Security assessments for software that matters.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Pentest-Report PDFgear Desktop App & Codebase 02.-03.2026

Cure53, Dr.-Ing. M. Heiderich, Dipl.-Ing. C. Steinauer, H. Crawford

Index

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

“PDF Tasks Made Easy - Read, edit, convert, merge, and sign PDF files across devices, for completely free and without signing up.”

From https://www.pdfgear.com/

The assessment focused on a set of application behaviors that had previously been flagged as potentially suspicious by the tria.ge sandbox analysis platform[1], with the objective of evaluating whether these behaviors represent malicious activity or legitimate software functionality.

The work is tracked as *PGR-01* and represents the first cooperation between Cure53 and PDF GEAR TECH PTE. LTD, which requested this inspection in January 2026. As scheduled, Cure53 carried out the inspection in late February and early March 2026, namely from CW07 to CW09.

Finalizing the notes on resources allocated to *PGR-01,* a total of nine days were invested to reach the coverage expected for this project. A team consisting of three senior testers was created and subsequently assigned to this project’s preparation, execution and finalization.

The work was split into three separate work packages (WPs):

- **WP1**: Code review of flagged behaviors in PDFgear’s Windows client
- **WP2**: Validation of PDFgear’s global hook usage & UI theme implementation
- **WP3**: Analysis of PDFgear’s *root* certificate update activity attribution

Cure53 was provided with source code snippets, the Windows application, as well as all further means of access required to complete the tests. The overarching methodology chosen for this inspection was white-box.

All project preparations were done in early February 2026, namely during CW06, thus ensuring a smooth start of the testing phase. Communications during the test were supported by a dedicated and shared Slack channel set up to connect the PDFgear and Cure53 teams.

All involved personnel from both parties could join the test-related discussions on Slack. Communications were optimized and flowed well. Not many questions had to be asked, largely due to the scope being well prepared and clear. Although Cure53 supplied frequent status updates about the test, live-reporting was not specifically requested for this audit. On the whole, no noteworthy roadblocks were encountered during this *PGR-01* test.

[1] https://tria.ge/250712-qsaa1a11cx/behavioral1

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The Cure53 team managed to get good coverage over the WP1-WP3 scope items, but only managed to spot one individual finding, which was classified as a general weakness with lower exploitation potential. While this is clearly a positive indicator, it should not be viewed as a deterministic indicator of the application's overall security.

In particular, Cure53 was not provided with the full source code. PDFgear provided code segments reported to implement the behaviors flagged by the tria.ge sandbox, and Cure53 reviewed these segments. However, without access to the complete codebase, it was not possible to independently verify that the provided code segments are representative of the actual implementation of these behaviors or to exclude the possibility of alternative code paths implementing the same functionality.

The review of the provided code snippets suggested that certain components had been implemented with security in mind. However, as those excerpts represented only a small fraction of the codebase, they must be seen as offering limited insight into the software's overall security posture and inner dependencies. To ensure an upkeep of this good level, Cure53 recommends continuous audits and tests, especially once new features are introduced to the PDFgear client.

The report will now shed more light on the scope and test setup as well as the available material for testing.

This section will be followed by a chapter that details the technical analysis of the sandbox-flagged behaviors. This is to elaborate on areas of the software in scope that have been covered. Cure53 clarifies what tests have been executed despite no findings associated with various areas.

The report then moves on to discussing the one general issue, clarifying its perimeter and parameters, as well as possible mitigations.

The report ends with a general conclusion regarding the perceived security posture of the scope that is the PDFgear Windows client, with special attention to the array of the flagged behaviors and implementations.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Source code audits & analysis of PDFgear desktop app & codebase**
  - **WP1:** Code review of flagged behaviors in PDFgear's Windows client
    - **Source code:**
      - Code snippets for six out of eight flagged behaviors provided in the form of Markdown files
      - *Code from pdfgear.md*
      - *Code from pdfgear-2.md*
    - **Scope information:**
      - Helper script to detect and terminate running PDFgear instances.
      - Code about the implementation to register PDFgear as default viewer and pinning the application to the taskbar.
      - Source code snippets regarding the implemented update mechanism.
      - Launcher implementation for the integrated file format converter.
      - Source code of launcher to open and edit PDF files.
      - Window events monitoring implementation.
  - **WP2:** Validation of PDFgear's global hook usage & UI theme implementation
    - **Source code:**
      - Code snippets provided in the form of a Markdown file
      - *Code from pdfgear.md*
  - **WP3:** Analysis of PDFgear's *root* certificate update activity attribution
    - **Windows application:**
      - **URL:**
        - https://downloadfiles.pdfgear.com/releases/windows/pdfgear_setup_v2.1.14.exe^
      - **SHA256:**
        - b668557a3874cc6d0775d1b4375b23908226f7af4f84515f3f4791b7d0809d13
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Technical analysis of flagged behaviors

This chapter describes the results of a detailed analysis of specific code snippets identified as the likely triggers for behaviors marked as potentially malicious by the *tria.ge* sandbox. The primary objective of this audit was to determine whether the flagged behaviors represent actual malicious activity or if they constitute false positives generated by the heuristic analysis that the sandbox performs.

Each flagged behavior was reviewed to identify potential malicious intent, suspicious code patterns and deviations from standard operating system interactions. Because the full source code of the application was not provided by the customer, all findings and statements within this section apply solely to the reviewed snippets.

Consequently, while this analysis aims to validate or refute the specific behaviors identified by *tria.ge*[2], it cannot definitively prove the presence or absence of malicious functionality within the application as a whole. To reiterate, only a limited subset of the total source code was available for inspection.

Marked behavior #1: Tasklist execution

Following an investigation into the usage of *tasklist /nh* flagged by *tria.ge*, it has been determined that this command is utilized exclusively to verify whether specific applications are active during certain installation states. This encompassed *InitializeSetup, PrepareToInstall* and *InitializeUninstall*. The process involves the sequence described next.

Process flow:

1. Running *tasklist* and piping the output to a temporary file.
2. Searching the temporary file for specific executables, including *FileWatcher.exe*, *pdfeditor.exe*, *PDFLauncher.exe* and *pdfconverter.exe*.
3. Executing a *taskkill* command if a match is found.

Implementation details:

```
[...]
function CheckSoftRun(strExeName: String): Boolean;
var ErrorCode: Integer
var bRes: Boolean
var strFileContent: AnsiString
var strTmpPath: String
var strTmpFile: String
var strCmdFind: String
var strCmdKill: String
```

[2] https://tria.ge/250712-qsaa1a11cx/behavioral1

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
begin
  strTmpPath := GetTempDir();
  strTmpFile := Format('%sfindSoftRes.txt', [strTmpPath]);
  strCmdFind := Format('/c tasklist /nh|find /c /i "%s" > "%s"',
[strExeName, strTmpFile]);
  strCmdKill := Format('/c taskkill /f /t /im %s', [strExeName]);
  bRes := ShellExec('open', ExpandConstant('{cmd}'), strCmdFind, '',
SW_HIDE, ewWaitUntilTerminated, ErrorCode);
  if bRes then begin
    bRes := LoadStringFromFile(strTmpFile, strFileContent);
    strFileContent := Trim(strFileContent);
    if bRes then begin
      if StrToInt(strFileContent) > 0 then begin
        ShellExec('open', ExpandConstant('{cmd}'), strCmdKill, '', SW_HIDE,
ewNoWait, ErrorCode);
        Result:= true
      end else begin
        Result:= true;
        Exit;
      end;
    end;
  end;
  Result :=true;
End;
[...]
```

This mechanism seems to be a procedural way to ensure that relevant processes are not in-use during an update or uninstallation, as active executables cannot be replaced or deleted by the operating system. No malicious activity was observed, as the code is restricted to managing applications belonging to the PDFgear suite to ensure a successful installation or uninstallation process.

The behavior in question is likely flagged as malicious because the execution of system commands via *ShellExec* - specifically *tasklist* - might be used by malware during the initial discovery stage. Threat actors often enumerate active processes to identify security software, sandboxes, or virtual machine environments before proceeding with an infection.

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Marked behavior #2: *RegExt.exe* & file associations

While analyzing the second marked behavior, it could be determined that this program's logic is used to set the default application for PDFs to PDFgear at install. It also allows the user to pin the application to the taskbar, if requested. While digging deeper into the source code, it became apparent that the software is first registered via standard registry writes to *Software\Classes* to establish its *ProgID* and supported file extensions.

This is generally a common process to register a new application and specify which file extensions the software can handle, as well as adding PDFgear to the right-click *Open with* context menu. However, further analysis revealed that the software circumvents Windows security features designed to prevent defining a default application without explicit user interaction.

Starting with Windows 8, Microsoft introduced protections to prevent silent hijacking of file-type defaults[3]. The system validates the *UserChoice* registry key using a hash value computed from the *ProgId*, *extension*, *user SID*, and a timestamp. To further secure this, Windows places explicit *deny* action on ACL entries on the *UserChoice* key to prevent programmatic writes. The source code utilizes a function named *ResetUserChoice* which gathers all required information to overwrite the user's choice without prompting.

Implementation details:

```
[...]
private static void ResetUserChoice(string keyPath, string progId, string
ext)
{
    try
    {
        var keyRoot =
Microsoft.Win32.RegistryKey.OpenBaseKey(Microsoft.Win32.RegistryHive.Curren
tUser, Microsoft.Win32.RegistryView.Default).OpenSubKey(keyPath, true);

        var hash = DefaultAppHashHelper.GetHash(progId, ext);

        SetUserChoiceKeyAccessControl(keyRoot);
        keyRoot.DeleteSubKeyTree("UserChoice", true);

        Microsoft.Win32.RegistryKey userChoice = null;
        try
        {
            userChoice =
Microsoft.Win32.RegistryKey.OpenBaseKey(Microsoft.Win32.RegistryHive.Curren
tUser, Microsoft.Win32.RegistryView.Default).CreateSubKey($"{keyPath}\\
UserChoice", true);
```

[3] https://learn.microsoft.com/en-us/windows/win32/shell/default-programs

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        }
        catch
        {
            try
            {
                SetUserChoiceKeyAccessControl(keyRoot);
                userChoice =
Microsoft.Win32.RegistryKey.OpenBaseKey(Microsoft.Win32.RegistryHive.Curren
tUser, Microsoft.Win32.RegistryView.Default)
                    .OpenSubKey($"{keyPath}\\UserChoice", true);
            }
            catch { }
        }

        userChoice.SetValue("Hash", hash);
        userChoice.SetValue("ProgId", progId);
    }
    catch { }
}
[...]
```

The process involves several steps, as enumerated and discussed below.

Process flow:

1. The parent key of the registry key *UserChoice* is opened with write-access via *OpenSubKey(keyPath, true)*
2. The software computes a valid hash via *DefaultAppHashHelper.GetHash(progId, ext)*. This seems to be a reverse-engineered implementation of Microsoft's internal computation algorithm.
3. *SetUserChoiceKeyAccessControl(keyRoot)* programmatically removes the *Deny* ACL entries. *keyRoot.DeleteSubKeyTree("UserChoice", true)* deletes the existing protected subkey entirely.
4. The *UserChoice* subkey is recreated with the forged hash and the PDFgear *ProgId* to define PDFgear as the new default application for PDF files.

After this method executes, the Windows operating system is deceived into believing that the user manually selected PDFgear as the default viewer through the official *Settings* UI.

While analyzing the taskbar pinning logic, another non-standard methodology was identified. Instead of using the supported Microsoft developer path, the application utilizes undocumented internal Windows COM interfaces to pin the application onto the taskbar. Microsoft's official stance is that the taskbar is a user-controlled area[4]. Since Windows 10, software has been strictly prohibited from pinning itself without explicit user consent to prevent *taskbar clutter*.

[4] https://learn.microsoft.com/en-us/windows/apps/develop/windows-integration/pin-to-taskbar

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

As shown in the following code excerpt, certain COM interfaces are utilized despite not being officially documented by Microsoft. Searching for specific GUIDs (Global Unique Identifiers) online[5] quickly reveals that this is a well-known bypass technique used to pin an application to the taskbar while circumventing official Windows features.

Referenced undocumented COM interfaces:

```
[...]
public static class ShellHelper
{
    private const string CLSID_TaskbandPin = "90AA3A4E-1CBA-4233-B8BB-
535773D48449";
    private const string CLSID_ShellLink = "00021401-0000-0000-C000-
000000000046";
    private const string IID_IPinnedList3 = "0DD79AE2-D156-45D4-9EEB-
3B549769E940";
    private const string IID_IShellLink = "000214F9-0000-0000-C000-
000000000046";
[...]
```

The behavior observed in *RegExt.exe* might be flagged by automated analysis because of its reliance on a combination of techniques that deviate from standard Windows development practices. While the program is designed to manage file associations and taskbar shortcuts, it interacts with the operating system in an unconventional manner.

Central to this assessment is the bypass of the *UserChoice* protection mechanism. Instead of using officially supported APIs that rely on user-facing confirmation dialogs, the logic programmatically modifies registry keys. This involves calculating a specific hash - a process that appears to mirror internal algorithms in Windows - and adjusting ACLs to remove the default *Deny* restrictions that normally protect these keys from programmatic changes.

Furthermore, the application reads raw binary data from *shell32.dll* to obtain version-specific strings, a method of data gathering that is rarely seen in standard application installers. Additionally, the use of undocumented COM interfaces like *IPinnedList3* allows the application to pin itself to the taskbar without triggering the standard system prompts.

Nevertheless, the primary purpose of the inspected functions is to ensure that the application is set as the default PDF handler and to facilitate pinning the program to the taskbar. It is important to note that the pinning process is only initiated after the user explicitly confirms that this action should be performed.

[5] https://groups.google.com/g/innosetup/c/oskWRaFzV1I?pli=1

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

While these methods are technically functional for completing the installation setup, they mirror defense evasion patterns. Because there is no official documentation supporting these methods - and because they circumvent established Windows security and user-consent boundaries - they are likely to be flagged by automated analysis as suspicious.

To prevent this unconventional implementation from raising future security concerns or triggering false positives, Cure53 recommends relying solely on documented Windows functionality and official APIs to achieve the goals and complete the necessary tasks.

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

Marked behavior #3: Update logic

While analyzing the third behavior marked by *tria.ge*, it was determined that the program logic manages PDFgear's automatic update mechanism. This functionality is designed to streamline the maintenance process for the software by automatically checking for new versions, downloading the latest installer, and initiating the update. The implemented process is a standard lifecycle common to many legitimate desktop applications.

The update flow begins by contacting a remote server to compare the local version against the latest release. It specifically validates that the server-side version is newer to prevent downgrades. Once a new version is identified and the user provides explicit consent through a confirmation dialog, the download is initiated. The software creates a subdirectory within the user's temporary folder (*%TEMP%\PDFgear*\) and downloads the installer executable.

Filedownload implementation:

```
[...]
internal static async Task<string> DownloadUpdateFile(string downloadUrl,
string validationMD5, Action<HttpHelperDownloadResponse> progressReporter,
CancellationToken cancellationToken)
{
    if (string.IsNullOrEmpty(downloadUrl)) return null;
    if (cancellationToken.IsCancellationRequested) return null;

    var tempFolderRoot = Path.GetTempPath();
    var tempFolder = Path.Combine(tempFolderRoot,
       UtilManager.GetProductName());
    var fileName = "";

    try
    {
        if (Uri.TryCreate(downloadUrl, UriKind.Absolute, out var result))
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        {
            fileName = result.AbsolutePath.Split('/').LastOrDefault();
            if (string.IsNullOrEmpty(fileName) ||
             fileName.IndexOfAny(Path.GetInvalidFileNameChars()) > 0)
            {
                fileName = "Setup.exe";
            }
        }

        Directory.CreateDirectory(tempFolder);
    }
[...]
```

To ensure the file was not corrupted or tampered with during transit, the logic performs an integrity check. This is done by comparing the file's MD5 hash against a value provided by the server.

File integrity check:

```
[...]
try
{
    if (File.Exists(filePath))
    {
        if (!string.IsNullOrEmpty(validationMD5))
        {
            using (var tmpStream = File.OpenRead(filePath))
            {
                var md5 = await GetMD5(tmpStream);

                if (string.Equals(md5, validationMD5,
                    StringComparison.OrdinalIgnoreCase))
                {
                    return filePath;
                }
                else
                {
                    throw new ArgumentException(nameof(filePath));
                }
            }
        }
        else
        {
            throw new ArgumentException(nameof(filePath));
        }
    }
}
[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

In the last phase, the application launches the downloaded setup file with */silent /update* flags. Crucially, it requests administrative privileges using the *runas* verb, which triggers a standard Windows User Account Control (UAC) prompt for the user.

Update installation:

```
[...]
private static async Task<bool> ExecuteFile(string filePath, Window window)
{
    return await Task.Run(async () =>
    {
        try
        {
            var psi = new System.Diagnostics.ProcessStartInfo(filePath,
"/silent /update")
            {
                Verb = "runas",
                UseShellExecute = true
            };

            var process = System.Diagnostics.Process.Start(psi);
[...]
```

Several aspects of the update process may be falsely identified as malicious by automated security tools because they mirror behavioral indicators used to detect malware. Primarily, the application's download-and-execute routine is a common pattern which might be flagged by a sandbox or antivirus software due to the fact that the parent process fetches an executable from the Internet and saves it directly to a user-writable temporary directory (*%TEMP%*) before launching it.

This signal is further amplified by the use of the *runas* verb to request administrative privileges. This combination of a temporary staging path and a UAC elevation request might be interpreted as a potential privilege escalation attack. Additionally, the inclusion of the */silent* installation flag - while functionally intended to streamline the update experience - may be flagged because it is a common tactic to minimize user oversight during high-privilege system modifications.

Collectively, these behaviors are benign components of a functional software updater, yet they fall within the scope of defensive heuristic rules. These rules explicitly prioritize detecting unauthorized tampering with the system and stealthy software delivery.

In conclusion, the investigation into the third marked behavior confirms that the program flow represents a typical software update mechanism and no malicious actions are performed. This process is consistent with the maintenance routines found in various other legitimate desktop software.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

Marked behavior #4: Inno setup temporary files

While analyzing this behavior, Cure53 determined that the identified activity is a characteristic byproduct of the Inno setup installer framework used by PDFgear. The directory identified in the *tria.ge* analysis, starting with the prefix *is-*, is a standard temporary staging area created by Inno setup to house the files necessary for the installation and uninstallation processes.

Technical validation confirms that this naming convention is consistent with the official Inno setup's source code[6]. As seen in the framework's logic, the installer generates a unique directory name using a randomized string. The following code snippet shows the identified Inno setup's source code relevant to this analysis.

Inno setup *tmp* directory logic:

```
[...]
function IntToBase32(Number: Longint): String;
const
  Table: array[0..31] of Char = '0123456789ABCDEFGHIJKLMNOPQRSTUV';
var
  I: Integer;
begin
  Result := '';
  for I := 0 to 4 do begin
    Insert(Table[Number and 31], Result, 1);
    Number := Number shr 5;
  end;
end;


function GenerateUniqueName(const DisableFsRedir: Boolean; Path: String;
  const Extension: String): String;
var
  Rand, RandOrig: Longint;
  Filename: String;
begin
  Path := AddBackslash(Path);
  RandOrig := Random($2000000);
  Rand := RandOrig;
```

[6] https://github.com/jrsoftware/issrc/blob/200901[...]/Projects/Src/SetupLdrAndSetup.InstFunc.pas#L277

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
  repeat
    Inc(Rand);
    if Rand > $1FFFFFF then Rand := 0;
    if Rand = RandOrig then
      { practically impossible to go through 33 million possibilities,
        but check "just in case"... }
      raise Exception.Create(FmtSetupMessage1(msgErrorTooManyFilesInDir,
        RemoveBackslashUnlessRoot(Path)));
    { Generate a random name }
    Filename := Path + 'is-' + IntToBase32(Rand) + Extension;
  until not FileOrDirExistsRedir(DisableFsRedir, Filename);
  Result := Filename;
End;
[...]
```

The observation that the flagged directory contains only five random characters suggests that PDFgear utilizes an older version of Inno setup. Current versions in the main branch on GitHub have been updated to use ten random characters for enhanced uniqueness.

This behavior might be flagged by automated analysis because of the inherent nature of Inno setup's handling of custom logic. Beyond simple file extraction, Inno setup allows execution of actions defined via Pascal Scripts. As previously described in the analysis of the first behavior, the installer uses these scripts to execute system commands, such as *tasklist /nh*, to check for running processes.

When a sandbox detects a process originating from a randomized temporary directory (*%TEMP%\is-XXXXX.tmp*) that subsequently performs system discovery tasks like process enumeration, it might trigger heuristic alerts. Although the combination of a transient execution path and command-line utility invocation can mirror malware behavior, these actions are necessary to detect running PDFgear instances for a successful update or removal. As detailed in Marked behavior #1: Tasklist execution, this process is a functional requirement rather than a malicious action.

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

Marked behavior #5: Chain call on *pdfconverter.exe*

The fifth marked behavior was determined to be due to the execution of a separate binary within the PDFgear program files directory. This helper application named *pdfconverter.exe* is launched using a method named *RunProcess*.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Process launcher:

```
[...]
public static void RunProcess(string cmd, string Parameters)
{
    try
    {
        Process p = new Process();
        p.StartInfo.FileName = cmd;
        p.StartInfo.Arguments = Parameters;
        p.StartInfo.UseShellExecute = false;
        p.Start();
    }
    catch { }
}
[...]
```

This method accepts an input parameter named *cmd*, which is used as the input for the creation of a new process on the host system. This method was only used in one location within the provided application's source code.

Converter application launcher:

```
[...]
var converterExe = GetRootDirectoryFile("pdfconverter.exe");
[...]
ProcessManager.RunProcess(converterExe, cmd.ToString());
[...]
```

In addition to the launched helper application, parameters are appended to the process creation command. Since unsafe use of input passed to this process could result in command injection, this was marked as a special concern.

Parameter handling:

```
[...]
OpenConverterCore("app2", type.ToString(), files, password);
[...]
private static void OpenConverterCore(string appTag, string type, string[]
files, string password)
[...]
var cmd = new StringBuilder();

cmd.Append('"').Append(appTag).Append('"').Append(' ');
cmd.Append('"').Append(type).Append('"').Append(' ');
[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

As shown in the above code snippet, it was not possible to validate the value of the *type* parameter with the limited source code access available to the testing team. At the same time, some examples were provided to indicate the intended operation.

Program execution example:

```
"C:\Program Files\PDFgear\pdfconverter.exe" "app2" "WordToPDF"
```

The current implementation may be flagged as suspicious because it starts external processes while silently suppressing all errors. Security tools often flag this pattern since it resembles techniques used by malware to run system utilities without detection. While no malicious actions were identified within the analyzed code, the practice of accepting arbitrary commands and parameters increases risk and reduces visibility. Hence, it makes this behavior appear evasive even if it is legitimate.

To improve, the application should handle exceptions with logging, validate or restrict commands to a known set of executables, and use a fully configured *ProcessStartInfo* object. Capturing output and errors, avoiding *shell* interpreters, and logging executed commands would make the process transparent and safer, reducing the likelihood of being flagged by security tools.

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

***PDFgear Note**: At the time of this report's completion, PDFgear has confirmed that in the actual implementation, exception handling (try–catch) is already implemented, and input validation is also present. However, these implementations have not yet been audited or confirmed by the Cure53 team.*

Marked behavior #6: Chain call on *pdfeditor.exe*

While analyzing the sixth marked behavior, Cure53 observed the same helper application execution implementation as the one documented in the fifth (previous) marked behavior. The very limited scope of the application's source code provided to the testers indicated that the *pdfeditor.exe* helper application also makes use of the *RunProcess* method.

Helper application execution:

```
[...]
public static void OpenEditor(string fullFileName, string action = null){
[...]
var pdfExe = GetRootDirectoryFile("pdfeditor.exe");
var argument = $"\"{fullFileName}\"";
[...]
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
ProcessManager.RunProcess(pdfExe, argument);
[...]
```

Similarly to the fifth marked behavior, Cure53 was unable to fully audit arguments passed to the *RunProcess* method due to very limited source code access. In this case, the *fullFileName* input parameter is passed as an argument to *RunProcess* but its definition is not included within the accessible source code.

Use of *new Process()* represents a typical way of instantiating helper applications. While no malicious actions were identified in the analyzed code, this implementation should be combined with proper exception handling, input validation and output logging. These measures are essential to ensure transparency, prevent command injection, and reduce the risk of this behavior being flagged as suspicious by security tools.

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

***PDFgear Note**: At the time of this report's completion, PDFgear has confirmed that in the actual implementation, exception handling (try–catch) is already implemented, and input validation is also present. However, these implementations have not yet been audited or confirmed by the Cure53 team.*

Marked behavior #7: *SetWindowsHookEx*

While reviewing the seventh marked behavior using the limited access to the relevant source code, Cure53 observed the presence and usage of low-level (Win32) APIs that are hooked when windows in the current process are created and destroyed.

Windows event hooking:

```
[...]
var threadId = GetCurrentThreadId();
hookProc = new HookProc(OnHookProc);
hhook = SetWindowsHookEx(HookType.WH_CBT, hookProc, IntPtr.Zero, threadId);
[...]
```

The system calls a *WH_CBT* hook procedure before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window[7]. Based on the provided source code, the use-cases of these low-level Windows APIs are limited to hooking changes to windows within the current process. No hooking of foreign or external processes could be identified.

[7] https://learn.microsoft.com/en-us/windows/win32/winmsg/about-hooks#wh_cbt

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Using *SetWindowsHookEx* with a *WH_CBT* hook for window lifecycle events can be legitimate, but it may still be flagged as suspicious because windows-hooks are commonly associated with malware techniques. Security tools often treat any use of low-level system APIs as high risk since they are frequently used for code injection, keylogging, or intercepting user activity.

Even when the hook is limited to the current thread, behavioral detection engines may not immediately distinguish it from broader system-wide monitoring, leading to false positives. For handling window lifecycle events within an application's own process, higher level framework mechanisms are generally safer and more maintainable.

In *WinForms*, overriding *WndProc* or using built-in form events such as *HandleCreated*, *Shown*, or *Activated* is typically sufficient. In *WPF*, *HwndSource.AddHook* or standard window lifecycle, events provide similar control without installing a low-level hook. These approaches reduce security concerns and improve readability. They are also less likely to trigger antivirus or endpoint detection alerts.

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

Marked behavior #8: *Root* certificate update

While performing the analysis of the eighth marked behavior, Cure53 monitored the Windows certificate store for changes. This pertained to both the installation process and the first launch of the application. The addition of three certificates was observed; two during the installation process and one after first launch.

Although installation of *root* CA certificates by an application is highly unusual and suspicious, Cure53 was able to validate that all of the added certificates were valid and trusted *root* CA certificates. This likely indicates that the certificates were not installed by PDFgear but by legitimate Windows processes.

CA serial numbers:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 8875640296558310041 (0x7b2c9bd316803299)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=Texas, L=Houston, O=SSL Corporation, CN=SSL.com
Root

Certificate:
    Data:
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        Version: 3 (0x2)
        Serial Number: 3182246526754555285 (0x2c299c5b16ed0595)
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, ST=Texas, L=Houston, O=SSL Corporation, CN=SSL.com EV
Root Certification Authority ECC

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            04:00:00:00:00:01:15:4b:5a:c3:94
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA
```

To validate the legitimacy of the installed certificates, Cure53 compared the serial numbers with official online repositories, in particular:

- *https://www.ssl.com/repository/*
- *https://support.globalsign.com/ca-certificates/globalsign-root-certificates*

Since the installed certificates were validated to be trusted *root* CA certificates, the marked behavior of PDFgear installing its own *root* CA certificates during the installation process or on first launch for malicious purposes can be discarded. However, it is still possible that such certificates may be installed under certain conditions or during the use of specific functionality. During this assessment, Cure53 did not observe installations of any malicious *root* CA certificates on the host system.

***Verdict**: No malicious activity identified. This verdict is based on the source code segments provided and reviewed, focusing on the behaviors flagged by the tria.ge sandbox. Within this scope, no indicators of malicious functionality were observed. However, it is important to highlight that this review did not constitute a comprehensive security assessment of the entire application.*

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible. Each finding has been given a unique identifier (e.g., *PGR-01)* to facilitate any follow-up correspondence, if necessary.

PGR-01-001 WP1: Update process relies on insecure MD5 for integrity check *(Info)*

During the audit of the provided source code, it was identified that the update process validates downloaded files by comparing a locally generated MD5 hash with one retrieved from the server. Because MD5 is cryptographically broken and vulnerable to collision attacks, this mechanism fails to ensure the integrity of the executable. An attacker in a Man-in-the-Middle (MitM) position could intercept the *update traffic* flow and replace the legitimate file with a malicious payload that shares the same MD5 signature.

Affected code:

```
[...]
internal static async Task<string> DownloadUpdateFile(string downloadUrl,
string validationMD5, Action<HttpHelperDownloadResponse> progressReporter,
CancellationToken cancellationToken)
{
    [...]

    try
    {
        if (File.Exists(filePath))
        {
            if (!string.IsNullOrEmpty(validationMD5))
            {
                using (var tmpStream = File.OpenRead(filePath))
                {
                    var md5 = await GetMD5(tmpStream);

                    if (string.Equals(md5, validationMD5,
                           StringComparison.OrdinalIgnoreCase))
                    {
                        return filePath;
                    }
                    else
                    {
                        throw new ArgumentException(nameof(filePath));
                    }
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
                }
            }
            else
            {
                throw new ArgumentException(nameof(filePath));
            }
        }
    }
[...]
```

Although the described attack scenario is largely theoretical in a real-world environment, Cure53 suggests transitioning to a state-of-the-art hashing algorithm, such as SHA-256, to further bolster the overall security of the update process in the tested complex.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

As noted in the Introduction to this PGR-01 report, Cure53 was tasked with performing a source code and behavioral audit of specifically flagged application components within the PDFgear project. Automated sandbox analysis platforms such as tria.ge rely on heuristic detection mechanisms and may flag legitimate software behavior as suspicious. Based on Cure53's review of the provided source code segments and behavioral analysis of the flagged implementations, no malicious functionality was identified within the analyzed components of the PDFgear Windows client, though the analysis was limited to code segments provided by PDFgear and did not encompass the complete source code. The inspection took place in February and March of 2026.

The primary objective of *PGR-01* was to evaluate the technical nature of several functions marked as potentially malicious by the *tria.ge* sandbox. The Cure53 testers reviewed and determined whether these triggers represented actual security threats or false positives resulting from unconventional implementation methods.

Throughout the assessment, communication was maintained via a dedicated Slack channel, ensuring a seamless and efficient exchange of technical queries and clarifications regarding the reviewed code snippets. The PDFgear team was highly responsive, providing prompt answers and additional information on certain aspects whenever requested.

In the frames of this audit, only the code snippets implementing the specific features flagged as potentially malicious were shared with the Cure53 team. As PDFgear is a commercial proprietary product, the customer was restricted from sharing the complete source code.

As the main focus was to evaluate why certain actions were flagged by sandboxes, no overall security audit was explicitly conducted. However, it was agreed with the customer that Cure53 would highlight potential improvements if any vulnerabilities or security-related weak spots were observed. This was done to increase the overall security of the application.

In this regard, a miscellaneous issue was identified in the *update* process concerning the usage of the broken MD5 hashing algorithm, which is currently used to validate the integrity of the downloaded installer. As previously noted, this hashing algorithm is no longer considered secure and is prone to collision attacks, as described in PGR-01-001.

To thoroughly validate whether the application performs any malicious activities, the provided code snippets were reviewed for typical malware indicators, such as keylogging, data exfiltration, password harvesting, or the injection of rogue *root* CA certificates intended to intercept encrypted communications. As detailed throughout this report, no such malicious actions were identified.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The purpose of the inspected functionalities was confirmed as solely related to standard PDF viewer operations, including installation, update process, application startup, and the management of window lifecycle events. This is a positive outcome for PDFgear. At the same time, several of the identified implementations utilize unconventional methods to complete their objectives, which likely explains why certain features were falsely marked as malicious by automated sandboxes.

Specifically, low-level APIs were utilized, and certain Windows security features were bypassed through undocumented COM interfaces and reverse-engineered internal functions. These techniques effectively prevent the user from being prompted to review certain actions, allowing them to occur in the background without direct interaction.

While several methods may have been implemented to streamline the user experience, they deviate from established best practices. Consequently, Cure53 recommends transitioning these implementations to officially documented Windows capabilities. Adhering to standard APIs would not only align the software with Microsoft-supported development model, but would also significantly reduce the likelihood of the application's legitimate actions being flagged by security software in the future.

In conclusion, it can be said that no actual malicious activity could be identified within the provided source code snippets. However, it must be noted that availability of the specific code segments deemed relevant to the flagged behaviors with Cure53 changes the overall scope of possible conclusions. To that end, Cure53 cannot testify to the presence or absence of malicious actions within the application as a whole, as the audit was restricted to a limited subset of the project’s codebase.

Moving forward, PDFgear would benefit from recurrent security assessments focusing on the application itself. While the current audit was limited to validating specific flagged behaviors, a full-scale source code review would be beneficial, as it would offer a way to systematically identify and mitigate latent security vulnerabilities across the entire codebase. As the application evolves and new features are deployed, a stringent analysis from a security perspective ensures that no major weaknesses are introduced.

Cure53 would like to thank the PDFgear team from PDF GEAR TECH PTE. LTD for their excellent project coordination, support and assistance, both before and during this assignment.