Sicherheit von Betriebssystemen – VO 04: Sicherheitskonzepte von macOS – Vertiefung

Rafael Vrecar

Research Group for Industrial Software (INSO)

`https://www.inso-world.com`

FACHHOCHSCHULE
WIENER NEUSTADT
University of Applied Sciences - Austria

Agenda

- Einleitung
- Installation von Programmen im Detail
- Gatekeeper & Quarantine Flag
- Notarisierte Apps
- Exploiting (Beispiel)
- Sandboxing
- Bedrohungsmodell
- Lebenszyklus
- System Integrity Protection (SIP)

Vorbemerkungen

- Apples Software ist im Allgemeinen (Darwin ausgenommen) „Closed Source“
- $\Rightarrow$ bzgl. Informationen *abhängig* von ...
  - Apple
  - Researcher:innen, die „Reverse Engineering“ betreiben
- $\Rightarrow$ Vorlesungseinheit basiert stark auf ...
  (Details im Literaturverzeichnis)
  - Apples Dokumentation
  - Papers von Researcher:innen
- einige der vorgestellten Konzepte werden auch von anderen Betriebssystemen/Plattformen verwendet

Klassifikation von macOS Sicherheitsmechanismen

- POSIX-traditional
  - POSIX-Berechtigungen (ugo+rwx)
- BSD-based
  - Mach Ports
- Apple-proprietary
  - Apples Sandbox
  - System Integrity Protection (SIP)
  - ...

(Vergleiche Jonathan Bar Or (JBO), Talk bei BlueHat IL 2022)

Diskussion

Welche Möglichkeiten gibt es, Programme auf macOS zu installieren?

Mac App Store

© Apple

- Programme werden Prüfung durch Apple unterzogen 👍
- laufen in einer Sandbox 👍
- Apple bezeichnet Mac App Store als „sichersten Ort“, um Software für den Mac zu beziehen 👍

- Sandbox schränkt auch Feature-Möglichkeiten ein 👎
  (in das System eingreifende Plugins, bspw. Dropbox)
- aus Developer:innen-Sicht: zusätzliche Kosten durch Abgaben an Apple 👎
- geringere Flexibilität bzgl. Updates, Releases: abhängig von Apples Review-Prozess 👎

Gatekeeper

© Apple

- prüft Anwendungen, die nicht über App Store installiert wurden
- **Developer ID:**
  - kann bei Apple bezogen werden, um sich als Developer:in auszuweisen
  - notwendig, um Apps für Gatekeeper zu *signieren*
- zusätzlich *Notarisierung*: Apps werden an Apple zur Security-Prüfung übermittelt ⇒ wenn keine Fehler gefunden, wird „Ticket“ an Softwarepaket angehängt, um Notarisierung nachzuweisen.
- seit macOS 10.15 (Catalina, 2019): Notarisierung wird per default eingefordert

Quarantine Flag

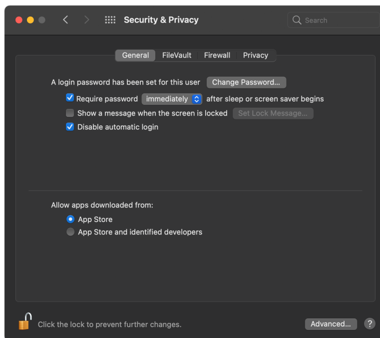
- wird aus dem Internet geladenen Anwendungen gesetzt
- extended Attribute
- entfernbar mit:

  ```
  xattr -d com.apple.quarantine TotallyNotAVirus.jpg.app
  ```

- i.d.R. nach erster Ausführung & damit verbundener User:innen-Bestätigung entfernt
- von Gatekeeper verwendet

Systemeinstellungen

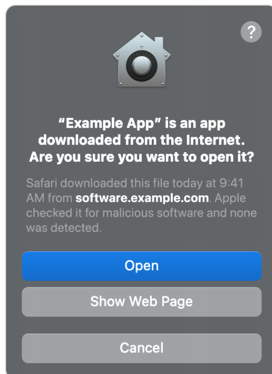
- per default: Apps aus dem App Store und von identifizierten Entwickler:innen zur Ausführung berechtigt
- kann auf App Store eingeschränkt werden



© Apple

Notarisierte App erstmalig öffnen

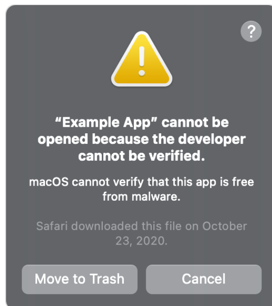
- einmalige Rückfrage, ob App wirklich geöffnet werden soll
- sofern erlaubt, wird nicht weiter nachgefragt, solange sich .app-Datei nicht ändert, abhängig von Update-Implementierung: erneute Rückfrage nach Update



© Apple

Nicht notarisierte App erstmalig öffnen

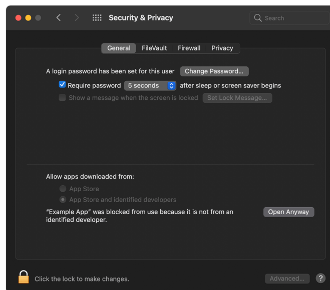
- per default: verboten
- oft begründete Fälle, die nicht notarisierte App erfordern: vertrauenswürdige:r Developer:in, App alt bzw. lange nicht aktualisiert, ...



© Apple

Nicht notarisierte App öffnen in Settings erlauben

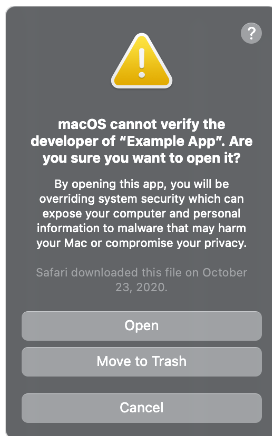
- nach dem Pop-Up in die System Preferences (Systemeinstellungen) navigieren
- … Security & Privacy beinhaltet nun neuen Punkt …



© Apple

Nicht notarisierte App nach Erlaubnis öffnen

- nach Erlauben des Öffnens erscheint nun beim erstmaligen Öffnen folgende Meldung



© Apple

Sicherheitsmechanismen aushebeln (Beispiel, bereits fixed)

- Kontext: schadhaftes Shell-Script als .app-Datei „getarnt“
- Gatekeeper, File Quarantine Flag & Notarisierung schlagen nicht an
- Proof-of-Concept .app-Datei nicht als „bundle“ klassifiziert, da keine Info.plist-Datei enthalten (fehlerhafter Check)
- $\Rightarrow$ Öffnen erlaubt, obwohl(!) executable
- Fix: Prüfung, ob .app-Datei oder `Contents/MacOS` enthalten $\Rightarrow$ bundle

.app-Dateien & darüber hinaus

- Anwendungen: .app-Dateien, liegen i.d.R. im `/Applications`-Verzeichnis (wenn für alle User:innen zugreifbar), können aber prinzipiell überall liegen (vgl. .exe-Dateien unter Windows)
- Öffnen von „standalone“ Apps oft nicht ausreichend, evtl. weitere Änderungen am System notwendig ...
  - Apps automatisch mit Login starten
  - Apps, die im Hintergrund laufen müssen (Dienste etc.)
  - Apps, die Finder o. a. Apps „modifizieren“ bzw. mit ihnen interagieren müssen (bspw. Dropbox)
  - ...

$\Rightarrow$ Installer (i.d.R. .pkg-Dateien)

Verwendung von Installern & damit verbundene Implikationen

- Installation erfordert möglicherweise privilegierte Aktion $\Rightarrow$ Installer benötigt Root-Rechte
- böswilliger Installer $\Rightarrow$ böswillige Aktionen, wenn User:innen dazu gebracht werden, Installer erweiterte Berechtigungen geben
- … Szenario des per se böswilligen Installers hier nicht weiter besprochen, da inhärent schadhaft
- interessant: schlecht programmierter Installer bekommt Aktionen zur Ausführung „untergeschoben“
- z. B.: User:in wird dazu gebracht, Script ohne(!) root-Rechte auszuführen $\Rightarrow$ schlecht programmierte Installer könnten dazu führen, das besagtem schadhaften Script *Privilege escalation* gelingt

Schwachstelle(n) & Ablauf einer Attacke

- `AuthorizationExecuteWithPrivileges` API zur Ausführung eines Binaries als root (Fenster mit Passwortabfrage)
- ist deprecated, da sie das auszuführende Binary nicht validiert
- beispielhafter Ablauf einer Attacke:
  1. Infektion (E-Mail, Trojan, ...): Script (nicht als root) ausführen
  2. „Watch & Wait": Script wartet auf infizierbaren Installer
  3. Exploit: Script nutzt Lücke aus, indem es Datei infiziert.
     Wie? Datei wird in schreibbarem TMP-Verzeichnis platziert, um in weiterer Folge als root ausgeführt zu werden, kann als nicht-privilegierte:r User:in verändert werden ⇒ (rechtzeitig!) Code einfügen ⇒ User:in autorisiert root-Ausführung ⇒ Schadcode wird ausgeführt
  4. Angreifer:in hat nun root-Rechte!

Implikationen der Schwachstellen & Schutz

- wie von Wardle erläutert: viele Anwendungen hatten eine derartige Lücke bereits in ihren Installern (Zoom, Little Snitch, Sophos, Google Chrome, VMWare Fusion, DropCam, ...)
- z. B. Process Monitor (bspw. von objective-see.com) verwenden, um Vorgänge nachzuvollziehen
- immer hinterfragen: Warum könnte dieser Installer root-Rechte benötigen, und will ich das?
  $\Rightarrow$ Anwendung lokal installieren reicht oft aus (bspw. Zoom in 2020, mittlerweile repariert)
- für Entwickler:innen: `SMAppService` verwenden, da `SMJobBless` (von Wardle 2017 erläuterte Alternative) mit macOS 13 Ventura (Oktober 2022) deprecated ist

Paketmanager

- nicht von Apple, aber Dritten
- machen Update-Prozess bzw. Verwaltung von Anwendungen einfacher
- Beispiele:
  - Homebrew (`https://brew.sh/`)
  - MacPorts (`https://macports.org/`)

Diskussion

Vor- & Nachteile
der verschiedenen Installationswege

Vor- & Nachteile der verschiedenen Installationswege

- Flexibilität für Entwickler:innen & User:innen 👍
- plattformübergreifend ähnliche User:innen-Experience (Installer unter macOS & Windows) 👍
- Software-Verteilung auch ohne Abgabe an/Kontrolle durch Apple möglich 👍
- Updates == Nightmares ?? 👎
  (kein zentraler Mechanismus, wenn nicht über App Store bezogen)
- Installer (.pkg) nehmen potenziell tiefergehende Änderungen vor … wurde auch über Deinstallation nachgedacht? Deinstallation überhaupt sinnvoll möglich? 👎
- unterschiedliche Features von App, wenn Installation über App Store vs. extern, vgl. iStat Menus, Parallels Desktop, … 👎

Sandboxing: Konzept

- $=$ Isolierung von Anwendungen
- Zugriff auf Systemressourcen wird eingeschränkt
- begrenzt potenziellen Schaden, den Apps – selbst wenn kompromittiert – dem System zufügen können
- wesentlich: Container-Konzept $\Rightarrow$ jede App bekommt eigenes „home“-Verzeichnis
- Container-Daten unter `/Library/Containers/<bundleId>`
- Apps können nur auf „Data“-Unterverzeichnis des Containers zugreifen $\Rightarrow$ fungiert als eigenes „home“-Verzeichnis
- seit 2012: Sandboxing für Apps aus Mac App Store vorgeschrieben
- externe Apps: optional(!)
- noch immer „closed source“ & technisch nicht dokumentiert

Sandboxing: Hintergrund

- mit Mac OS X 10.5 (Leopard, 2007) als optionale Sicherheitsfunktion eingeführt ⇒ konnte leicht umgangen werden
- ursprünglich: Funktion nur über undokumentierte, Scheme-ähnliche Sprache Sandbox Profile Language (SBPL) konfigurierbar
- von Apple fast ausschließlich für Systemprozesse verwendet
- mit Mac OS X 10.7 (Lion, 2011): „App-Sandbox“ als benutzbare Schnittstelle zur Sandbox eingeführt ⇒ für Anwendungen von Dritten gedacht

Sandboxing: Bedrohungsmodel (Threat Model)

- öffentliches Marketing Material: Sandboxing schützt „Ihren Computer und Ihre Daten“ vor Anwendungen, die sich aufgrund von unbeabsichtigten Programmierfehlern („Bugs“) falsch verhalten, und sogar vor Anwendungen, die „von bösartiger Software kompromittiert“ werden
- Developer:innen-Dokumentation: App-Sandbox wurde entwickelt, um Schaden für System & Daten der User:innen einzudämmen, wenn App kompromittiert
- *absichtlich bösartige* Software ausdrücklich aus Bedrohungsmodell ausgeschlossen, …

Sandboxing: Bedrohung in der Praxis

- … allerdings kein praktischer Unterschied zwischen absichtlich bösartigen Anwendungen & jenen, die kompromittiert werden ⇒ in beiden Fällen bösartiger Code ausgeführt
- Sandboxing kann per se bösartige Anwendungen nicht davon abhalten, offiziell gewährte Privilegien zu missbrauchen
- Sandboxing soll absichtlich bösartige Software auf ihre begrenzten Möglichkeiten einschränken
- Patente von Apple begründen Notwendigkeit von Sandboxing: „[…] Programm ein bösartiges Programm sein kann, das entwickelt wurde, um absichtlich Schäden zu verursachen“ ⇒ durch Sandboxing können „solche Schäden stark reduziert werden […]“

Sandboxing: Lebenszyklus

- 1) Konfiguration $\Rightarrow$ 2) Initialisierung $\Rightarrow$ 3) Durchsetzung
- Sandbox-Konfiguration: Prozess, mit dem Sandboxing während Entwicklungsphase einer App aktiviert & konfiguriert wird
- während Laufzeit einer App wird Sandbox zunächst initialisiert
- wenn App Ressourcen anfordert, die durch Sandboxing geschützt sind, prüft Betriebssystem, ob App den angeforderten Vorgang durchführen darf

Sandboxing: Konfiguration

- App-Sandbox über Berechtigungen aktiviert & konfiguriert, welche Programmen bestimmte Ressourcen & Fähigkeiten zuweisen
- Berechtigungen = Schlüssel-Wert-Paare
- Strings als Schlüssel verwendet, um Fähigkeit zu identifizieren
- Werte können verwendet, um Berechtigungen weiter zu konfigurieren; können verschiedene Typen haben – im einfachsten Fall boole'sche Werte
- um verpflichtende Sandbox-Anforderung für Einreichungen im Mac App Store zu erfüllen, aktivieren Entwickler:innen Sandbox-Berechtigung `com.apple.security.app-sandbox`
- App-Sandbox standardmäßig für Apps aktiviert, die mit Xcode erstellt wurden

Sandboxing: Berechtigungen

- Apps mit Sandbox in Ausführung eingeschränkt; können z. B. nicht auf User-Files, Mikrofon, Kamera zugreifen
- Entwickler:innen können zusätzliche Berechtigungen festlegen, um auf bestimmte Ressourcen zugreifen zu können:
  - Berechtigung `com.apple.security.device.microphone` ⇒ auf Mikrofon zugreifen
  - Berechtigung `com.apple.security.network.client` ⇒ Zugriff auf das Netzwerk ermöglicht
- insgesamt etwa 50 Sandbox-bezogene Berechtigungen dokumentiert
- Berechtigungen als Eigenschaftsliste in ausführbare Datei der App eingebettet
- Integrität durch Codesignatur der App geschützt

Sandboxing: App Store Review

- wird vor Veröffentlichung in Mac App Store durch Apple durchgeführt
- statische & dynamische Analyse
  $\Rightarrow$ Details in unserer LVA „Mobile Security“ ;)
- prüft u. a. Berechtigungen der App
- wenn unklar ist, warum bestimmte Berechtigungen benötigt, muss Entwickler:in möglicherweise schriftliche Erklärung abgeben
- Apple behält sich Recht vor, eingereichte Apps abzulehnen, wenn böswillig oder bestimmte Berechtigungen übermäßig ausnutzen

Sandboxing: Initialisierung – Profil erstellen

- alle Anwendungen starten ohne Sandbox & können erst später in Sandbox wechseln
- zu Beginn des Startvorgangs prüft *Dynamic Linker*, ob App Sandbox-Berechtigung aktiviert hat
- wenn ja: *libsandbox* (`/usr/lib/libsandbox.dylib`) aufgerufen & extrahiert eingebettete Berechtigungen, um endgültiges „Sandbox-Profil“ zu kompilieren
- $\Rightarrow$ Profil wird zwischengespeichert & bei nachfolgenden Starts direkt verwendet $\Rightarrow$ muss nicht mehrfach kompiliert werden

Initialisierung – Kontrollübergabe

- mit diesem Profil macht *Dynamic Linker* Systemaufruf, um Sandbox zu initialisieren, bevor Kontrolle an Anwendungscode übergeben (d. h. an `main()`-Funktion oder an Konstruktoren der gemeinsamen Bibliothek)
- da Sandbox normalerweise initialisiert, bevor Kontrolle an Code von Dritten übergeben ⇒ sollte vor bösartigem Anwendungscode schützen
- *Dynamic Linker* läuft im Kontext der Anwendung

Initialisierung – Fehlerbehandlung & Vergleich zu iOS

- wenn Initialisierung der Sandbox fehlschlägt, oder wenn möglich, dass App-Code ausgeführt, bevor Initialisierung abgeschlossen, sind *alle(!)* von App-Sandbox gebotenen Schutzmaßnahmen *wirkungslos(!)*
- im Vergleich dazu: Sandbox unter iOS von Kernel durchgesetzt $\Rightarrow$ Apps ohne Container werden beendet

Durchsetzung

- während Normalbetrieb rufen Apps OS auf, um grundlegende Aktionen in ihrem Namen durchzuführen (z. B. Öffnen von Dateien, Senden/Empfangen von Netzwerkpaketen, Kommunikation mit anderer Software, ...)
- in Sandbox-Umgebung: OS muss sicherstellen, dass aufrufende App angeforderte Aktionen ausführen darf
  - macOS-Kernel ruft Sandbox-Kernel-Erweiterung auf, welche während Initialisierung kompiliertes Sandbox-Profil verwendet, um zu entscheiden, ob Anfrage zulässig
  - $\Rightarrow$ angeforderte Ressource oder Fehlercode wird an App zurückgegeben

SIP: Allgemeines

- dt. „Systemintegritätsschutz“ wurde in macOS 10.11 (El Capitan) eingeführt, kann deaktiviert werden (siehe Literaturverzeichnis für Details)
- auch bekannt als „Rootless“
- bietet systemweit auferlegte Einschränkungen $\Rightarrow$ betreffen auch root(!)
- root verliert Berechtigungen für SIP-geschützte Dateien, Geräte, Prozesse, ...
- soll Kernbetriebssystem vor dauerhaftem Verlust der Integrität schützen
- SIP ist anwendungsbasierte Zugriffskontrolle, keine(!) benutzer:innenbasierte

SIP: Design

- macOS hat Wurzeln in UNIX (Darwin)
  $\Rightarrow$ es gibt root & normale User:innen
- SIP unterscheidet zwischen eingeschränkten & nicht eingeschränkten Objekten
- eingeschränkte Objekte erhalten Schutzstufe, welche sie u. a. auch vor root schützt
  - als Dateien können sie nicht geändert oder entfernt werden
  - als Prozesse können sie nicht debugged oder manipuliert werden

SIP: Implementierung 1/2

- Implementierung von SIP läuft auf Sandbox-Profil hinaus
  $\Rightarrow$ `platform_profile`
- = Default-Profil für fast alle Anwendungen im System, auch, wenn diese (mit wenigen Ausnahmen) nicht in der Sandbox laufen
- beim Booten startet `launchd(8)` `/usr/libexec/rootless-init` $\Rightarrow$ Binärprogramm, das die `/System/Library/Sandbox/rootless.conf` öffnet
- Konfigurationsdatei `rootless.conf` wendet Schutzmaßnahmen auf die darin angeführte Dateien an
- Schutz wird durch Setzen des `com.apple.rootless` extended Attributes auf den Label-Wert aus der conf-Datei für das Verzeichnis gesetzt, wobei bestimmte Unterverzeichnisse durch Angabe von ausgeschlossen werden

SIP: Implementierung 2/2

- bei Verwendung der `rootless.conf`-Datei werden Schutzmaßnahmen auf Dateien anwendet, welche Apple als vertrauenswürdig erkennt
- wenn System von früherer Version auf 10.11 aktualisiert wird, werden alle Dateien, welche nicht von Apple stammen, nicht als eingeschränkt markiert werden ⇒ können geändert & manipuliert werden – aber wenn entfernt, nicht wieder hinzugefügt
- `/System/Library/Sandbox/Compatibility.bundle/Contents/Resources/paths` ist zweite Ausnahmeliste, meist für Drittanbietenden-Binärdateien, welche sich in SIP-geschützte Speicherorte verschoben haben

Dateisystem-Schutzmaßnahmen

- Unterscheidung zwischen eingeschränkten & nicht eingeschränkten Objekten, wird durch zwei Methoden umgesetzt: „restricted“-Flag & erweitertem `com.apple.rootless`-Attribut
- „restricted“-Flag:
  - sichtbar durch `ls -lo`
  - Versuch, mit `chflags(2)` zu entfernen, vereitelt, wenn SIP aktiv
- erweitertes `com.apple.rootless`-Attribut
  - eingeschränkten Objekten zugewiesen (Links, Verzeichnisse, ...)
  - durch `ls -le` sichtbar
  - für die meisten Objekten enthält es keinen Wert
  - in einigen Fällen enthält Bezeichnung aus Datei `rootless.conf`

Berechtigungen: Überblick

- speziell für SIP eingeführte Berechtigungen an Präfix `com.apple.rootless` erkennbar
- macOS & iOS Entitlements: `http://newosxbook.com/ent.jl`
- Apple muss sich Recht vorbehalten, alle Dateien & Verzeichnisse im System zu ändern
  $\Rightarrow$ erforderlich, um z. B. Systemaktualisierungen durchzuführen
- geschieht, indem benötigten ausführbaren Dateien bestimmte Berechtigungen zugewiesen werden
- Berechtigungen = Daten, die in signierter Anwendung enthalten & System mitteilen, dass Anwendung berechtigt, bestimmte Aufgaben auszuführen

Berechtigungen: SIP-spezifisch

- für SIP relevante Berechtigungen:
  - `com.apple.rootless.install`: hebt alle Dateisystembeschränkungen von SIP auf
  - `com.apple.rootless.install.heritable`: identisch, aber Kindprozesse erben Berechtigung
- Apple lässt nicht zu, dass beliebiges Programm mit genannten Berechtigungen signiert wird, aber einige von Apple signierte Programme im System besitzen diese Berechtigungen
- Berechtigungen einer ausführbaren Datei können mit Dienstprogramm `codesign` betrachtet werden

Berechtigungen: `system_installd`

- Beispiel ist `system_installd`, ein Teil des `PackageKit`-Frameworks, das für Installation von Apple-signierten Paketen verantwortlich
  - `system_installd` hat die mächtigere `com.apple.rootless.install.heritable`-Berechtigung
  - wurde in Vulnerability CVE-2022-22583 verwendet: Schwachstelle in macOS, welche es Angreifer:innen ermöglicht, SIP zu umgehen ⇒ volle Kontrolle über System erhaltbar

SIP im abschließenden Überblick

- SIP versucht, benutzer:innenbasierte Berechtigungen durch anwendungsspezifische Berechtigungen zu ersetzen
- kein Schreibzugriff auf: `/System`, `/bin`, `/sbin`, `/usr` (außer `/usr/local`)
- kein Zugriff auf von Apple signierte Prozesse, inklusive Speicherdumping, `ptrace()` und DTrace-Zugriff
- kein Laden von unsignierten Kernel-Erweiterungen (`kext`)
- kein Schreibzugriff auf Boot- und SIP-bezogene NVRAM-Einstellungen
- schützt symbolische Links innerhalb von `/etc`, `/tmp`, `/var`
- schützt Systemanwendungen unter `/Applications`
- schützt vor dem Entfernen ausgewählter `launchd`-Dienste.

Zusammenfassung

- verschiedene Ebenen, wo Schutzmechanismen ansetzen können.
- verschiedene Arten, Programme zu installieren
  - Mac App Store
  - heruntergeladene .app-Dateien (vgl. Signatur & Notarisierung!)
  - Installer
  - Paketmanager
- schlecht programmierte Installer können zu hohen Sicherheitsrisiken führen
- Sandboxing soll verhindern, dass kompromittierte Apps schaden außerhalb ihres „home“-Verzeichnisses anrichten können
- System Integrity Protection schwächt root-Berechtigungen ab, gewisse Teile des Systems während des Betriebs unveränderlich

Literaturverzeichnis 1/6

- Apple Developer. App Sandbox. https://developer.apple.com/documentation/security/app_sandbox.
- Apple Developer. Disabling and Enabling System Integrity Protection. https://developer.apple.com/documentation/security/disabling_and_enabling_system_integrity_protection.
- Apple Developer. Distributing software on macOS. https://developer.apple.com/macos/distribution/.

Literaturverzeichnis 2/6

- Apple Developer. Notarizing macOS Software Before Distribution. `https://developer.apple.com/documentation/security/notarizing_macos_software_before_distribution`.
- Apple Developer. Signing Your Apps for Gatekeeper. `https://developer.apple.com/developer-id/`.
- Apple Support. About System Integrity Protection. `https://support.apple.com/en-us/HT204899`.
- Apple Support. Safely open apps on your Mac. `https://support.apple.com/en-us/HT202491`.

Literaturverzeichnis 3/6

- Eclecticlight.co. Quarantine, SIP, and MACL: macOS per-file security controls. `https://eclecticlight.co/2020/01/30/quarantine-sip-and-macl-macos-per-file-security-controls/`.
- Heise.de. Tipp: System Integrity Protection (SIP) deaktivieren. `https://heise.de/-3946690`.
- Jonathan Levin: MacOS and iOS Internals, Volume III: Security & Insecurity. Technologeeks Press. 2016. ISBN: 978-0991055531.

Literaturverzeichnis 4/6

- Maximilian Blochberger, Jakob Rieck, Christian Burkert, Tobias Mueller, Hannes Federrath. 2019. State of the Sandbox: Investigating macOS Application Security. In Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society (WPES'19). Association for Computing Machinery, New York, NY, USA, 150–161. `https://doi.org/10.1145/3338498.3358654`.
- Microsoft Israel R&D Center. BlueHat IL 2022 - Jonathan Bar Or (JBO) - Learning macOS Security by Finding Vulns. `https://www.youtube.com/watch?v=jBvE0kciSx8`.

Literaturverzeichnis 5/6

- Perception-Point.io. Technical Analysis Of CVE-2022-22583: Bypassing macOS System Integrity Protection (SIP). `https://perception-point.io/technical-analysis-of-cve-2022-22583-bypassing-macos-system-integrity-protection/`.
- Objective-See.com, Patrick Wardle. The ’S’ in Zoom, Stands for Security. `https://objective-see.com/blog/blog_0x56.html`.

Literaturverzeichnis 6/6

- Speakerdeck.com, Patrick Wardle. Bundles of Joy: Breaking macOS via Subverted Applications Bundles. https://speakerdeck.com/patrickwardle/bundles-of-joy-breaking-macos-via-subverted-applications-bundles.
- Speakerdeck.com, Patrick Wardle. [DefCon 2017] Death by 1000 Installers; it's All Broken!. https://speakerdeck.com/patrickwardle/defcon-2017-death-by-1000-installers-its-all-broken.
- Joshua Long. n-1 and n-2: Should we really trust in you? An examination of macOS security updates, 2021. https://objectivebythesea.com/v4/talks/OBTS_v4_jLong.pdf

Vielen Dank!

`https://establishing-security.at/`