during the CLI tool security audit in March 2022 and were retested in the current assessment.

4.1.1 Lax Parsing for Dotenv Files

Summary

Type	Location	Severity	Status
Code	godotenv (third-party dependency)	Low	Partially Fixed

Technical Description

While reviewing the dotenv file parsing of the `op inject` command, it was found that the used parser implementation is rather lax. A cursory inspection of the parser code[5] revealed that - for instance - the handling of comments does not appear to properly address situations where multiple quotes occur in a nested fashion:

```
func parseLine(line string, envMap map[string]string) (key string, value
string, err error) {
	if len(line) == 0 {
		err = errors.New("zero length string")
		return
	}

	// ditch the comments (but keep quoted hashes)
	if strings.Contains(line, "#") {
		segmentsBetweenHashes := strings.Split(line, "#")
		quotesAreOpen := false
		var segmentsToKeep []string
		for _, segment := range segmentsBetweenHashes {
			if strings.Count(segment, "\"") == 1 || strings.Count(segment,
"'") == 1 {
				if quotesAreOpen {
```

5 https://github.com/joho/godotenv/blob/c40e9c6392b05ba58e6fea50091ce35a1ef020e7/godotenv.go#L100

```
                        quotesAreOpen = false
                        segmentsToKeep = append(segmentsToKeep, segment)
                    } else {
                        quotesAreOpen = true
                    }
                }

                if len(segmentsToKeep) == 0 || quotesAreOpen {
                    segmentsToKeep = append(segmentsToKeep, segment)
                }
            }

            line = strings.Join(segmentsToKeep, "#")
    }
```

It can be observed that the code would treat a string like `"'" # foo` as one actual literal - contrary to what one might expect.

An attacker with the ability to create dotenv files might abuse this behaviour in order to craft seemingly benign dotenv files, which would inadvertently leak secret information into unrelated environment variables.

It should be noted that no in-depth review of the parser implementation has been performed and that the presence of other issues can hence not be ruled out.

Recommended Action

In order to address this issue, a first step could be to perform a more in-depth review of the used dotenv parser in order to identify further possible issues. All identified problems should subsequently be reported upstream in order to be addressed by the developers of the library.

Reproduction Steps

In order to reproduce this issue, please install godotenv and create a dotenv file `/tmp/test.env` with the following contents:

```
PASSWORD=super secret
USERNAME='as908dzf/"' # Has to be quoted, because we hard-code it. When using
variable references like $PASSWORD, this is not needed
```

Then, please use a command as shown below to observe the behaviour of the parser:

```
λ go/bin/godotenv -f /tmp/test.env env
USERNAME='as908dzf/"' # Has to be quoted, because we hard-code it. When using
variable references like super secret, this is not needed
PASSWORD=super secret
```

It can be observed that the `USERNAME` variable - contrary to what one might expect - not only contains the full comment, but also exposes the contents of the `PASSSWORD` variable within this comment.

Retest Status

The new parser implementation appears to be in `op-cli/command/refsyntax/dotenv/dotenv.go`. Dynamic tests showed that the parser still has troubles with quoting. Consider a `.env` file like this:

```
FOO=f'oo
BAR=baz'
```

Using `bash --posix`, the above snippet would create a single variable `$FOO`:

```
echo $FOO
foo BAR=baz
```

`op`, however, interprets this as two variables, `f'oo` and `baz'` respectively. Another observation was that comments are not handled correctly, e.g., `FOO=bar#`, which is parsed as "bar" in `op` but as "bar#" in shells.

This means that there is still potential for certain attacks.

Further, it was found that similar code to the originally used vulnerable `godotenv` library now appeared in `b5-b5app-release-1507/tools/ci/b5envchecker/loader.go`.

4.1.2 Output of Escape Sequences

Summary

Type	Location	Severity	Status
Code	CLI Tools	Low	Closed

Technical Description

While reviewing the implementation of the CLI tools, it was found that the `op` tool generally does not filter terminal escape sequences when writing data to `stdout`. An attacker with the ability to control (parts of) the tool's output could therefore inject malicious escape sequences.

Depending on the used terminal emulator, this can lead to a number of possible issues. Historically, there have been code execution vulnerabilities in a number of terminal emulators, such as xterm or more recently in xterm.js[6].

However, even without such issues, outputting untrusted escape sequences can result in potential problems. Attackers could for instance use escape sequences in order to display misleading information to the user. One obvious example of such misleading information could be fake password prompts, aiming to trick the user into entering sensitive passwords into their command shell.

Recommended Action

In order to address this issue, it is recommended to filter terminal escape sequences when the `op` tool interacts with a tty.

Reproduction Steps

In order to demonstrate the presence of the problem, the following Python script can be used to generate a file named `poc.json`, which contains a sequence of terminal escape sequences aiming to trick a user into entering their password:

```python
import sys

with open('poc.json', 'w') as f:
    f.write(u"\u001b[?25l\u001b[2J\u001b[1;1HPlease enter your password:")
    for i in range(1000000):
        f.write(u"{}\u001b[50D".format(' '*50))
```

The file `poc.json` can subsequently be added to a vault as shown below:

```
$ op document create poc.json
```

6 https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0542

After adding the file, the target user could be tricked into displaying the file:

```
$ op document get poc.json
```

The escape sequences in the above PoC code will display a password prompt, which could make the user assume they are still interacting with the `op` tool - while in reality, they would enter their password into their command shell, which in turn might for instance save it in the history file.

Retest Status

While dynamically retesting this issue, it was found that `\x1b` (ESC) characters were removed from the output and the rest of the escape sequences are therefore now rendered in a secure way in the terminal.

This results from filtering unprintable characters in the function `FilterUnprintable` defined in `core/input/userinput.go` and utilized by the CLI's IO interface (`op-cli/command/opio/io.go`):

```go
func FilterUnprintable(p string) string {
	charFilter := func(r rune) rune {
		if unicode.IsGraphic(r) || unicode.IsSpace(r) {
			return r
		}
		return -1
	}
	return strings.Map(charFilter, p)
}
```

Accordingly, this issue is considered to be closed.

4.1.3 Race Condition in File Creation

Summary

Type	Location	Severity	Status
Code	op-cli/command/opio/file.go	Low	Partially Fixed

Technical Description

The CLI tools offer a number of ways for creating files. For instance, the `op inject` command can be used to parse template files, fill in secret information and create new files. While reviewing the file creation logic in `op-cli/command/opio/file.go`, it was identified that the implementation of `CreateFile` contains a number of possible race conditions. Further, it also does not appear to properly consider symbolic links.

Please consider the below excerpt from the code:

```
func CreateFile(name string, fileMode os.FileMode, io Stdinout, force bool)
(io.WriteCloser, error) {
   var file *os.File
   _, err := os.OpenFile(name, os.O_WRONLY, fileMode)
   if err == nil {
      // Return error when in pipe
      if io.IsOutputPiped() && !force {
         return nil, fmt.Errorf("file %s already exists", name)
      }

      if !force {
         prompt := fmt.Sprintf("File %s already exists, overwrite it? [Y/n] ",
name)
         if ok, _ := input.ProcessYesNoReturnPromptRW(prompt, io.TTYOrStdin(),
os.Stderr); !ok {
            fmt.Println("Aborting.")
            return nil, ErrAborted
         }
      }
      // This is necessary in order to wipe the contents of the previously
existing file.
      file, err = os.Create(name)
      if err != nil {
         return nil, fmt.Errorf("could not overwrite file %s: %s", name, err)
      }

   } else if os.IsNotExist(err) {
      file, err = os.Create(name)
      if err != nil {
         return nil, fmt.Errorf("could not create file %s: %s", name, err)
      }
   } else {
      return nil, fmt.Errorf("could not open file %s: %s", name, err)
   }

   err = file.Chmod(fileMode)
   if err != nil {
      return nil, fmt.Errorf("could not set permissions on file at %s: %s",
name, err)
   }
   return file, nil
```

It can be observed that the code first checks if the target file already exists. If it exists, it prompts the user whether to overwrite the existing file. Subsequently, it clears the file's contents and changes the file permissions. It should be noted that these operations are not performed in an atomic manner. For instance, the target file could be created just after the code checks for its existence. Furthermore, if the target file is a symbolic link, the code could write files to unintended locations. Depending on the respective environment, this might turn into an exploitable condition.

One obvious example of such an exploitable condition leverages the missing check for a symbolic link: Assume that a legitimate user Alice uses the `op inject` command to create a file named `/tmp/foo`. The attacker, Bob, knows about this and creates a symbolic link `/tmp/foo` beforehand. The symbolic link points to Alice's `~/.bashrc` or a similar sensitive configuration file. Alice now runs the `op inject` command. As the target of the symbolic link already exists, the code will prompt her to overwrite the file `/tmp/foo`. She confirms this prompt and thereby accidentally overwrites her `~/.bashrc`.

In order to improve the attack, Bob might attempt to create the symbolic link `/tmp/foo` directly after the code checked for the file's presence. In this case, no prompt would be shown to Alice.

It should be noted that the above example relies on the `fs.protected_symlinkssysctl` variable being set to zero. However, on the one hand the value of this variable is not under the control of the analyzed codebase. On the other hand, more involved attacks, which do not rely on `fs.protected_symlinks` being set to zero, cannot fully be ruled out.

Recommended Action

In order to address this issue, the following approach for creating a file is recommended:

1 Attempt to open the file using the `O_CREAT|O_EXCL|O_NOFOLLOW` flags, while directly passing the desired file permissions as well

2 When the result of the above operation is `EEXIST`, prompt the user to overwrite the file. In case of `ELOOP`, inform the user about the fact that a symbolic link has been detected and abort the

process

3 If the user confirms, `unlink` the file and re-start the process at step 1

Reproduction Steps

This finding has been identified in a static source code review and has not been reproduced dynamically. Hence, no reproduction steps are provided.

Retest Status

It was found that the updated source code does not reject directories in the given file path that are a symbolic link. This means that the described attack scenario can still be applied on a directory basis. The situation changes in so far that the filename part of the output path has to match the filename of the finally written file. The rest of the location, however can by hidden from the user.

Consider the following steps for Unix to create or overwrite a file in the home directory of a user:

1 Create a new system user with the command `sudo useradd otheruser`

2 Switch contexts using `sudo su - otheruser`

3 Create a symbolic link to the home directory of the `op` user via `ln -s /home/<op_user> /tmp/safedir`

4 Switch back to the attacked `op` user context

5 Execute `echo whatever | op inject -o /tmp/safedir/somefile`

6 Observe the created file and its contents with the path `/home/<op_user>/somefile`

The approval prompt can no longer be bypassed by an attacker by exploiting a race condition, since the file is created by the same system call that is used to check its presence.

It should further be noted, that an existing empty directory provided as output path would be deleted by the call to `os.Remove` used in the implementation on Unix systems.

4.1.4 Access to Parent Environment in op run

Summary

Type	Location	Severity	Status
Code	op-cli/command/run.go	Low	Won't Fix

Technical Description

The `op run` command allows users to start processes, passing secrets stored in 1Password via environment variables. In order to prevent the started process from directly interacting with the running OP8 instance, the code removes a number of sensitive environment variables prior to executing the child process.

However, it should be noted that the started child process could still be able to obtain such environment variables from its parent process by reading `/proc/<ppid>/environ`. This could allow malicious child processes to interact with the running OP8 instance.

Recommended Action

In order to address this issue, the `PR_SET_DUMPABLE` attribute of `prctl` could be used to make the `op` process inaccessible by its child processes.

Reproduction Steps

In order to reproduce this issue, please use the following Python script:

```python
#!/usr/bin/env python3

import os

print("My environment:")
for e in os.environ:
    if 'OP_SESSION' in e: print(e)

stat = open("/proc/self/stat").read()
ppid = stat.split(" ")[3]
print("ppid = " + ppid)

penv = open("/proc/" + ppid + "/environ").read()
entries = penv.split('\0')
print("Parent's environment:")
for e in entries:
    if 'OP_SESSION' in e: print(e)
```

When started with `op run`, the script should first display the environment variables it can directly access. Please note that the `OP_SESSION` variable is not set. However, by reading the environment of its parent process, the script can still access the contents of the `OP_SESSION` variable, as the below excerpt illustrates:

```
λ op run python poc.py
My environment:
GNOME_DESKTOP_SESSION_ID
DESKTOP_SESSION
ppid = 2740520
Parent's environment:
DESKTOP_SESSION=regolith
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
OP_SESSION_secfaulttest1=nveh5yNn4NzHQLbPrVm6vQOQf-ch8o0ZoND_GOGPzk8
```

Retest Status

This issue has been addressed by removing the feature to filter environment variables. Dynamic tests verified this.

Agilebits Inc dba 1Password 's Response:

> *We've investigated this finding and noted that our attempt to filter out CLI specific environment variables was ineffective and set the wrong expectations. As a result, version 2.0.1 of the 1Password CLI no longer attempts to filter out the parent environment.*

This issue is therefore set to the state "Won't Fix".

4.1.5 Missing Privilege Dropping in op run

Summary

Type	Location	Severity	Status
Code	op-cli/command/run.go	Medium	Closed

Technical Description

While reviewing the implementation of the `op run` command, it was found that the `op` tool does not drop its privileges prior to executing its child process. For enabling the new "Biometric Unlock" feature on Linux, the `op` binary belongs to the group `onepassword-cli` and has the `setgid` flag set. This means that the binary will have its group set to `onepassword-cli` even if it is started by a user who is not a member of this group. This mechanism serves the purpose of being able to identify the binary when it communicates with OP8 via a Unix socket.

The fact that the `op` binary does not drop its privileges however means that processes started by `op run` will also have the group `onepassword-cli`, which could enable them to directly interact with the running OP8 instance via its Unix socket.

Recommended Action

During the execution of the project, the issue was communicated to Agilebits Inc dba 1Password, who stated that the issue has already been addressed by dropping privileges in the `op` process.

Reproduction Steps

In order to demonstrate the presence of this issue, the following command can be used:

```
op run -- id -g --name
```

The output of the `id` command started via `op run` indicates, that the child process indeed belongs to the `onepassword-cli` group.

Retest Status

Dynamic tests showed, that the output of `id -g --name` and `op run -- id -g --name` command is now identical. The code was found to set the effective group id when the program starts, to drop its privileges.

Note the below excerpt of the `op-cli/desktopapp/ipc/processprivilege/linux.go` file:

```
func Drop() {
	lock.Lock()
	defer lock.Unlock()

	effectiveGID := os.Getegid()
	realGID := os.Getgid()
	if effectiveGID != realGID {
		savedSetGID = effectiveGID
		if err := syscall.Setegid(realGID); err != nil {
			// According to the Linux kernel documentation, this syscall
should not fail if provided with the real
			// group ID.
			panic(fmt.Errorf("setegid() drop: %s", err))
		}
	}
}
```

Then, when needed for IPC authentication, it restores the saved group ID from the `savedSetGID` variable, as depicted in the excerpt from the `connect` function defined in `op-cli/desktopapp/ipc/ipc_unix.go`:

```
func connect(ctx context.Context) (net.Conn, error) {
	runtimeDir := os.Getenv("XDG_RUNTIME_DIR")
	if runtimeDir == "" {
		runtimeDir = fmt.Sprintf("/run/user/%d", os.Getuid())
[...]
	processprivilege.Restore()
	conn, err := dialer.DialContext(ctx, "unix", socketPath)
	processprivilege.Drop()
```

The `Restore` function was found to be used solely in this context, therefore the issue is considered "Closed".

Since other platforms do not require the assignment of special permissions to the CLI binary, the aspect is considered to affect only Linux installations.

4.1.6 Secret Data in Command Arguments

Summary

Type	Location	Severity	Status
Code	op-cli/command/ signin.go	Low	Closed

Technical Description

While reviewing the implementation of the `op` CLI tool, it was found that the `op signin` command accepts possibly sensitive information via command line arguments. The syntax for invoking the command is `signin [<sign_in_address> [<email_address> [<secret_key>]]]`, which indicates that the `secret_key` for a user account could be provided on the CLI. This might leak the secret key info the user's shell history, as well as in the system's process list (e.g., accessible via the `ps` command).

Recommended Action

In order to address this issue, it is recommended to at least inform users about the potential risks of passing data as command arguments. If using command arguments is required for convenience, it could be advisable to process such arguments only if a specific flag (such as `--insecure`) has been provided.

Reproduction Steps

In order to reproduce the issue, please use the `op signin` command, providing the `secret_key` as a parameter. Then, please inspect the process list of the system, as well as the respective user's shell history file.

Retest Status

When using `op signin` the CLI requests the secrets via `stdin` now. The password is not rendered when typed in by the user. No sensitive info is ever passed by the command line, which means this issue was addressed. The same is true for other sensitive actions, i.e., `op account add`.

4.1.7 ToCToU Weakness in Windows Peer Verification

Summary

Type	Location	Severity	Status
Code	CLI Tools	Medium	Closed

Technical Description

The Windows CLI tools connecting to a named pipe to communicate to the main application are verified based on Authenticode signatures. The signature of the executable is checked and its issuer is ensured to belong to AgileBits. This procedure consists of multiple steps:

1. First, the PID of the connecting process is determined using the API function `GetNamedPipeClientProcessId`
2. The execution path of the process is requested by a call to `QueryFullProcessImageNameW`
3. The file at the execution path is opened
4. The execution path is provided as input to `WinVerifyTrust` to perform the Authenticode verification
5. The subject of the signature issuer, as retrieved from the trust store, is checked to belong to AgileBits

As known to Agilebits Inc dba 1Password this validation steps might be affected by Time-of-check-to-time-of-use (ToCToU) problems. For this reason, step 3 was implemented. It should lock the respective file, to prevent it from being renamed and overwritten before it is read as part of step 4 and 5. This should rule out that an attacker performs a connection with a malicious binary "A" and renames the CLI tools binary to binary "A" (and binary "A" to something else) directly after step 2. Thereby, the legitimate binary would be used in step 4 and 5 although it is not the actual executable.

The implementation of this approach, however, suffers from an implementation issue. It should be noted here, that this perception was gained by a static code review and was not confirmed dynamically.

The issue arises from the selection of the function `std::fs::File::open`[7] for acquiring a file lock. An inspection of its source code[8] showed the internal utilization of `std::fs::OpenOptions::open`:

```
pub fn open<P: AsRef<Path>>(path: P) -> io::Result<File> {
    OpenOptions::new().read(true).open(path.as_ref())
}
```

A further cursory inspection of the documentation for the Windows `std::fs::OpenOptions`

7 https://doc.rust-lang.org/stable/std/fs/struct.File.html#method.open
8 https://doc.rust-lang.org/stable/src/std/fs.rs.html#327-329

extension revealed the default sharing mode[9] requested for the file:

> *By default share_mode is set to FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE. This allows other processes to read, write, and delete/rename the same file while it is open. Removing any of the flags will prevent other processes from performing the corresponding operation until the file handle is closed.*

As the implementation of `std::fs::File::open` does not explicitly use the function `std::os::windows::fs::OpenOptionsExt::share_mode` to clear all flags (note the example in the documentation[10]), it is expected to utilize the mentioned defaults. As a result, it should still be possible to rename and overwrite the file after step 3.

On a successful attack, the connection of an attacker-controlled binary would be accepted by the named pipe to make requests on behalf of the CLI tools.

Recommended Action

Generally, a more in-depth analysis of the situation is recommended. The missing file locking should be added by using appropriate `open` flags on the Windows platform.

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.

Retest Status

During the execution of the retest, it was found that the originally reported issue has been addressed. However, while performing the retest, another issue could be identified, which is described in section 4.2.1 of this document.

4.2 New Findings

The below sections document new security issues observed during the assessment.

4.2.1 Windows Peer Verification depends on File System Type

Summary

Type	Location	Severity
Code	CLI Tools	Medium

Technical Description

As part of the retest performed in the course of the assessment, the verification process utilized for

9 https://doc.rust-lang.org/stable/std/os/windows/fs/trait.OpenOptionsExt.html#tymethod.share_mode
10 https://doc.rust-lang.org/stable/std/os/windows/fs/trait.OpenOptionsExt.html#tymethod.share_mode

processes, such as the CLI tools, connecting to the main application was once again inspected. While the original implementation flaw was addressed as recommended, dynamic tests revealed another fundamental issue in the chosen approach.

The legitimacy of the connecting process is ensured by the means of Authenticode signatures. For this purpose, the file data of the respective binary must be provided to the respective Windows API function. Accordingly, the source file of the relevant process must be determined and read at a step previous to the actual signature verification. Without special precautions this would be prone to Time-of-check-to-time-of-use (ToCToU) attacks. Agilebits Inc dba 1Password was found to be aware of this risk and implemented a measure based on Windows file locking:

1. The file path of the process binary is determined by a call to `QueryFullProcessImageNameW`
2. The file is locked except for read-only operations
3. The file path of the process binary is again determined and compared to the result of step 1
4. The verification process is started

The third step is considered to be introduced to prevent a ToCToU issue, where the attacker replaces the file at the respective path location between step 1 and 2.

Taking into account the generally poor bonding between processes and the respective binary file as stored in the file system, dynamic tests were performed to examine the updating of the file path, in case of file system modifications. During this, it was observed that both the updating of the path information as well as the file locking in general depend on the involved file system type. On NTFS the expected behavior could be observed, while binaries accessed via Samba based file shares were not protected by the described routine against modification on the host system.

Since remote shares are regularly used, especially by Windows users, it is considered feasible to trick for example business customers to start a malicious binary made available this way. On starting it, the executable would connect to the named pipe exposed by the Agilebits Inc dba 1Password main application. The attacker can afterwards exchange the binary at the remote location determined in step 1 against an application signed by Agilebits Inc dba 1Password, e.g. the legit CLI tools. Since both step 2 and 3 take no effect in this scenario, the exchange can happen at any point before the file contents are read and passed to the Authenticode API. This way, the verification will succeed while the named pipe can still be accessed by the malicious binary.

It should be noted that the effect of the described ToCToU attack in its entirety was not verified by dynamic tests.

Recommended Action

It should be determined which file system types actually guarantee to update the information obtained by the `QueryFullProcessImageNameW` API call and implement mandatory file locking preventing content overwrites and file deletions. Afterwards the verification process should be altered to reject the respective pipe access, if the path determined in step 1 points to non supported file system type. The comparison in step 3 should be adjusted accordingly to detect any interim changes to the file system type.

Reproduction Steps

The dynamic tests were performed using a Samba 4.15.13 share hosted on Linux.

The below Rust test programs were compiled with Cargo adding the following lines to the `Cargo.toml` configuration:

```
[dependencies]
winapi = { version = "0.3.9", features = ["winbase" ] }
```

Execute the below program after substituting the `<targe_file_path>` place holder twice. First pointing to a file stored using NTFS and afterwards to a file on the mounted Samba share.

```
use std::{
    thread::sleep,
    time::Duration,
    fs::OpenOptions,
    os::windows::fs::OpenOptionsExt,
};

fn main() {
        let mut open_options = OpenOptions::new();
        let open_options = open_options
            .read(true)
            .share_mode(winapi::um::winnt::FILE_SHARE_READ);

        if let Ok(_) = open_options.open("<targe_file_path>") {
            println!("File should be locked!");
            sleep(Duration::new(30,0));
        }
        else {
            println!("Did not get lock!");
        }
}
```

During the 30 seconds that the file will be locked, try to change the contents or delete/rename the file. Note that for the NTFS location, these actions will be refused. However, in the case of the remote share, the host can still move the file and change its contents by directly accessing the underlying directory without using Samba.

The next program shows that the binary path location will not be updated in the result of the `QueryFullProcessImageNameW` function, when the host is moving the file in the shared directory.

```
use std::{
    ffi::OsString,
    os::windows::ffi::OsStringExt,
    thread::sleep,
    time::Duration,
};
use winapi::{
    um::{
        processthreadsapi::OpenProcess,
        winbase::QueryFullProcessImageNameW,
        winnt::PROCESS_QUERY_LIMITED_INFORMATION,
    },
};

fn file_path() {
        let pid = std::process::id();
        let mut buffer = [0u16; 1024];
        let mut blen = buffer.len() as u32;
        let handle = unsafe { OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, 0,
pid) };
        unsafe { QueryFullProcessImageNameW(handle, 0, buffer.as_mut_ptr(),
&mut blen); };

        let len = blen as usize;
        if len <= buffer.len() {
            if let Ok(path) = OsString::from_wide(&buffer[..len]).into_string()
{
                println!("Path: {}", path);
            }
            else{
                println!("Did not work");
            }
        }
}

fn main() {
        file_path();

        sleep(Duration::new(30,0));
        println!("I'm awake!");

        file_path();
}
```

Place the build executable in the shared directory on the host and ensure that its execution is permitted by setting the correct access rights. Now, execute it on the Windows machine by accepting the general warning shown for running executables from remote shares. During the 30 seconds between the calls to `file_path` function, rename the file on the Samba host and observe that the changes are reflected in the respective directory on the Windows machine. Nevertheless, the test program will output the same path twice. Executing the same binary in an NTFS location will show the updated file path after renaming it.

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 Use of Deprecated Endpoints in CLI Item Share Command

While reviewing the API endpoints used for the item sharing feature recently implemented in the CLI tool, it was identified that the endpoints in question, e.g., `/api/v3/itemshare` are marked as deprecated in the B5 source code. Using deprecated API endpoints is generally considered to be bad practice and should be avoided, especially in recent implementations. As deprecated endpoints are often not updated, they can be considered potential security risks.

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 Lax Parsing for Dotenv Files (Finding 4.1.1)

1Password considers the findings resolved. While Secfault noted some caveats about our remediation, any remaining concerns don't represent any significant risk. As a result we consider these findings closed and fixed.

6.2 Race Condition in File Creation (Finding 4.1.3)

1Password considers the findings resolved. While Secfault noted some caveats about our remediation, any remaining concerns don't represent any significant risk. As a result we consider these findings closed and fixed.

6.3 Access to Parent Environment in op run (Finding 4.1.4)

We've investigated this finding and noted that our attempt to filter out CLI specific environment variables was ineffective and set the wrong expectations. As a result, version 2.0.1 of the 1Password CLI no longer attempts to filter out the parent environment.

6.4 Windows Peer Verification depends on File System Type (Finding 4.2.1)

We accepted this issue and implemented a fix in the July 2023 release.

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
API	Application Programming Interface
CLI	Command Line Interface
ID	Identification
IO	Input/Output
IPC	Inter-Process Communication
NTFS	New Technology File System
PoC	Proof-of-Concept
SSH	Secure Shell